FreeCore : un système d'indexation de résumés de document sur une Table de Hachage Distribuée (DHT)

Bassirou Ngom

Sorbonne Université

Université Cheikh Anta Diop de Dakar

École doctorale Informatique, Télécommunications et Électronique (Paris)

Laboratoire d'informatique de Paris 6 (LIP6) / Équipe de recherche : REGAL

École doctorale Mathématique-Informatique (Dakar)

Laboratoire d'informatique de Dakar (LID) / Équipe de recherche : Bases de Données et Data Mining

Titre de la thèse

FreeCore : un système d'indexation de résumés de documents sur une Table de Hachage Distribuée (DHT)

Présentée par :

Bassirou NGOM

Dirigée par :

M. Mesaac MAKPANGOU	HDR, Chargé de recherche	INRIA, Sorbonne Université	Directeur de thèse
M. Samba NDIAYE	Maître de conférences	UCAD	Co-Directeur de thèse

Soutenue le 13/07/2018

Devant le jury composé de :

M. Lionel SEINTURIER	Professeur	Université de Lille	Rapporteur
M. Eddy CARON	Maître de conférences (HDR)	ENS de Lyon	Rapporteur
Mme. Maria POTOP-BUTUCARU	Professeur	Sorbonne Université	Examinateur
M. Moussa LO	Professeur	UGB (St Louis - Sénégal)	Examinateur
M. Pierre SENS	Professeur	Sorbonne Université	Examinateur

Remerciements

Depuis l'âge de raison jusqu'au jour où j'eus terminé ma thèse, j'ai observé, appris, écrit, lu sans relâche, et ma vie fut comme un long pensum que je n'aurais jamais supporté sans la participation d'un grand nombre de personnes.

> « Aucun de nous, en agissant seul, ne peut atteindre le succès. »
>
> *Nelson Mandela*

Je vais essayer de remercier ces personnes tout en sachant que mes mots ne suffiront pas pour décrire le sentiment de gratitude et de reconnaissance envers eux.

En premier lieu, je tiens à remercier mes directeurs de thèse, messieurs Mesaac MAKPANGOU et Samba NDIAYE, pour la confiance qu'ils m'ont accordée en acceptant d'encadrer mes travaux doctoraux. Je les remercie pour leur soutien constant, leurs maints conseils et pour toutes les heures qu'ils ont consacrées à diriger cette recherche.
J'aimerais également dire à quel point j'ai apprécié la grande disponibilité du Dr MAKPANGOU et l'humilité sans faille qu'il a fait preuve à mon égard durant toute la thèse. Il m'accueillait à Paris aussi bien au laboratoire que chez lui sans aucune condition.
En dehors du travail d'encadrement, j'ai été extrêmement sensible aux qualités humaines d'écoute et de simplicité que les sieurs MAKPANGOU et NDIAYE m'ont montrées tout au long de cette thèse. Leurs caractères continueront de m'inspirer pour toujours.

Mes remerciements vont également aux membres du jury pour avoir accepté d'évaluer mon travail. Je remercie les professeurs Lionel SEINTURIER et Eddy CARON qui, malgré les emplois du temps chargés, ont accepté de lire et de corriger cette thèse. Mes remerciements au Professeurs Maria POTOP-BUTUCARU et Moussa LO pour s'être rendus disponibles pour mon jury de thèse.

Je remercie le Service de Coopération et d'Action Culturelle (SCAC) et l'Institut National de Recherche en Informatique et en Automatique (INRIA) pour avoir financé mes séjours à Paris et pour m'avoir permis de bien mener mes recherches au sein du LIP6.

Je souhaiterais exprimer ma gratitude à monsieur Philippe DARCHE de s'être rendu disponible pour les relectures de mes travaux et les précieux conseils qu'il m'a prodigués. Je le remercie aussi pour son accueil chaleureux dans le bureau et ses encouragements qui m'ont beaucoup aider.

Je remercie aussi les membres de l'équipe DELYS (ex REGAL) du LIP6 qui m'ont chaleureusement accueilli et facilité mon intégration et mon épanouissement au sein de l'équipe. Mes sincères remerciements à Pierre SENS pour les relectures de mon manuscrit et pour son aide précieuse pour le financement de la thèse. Je remercie également Luciana ARANTES pour son soutien constant, ses encouragements et ses corrections. Également, j'exprime ma gratitude à Eugène KAMDEM mon gestionnaire qui s'est bien occupé de moi durant mes séjours à Paris.

Je remercie l'ensemble du corps professoral de la section informatique de l'UCAD. Mes remerciements à tous les membres de l'équipe Base de données et Dataminig Idrissa SARR, Modou GUEYE, Aliou BOLY et Ndiouma BAME pour leurs conseils et discussions fructueux. Je remercie aussi mes autres collègues et amis Ibrahima GUEYE, Ibrahima DIANE , Soidridine Moussa MOINDZE , Bassirou DIENE, Ousmane DIALLO, Mamadou THIONGANE, Fodé CAMARA et Joseph NDONG pour les échanges intéressants que nous avons souvent eus, leur amitié et soutien constant.

Je ne saurais terminer ces remerciements sans dire quelques mots aux membres de ma famille. Je les remercie tous pour le rôle important qu'il ont joué durant toute ma carrière.

Je remercie feue ma mère Astou TAMBEDOU, partie durant cette thèse. Bien que gravement malade, tu me remontais le moral quand je t'appelai depuis Paris. Merci maman, tu as fait de moi, un vrai homme capable d'endurer toutes situations difficiles.

Je remercie aussi mon père Fallou NGOM, qui ne s'est jamais lassé de me soutenir. Merci papa pour l'amour inconditionnel que tu portes envers moi. Je te souhaite une très longue vie papa !

Je remercie aussi mes frères et sœurs pour leur présence constante à mes cotés et le soutien sans faille que vous m'avez apporté et d'avoir toujours pris soin de ma femme et de ma fille durant mes absences. Modou, Dame, Serigne Dia, Oumy et Saye, je vous aime !

J'exprime une immense gratitude à ma chère épouse Aida BA et à ma princesse Sokhna Faty pour leur patience, leur compréhension et leur soutien durant ces longues années de thèse. Vous m'avez facilité mes recherches en me permettant de s'absenter plusieurs fois, vous avez ignoré toutes les manques d'attention que la thèse m'a souvent poussé à faire, vous m'avez épaulé en me facilitant le travail de nuit et de jour que demandait la thèse. Bref, vous avez tout fait pour que je réussisse cette thèse.
Je remercie aussi mon oncle Dame TAMBEDOU et son épouse, ma tante Sokhna NDIAYE et ma belle mère Fatou Kiné MBAYE pour leur soutien et les prières qu'ils ont formulées pour moi.

Je remercie aussi la famille MAKPANGOU sise à Montigny Le Bretonneux pour la compréhension et l'accueil qu'elle m'accordait pendant les longues heures de travail à leur domicile avec mon encadrant.

Enfin, merci à vous qui lisez cette phrase pour l'intérêt que vous portez à ce travail.

À feue ma mère Astou TAMBEDOU.

Résumé

Cette thèse étudie la problématique de l'indexation et de la recherche dans les tables de hachage distribuées –*Distributed Hash Table (DHT)*. Elle propose un système de stockage distribué des résumés de documents en se basant sur leur contenu. Concrètement, la thèse utilise les Filtres de Bloom (FBs) [1] pour représenter les résumés de documents et propose une méthode efficace d'insertion et de récupération des documents représentés par des FBs dans un index distribué sur une *DHT*.

Le stockage basé sur contenu présente un double avantage, il permet de regrouper les documents similaires afin de les retrouver plus rapidement et en même temps, il permet de retrouver les documents en faisant des recherches par mots-clés en utilisant un FB. Cependant, la résolution d'une requête par mots-clés représentée par un filtre de Bloom constitue une opération complexe, il faut un mécanisme de localisation des filtres de Bloom de la descendance qui représentent des documents stockés dans la *DHT*.

Ainsi, la thèse propose dans un deuxième temps, deux index de filtres de Bloom distribués sur des *DHTs*. Le premier système d'index proposé combine les principes d'indexation basée sur contenu et de listes inversées et répond à la problématique liée à la grande quantité de données stockée au niveau des index basés sur contenu. En effet, avec l'utilisation des filtres de Bloom de grande longueur, notre solution permet de stocker les documents sur un plus grand nombre de serveurs et de les indexer en utilisant moins d'espace.

Ensuite, la thèse propose un deuxième système d'index qui supporte efficacement le traitement des requêtes de sur-ensembles (des requêtes par mots-clés) en utilisant un arbre de préfixes. Cette dernière solution exploite la distribution des données et propose une fonction de répartition paramétrable permettant d'indexer les documents avec un arbre binaire équilibré. De cette manière, les documents sont répartis efficacement sur les serveurs d'indexation. En outre, la thèse propose dans la troisième solution, une méthode efficace de localisation des documents contenant un ensemble de mots-clés donnés. Comparé aux solutions de même catégorie, cette dernière solution permet d'effectuer des recherches de sur-ensembles en un moindre coût et constitue est une base solide pour la recherche de sur-ensembles sur les systèmes d'index construits au-dessus des *DHTs*.

Enfin, la thèse propose le prototype d'un système pair-à-pair pour l'indexation de contenus et la recherche par mots-clés. Ce prototype, prêt à être déployé dans un environnement réel, est expérimenté dans l'environnement de simulation peersim qui a permis de mesurer les performances théoriques des algorithmes développés tout au long de la thèse.

Mots-clés
table de hachage distribué, indexation, recherche par mots-clés, filtres de Bloom, FreeCore.

1. Un filtre de Bloom est une structure de données probabiliste qui permet de réaliser des tests d'appartenance d'un élément à un ensemble.

Abstract

This thesis examines the problem of indexing and searching in *DHTs*. It provides a distributed system for storing document summaries based on their content. Concretely, the thesis uses Bloom filters (BF) to represent document summaries and proposes an efficient method for inserting and retrieving documents represented by BFs in an index distributed on a *DHT*.

Content-based storage has a dual advantage. It allows to group similar documents together and to find and retrieve them more quickly at the same by using Bloom filters for keywords searches. However, processing a keyword query represented by a Bloom filter is a difficult operation and requires a mechanism to locate the Bloom filters that represent documents stored in the *DHT*.

Thus, the thesis proposes in a second time, two Bloom filters indexes schemes distributed on *DHTs*. The first proposed index system combines the principles of content-based indexing and inverted lists and addresses the issue of the large amount of data stored by content-based indexes. Indeed, by using Bloom filters with long length, this solution allows to store documents on a large number of servers and to index them using less space.

Next, the thesis proposes a second index system that efficiently supports superset queries processing (keywords-queries) using a prefix tree. This solution exploits the distribution of the data and proposes a configurable distribution function that allow to index documents with a balanced binary tree. In this way, documents are distributed efficiently on indexing servers. In addition, the thesis proposes in the third solution, an efficient method for locating documents containing a set of keywords. Compared to solutions of the same category, the latter solution makes it possible to perform subset searches at a lower cost and can be considered as a solid foundation for supersets queries processing on over-dht index systems.

Finally, the thesis proposes a prototype of a peer-to-peer system for indexing content and searching by keywords. This prototype, ready to be deployed in a real environment, is experimented with peersim that allowed to measure the theoretical performances of the algorithms developed throughout the thesis.

Mots-clés
Distributed Hash Tables, indexing, keywords search, Bloom filters, FreeCore.

Table des matières

Table des figures

Liste des tableaux

Contexte et motivations

Dans les pays en voie de développement comme le Sénégal, l'accès à la production scientifique nationale reste un challenge. Répartis au sein d'une dizaine d'universités et de grandes écoles, les chercheurs sénégalais éprouvent d'énormes difficultés pour trouver les travaux nationaux connexes à leurs sujets de recherche. Les bibliothèques universitaires chargées de la numérisation et de la diffusion des résultats scientifiques ne disposent pas de suffisamment de ressources matérielles (bande passante réseau, capacité de traitement, capacité de stockage, connectivité), financières et humaines pour assurer l'accès efficace aux informations nécessaires aux chercheurs. Par conséquent, on voit souvent des étudiants en master travailler sur des sujets de stage ou de mémoire qui sont déjà traités dans d'autres universités.

Un catalogue national ouvert similaire à l'archive Hyper Articles en Ligne (HAL) accessible à tous et hautement disponible permettrait d'une part, de trouver facilement la bibliographie relative à des études locales et d'autre part, d'améliorer les travaux effectués dans d'autres unités de recherche. Cela contribue considérablement au développement de la recherche au niveau national.

Les chercheurs dans ces pays en développement ont aussi des difficultés pour accéder aux productions scientifiques internationales. Même si les structures universitaires investissent plus de moyens pour l'accès aux catalogues internationaux tels que ACM [2] et IEEE [3], les chercheurs peinent à télécharger rapidement les ressources qu'ils souhaitent. Cela peut les décourager et les inciter à abandonner le téléchargement. Or, il se trouve souvent que les articles recherchés ont été déjà téléchargés par d'autres.
Un cache partagé [4] des ressources scientifiques que les utilisateurs trouvent intéressantes ou qu'ils utilisent dans leurs travaux pourrait permettre aux universités de réduire les coûts dus aux téléchargements.

Enfin, l'insuffisance de moyens empêche certains chercheurs de participer aux meilleures conférences de leurs domaines. De même, l'organisation de séminaires avec des invités de marque a un coût non négligeable. De ce fait, il est très difficile pour un chercheur de suivre l'évolution de l'état de l'art dans un domaine donné. Les outils d'alerte tels que les flux RSS [5] ou Alerte-Google permettent aux chercheurs de recevoir des alertes concernant des thèmes

2. *Association for Computing Machinery (ACM), http ://dl.acm.org/*
3. *Institute of Electrical and Electronics Engineers (IEEE), https ://ieee.org*
4. respectant les réglementations en vigueur
5. http ://rssboard.org/rss-specification

de recherche mais il n'existe pas de support pour partager les notifications intéressantes. Un système facilitant la notification des travaux découverts (articles, projets, livres, etc.) pourrait aider les utilisateurs à suivre facilement l'état de l'art en étant notifiés des nouveaux documents téléchargés.

Afin de répondre aux attentes des utilisateurs, une première solution consisterait à utiliser les systèmes de notification de contenus pour diffuser des flux de données résumant les nouvelles ressources téléchargées ou enregistrées par les bibliothèques. Par exemple, chaque université du pays pourrait utiliser les systèmes de notification tels que les médias sociaux (Twetter[6] , facebook[7] , etc.) ou les systèmes de publication/abonnement (*Really Simple Syndication (RSS)*[8] , Atom[9]) pour notifier régulièrement les utilisateurs des nouveaux articles disponibles dans ses bibliothèques. Mais, vues les largeurs limitées de bande passante, l'utilisation des systèmes de notification présenterait trois problèmes :

1 D'abord, certaines universités peinent à satisfaire les demandes de téléchargement des nouveaux articles effectuées par un grand nombre d'utilisateurs. Cela s'aggrave si le taux de mise à jour devient important. Ce phénomène est en réalité commun aux systèmes de notification et peut conduire un fournisseur de contenus à utiliser des GigaOctets de bande passante pour satisfaire les demandes de téléchargement. Par exemple TechCrunch, le plus influent site américain sur l'actualité technologique, possède plus de 2 millions d'abonnés à ses flux et consomme une bande passante de 2400 Go quotidiennement ; en 2011, [Rao and Chen 11] soulignait que le site Boing Boing disposa de 11500 abonnés sur ses flux *RSS* et Atom et sa consommation en bande passante fut de 22 Go par jour.

2 Ensuite, les utilisateurs ne supportent pas le téléchargement de toutes les ressources partagées dont certaines ne sont pas pertinentes pour leurs domaines de recherche. Il faut un moyen de filtrer les ressources pour envoyer à chaque utilisateur les publications utiles. Mais compte tenu du grand nombre d'utilisateurs et de la diversité des attentes, un tel filtrage n'est pas réalisable au niveau de chaque bibliothèque.

3 Enfin, pour satisfaire un besoin d'informations au niveau national, les utilisateurs devraient souscrire aux flux de toutes les bibliothèques puis filtrer les informations reçues. Cela revient à parcourir les différents catalogues et à faire soi-même le tri. Ce qui serait un travail fastidieux pour les chercheurs.

Une autre solution consisterait à se reposer sur les moteurs de recherche du *World Wide Web (WEB)* tels que Google, Bing ou Yahoo pour faire indexer les productions scientifiques enregistrées par les bibliothèques universitaires et permettre aux utilisateurs de les retrouver à partir d'Internet. Cette solution est facile à mettre en oeuvre et ne nécessite pas beaucoup d'effort de la part des structures universitaires mais elle ne facilite pas le partage de documents à l'échelle nationale. Les politiques appliquées par les moteurs de recherche *WEB*

6. https ://twitter.com/company
7. https ://facebook.com/
8. http ://rssboard.org/rss-specification
9. http ://tools.ietf.org/html/rfc4287

pour retourner les meilleurs résultats d'une recherche ne privilégient pas les articles nationaux qui peuvent être noyés dans des milliards de résultats issus du monde entier.

La satisfaction des attentes de la communauté universitaire du Sénégal nécessite un système permettant aux bibliothèques ou à toute personne disposant des contenus de les rendre accessibles à tous les chercheurs de cette communauté. Ce système doit pouvoir indexer des millions de documents, servir des centaines de milliers d'utilisateurs répartis dans le pays et être hautement disponible.

Ainsi, la principale motivation de notre travail de thèse est de proposer un système de stockage, d'indexation et de notification permettant à une communauté d'utilisateurs comme celle des chercheurs sénégalais de partager efficacement des documents.

1 | Introduction

Au moment où l'expression "univers digital" se concrétise grâce au développement du web, une grande partie de la population mondiale réalise des milliers d'activités journalières sur Internet. Allant de la consultation des emails au visionnage de vidéos et de photos, beaucoup de nos activités se déroulent sur Internet. Parmi les tâches que nous faisons quotidiennement, figurent en bonne place le partage et la recherche d'informations.

Cette thèse propose FreeCore, un système de stockage et d'indexation de résumés de documents qui permet à des communautés d'utilisateurs comme la communauté universitaire du Sénégal de partager efficacement des documents.

Globalement, FreeCore (*c.f.* Figure 1.1) offre à ses utilisateurs les possibilités de publier les résumés des documents scientifiques en leur possession et de rechercher en utilisant les mots-clés des documents partagés par les membres de la communauté.

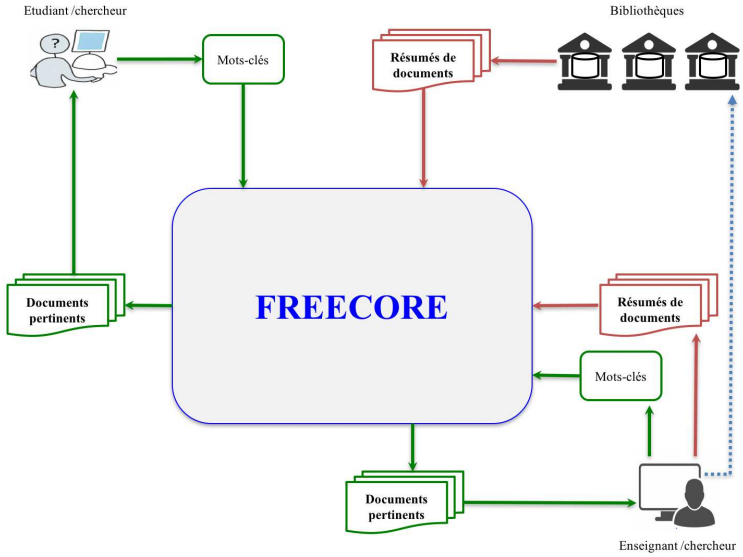


FIGURE 1.1 – Architecture globale de FreeCore

.

Pour assurer ses services, FreeCore adopte un principe simple. Il s'agit d'avoir une

collection de données sur lequel une donnée particulière peut être retrouvée à partir d'une liste de mots-clés. Les données de la collection peuvent être de diverses natures mais dans le cadre de cette thèse, nous considérons des données textuelles constituées de documents scientifiques.

Par ailleurs, FreeCore tient compte de la faiblesse des largeurs de bandes passantes octroyées à ses utilisateurs en considérant des résumés à la place des contenus entiers des documents. Ainsi, l'utilisation de FreeCore est facilitée pour les chercheurs qui vont transférer uniquement des résumés des documents scientifiques.

Par ailleurs, les documents et les utilisateurs sont souvent dispersés et se trouvent sur plusieurs sites distincts. La gestion des documents et des besoins des utilisateurs avec des serveurs centralisés n'est pas aisée et ne convient pas aux applications Web 2.0 qui exigent la haute disponibilité et une faible latence pour accéder aux données. La conception de FreeCore doit ainsi tenir compte de ces exigences et de l'insuffisance de ressources qui caractérise l'environnement des chercheurs sénégalais. Le choix d'un système pair-à-pair permet à FreeCore de mutualiser les ressources des utilisateurs afin d'offrir ses services à large échelle.

1.1 Choix d'une DHT comme infrastructure de stockage

Pour construire un système de stockage, d'indexation et de notification de documents qui passe à l'échelle, plusieurs infrastructures peuvent être utilisées. D'abord, nous avons le pair-à-pair qui est une infrastructure avec laquelle des utilisateurs peuvent partager spontanément des documents sans avoir besoin d'une coordination centrale.

Ensuite, il y a les grilles informatiques qui sont des infrastructures virtuelles constituées d'un ensemble d'ordinateurs (nœuds) potentiellement éloignés géographiquement mais fonctionnant en réseau pour fournir une puissance globale. Elles proposent l'agrégation des ressources informatiques (de calcul et/ou de stockage) réparties sur plusieurs sites pour la résolution d'un problème complexe – le plus souvent un problème scientifique ou technique qui nécessite un grand nombre de cycles de calcul ou l'accès à de grandes quantités de données [Jan 06].

Enfin, nous avons les nuages informatiques (*Cloud*) qui se présentent comme un modèle de grilles permettant à de multiples utilisateurs l'accès à des ressources matérielles et logicielles via Internet, de manière distribuée et sous forme de services.

Le point commun entre ces différentes infrastructures est la capacité de mobiliser beaucoup de nœuds généralement distants qui coopèrent pour la réalisation d'une tâche informatique comme le stockage d'un grand volume de données ou l'exécution d'un calcul complexe. Afin de faciliter la coopération des nœuds mobilisés, il s'avère utile de les organiser de sorte qu'ils soient facilement localisables. D'où la nécessité d'un réseau logique superposé – overlay en anglais – au réseau physique formé par les nœuds mobilisés.

Il existe plusieurs manières d'organiser et d'administrer les nœuds autour d'un réseau logique pour faciliter leur collaboration. Si le pair-à-pair et les grilles s'orientent vers une collaboration à large échelle pour l'organisation et la gestion des ressources informatiques, le *Cloud* prend une orientation plus contrôlée en choisissant une gestion centralisée au niveau d'un seul opérateur qui impose souvent des coûts financiers pour accéder aux nœuds

mobilisés. Ce qui peut rendre le *Cloud* inaccessible pour certains groupes d'utilisateurs. A l'opposé, les grilles informatiques et les réseaux P2P favorisent la collaboration à large échelle en mutualisant les ressources libres au niveau des nœuds mobilisés pour former un réseau logique complètement distribué. Avec ces infrastructures, les ressources sont organisées autour des réseaux logiques selon deux méthodes : soit les pairs assurent la gestion de leurs propres documents et pour localiser une ressource, il faut propager un message de recherche qui parcourt l'intégralité du réseau, c'est la méthode employée par les *overlays* non-structurés, soit la gestion des ressources est attribuée à un groupe de pairs précis en utilisant une table de hachage distribuée – *Distributed Hash Table (DHT)*–, il s'agit des *overlays* structurés.

Avec la première méthode, la localisation d'un document s'effectue en échangeant un nombre élevé de messages, ce qui participe à l'augmentation de la durée de recherche d'un document. Par contre les *overlays* structurés ont la particularité d'organiser le réseau en une topologie ayant des fonctionnalités de routage efficaces, permettant de localiser et de retrouver rapidement les documents.

Dans une communauté où les membres disposent de moyens financiers limités, une solution peu coûteuse pour réaliser un tel système capable de passer à l'échelle est la construction d'une infrastructure pour le stockage et le partage de contenus au-dessus d'une *DHT*.

1.2 Problématique

Nous considérons la réalisation d'un système de stockage de contenus reposant sur une table de hachage distribuée, destiné à une communauté d'utilisateurs telle que la communauté de chercheurs du Sénégal. Chaque membre publie les résumés et les URIs des documents en sa possession qu'il souhaite partager avec les autres membres. Pour retrouver des documents pertinents, un utilisateur soumet une requête constituée des mots-clés.

L'objectif de notre travail est la construction d'un système de stockage et d'indexation de documents sur une table de hachage distribuée permettant de retrouver efficacement, à partir d'une requête spécifiant des mots-clés, les documents publiés qui satisfont cette requête.

La problématique de stockage et de construction d'un index sur une table de hachage distribuée afin de permettre la recherche de contenus par mots-clés a été largement discutée par le passé.

Selon la stratégie utilisée pour indexer les documents, nous distinguons les trois grandes catégories de solutions qui sont les *over-DHT index* de mots-clés, les indexes ad'hoc de contenus et les indexes de contenus dans des *DHT* modifiées.

Dans les *over-DHT index* de mots-clés, les solutions proposées construisent des listes inversées de mots-clés et les répartissent sur un *over-DHT index*. Les listes inversées [Witten et al. 99] sont des paires (w, O), où w est un terme et O est l'ensemble des identifiants de document contenant ce terme. Avec ces listes inversées, on construit un index inversé où on associe chaque mot-clé à un serveur d'index chargé de stocker les documents contenant le mot. Pour retrouver les documents satisfaisant une requête de mots-clés, il suffit de faire l'intersection des listes associées aux termes de la requête.

Cette stratégie est confrontée aux trois problèmes rédhibitoires soulevés par [Tigelaar et al. 12] et [Liu et al. 04] que sont :

— le déséquilibrage de charges des nœuds d'indexation dû à l'absence d'uniformité dans la popularité des mots dans les documents ;
— la grande consommation de bande passante pour réaliser les intersections des listes en particulier lorsque la requête comprend un nombre élevé de mots clés ;
— le coût important d'indexation qui dépend du nombre de mots contenus dans les documents à indexer.

Ces problèmes découlent principalement de la distribution des mots dans les documents. La fréquence de certains mots augmente la charge de certains serveurs tandis que la rareté d'autres mots décharge d'autres serveurs. De plus, les longues listes de documents associées aux mots populaires doivent être fréquemment transférées dans le réseau pour répondre aux requêtes qui demandent souvent des mots célèbres.

Parmi les solutions qui utilisent cette stratégie, certaines tentent d'enrichir l'index inversé avec des résumés de contenus notamment avec des *Filtre de Blooms (FBs)* [Bloom 70] pour réduire le coût de l'intersection distribuée des résultats, c'est le cas des solutions proposées par [Reynolds and Vahdat 03], [Wang et al. 09], [Yang and Ho 06], [Chen et al. 12] et [Chen et al. 08]. D'autres solutions telles que celle de [Papapetrou et al. 10] et de [Ren et al. 10] adoptent des mécanismes de partitionnement (ou clustérisation) des serveurs d'index afin d'agréger partiellement les réponses au sein des groupes de serveurs.

Au niveau des indexes ad'hoc de contenus, il s'agit de construire une structure d'indexation capable de déterminer le serveur d'index en charge d'un document en se basant sur le contenu de ce dernier (c'est-à-dire de l'ensemble des termes que contient le document). Un exemple type de cette approche est proposé par [Joung et al. 05, Joung et al. 07] et [Szekeres et al. 11] qui utilisent des mots binaires de longueur r pour représenter le contenu d'un document. Dans ces solutions, la longueur r détermine le nombre de serveurs d'index utilisables et par conséquent, pour garantir le passage à l'échelle, cette longueur doit être très grande. Malheureusement, une grande valeur de r conduit à de mauvaises performances car la latence de la recherche est très élevée.

Enfin, dans la troisième catégorie de solution, il s'agit de modifier la *DHT* ou de construire une structure équivalente pour inclure les informations appropriées ou pour adapter les algorithmes internes. Par exemple, la solution Mkey [Jin et al. 06] crée une *DHT* dont les nœuds sont identifiés par des vecteurs de bits de longueur m, contenant au plus deux bits à 1. Mkey représente les documents par des filtres de Bloom de longueur m et construit une *DHT* où les identifiants sont des mots binaires de longueur m.

Les solutions de cette catégorie nécessitent d'une part une implémentation complexe : la gestion des clés *DHT* dans Mkey et la génération des identifiants dans la solution [Shi et al. 04] sont difficiles à mettre en œuvre. D'autre part, en modifiant les *DHTs*, il est difficile de conserver les propriétés de cette dernière et cela peut conduire à de mauvaises performances pour la recherche des documents.

Des stratégies décrites ci-dessus, nous retenons que d'une part, les indexes construits au-dessus des *DHT* ont l'avantage de garder les propriétés de passage à l'échelle, de routage efficace des messages et de haute disponibilité offertes par les *DHT*. D'autre part, les indexes ad'hoc et les indexes modifiant ou dépendant de la *DHT* utilisant des structures de représentation de contenus (filtres de Bloom, mots-binaires, résumés de listes inversées [Yang and Ho 06]) ont l'avantage de réduire les coûts d'indexation des documents et la bande passante des recherches par mots-clés.

1.3 Contributions de la thèse

Les principales contributions de cette thèse sont :

1.3.1 Représentation de contenus par des ensembles de mots binaires

Les listes inversées de termes sont des structures de base pour indexer des informations textuelles dans les *DHTs*. Cette approche est confrontée au problème de déséquilibrage de charge des serveurs d'index dû au fait que la popularité de termes suit la loi de Zipf.

La première contribution de cette thèse est la transformation du problème d'indexation de documents textuels en celui de l'indexation de filtres de Bloom caractérisés chacun par un ensemble de mots binaires. La représentation de chaque document par un *FB* garantissant un très faible taux d'erreur permet de réaliser des tests d'inclusion ensembliste de façon performante en exploitant uniquement des opérations bit à bit.

Si nous considérons des mots binaires de même longueur, le vocabulaire est fini et l'étude de la popularité des mots binaires montre que les fréquences d'apparition des mots binaires ne varient pas inversement en fonction du rang c'est-à-dire que la variation des fréquences des mots binaires ne suit pas la loi de Zipf. Ceci nous permet a priori de construire des indexes basés sur des listes inversées de mots binaires.

1.3.2 FragIndex

`FragIndex` est un schéma d'index de filtres de Bloom basé sur des listes inversées de mots binaires. Il exploite la représentation de contenus sous forme d'ensembles de mots binaires de même longueur. `FragIndex` s'inspire des indexes ad'hoc proposés par [Joung et al. 05, Joung et al. 07], mais implante des mécanismes qui permettent de répartir la charge d'indexation sur un plus grand nombre de serveurs d'index tout en assurant un coût de recherche comparable voire meilleur que celui offert par ces solutions.

Comparé aux systèmes d'index basés sur des listes inversés des termes textuels, `FragIndex` permet un meilleur équilibrage de charge ; toutefois le coût (en terme de trafic réseau, de bande passante et de latence) de son protocole de localisation de sur-ensembles est très élevé comparé à celui des systèmes à base de listes inversées de termes.

1.3.3 Prefix Matching Trie (PMT) et Summary Prefix Tree (SPT)

D'une part, pour localiser les sur-ensembles d'un mot binaire, `FragIndex` explore plusieurs entrées de la table de hachage dont le nombre et les adresses dépendent du mot binaire considéré. `FragIndex` accède à des entrées qui pourraient être vides. D'autre part, les arbres de préfixes sont des structures simples et efficaces utilisés par plusieurs systèmes d'index tels que *Prefix Hash Tree (PHT)* [Ramabhadran et al. 04], LIGHT [Tang et al. 10]. *PHT* et LIGHT sont construits au dessus d'une DHT et s'intéressent à la résolution des requêtes complexes de type intervalle. Les données considérées par ces solutions sont liées par une relation d'ordre total. Ainsi, les méthodes de recherche qu'ils proposent ne conviennent pas à la recherche de sur-ensembles car il n'y a pas une relation d'ordre total entre les ensembles.

Nous proposons *Prefix Matching Tree (PMT)*, un schéma d'indexation de filtres de Bloom basé sur le matching de préfixes. L'objectif de cette proposition est d'éliminer les accès aux entrées vides de la table distribuée. Les filtres de Bloom représentant les contenus sont stockés en utilisant un arbre de préfixes. *PMT* implante des heuristiques de répartition de données permettant de construire un arbre de préfixes équilibré. *PMT* propose un protocole de recherche de sur-ensembles qui exploite la relation d'ordre partiel qui existe entre les filtres de Bloom. Les évaluations montrent que *PMT* équilibre la charge des serveurs d'indexation en garantissant que plus de 80% des serveurs utilisent au moins 40% de leur capacité de stockage. De plus le coût de la recherche d'un document avec *PMT* est plus faible que la solution *PHT*.

En plus de la solution *PMT* qui s'inspire de *PHT*, nous avons aussi proposé *Summary Prefix Tree (SPT)* [Ngom and Makpangou 17] une autre version simplifiée s'inspirant de la solution [Tang et al. 10].
SPT étend l'utilisation des arbres de préfixe en construisant un index supportant des recherches par mots-clés. *SPT* approxime la recherche de sur-ensembles de mots-clés à la recherche de descendances de filtres de Bloom. Cette approximation permet de faire des recherches avec des ensembles compacts et d'utiliser des comparaisons bit-à-bit simples et efficaces. *SPT* définit aussi un mécanisme de recherche hybride qui convient spécifiquement à la nature creuse des filtres de Bloom utilisés comme clés d'indexation. Contrairement aux méthodes de recherche linéaires des arbres de préfixes qui testent tous les préfixes jusqu'à atteindre le nœud feuille recherché, la méthode hybride de *SPT* saute directement aux préfixes qui sont significatifs. Grâce à cette méthode, *SPT* dispose d'un algorithme réduisant le coût des recherches de sur-ensembles au strict minimum.

1.3.4 Un prototype expérimental pour la simulation de FreeCore

Enfin, la dernière contribution de ce manuscrit est la réalisation d'un prototype expérimental de FreeCore qui explique son fonctionnement, son implémentation distribuée et son évaluation sur l'environnement de simulation peersim.

1.4 Organisation du manuscrit

La suite du manuscrit comprend les chapitres suivants :
Le Chapitre 2 présente l'état de l'art de la recherche de contenus dans les systèmes distribués

et étudie particulièrement les systèmes d'indexation et de recherche construits avec des tables de hachage distribuées. Dans ce chapitre, nous allons passer en revue les mécanismes de recherche et les méthodes d'évaluation des requêtes avec les documents *Extensible Markup Language (XML)*, les techniques de résolutions de requêtes complexes et les systèmes d'indexation et de recherche par mots-clés dans les *DHT* qui constituent le cœur de notre travail.

Le Chapitre 3 décrit globalement la solution FreeCore et les principes de base qui sous-tendent nos contributions. D'abord, nous présentons notre système de stockage des documents sur une *DHT* utilisant les filtres de Bloom pour résumer les documents et leur contenu. Nous verrons ensuite les propriétés régissant les filtres de Bloom et définissons le principe de la recherche de sur-ensemble de filtres de Bloom. Enfin, le chapitre présente les challenges auxquels l'indexation et la recherche de filtre de Bloom sont confrontées.

Le Chapitre 4 présente le schéma d'indexation basé sur des listes inversées mots binaires – FragIndex. Nous donnons dans ce chapitre les algorithmes d'indexation et de recherches de sur-ensembles de filtres de Bloom puis nous évaluons leurs performances.

Le Chapitre 5 présente notre solution d'indexation basée sur un arbre de préfixe –*PMT* en spécifiant ses différents algorithmes et étudie ses performances.

Le Chapitre 6 est consacré au prototypage du système FreeCore et à son implémentation.

Enfin, la dernière partie du manuscrit comprend la conclusion générale qui résume les différentes démarches que nous avons adoptées et les contributions réalisées, les perspectives de recherche qui décrivent les futurs travaux et les axes de recherche que nous envisageons et l'annexe qui présente l'implémentation et la mise en œuvre dans l'environnement de simulation peersim.

2 | État de l'art

Dans ce chapitre, nous faisons une revue des travaux connexes à la problématique de la thèse. Il s'agit principalement d'étudier les systèmes d'indexation et de recherche dans les réseaux *pair-à-pair (P2P)* construits pour les documents structurés (tels que les documents *Extensible Markup Language (XML)*) et les documents non structurés (les contenus textuels libres).

2.1 Systèmes d'index et de recherche dans les documents structurés

Dans une communauté d'utilisateurs, il est essentiel de disposer d'outils efficaces permettant aux membres d'échanger des informations utiles. Sur ce, beaucoup de travaux ont été réalisés afin de faciliter le partage et la recherche d'informations structurées notamment dans des communautés d'utilisateurs répartis à large échelle.

Nous étudions dans cette section quelques systèmes de recherche et de partage d'informations dans les réseaux *P2P* structurés. Nous distinguons deux principaux groupes de systèmes : les systèmes de recherche d'informations sur des documents structurés (généralement sur les documents *XML*) et les systèmes de partage d'informations structurées dans les réseaux *P2P*.

2.1.1 Recherche dans les documents XML

XML est un métalangage informatique utilisé pour l'échange et l'intégration de données sur le web. Plusieurs données échangées sur le web sont représentées par des documents ou fragments de document *XML*. Par exemple, nous pouvons citer les flux *Really Simple Syndication (RSS)* et Atom qui sont utilisés pour la diffusion de résumés d'informations. Ainsi, la problématique de la recherche efficace de données *XML* dans les réseaux *P2P* a sollicité beaucoup d'intérêts dans la communauté scientifique.

Dans les systèmes présentés dans [Bonifati et al. 04] [Koloniari and Pitoura 04], les auteurs s'intéressent à la résolution des requêtes XPATH dans les réseaux P2P.

XP2P [Bonifati et al. 04] considère un document XML comme un arbre préfixé comprenant un ensemble de chemins qui mènent vers les éléments du document. Un élément et les fragments du chemin menant vers l'élément sont stockés sur un nœud de la DHT et une

simple recherche permet de les retrouver sans difficulté.
En utilisant des empreintes constituées de jetons de bit pour localiser les nœuds stockant les fragments de chemins et les éléments liés à une requête XPATH, XP2P propose deux types de recherche. Une recherche partielle qui retourne uniquement le fragment correspondant à la requête et une recherche complète qui retourne le fragment et les sous fragments correspondant à la requête.
Dans le cas d'une recherche partielle, la requête XPATH est modélisée sous forme d'un arbre et son empreinte est construite puis appliquée à la DHT définie. La recherche réussit s'il y a exactement un chemin qui correspond au chemin spécifié par la requête. Sinon la recherche échoue, et la requête est tronquée et reprise à une étape antérieure du chemin.
Pour une recherche complète, XP2P effectue une recherche locale sur le pair courant et poursuit suivant les directions prometteuses indiquées par les sous fragments dont les racines constituent les éléments recherchés.

Une autre solution proposée par [Koloniari and Pitoura 04] réduit la quantité de données échangée entre les nœuds d'un réseau P2P lors la résolution des requêtes XPATH.
À travers les filtres de Bloom spéciaux `Breadth Bloom Filter` et `Depth Bloom Filter` qui résument le contenu des documents et qui supportent des requêtes XPATH, chaque nœud maintient deux types de filtre de Bloom, un filtre local qui représente l'ensemble des documents XML stockés sur le nœud et un ou plusieurs filtres de Bloom fusionnés qui résument les documents stockés par les nœuds voisins.
Pour effectuer une recherche, on vérifie si les filtres de Bloom contiennent les éléments recherchés. Ces filtres de Bloom facilitent le routage entre les pairs du réseau et permettent de les regrouper en fonction de leur contenu. Les nœuds ayant un contenu similaire forment un arbre et toutes les racines des arbres sont connectées afin de faciliter la communication entre les groupes.

L'idée de base des solutions proposées par [Bonifati et al. 04] et [Koloniari and Pitoura 04] est d'élaborer des algorithmes distribués pour la résolution des requêtes XPATH. Mais la syntaxe particulière utilisée pour la formulation des requêtes est contraignante au niveau des utilisateurs qui, en plus, ont souvent besoin de rechercher dans les contenus textuels des balises XML.

Les solutions présentées par [Jamard et al. 07] et [Abiteboul et al. 08] vont plus loin en considérant les contenus textuels des documents XML.

Par exemple, le travail décrit dans [Jamard et al. 07] utilise des filtres de Bloom distribués – *Distributed Bloom Filter (DBF)* pour indexer les structures des documents et leur contenus textuels dans un réseau *P2P*. Ici, chaque document est représenté comme un ensemble contenant tous les chemins qui mènent vers chaque mot du document. Afin de prendre en charge tous les types de requêtes, la solution utilise trois filtres de Bloom qui résument les chemins, les balises *XML* et les mots.
Les Filtres de Bloom sont ensuite divisés en segments de tailles égales et ces segments sont identifiés par des numéros déterminés par une fonction de hachage. Une fois que le numéro de segment est déterminé, d'autres fonctions de hachage associent les clés des données aux segments, ainsi chaque segment peut être considéré comme un sous filtre de Bloom. S'il y a

un segment avec une forte probabilité de faux positifs, le segment est remplacé par un segment plus grand.
L'évaluation d'une requête est effectuée en deux étapes : une première étape qui vérifie le *DBF* et une deuxième étape qui vérifie le pair stockant le document source pour déterminer l'exactitude de la réponse.

On notera que les travaux [Jamard et al. 07, Abiteboul et al. 08] sont fortement dépendants de la structure des documents *XML*. De là, on s'interroge sur leur capacité à prendre en compte des documents ayant des formats différents surtout dans le cas où les documents sont représentés par des ensembles de mots-clés.
Pour que ces solutions soient efficaces, les filtres de Bloom doivent avoir une grande longueur pour résumer l'ensemble des mots dans [Jamard et al. 07] ou les listes inversées [Abiteboul et al. 08]. Cette situation mènerait à la dégradation des performances de ces solutions.

2.1.2 Filtrage et partage de documents structurés

L'importante quantité d'informations publiées sur le web nous amène à poser la question de savoir comment retrouver facilement les documents moins populaires avec les moteurs de recherche du web.

Parmi les solutions les plus adoptées pour partager des informations moins populaires, nous avons les systèmes publication/abonnement où les utilisateurs publient des informations et s'abonnent sur des producteurs d'informations en utilisant des requêtes de longue durée basées sur un thème donné ou sur une liste de mots-clés (souscription basée sur contenu).

Prenons le cas des flux *RSS* qui sont des documents au format *XML* loin d'être célèbres au niveau des moteurs de recherche du web. Pour faciliter leur partage, des travaux ont été mené dans le passé.

D'abord, nous avons les solutions proposées par [Sandler et al. 05], [Jun and Ahamad 06] et [Ngom et al. 10] qui proposent des systèmes *P2P* dans lesquels les fournisseurs des documents collaborent pour la dissémination des flux vers les consommateurs. Par exemple, dans FeedTree [Sandler et al. 05], les consommateurs récupèrent les notifications et les propagent à travers des arbres de diffusion.

La solution de [Zhou et al. 06] propose le moteur de recherche SoSpace dédié aux flux *RSS* fonctionnant sur un réseau *P2P* structuré. Tandis que [Rao and Chen 11] proposent un système *P2P* de dissémination de flux *RSS* qui permet aux utilisateurs de souscrire à des contenus à travers des mots-clés. Le système proposé est construit sur un *overlay* de pairs qui coopèrent pour effectuer le téléchargement, le filtrage, l'agrégation et la distribution des flux. Contrairement à SoSpace, le système proposé par [Rao and Chen 11] permet d'effectuer des notifications aux utilisateurs si une mise à jour est publiée. Par contre, elle ne distribue pas équitablement la charge du système sur les pairs du réseau. Le pair qui gère un terme célèbre devient un goulot d'étranglement, empêchant ainsi le passage à l'échelle de ce système.

Les solutions discutées dans cette section sont toutes conçues pour des documents structurés et comme nous l'avons souligné auparavant, il n'est pas évident qu'elles puissent s'adapter au contexte d'une recherche par mots-clés demandant plus de flexibilité que la recherche XPATH. Les sections suivantes discuteront des solutions plus adaptées au traitement des requêtes plus flexibles telles que les recherches par intervalle de valeurs et les recherches par mots-clés.

2.2 Systèmes d'index pour le traitement des requêtes complexes sur les DHT

Les systèmes distribués construits avec les tables de Hachage distribuées – *Distributed Hash Table (DHT)* supportent efficacement des requêtes exactes (c'est-à-dire qu'ils permettent de trouver les données dont les identifiants correspondent exactement à une clé). Pour de nombreuses applications, le traitement des requêtes exactes ne suffit pas, il faut un support efficace pour réaliser des requêtes complexes telles que les requêtes d'intervalles ou les requêtes de sur-ensembles.

Plusieurs solutions ont été proposées pour supporter les requêtes complexes en étendant les fonctionnalités des *DHT* [Ramabhadran et al. 04, Tang et al. 10, Hidalgo et al. 11, Caron et al. 06, Cortés et al. 16, Hidalgo et al. 16]. Le principe est simple et consiste à indexer les clés utilisées pour stocker les objets dans la *DHT* : c'est ce qu'on appelle l'indexation au dessus de la DHT ou *over-DHT indexing*.

Un index over-DHT est une structure ad'hoc construite au dessus d'une table de hachage distribuée et dont l'objectif est de traiter efficacement les requêtes complexes sans modifier la *DHT*. Parmi les structures les plus utilisées, nous avons les arbres de préfixes. Cette section étudie quelques solutions proposées pour traiter les requêtes complexes sur les DHT.

2.2.1 Prefix Hash Tree (PHT)

Prefix Hash Tree (PHT) [Ramabhadran et al. 04] est un arbre binaire de préfixes destiné à indexer un ensemble de données dans un réseau *P2P* structuré. Les nœuds de l'arbre binaire sont identifiés par un label (composé des symboles "/", "0" et "1"). Chaque nœud interne possède deux fils. Le fils gauche (resp droit) a pour label 0 (resp. 1). Tout nœud est désigné par la concatenation des labels des nœuds qui compose le chemin qui le relie à la racine (/).

La structure de PHT contient trois types de nœuds, les nœuds feuilles qui stockent les données, les nœuds internes qui participent au maintien de l'arbre et les nœuds externes qui n'appartiennent pas à l'arbre de préfixe mais qui sont présents dans la DHT.
L'ensemble des nœuds de PHT est réparti entre les pairs du réseau DHT en associant les préfixes des nœuds aux identifiants des pairs de la DHT. Ainsi, chaque nœud de l'arbre PHT correspond à un nœud dans la DHT et ses objets peuvent être localisés par une simple opération lookup sur la DHT.

La recherche dans PHT est effectuée par l'opération PHT-Lookup. Considérant une donnée représentée par une chaine binaire, PHT-Lookup retourne l'unique nœud feuille dont l'identifiant est un préfixe de la donnée, c'est-à-dire, le nœud qui stocke la donnée. Pour cela,

PHT propose deux méthodes de recherche.
D'abord il y a la recherche linéaire qui commence par effectuer une opération DHT-Lookup en utilisant le plus court préfixe de la donnée. Si un nœud interne est atteint, une nouvelle DHT-Lookup est réalisée en utilisant un préfixe plus long. Ce processus est répété jusqu'à ce qu'une feuille soit trouvée. Ensuite, il y a la recherche binaire qui divise la longueur du préfixe de la donnée en deux et commence par rechercher la moitié du préfixe. Si cette recherche retourne un nœud interne de PHT, le processus recommence avec un nouveau préfixe égale au milieu de la moitié supérieure. Si le préfixe correspond à un nœud externe, la moitié supérieure de l'intervalle est supprimée et la recherche recommence avec la moitié inférieure.

Pour effectuer une recherche d'intervalle, la plus petite borne de l'intervalle est utilisée pour trouver le premier nœud feuille. Une fois ce nœud atteint, PHT utilise une liste liant les nœuds feuilles pour retrouver toutes les données en parcourant la liste vers le nœud feuille qui est en charge de la borne supérieure de l'intervalle.

2.2.2 TPT-C

TPT-C [Hidalgo et al. 11] améliore les performances des recherches sur les arbres de préfixes en évitant de revisiter les nœuds internes des arbres. Pour cela, TPT-C propose de maintenir sur chaque nœud interne de l'arbre un cache qui stocke les labels des nœuds visités lors des recherches précédentes. Chaque nœud contient un nombre limité d'entrées dans son cache et chaque entrée est un préfixe de label d'un nœud interne.

Pour effectuer la recherche (binaire ou linéaire), TPT-C utilise le cache pour récupérer l'entrée correspondant au plus long préfixe commun avec la requête. Si un tel préfixe est trouvé, la recherche continue avec ce préfixe et réduit l'espace de recherche, sinon la recherche est effectuée normalement comme dans *PHT*. TPT-C optimise ainsi le coût de la recherche dans les arbres de préfixes en réduisant l'espace des préfixes candidats.

2.2.3 LIGHT

Comme *PHT*, LIGHT [Tang et al. 10] permet aussi de traiter les requêtes complexes à un moindre coût. Pour cela, LIGHT définit : un arbre de préfixe qui indexe les données, une structure de données qui stocke le résumé de l'arbre et une fonction de nommage qui associe la structure de données et la *DHT*.

LIGHT partitionne récursivement l'espace des données en deux sous-espaces de taille égales jusqu'à ce que chaque sous-espace contienne un nombre limité de données.
Tout comme *PHT*, LIGHT utilise un arbre de préfixes avec lequel, les données sont stockées dans les nœuds feuilles mais l'arbre de LIGHT dispose de deux racines dont une virtuelle ayant un seul fils qui est la racine normale.

Chaque branche de l'arbre est étiquetée par un nombre binaire, 0 pour les branches vers les fils gauches et 1 pour les fils droits. La branche spéciale entre la racine virtuelle et la racine normale est étiquetée par 0.
Ainsi chaque nœud possède un préfixe unique qui est la concaténation des nombres binaires composant son chemin depuis la racine virtuelle. Le préfixe de la racine virtuelle est le caractère spécial #.

À la différence de *PHT*, LIGHT ne maintient pas une liste liant les feuilles mais il propose à la place, une structure de données distribuée pour stocker les données et un résumé des informations de l'arbre de préfixes.

Pour rechercher une donnée, LIGHT-lookup retourne la clé DHT correspondante. Il s'agit de trouver le préfixe du nœud qui couvre la donnée sur lequel LIGHT applique sa fonction de nommage pour obtenir la clé DHT.
LIGHT propose un algorithme de recherche binaire similaire à celui de *PHT*. À chaque itération, LIGHT calcule le milieu de l'intervalle et effectue une opération DHT-get de la clé correspondant au préfixe obtenu. Si l'opération DHT-get échoue, le préfixe courant correspond à un nœud qui n'existe pas et la borne supérieure est réduite à la longueur du préfixe du nœud interne contenant le préfixe courant ; Si la DHT-get réussit, soit le préfixe retourné couvre la donnée dans ce cas la recherche retourne le préfixe du nœud interne contenant la clé-DHT courante, soit le préfixe retourné est un ancêtre de la feuille recherchée et LIGHT augmente la valeur de la borne inférieure.

Pour trouver un intervalle, LIGHT recherche une des bornes en utilisant la recherche binaire. Une fois que la feuille responsable de la borne est repérée, LIGHT transmet récursivement la requête aux nœuds de la vue local de la feuille. La transmission est effectuée jusqu'à ce que tous les nœuds internes couvrant l'intervalle soient visités. Grâce à la structure de donnée distribuée LIGHT parvient à maintenir des vues logiques au sein des nœuds feuilles et à explorer les contenus des sous arbres de la vue.

La flexibilité de la structure de LIGHT peut être exploitée pour optimiser la recherche binaire. En effet, le préfixe d'une feuille stocké sur un nœud interne ou sur une racine révèle la profondeur d'un des deux sous arbres issus de ce nœud : si le préfixe termine par 0, c'est la profondeur de la partie gauche du sous arbre qui est indiquée et si le préfixe termine par 1, c'est la profondeur du sous arbre droit qui est indiquée.
La solution de [Fu et al. 11] exploite le préfixe de la feuille trouvée pour déterminer le nœud fils le plus proche de la clé recherchée et utilise ce préfixe comme la borne supérieure de l'intervalle de la recherche binaire.

2.2.4 DST

DST [Zheng et al. 06] (*Distributed Segment Tree*) est un index over-DHT qui partitionne l'espace des données en segments d'intervalles fixes. L'index supporte les requêtes d'intervalles et les requêtes de couverture sans modifier la structure de la *DHT* sous-jacente. Les requêtes de couverture sont des requêtes d'intervalles spéciales qui récupèrent toutes les parties d'une donnée décomposée.

La structure de l'index de DST (*i.e.* un arbre de segments [de Berg et al. 97]) est un arbre totalement binaire construit au dessus d'une *DHT*. Les nœuds de l'arbre de segments sont associés aux pairs de la couche *DHT* en appliquant une fonction de hachage sur les bornes des intervalles. DST réplique les données de chaque niveau de l'arbre vers les nœuds ancêtres pour maintenir une vue globale sur l'ensemble des données de l'arbre.
Cette réplication rend les nœuds de niveaux supérieurs surchargés car la racine doit stocker tout l'intervalle indexé par l'arbre et devient un point de congestion. Pour éliminer ce

problème, une stratégie a été adoptée pour limiter la quantité de donnée qu'un nœud peut stocker. Quand un nœud atteint sa capacité, il arrête de stocker des données supplémentaires.

L'évaluation d'une requête d'intervalle est simple et efficace car les intervalles indexés sont partitionnés statiquement et sont connus à tous les niveaux de l'arbre. La recherche consiste alors à trouver le nombre minimal de nœuds couvrant complètement l'intervalle recherché. Pour ce faire, DST utilise un algorithme d'éclatement pour diviser l'intervalle recherché en des segments minimaux. Ensuite une opération DHT-lookup est effectuée pour localiser le nœud responsable de chaque segment. Enfin, pour obtenir les résultats de la requête, DST réalise l'union des clés retrouvées.

2.2.5 Limites des index conçu pour le traitement des requêtes complexes

Les solutions discutées dans cette section proposent des algorithmes efficaces pour la recherche d'intervalle. Bien qu'il existe des améliorations sur leurs performances notamment sur la tolérance aux pannes [Caron et al. 16], elles n'offrent pas la possibilité de réaliser des recherches de mots-clés. Notre solution *Prefix Matching Tree (PMT)* (*cf.* chapitre 5) s'inspire de ces travaux et propose un algorithme de recherche de sur-ensembles adapté pour un ensemble de données liées par une relation d'ordre partiel.

2.3 Systèmes d'index pour la recherche par mots-clés dans les DHT

Les *DHT* utilisent des valeurs numériques (clé de stockage) pour stocker et récupérer une donnée dans un réseau *P2P* structuré et ne tiennent pas compte des propriétés des données manipulées. Malheureusement, cela ne convient pas à la recherche de données avec des requêtes par mots-clés car dans une telle recherche, il ne s'agit pas de trouver des égalités parfaites entre les documents et les requêtes mais plutôt de trouver des documents qui contiennent une requête ou qui la décrivent partiellement. Afin de supporter la recherche par mots-clés, la stratégie la plus adoptée est la construction d'un index pour localiser efficacement les documents en fonction des mots-clés d'une requête.

2.3.1 Systèmes basés sur les listes inversées de mots-clés

En considérant les documents comme des points d'un espace multi-dimensionnel où chaque mot est une dimension, la recherche par mots-clés consiste à trouver l'ensemble des points contenant les mots de la requête [Singh and Singh 14]. Afin de supporter cette recherche sur une *DHT*, certaines solutions proposées considèrent chaque dimension comme une clé d'indexation permettant de regrouper les documents contenant un mot donné. Cette méthode est communément appelé indexation basée sur des listes inversées de mots. D'autres solutions proposent de ramener l'espace multi-dimensionnel en une seule dimension et d'utiliser une structure d'indexation.

La plupart des solutions de recherche par mots-clés utilisent des listes inversées comme structure de données [Witten et al. 99] et [Liu et al. 04]. Pour faciliter les recherches par mots-clés, l'approche naïve consiste à distribuer les listes de sorte que chaque terme soit attribué à un pair de la *DHT* qui stocke les documents le contenant. Les mots sont utilisés pour déterminer les identifiants des nœuds d'index chargés de stocker les listes des identifiants des documents contenant ces mots. Avec une simple opération DHT-get, la liste des documents contenant un mot-clé est récupérée.

La faiblesse de cette approche se trouve au niveau de la résolution des requêtes contenant plusieurs mots-clés (requête multi-termes). Dans ce cas, la recherche est coûteuse en consommation de bande passante car il faudra récupérer les listes associées à tous les termes de la requête. Ce problème s'aggrave si les mots sont célèbres c'est-à-dire s'ils sont associés à plusieurs documents. En effet, si une requête porte sur les termes A et B, l'intersection des résultats consiste à visiter le responsable du mot A ($pair_A$) et à transférer tous les documents contenant A au pair DHT responsable du mot B ($pair_B$) qui effectue l'intersection et retourne les réponses au client.

Afin de réduire la quantité d'information échangée et économiser la bande passante consommée par une recherche multi-termes, plusieurs solutions basées sur les filtres de Bloom ont été proposées [Tigelaar et al. 12].

D'abord, [Wang et al. 09] et [Reynolds and Vahdat 03] proposent d'utiliser un filtre de Bloom $F(A)$ qui représente l'ensemble des documents du $pair_A$ et d'envoyer $F(A)$ au $pair_B$. Celui-ci teste si chaque document de B appartient à $F(A)$ et retourne le résultat de $B \cap F(A)$ au $pair_A$. Enfin, ce dernier élimine les faux positifs en calculant $A \cap (B \cap F(A))$ qui est équivalent à $A \cap B$. Pour réduire la latence, le $pair_B$ peut envoyer directement l'intersection $B \cap F(A)$ au client plutôt que de l'envoyer au $pair_A$, mais cela augmente le nombre de réponses erronées.
En effet, avec la probabilité de faux positif des filtres de Bloom, de fausses réponses seront considérées comme des bonnes. Ainsi, la bande passante utilisée pour retourner tous les résultats au client croît linéairement avec la taille du corpus de documents.

Ensuite, [Reynolds and Vahdat 03] propose de tronquer les résultats pour rendre le coût constant, indépendamment du nombre de documents dans le réseau. Lorsqu'un client recherche un nombre fixe de résultats, les pairs $pair_A$ et $pair_B$ communiquent progressivement jusqu'à ce que ce nombre est atteint. Le $pair_A$ envoie son filtre de Bloom par morceaux et le $pair_B$ envoie les résultats par bloc jusqu'à ce que le $pair_A$ obtienne assez de résultats.

Enfin, contrairement à [Reynolds and Vahdat 03], les travaux de [Chen et al. 08, Chen et al. 12] montrent que le coût en bande passante d'une recherche multi-termes est déterminé par la popularité des mots dans le corpus de documents et que pour réduire la bande consommée, l'ordre d'intersection des résultats doit être prise en compte. En effet, en visitant les pairs responsables des mots célèbres, le nombre de documents récupéré est très grand. Il est donc préférable de commencer par les pairs responsables des

mots moins célèbres.

Ainsi, [Chen et al. 08] définit une mesure de similarité basée sur les *Filtre de Blooms (FBs)* pour déterminer le pair indexant le moins de documents. Mais cela nécessite une connaissance à priori des statistiques des mots indexés dans le système que la solution essaie de résoudre en effectuant la collecte des statistiques des mots avec un protocole basé un réseau non structuré.

Une autre approche développée pour réduire le coût d'une recherche multi-termes est l'utilisation de groupes de mots pour construire les listes inversées. Ce regroupement des mots permet de stocker sur un nœud uniquement les documents contenant les mêmes termes. Cette méthode est employée par [Skobeltsyn et al. 07] pour réduire la taille des listes de documents associées aux termes. Au lieu de construire des listes inversées pour les termes simplement, les mots des requêtes sont utilisés pour fabriquer des termes composés et associés aux documents qui les contiennent. On obtient ainsi un index de listes inversées représentant les contenus des documents recherchés.

Keyword Fusion [Liu et al. 04] est une autre solution construite sur la DHT Chord et utilise le même principe en maintenant un grand dictionnaire des mots célèbres. Si un mot devient célèbre, il est supprimé du pair et placé au niveau du dictionnaire de mots célèbres. Ensuite, de nouveaux mots formés par concaténation des mots supprimés du dictionnaire sont générés et stockés sur la DHT.
Une requête est routée au pair responsable du mot le moins célèbre et les résultats sont filtrés en utilisant les mots célèbres qui se trouvent dans la requête. La gestion du dictionnaire de mots célèbres est un frein pour cette solution dans un environnement dynamique. En effet le coût de maintenance est élevé (échange de messages pour synchroniser les mises-à-jour du dictionnaire). De plus, la création de nouveaux mots par concaténation équilibre la charge mais augmente la taille de l'index.

2.3.2 Linéarisation de l'espace multi-dimensionnel

À la différence des solutions précédentes, nous discutons dans cette partie des solutions qui transforment l'espace multi-dimensionnel des mots en un espace linéaire à une dimension.

pSearch [Tang et al. 03] utilise les modèles VSM [Salton et al. 75] et LSI [Deerwester et al. 90] pour représenter les documents et les requêtes comme des vecteurs dans lesquels chaque élément correspond au poids d'un mot. Pour indexer un document, pSearch considère son vecteur comme les coordonnées d'un point géométrique d'un espace cartésien et utilise la DHT CAN pour placer le document sur le pair responsable de la zone contenant le point. Pour la comparaison des requêtes aux documents, pSearch calcule leur similarité en utilisant le produit scalaire de leur représentation vectorielle.
Lors de l'évaluation d'une requête, sa représentation vectorielle est utilisée pour déterminer le pair responsable de la zone contenant le point. Ce dernier applique la fonction de similarité

pour déterminer les documents satisfaisants puis transmet la requête à ses voisins jusqu'à ce qu'un nombre de réponse attendu ou un seuil de pertinence soit atteint.

Suivant le même procédé, Squid [Schmidt and Parashar 04] utilise la technique *Hilbert Space Filling Curve (HSFC)* [Bially 67] pour transformer les documents et les requêtes en des points d'un espace uni-dimension qui peuvent être utilisés avec une *DHT*. *HSFC* traduit un point d'un espace à d-dimension à une courbe dans un espace d'une dimension de sorte que les points proches dans l'espace de d-dimensions soient associés à des valeurs adjacentes au niveau de l'espace unidimensionnel. Ainsi, *HSFC* préserve la localisation des documents lors du passage inter-dimensionnel.
Concrètement, à partir de d termes d'un document, Squid construit un point dans un espace de d-dimension, ensuite *HSFC* est utilisé pour générer un indice en une dimension correspondant au point, enfin, l'indice est utilisé pour stocker le document sur la *DHT*. Pour effectuer une recherche, il suffit de trouver le grappe (*cluster*) approprié en utilisant l'indice obtenu à partir des mots-clés recherchés et d'effectuer une recherche par inondation sur les nœuds du *cluster*.

La projection (*mapping*) des termes dans un espace multidimensionnel nécessite une connaissance *a priori* de l'ensemble des mots-clés possibles du système pour dimensionner l'espace de la *DHT*. Cela pourrait contraindre les solutions de [Tang et al. 03] et de [Schmidt and Parashar 04] à reconstruire entièrement les structures d'index lorsque les statistiques des mots-clés évoluent.

2.3.3 Systèmes d'index ad'hoc de contenu

Il existe d'autres solutions qui effectuent la linéarisation en représentant les contenus par des vecteurs de bits mais elles modifient les propriétés de la DHT ou créent une structure ad'hoc équivalente. Parmi ces solutions, nous avons la proposition de [Joung et al. 05] et celle de [Jin et al. 06].

2.3.3.1 Index basé sur un hypercube

[Joung et al. 05] propose un système distribué de localisation et de routage d'objet – *Distributed Object Location and Routing (DOLR)* modélisée sous forme d'un graphe direct $G = (V, E)$ et associe chaque nœud à un identifiant unique de $a - bit$.
Le graphe G est construit au dessus du réseau physique et permettant d'accéder aux documents stockés sur le réseau. Les références des documents sont enregistrés sur le *DOLR* à partir d'une fonction de hachage qui place chaque document identifié par son ensemble de mots sur un nœud représenté sous forme de chaine binaire de $r - bit$.

Un index est construit sur un hypercube de dimension r et permet d'associer un document à l'ensemble des mot-clés qu'il contient. Cet hypercube possède 2^r nœuds représentés chacun par une chaine binaire unique de longueur $r - bit$.
En désignant par W l'ensemble des termes du vocabulaire, une fonction de hachage $h : W \longrightarrow \{0, 1, ..., r - 1\}$ fait correspondre chaque sous-ensemble de termes à un nœud unique dans l'hypercube. Un nœud de l'hypercube est responsable d'un ensemble de mots E

si les bits 1 au niveau de son $r - bit$ sont fournies par le hachage des mots de E.
Un document d contenant un ensemble de mots E_d est alors indexé sur le nœud de l'hypercube responsable de l'ensemble E_d. Ce nœud stocke une entrée de la forme $< E_d, d >$ pour matérialiser ce document dans son index local.

Deux types de recherche peuvent être effectuées : la recherche exacte (*pin search*) et la recherche de sur-ensemble (*superSet search*).
La recherche exacte permet de retrouver l'ensemble des entrées correspondant à un ensemble de mots-clés E. Il s'agit de localiser le nœud de l'hypercube identifié par le mot-binaire construit à partir de E. Ensuite le document contenant l'ensemble de mots-clés est recherché sur la table d'index du nœud et enfin la fonction $read$ du *DOLR* est appelé pour récupérer le document.
La recherche de sur-ensemble est réalisée à partir d'un arbre couvrant construit à partir du sous-hypercube issu du nœud correspondant à l'ensemble des mots-clés recherché. À partir de cet arbre couvrant, tous les nœuds responsables du sur-ensemble de mots-clés contenant l'ensemble recherché peuvent être retrouvés.
En désignant par O_E l'ensemble des documents contenant E, la recherche des sur-ensembles de E et d'un seuil t consiste à retourner $min(t, |O_E|)$ documents qui contiennent E. Tout nœud du sous-hypercube peut potentiellement indexer des documents contenant E. La solution propose de parcourir l'arbre "en profondeur d'abord" pour retrouver tous les sur-ensemble recherchés. Tant que le seuil n'est pas atteint, la recherche continue en suivant le sens de parcours de l'arbre. Et à chaque étape une vérification est effectuée sur les nœuds susceptibles de répondre à la requête.

Pour améliorer les performance de la recherche de sur-ensembles dans la solution de cette solution, [Szekeres et al. 11] définit un algorithme d'indexation et de recherche basé sur un arbre couvrant permettant de construire tous les chemins partant de l'ensemble des mots d'un document à tous les sous-ensembles de mots qu'il contient. Ainsi, un document pourra être récupérer à partir d'un sous-ensemble minimal contenant les mots-clés d'une requête.

Si un nœud reçoit un document à indexer et constate qu'il est le seul nœud responsable de l'ensemble de termes du document, il envoie son identifiant à ses parents responsables des sous-ensembles inférieurs qui, à leur tour, envoient leur identifiants à leur parents. S'il constate qu'il n'est pas le seul nœud responsable d'un sous-ensemble de mots (c'est-à-dire qu'il a un fils), il garde l'identifiant de son fils et transmet les sous-ensembles inférieurs à ses parents en vérifiant que les sous-ensembles n'ont pas été transmis par un autre nœud. La procédure d'indexation continue jusqu'à obtenir le sous-ensemble minimal qui ne peut plus être décomposé.

Pour la recherche, le nœud recevant une requête de mots-clés répond en envoyant son identifiant et l'ensemble de mots-clés demandé, s'il constate qu'il est le sous ensemble minimal couvrant l'ensemble des mots-clés recherché, il transmet la requête à tous ses fils. Tout nœud qui reçoit la requête procède de la même manière jusqu'à ce que le sur-ensemble maximal soit atteint. La réponse à la requête de mots-clés est la fusion des résultats issus de chaque nœud visité.

L'inconvénient de cette approche est le nombre élevé de nœuds explorés lors de la recherche *superset*. Plusieurs nœuds peuvent être visités inutilement, ceci augmente le nombre de messages pour une recherche notamment avec les requête contenant peu de mots-clés. Le nombre de nœuds visités varie exponentiellement avec le nombre de bit 0 se trouvant dans les représentations des requêtes de mots-clés : une requête de n bits 0 doit visiter 2^n serveurs d'index.

Afin de limiter le nombre de nœuds visités, la solution préconise de prendre des valeurs de r très petites (environ r = 10), ce qui limite le nombre de serveurs d'indexation à 2^r et engendre une grande similarité des représentations des documents lorsque ces derniers contiennent plus de r mots. Du coup, le déséquilibrage de charge devient énorme car tous les documents de plus de r mots auront les mêmes représentations et seront indexés par les mêmes nœuds.

[Szekeres et al. 11] réduit le coût des recherches de sur-ensembles mais les algorithmes qu'il propose sont fortement dépendants de la nature des mots qui sont supposés ordonnés de manière lexicographique. Le problème des mots célèbres connu avec les index inversés reste non résolu car chaque mot est un sous-ensemble minimal et le nœud responsable va subir une forte charge d'indexation. De plus, les sur-ensembles des mots célèbres peuvent aussi devenir plus célèbres que le sont les mots, cela aggrave le déséquilibrage de charge et augmente le coût des recherches de mots-clés.

2.3.3.2 Mkey

Mkey est une autre approche proposée par Jin et al.[Jin et al. 06]. La structure de Mkey est hybride et composée d'un réseau structuré construit avec la *DHT* Chord qui regroupe un ensemble de pairs dénommés *hubs*. Chaque *hub* est le point d'entrée d'un *cluster* de nœuds regroupés dans un réseau non structuré.

Mkey utilise les *FBs* comme structure de donnée pour construire un index distribué au sein des *hubs* c'est-à-dire sur la *DHT* Chord. Pour ce faire, chaque nœud *hub* est identifié par un vecteur de bit qui contient seulement un ou deux bit de valeur 1 et chaque document est associé à un *FB* qui représente les mots-clés qu'il contient.

Pour stocker un document, son filtre de Bloom est stocké sur plusieurs pairs (hub) de la *DHT*. Les clés DHT utilisée pour stocker un *FB* sont obtenues en construisant séquentiellement des vecteurs binaire ayant deux bit à 1.
Plus précisément, le *FB* du document est parcouru séquentiellement de gauche à droite et chaque couple de bits est pris individuellement pour construire une clé de la *DHT*.
Par exemple, si le filtre de Bloom 01101110 représente un document, le *FB* sera stocké sur les *hubs* identifiés par 01100000, 00001100 et sur 00000010.

Pour effectuer une recherche, la requête est représentée par un filtre de Bloom et plusieurs clés DHT sont générées à partir du *FB* obtenu. Pour obtenir les clés DHT à partir du *FB* de la

requête, seuls les trois premiers bits à 1 sont considérés (en commençant par la droite). Les clés DHT des *hubs* à visiter sont obtenues en construisant des vecteurs binaires ayant deux bits à 1 choisis comme suit : si l, m et r désignent les 3 premiers bits à 1 dans la requête, les clés DHT sont obtenues en combinant le deuxième bit m avec chaque bit compris entre l et r en excluant le bit m.

Par exemple, la requête 01101010 sera envoyée aux *hubs* identifiés par les vecteurs binaires suivants : 01100000, 00110000 et 00101000. Une fois la requête est routée sur les différents *hubs*, elle continue dans chaque *cluster* et la recherche s'effectue par inondation.

Pour que cette solution fonctionne, il faut impérativement un mécanisme de gestion des clés de la *DHT*. Pour cela, un nœud de rendez-vous attribue les clés *DHT* aux *hubs* de niveau 1 c'est-à-dire les clés de la *FB* contenant seulement un bit à 1. Ensuite un nœud de classe 1 attribue les clés *DHT* de classe 2 issu de lui c'est-à-dire ayant le même bit 1 que lui. Il est évident que l'existence d'un nœud rendez-vous constitue un facteur de congestion et peut empêcher le passage à l'échelle de cette solution.

Par ailleurs, le coût de la recherche est proportionnel au nombre de mots et à la taille des filtres de Bloom. En effet, le nombre de mots et la taille des *FB* influent sur la répartitions des bits à 1 dans les *FBs* des requêtes. Enfin, toutes les requêtes sont réparties sur un nombre limité de *hub*, ce qui rend cette solution incapable de passer à l'échelle.

2.3.3.3 Multi-Level Partitioning –MLP

Le partitionnement multi-niveaux ou *Multi-Level Partitioning (MLP)* [Shi et al. 04] est une stratégie utilisant SkipNet [Harvey et al. 03] pour construire un index formé de nœuds groupés de manière hiérarchique. L'ensemble des nœuds est logiquement divisé en groupes. Puis chaque groupe de nœuds est à son tour divisé en sous-groupes, et ainsi de suite jusqu'à obtenir le niveau de hiérarchisation souhaité.

Dans SkipNet, un nœud possède un identifiant lexicographique (LexID) et un identifiant numérique (NumID) qui peut être généré par une fonction de hachage comme dans les *DHT*. Pour trouver les LexID, les auteurs modélisent Internet comme un espace géométrique et calculent les coordonnées des nœuds dans un espace cartésien. Puis dans une seconde étape, le LexID de chaque nœud est généré par un mapping de ses coordonnées sur un espace circulaire d'identifiants.

La hiérarchisation des groupes de nœuds est construit en divisant le LexID chiffre par chiffre, ainsi tous les nœuds d'un même groupe ont un préfixe de LexID commun.

Pour indexer les documents, la liste inversée de base est répartie au sein des groupes. Ainsi chaque groupe maintient un index inversé local des documents appartenant à ce groupe.
Ensuite, les documents de chaque groupe sont partitionnés en fonction des mots-clés qu'ils contiennent, puis distribués au sein des nœuds du groupe. Chaque nœud dans le groupe est donc responsable de la liste inversée de certains mots-clés.

Afin d'effectuer une recherche par mots-clés, la requête est envoyée à tous les groupes de niveau 1 qui la diffusent successivement aux prochains niveaux jusqu'à ce que le niveau de hiérarchisation soit atteint. Au niveau de chaque groupe, les nœuds ayant la liste inversée des mots-clés sont chargés de répondre à la requête. Les listes inversés sont rassemblées pour produire le résultat du groupe. Ensuite, les résultats de tous les groupes sont combinés niveau par niveau et renvoyés au nœud initiateur.

La solution *MLP* nécessite d'une part un processus de génération complexe des identifiants rendant sa mise en œuvre fastidieuse. D'autre part, elle peut conduire à de mauvaises performances pour la recherche par mots-clés dans la mesure où les propriétés de passage à l'échelle et de tolérance aux pannes offertes par les *DHT* ne sont pas garanties.

2.4 Systèmes de filtrage des documents publiés en fonction des requêtes

Pour terminer notre revue de l'état de l'art, jetons un coup d'œil sur les mécanismes de filtrage des informations. Étant donné les faibles débits de téléchargement des utilisateurs de FreeCore, il y a un besoin vital de disposer des outils efficaces pour le filtrage des informations partagées permettant aux utilisateurs de recevoir les informations personnalisées et pertinentes.

Parmi les solutions que nous avons étudiées, certaines ont choisi d'indexer les requêtes et de filtrer les contenus pertinent en fonction des requêtes indexées et d'autres prennent le sens inverse en indexant les documents et en définissant des moyens d'interrogation tenant compte des requêtes et des documents partagés.

Dans le premier cas, les requêtes formulées à travers des mots-clés seront comparées à la volée aux contenus des documents entrant en provenance de différentes sources [Hmedeh et al. 12]. Cette approche s'intéresse uniquement à la diffusion des flux de données et n'intègre pas des mécanismes de recherche instantanée c'est-à-dire des recherches faites avec des mots-clés.

L'autre approche utilisée est l'indexation des documents et la définition d'algorithmes de filtrage tenant compte des mots-clés dans les requêtes.
Par exemple, dans le contexte des environnements mobiles, [Yu and Shoon 11] propose un mécanisme de filtrage des documents XML qui représente les documents et les requêtes par des filtres de Bloom.
Les documents *XML* sont modélisés comme des arbres ordonnés étiquetés dans lesquels les nœud constituent les éléments et les branches sont les relations existant entre ces éléments (parent-fils). Une requête est formulée avec XPATH et peut être découpée en un ensemble d'éléments et de relations.
La solution est composée d'un moteur de filtrage et d'un module de pré-traitement des documents *XML* et des requêtes XPATH. Le module de pré-traitement analyse les documents

XML publiés et les requêtes puis les associe à des filtre de Bloom. Le moteur de filtrage est chargé de stocker les souscriptions faites sur le système et d'effectuer les comparaison de ces requêtes aux documents publiés. Dans cette solution, le problème de la comparaison des documents et des requêtes est résolue facilement car il suffit de réaliser des comparaisons bit-à-bit entre les filtres de Bloom.

Conceptuellement, nous suivons le même approche que la solution [Yu and Shoon 11] en représentant les documents et les requêtes par des filtres de Bloom. Nos principales différences avec cette solution est d'une part, nous construisons un système décentralisé basé sur une *DHT* alors que cette solution est destinée à un fonctionnement client-serveur dans lequel la gestion de son index de filtre de Bloom ne demande pas d'effort particulier.
D'autre part, notre solution vise à indexer des résumés de document et non des éléments des balises de documents XML dont le nombre est tellement limité par rapport au nombre de mots contenus dans un résumé.

2.5 Conclusion

Dans ce chapitre, nous avons effectué un état de l'art sur les travaux connexes à notre sujet en mettant particulièrement l'accent sur les méthodes d'indexation de contenus stockés dans une *DHT* et les systèmes d'indexation et de recherche par mots-clés. Le chapitre suivant va aborder les principes de base de nos contributions dans ces deux thèmes de recherche.

3 | La solution FreeCore

FreeCore est un système d'indexation de documents partagés par une communauté d'utilisateurs. Il propose deux services essentiels qui sont la publication de contenus et leurs résumés et la recherche de contenus à partir d'une liste de mots.

Ainsi, nous trouvons au cœur de FreeCore, un système de stockage et de recherche par mots-clés permettant de stocker les résumés publiés par des utilisateurs et de filtrer les résumés qui contiennent les ensembles de mots recherchés par d'autres utilisateurs .

Comme tout système de recherche, la réalisation de FreeCore est divisée en deux parties, le stockage des documents et l'indexation de leurs contenus [Barroso et al. 03, Ahmed and Boutaba 11]. Dans un environnement distribué, cela revient à définir un système réparti pour le stockage des documents et à maintenir un index de contenu distribué.

3.1 Stockage orienté contenu

Dans le contexte d'application de FreeCore, nous considérons que les utilisateurs publient les URIs de documents (articles, mémoires, ...) qu'ils mettent à disposition ainsi que leurs résumés. La construction du corpus est donc faite de manière progressive par les utilisateurs.

À la réception d'une publication de document, FreeCore procède à une phase de pré-traitement avant de le stocker. Dans cette phase, FreeCore associe le document à une paire d'informations formée d'un identifiant et d'une description.

3.1.1 Pré-traitement des contenus publiés

Pour obtenir l'identifiant d'un document, FreeCore utilise l'URI fournie par l'utilisateur lors de la publication.

Quant à la description du document, FreeCore construit une description à partir du résumé publié. Il existe plusieurs techniques permettant à FreeCore de transformer un contenu publié en une description. Parmi ces techniques, nous avons celles qui utilisent des groupes de mots ou des phrases comme unités de représentation [Salton 89, Schütze et al. 95, Mitra et al. 97] et celles qui prennent individuellement les mots pour représenter les contenus publiés par des sacs-de-mots [Salton et al. 75, Hofmann 99, Sharma and Jain 15].

Dans le cas de FreeCore dont le but est principalement de filtrer les documents suivant un ensemble de mots, la représentation par des sacs-de-mot est plus adaptée. De plus, elle est plus simple à mettre en œuvre et constitue la solution la mieux adoptée dans la littérature.

Ainsi pour stocker les documents, FreeCore résume leurs contenus par des sacs-de-mot (liste de mots-clés ou termes). Pour optimiser les ressources de stockage, FreeCore choisit des descriptions avec un nombre limité de mots. Cela s'effectue principalement en trois étapes.
D'abord, le résumé publié sous forme de texte intégral (composé de mots, ponctuations et formatage) est nettoyé en supprimant les éléments non significatifs tels que les articles, les mots de liaisons et les ponctuations.
Ensuite, les termes restants sont traités et ramenés à des termes de base. Cette phase s'appelle radicalisation (ou lemmatisation) et permet d'unifier les différentes formes des mots d'une même famille afin de les représenter par un seul terme (par exemple **ami** pour **amis** et **amies** ; **étudi** pour **étudier**, **étudiant**, **étudie**, ...).
Enfin, FreeCore choisit parmi l'ensemble des termes lemmatisés, un sous ensemble limité composé des mots les plus importants. L'importance d'un mot peut être trouvée avec des statistiques telles que sa fréquence dans le document (**tf**) et sa fréquence inverse de documents (**idf**). Elle peut être aussi déterminée à partir d'un mécanisme de pondération basé sur les requêtes des utilisateurs qui attribue à chaque terme un poids.

Au final, FreeCore dispose pour chaque résumé publié, une représentation sous forme d'un couple formé d'une URI et d'une description. Dès lors, le stockage des documents consistera à répartir les URIs et les descriptions correspondantes dans un ensemble de serveurs répartis. À cet effet, FreeCore utilise une table de hachage distribuée – *Distributed Hash Table (DHT)* et associe chaque document à une clé permettant de stocker sa description et son URI sur un pair de la *DHT*.

Pour cela, FreeCore construit une clé de stockage du document à partir de sa description en utilisant un filtre de Bloom qui représente la description par une structure compacte, puis utilise ce filtre de Bloom pour trouver le serveur *DHT* chargé du stockage du document.

3.1.2 Construction de clés de stockage basées sur contenu

FreeCore représente la description d'un document par un filtre de Bloom [Bloom 70] qui consiste en une structure de données compacte permettant de tester de manière simple si un élément appartient à un ensemble. C'est un tableau de m bits, tous initialisés à 0, et h fonctions de hachage. Chaque fonction de hachage fait correspondre un élément en une unique position du tableau. Ainsi, un élément est codé par h positions dans le tableau de m bits.

Pour insérer un élément dans le filtre [1], on le hache avec les h fonctions différentes. Chaque valeur retournée par une fonction de hachage correspond à une unique position dans le tableau et le bit à cette position est mis à 1.

Pour tester la présence d'un élément dans le filtre de Bloom, on calcule d'abord les valeurs retournées par les mêmes h fonctions de hachage utilisées pour insérer les éléments, ensuite on vérifie dans le tableau les bits qui se trouvent aux positions correspondant aux valeurs trouvées.

1. le mot filtre renvoie à filtre de Bloom

Si tous les bits sont à 1, on estime que l'élément est présent dans l'ensemble représenté. Si au moins un des bits est à 0, on est certain que l'élément ne se trouve pas dans l'ensemble représenté (voir Figure 3.1).

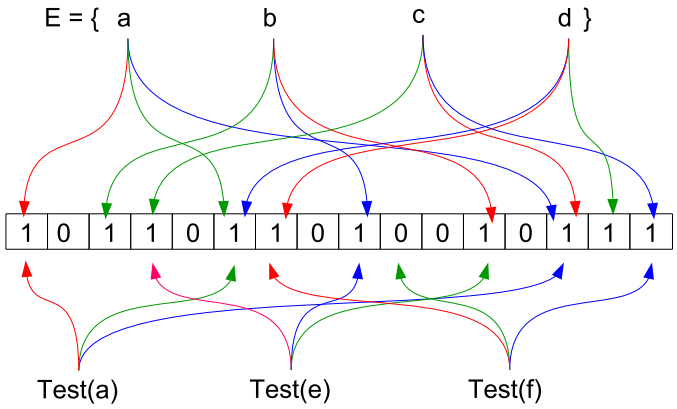


FIGURE 3.1 – Exemple de filtre de Bloom contenant les éléments de l'ensemble $E = \{a, b, c, d\}$ construit avec trois fonctions de hachage. Le test d'appartenance de l'élément e retourne Vrai alors que e n'appartient pas au filtre : c'est un **faux positif**

.

On caractérise un filtre de Bloom par le taux de faux positif (*false positif rate* (fp_r)) qui exprime la probabilité qu'un élément soit déclaré présent, à tord, dans l'ensemble représenté. Cette probabilité évolue avec le nombre d'éléments n insérés dans le filtre de Bloom et dépend du ratio $r = \frac{m}{n}$ [Fan et al. 00]. Pour garder l'efficacité d'un filtre de Bloom, sa longueur m doit croître proportionnellement en fonction du nombre d'éléments ajoutés.

FreeCore résume ainsi la description d'un document d par un filtre de Bloom FB_d de longueur m dans lequel tous les termes de la description sont insérés. Un document est donc transformé en une paire (clé, valeur) dans laquelle la clé correspond au filtre de Bloom FB_d et la valeur est constituée de l'identifiant et du résumé du document, ainsi nous avons $d = (FB_d, desc_d)$ où $desc_d = (uri_d, résumé_d)$.

3.1.3 Répartition des documents sur une DHT

Après avoir représenté un document par un couple $(FB, desc)$, FreeCore considère le filtre de Bloom comme une clé servant à stocker la donnée $desc$ représentant un identifiant et un résumé dans une table de hachage distribuée. Une fonction de hachage permet à FreeCore d'associer le filtre de Bloom FB à l'identifiant d'un serveur de stockage de la *DHT*.

De cette manière, tous les documents ayant le même filtre de Bloom correspondent à la même entrée de la *DHT* et ils sont stockés sur un même serveur. L'algorithme 3.1 montre le processus d'insertion d'un document dans l'index distribué. Il utilise principalement l'opération **put** offerte par la table de hachage distribuée pour enregistrer le document sur le serveur responsable de la clé obtenue avec la fonction de hachage.

Algorithme 3.1 insertion d'un document $(FB, desc)$ dans l'index distribué

Input: FB , $desc$
1: DHT-Key k $\longleftarrow$ $hash(FB)$
2: DHT-put(k, $desc$)

On note que la connaissance du filtre de Bloom d'un document permet de récupérer son identifiant et son résumé au niveau de la table de hachage distribuée, il suffit d'appliquer la fonction de hachage sur le filtre de Bloom et d'utiliser la clé obtenue avec l'opération **get** de la *DHT*. Ainsi, étant donné un ensemble de termes, FreeCore permet de retrouver en un seul accès à la *DHT* tous les documents stockés ayant exactement la même description.

Pour retrouver les documents dont les descriptions constituent des sur-ensembles, FreeCore exploite la nature des filtres de Bloom utilisés comme clés de stockage.

3.2 Support de la recherche par mots-clés

Un des objectifs de FreeCore est de fournir aux utilisateurs la possibilité de rechercher des documents stockés. Pour cela, les utilisateurs soumettent des requêtes sous forme d'un texte libre comme lors de la publication des documents.

Comme pour toute publication, pour traiter une requête, il est nécessaire de nettoyer le texte afin de supprimer les ponctuations et les mots non significatifs (articles et mots de liaisons) et de ramener les termes de recherche sous forme lématisée : on obtient ainsi une suite de mots caractérisant les documents recherchés.

Les documents ayant des descriptions contenant cette liste de mots seront considérés comme satisfaisants pour cette requête. Ainsi, soit F la liste de mots clés de la requête, cela revient à déterminer tout document ayant une description E qui est un sur-ensemble de F, c'est-à-dire $F \cap E = F$.

FreeCore se base sur l'intersection des filtres de Bloom représentant les ensembles E et F pour approximer le test d'inclusion.

3.2.1 De la recherche par mots-clés à la recherche de descendance de FB

Considérons deux ensembles de termes E et F résumés respectivement par les filtres de Bloom FB_E et FB_F. Si l'ensemble E contient l'ensemble F alors le filtre de Bloom FB_E contient le filtre de Bloom FB_F, c'est-à-dire si l'égalité $FB_{E \cap F} = FB_F$ est vérifiée. Ainsi, deux filtres de Bloom de même longueur sont liés par une relation d'ordre partiel (inclusion de filtres de Bloom) que nous formalisons par la définition suivante :

Définition 3.1. Soient FB_E et FB_F, deux filtres de Bloom de longueur m.
Nous disons que FB_E *contient* FB_F (ou que FB_F *mène* à FB_E), noté $FB_F \rightsquigarrow FB_E$, si et seulement si $\forall i, 0 \leq i < m, FB_F[i] = 1 \Rightarrow FB_E[i] = 1$; c'est-à-dire l'ensemble des bits à 1 dans FB_F est inclus dans l'ensemble des bits à 1 de FB_E.

Définition 3.2. Soit FB un filtre de Bloom.
Nous appelons descendance de FB (notée $\mathcal{D}(FB)$), l'ensemble des filtres de Bloom qui contiennent FB, i.e $f \in \mathcal{D}(FB) \Leftrightarrow f \wedge FB = FB$.

Exemple 3.1. Soit F_1 = 10011010, F_2 = 11011010 et F_3 = 10001010 trois filtres de Bloom de même longueur. Nous avons les relation suivantes : $F_1 \in \mathcal{D}(F_2)$, $F_3 \in \mathcal{D}(F_2)$ et $F_3 \in \mathcal{D}(F_1)$.

Propriété 3.1. Soient E et F deux ensembles de termes résumés respectivement par FB_E et FB_F deux filtres de Bloom de même longueur construits avec les mêmes fonctions de hachage. Si E contient F alors ($FB_E \wedge FB_F = FB_F$)

De la propriété ci-dessus, nous déduisons qu'étant donné un ensemble de termes F résumés par un filtre de Bloom FB_F, tous les filtres de Bloom des sur-ensembles de F construits avec les mêmes fonctions de hachage et de même longueur que FB_F appartiennent à la descendance de FB_F.

Toutefois, la propriété 4.1 est une condition nécessaire mais pas suffisante. En d'autres termes, le filtre de Bloom résumant les termes d'un ensemble S peut appartenir à la descendance de FB_F alors que S ne contient pas F
.

De manière simple, une recherche par mots-clés consiste à filtrer les documents stockés suivant une liste de termes recherchés. Les documents ayant des descriptions contenant cette liste seront considérés comme satisfaisants pour la recherche. Cela revient à déterminer tout document ayant une description E qui est un sur-ensemble d'un ensemble de termes recherchés F c'est-à-dire tout document dont l'intersection avec la requête retourne la requête : $F \cap E = F$.

FreeCore se base donc sur l'intersection des filtres de Bloom [Guo et al. 06, Mitzenmacher 01] et exploite le fait que si F est inclus dans E, $FB_F = FB_E \wedge FB_F$

Ainsi, si le résultat de l'intersection est égal au filtre de la requête, FreeCore conclut alors que le document contient la requête.

Cette conclusion est en fait une approximation. Pour réduire les erreurs d'approximation, FreeCore doit construire des filtres de Bloom qui garantissent un taux d'erreur très faible pour les approximations de tests d'inclusion des ensembles représentés.

3.2.2 Construction des filtres de Bloom efficaces pour les tests d'inclusion

L'utilisation des filtres de Bloom résumant des ensembles pour approximer les tests d'inclusion est sujette à des erreurs d'approximation qui ont un impact sur la précision des réponses obtenues. Ces erreurs conduisent à une augmentation du bruit dans les réponses à cause des ensembles considérés à tord comme de bons résultats.

La précision des réponses obtenues dépend de la probabilité de faux positif des filtres de Bloom résumant les documents et les requêtes. Plus cette probabilité est élevée, plus la probabilité de déclarer à tort un document comme un sur-ensemble de la requête est élevée.

3.2.2.1 Impact du ratio taille/nombre d'éléments

Les performances d'un filtre de Bloom dépendent de la probabilité de faux positif déterminée par le ratio $r = \frac{m}{n}$ où m est la taille du filtre (c'est-à-dire le nombre de bits qui lui est alloué) et n est le nombre d'éléments insérés dans ce filtre.

Puisque nous nous basons sur les filtres de Bloom pour dire qu'un document contient ou non une requête, nous pouvons nous tromper si les filtres de Bloom manipulés ont des probabilités de faux positif élevées. Pour éviter cette situation et rendre les filtres de Bloom efficaces pour l'intersection d'ensemble, nous devons les configurer avec une grande longueur m.

La Figure 3.2 montre la variation de la précision moyenne de 40 filtres de Bloom représentant des requêtes de mots-clés sur un corpus de 100000 filtres de Bloom représentant des documents ayant 60 termes au maximum (la valeur de n est fixée à 60).

Sur la Figure, nous voyons que la précision d'une recherche devient acceptable à partir de $r = 10$, c'est-à-dire quand la longueur m des filtres de Bloom est 10 fois supérieure au nombre maximal d'éléments n pouvant être présents dans les filtres. Nous voyons aussi que la probabilité de faux positif maximale doit être inférieure ou égale à 0,017 pour que l'intersection des filtres de Bloom puisse fournir des mesures de similarité précises entre les documents et les requêtes. Ainsi, nous voyons qu'une des conditions pour rendre efficace les filtres de Bloom est d'augmenter leur longueur m c'est-à-dire construire des filtres de Bloom avec un ratio r supérieur ou égal à 10 qui garantissent un taux de faux positif inférieur à 1,7%.

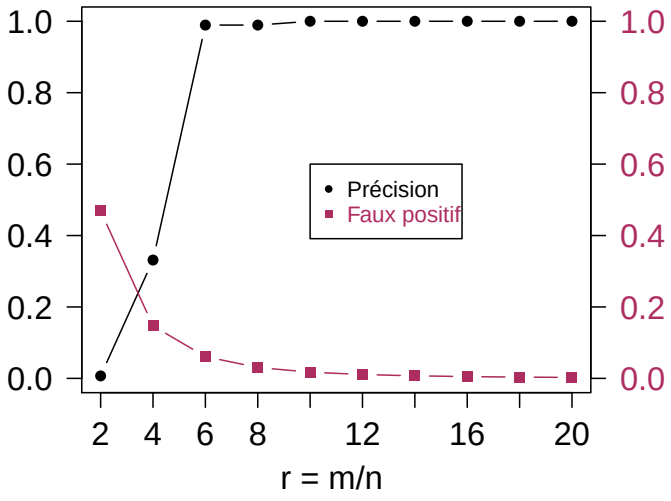


FIGURE 3.2 – Prob faux positif en fonction du r pour n = 60 termes

3.2.2.2 Paramétrage des filtres de Bloom

Les performances d'un filtre de Bloom dépendent de la probabilité de faux positif déterminée par le ration $\frac{m}{n}$ où m est la taille du filtre de Bloom (c'est-à-dire le nombre de bits qui lui est alloué) et n est le nombre d'éléments insérés dans le filtre de Bloom. Puisque nous nous basons sur les filtres de Bloom pour dire qu'un document contient une requête ou non, nous pouvons nous tromper si les filtres de Bloom manipulés ont des probabilités de faux positif élevées. Pour éviter cette situation et rendre les filtres de Bloom efficaces pour l'intersection d'ensemble, nous devons les configurer avec une grande longueur m.

Ainsi, nous paramétrons FreeCore de sorte que la probabilité de faux positif des filtres de Bloom reste très faible en représentant les documents et les requêtes par des filtres de Bloom ayant un ratio $\frac{m}{n}$ très élevé, c'est-à-dire que la longueur m est largement supérieure au nombre d'éléments à insérés dans chaque filtre de Bloom.

Pour cela, nous faisons l'hypothèse que le nombre n d'éléments qu'on peut insérer dans un filtre de Bloom est connu et fixé dans le système. En pratique, nous pouvons fixer n à une valeur permettant de coder le nombre moyen de termes attendu dans les documents à représenter.

Enfin, nous considérons qu'il existe des fonctions de hachage efficaces permettant de représenter des ensembles par des filtres de Bloom. Ces fonctions de hachage doivent être assez discriminantes, à défaut d'être parfaites, de sorte que deux éléments différents soient associés à des bits 1 différents qui sont uniformément répartis dans les filtres de Bloom.

Dans la suite du manuscrit, nous travaillerons principalement avec des filtres de Bloom paramétrés en fonction du jeu de donnée contenant les résumés étendus des pages en anglais de Wikipédia[2]. Nous fixons n à 64 et r à 16 afin de coder efficacement la majeure partie des résumés (*c.f.* Tableau 3.2). Et enfin, pour simplifier, nous utilisons la puissance de 2 supérieure ou égale à la valeur de m nécessaire pour coder la taille moyenne des résumés.

Tailles des résumés	
Min	1
1st Qu.	16
Median	28
Mean	41,24
3rd Qu.	52
Max	8915
Nb Total de résumés	4.636.000

Tableau 3.1 – Caractéristiques du jeu de données Wikipédia

Paramètres	Valeurs
n	64
r	16
h	5
m	1024
fp_r (attendu)	0.001

Tableau 3.2 – Paramétrage des filtres de Bloom : $fpr = (1 - e^{\frac{-hn}{m}})^h$

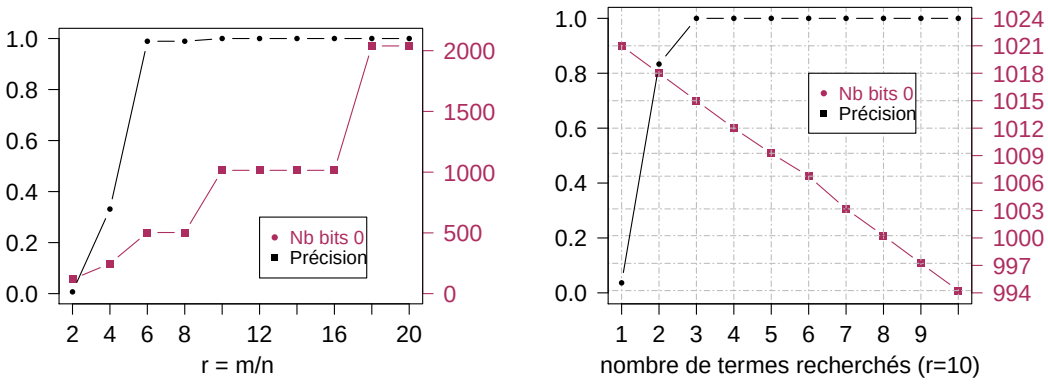
2. $http://data.dws.informatik.uni-mannheim.de/dbpedia/2014/en/long\_abstracts\_en.ttl.bz2$

3.3 Indexation des filtres de Bloom pour la recherche efficace de sur-ensembles

À partir d'une requête de mots-clés représentée par un filtre de Bloom, la recherche s'effectue en retrouvant tous les filtres appartenant à la descendance du filtre de Bloom qui résument des documents stockés sur la *DHT*.

La difficulté réside dans le fait que la connaissance du filtre de Bloom de la requête permet seulement de déterminer l'ensemble de ses descendants mais ne donne pas la possibilité de savoir, *a priori*, si ces filtres de Bloom représentent ou non des documents stockés dans la *DHT*.

Par ailleurs, afin de rendre possible la comparaison des filtres de Bloom des documents et des requêtes, les filtres de Bloom doivent avoir la même longueur. Le paramétrage des filtres en fonction du nombre de mots des documents conduit à la construction de filtres de Bloom creux pour les requêtes car ces dernières peuvent contenir des ensembles de mots plus petits [Efthimiadis 08, Silverstein et al. 99].
En conséquence, les quelques mots d'une requête sont résumés par un filtre de Bloom ayant peu de bits 1 (et donc beaucoup de bits 0). Par exemple, on voit sur la Figure 3.3a qu'il faut plus de 1015 bits à 0 dans les filtres des requêtes pour avoir des précisions acceptables (*i.e.* supérieure à 40%). Aussi, une grande valeur du ratio r permet d'augmenter le nombre de bits 0 dans les filtres de Bloom des requêtes et d'atteindre des précisions élevées. La Figure 3.3b montre qu'avec un ratio égal à 10, il faut au minimum trois mots-clés pour avoir une précision de 100%.



(a) Nb bit 0 /précision vs r

(b) Nb bit 0 /précision vs Nb mots-clés

FIGURE 3.3 – Variation du nombre de bit 0 dans les requêtes (n = 60 termes)

Avec une requête représentée par un filtre de Bloom ayant z bits 0, on aura 2^z descendants

parmi lesquels on ne pourra pas *a priori* distinguer ceux qui sont utilisés pour stocker des documents.

Afin d'effectuer une recherche, la méthode naïve est de prendre en compte tous les filtres descendants de la requête qui représentent potentiellement des documents stockés, et d'accéder à tous les pairs *DHT* responsables des ces filtres de Bloom pour récupérer les documents. Ce qui peut devenir coûteux en latence réseau. Parmi les filtres de Bloom de la descendance, certains peuvent ne pas être liés à aucun document, les entrées correspondantes à ces filtres mèneront donc vers des serveurs qui ne les stockent pas et par conséquent, l'accès à ces serveurs est inutile.

Considérons par exemple la requête représentée par le filtre de Bloom $q = 00100010\,10010010\,00000100\,00100010$ dans la *DHT* de la Figure 3.4 schématisant un système *pair-à-pair (P2P)* sur lequel on a réparti un corpus de 10 documents générés arbitrairement afin d'illustrer nos propos.

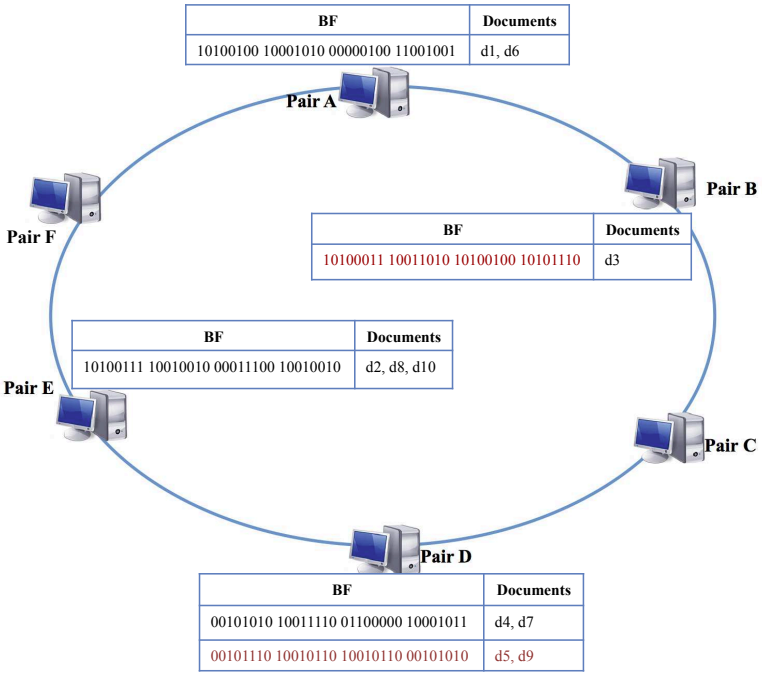


FIGURE 3.4 – Exemple de stockage distribué des documents dans une DHT

Puisque nous ne connaissons pas les identifiants des documents recherchés et que nous ne disposons que du filtre de Bloom de la requête et de sa descendance, la méthode naïve serait d'explorer tous les pairs que nous pourrons accéder pour retrouver des documents satisfaisants la requête. Dans l'exemple, il y a 24 bits 0 dans la requête q et la descendance de q contient 2^{24} soit 16.777.216 filtres de Bloom correspondant potentiellement à des entrées de la *DHT*. Or, la *DHT* contient seulement 5 filtres de Bloom parmi lesquels, seuls 2 filtres de Bloom appartiennent à la descendance de la requête q et sont utilisées pour stocker les

documents d_3, d_5, d_9.

L'exploration naïve va donc envoyer 2^{24} messages *DHT-GET* pour tenter de récupérer les documents stockés avec les filtres de Bloom appartenant à $\mathcal{D}(q)$ alors que deux messages seulement permettent de retrouver toutes les réponses.
Cela conduit évidemment à de mauvaises performances pour un système distribué à large échelle dans la mesure où un énorme trafic doit être généré pour traiter des milliers de requêtes.

Puisque le nombre de descendant dépend du nombre de bits 0, une solution est de réduire la longueur des filtres de Bloom (et donc des bits 0) afin de diminuer le nombre d'éléments de la descendance. Mais cela conduit à l'augmentation de la probabilité de faux positif, ce qui va impacter négativement sur la précision des réponses et donc sur les performances de la recherche.

Afin de rendre efficace la recherche, notre solution est de considérer seulement les filtres de Bloom correspondant à des entrées de la *DHT*. Pour cela, nous proposons un index de filtres de Bloom permettant d'effectuer des *DHT-GET* uniquement avec les filtres de Bloom de la descendance utilisés lors du stockage des documents.

Un tel index permettra de retrouver les filtres de Bloom de la descendance qui correspondent à des clés de stockage. Tout comme l'index à la fin d'un livre qui permet de repérer les pages contenant un mot, l'index de filtre de Bloom servira à identifier les clés de stockage contenant des documents satisfaisants pour une requête. Dans les prochains chapitres, nous proposons deux solutions permettant de résoudre efficacement les recherches par mots-clés. Au paravent, nous décrivons le fonctionnement du système FreeCore puis nous présentons son architecture construit sur un réseau pair-à-pair utilisant une *DHT*.

3.4 Architecture fonctionnelle de FreeCore

FreeCore est un système de stockage de contenus reposant sur une table de hachage distribuée. Il comprend un système de stockage distribué de type clé-valeur reposant sur une *DHT*, un système d'indexation de filtres de Bloom exploitant une *DHT* et une API qui permet d'accéder à ces deux composants. La Figure 3.5 montre l'architecture fonctionnelle de FreeCore et permet d'identifier ses trois composants : *Query Handler*, *DataStore* et *index* [Makpangou et al. 14].
Chaque utilisateur de FreeCore est associé à un pair de la DHT qui lui offre une API lui permettant de publier des documents ou d'initier des recherches par mots clés. La méthode *publier()* (resp. *rechercher()*) crée une requête de type *add* (resp. *superSetSearch*) comprenant notamment le filtre de Bloom correspondant au résumé du document publié (resp. de la requête), puis invoque le module d'indexation pour insérer le filtre (resp. rechercher les sur-ensembles du filtre) dans l'index, et ce, quel que soit le schéma d'indexation implanté par le module d'indexation.

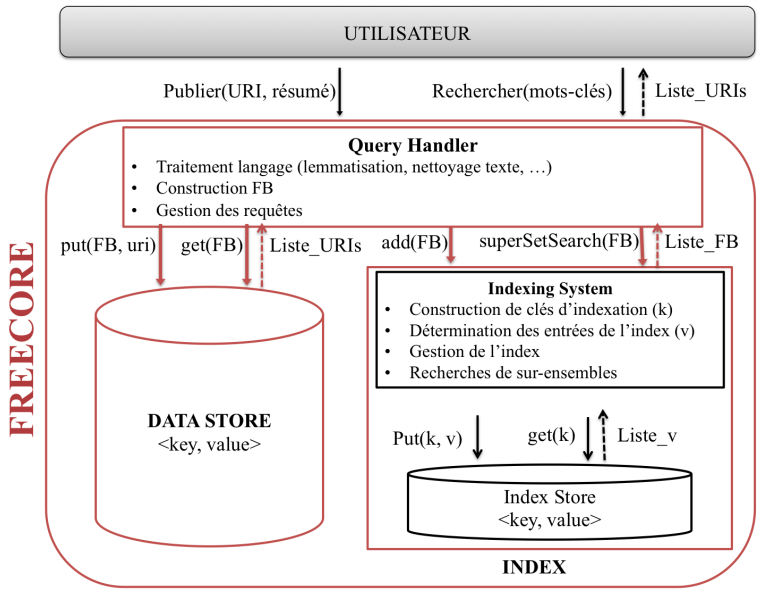


FIGURE 3.5 – Architecture fonctionnelle de FreeCore

Sur l'architecture fonctionnelle, nous avons d'une part les utilisateurs de FreeCore qui émettent deux types de requêtes. Nous avons les demandes de publication de document qui permettent à un utilisateurs d'envoyer à FreeCore l'URI et le résumé du document qu'il souhaite partager. Nous avons aussi les recherches de documents que les utilisateurs effectuent en envoyant à FreeCore la liste de mots-clés qui se trouvent dans les documents qu'ils recherchent.

D'autre part, nous avons la couche FreeCore avec ses trois composants :

3.4.1 Query Handler

Ce composant joue deux rôles. D'abord, il dispose d'une interface lui permettant de recevoir les tâches des utilisateurs de FreeCore. Ensuite, Il dispose d'un ensemble de routines lui permettant d'exploiter les autres composant de FreeCore. À travers ce composant, les résumés des documents publiés sont nettoyés et mis sous forme d'une liste de mots-clés. Ensuite, les mots-clés provenant des résumés publiés ou envoyés par un utilisateur sont représentés par des filtres de Bloom. La construction de filtres de Bloom s'effectue toujours de la même manière c'est-à-dire avec les mêmes paramètres. Ce qui donne à FreeCore un statut modulaire qui permet de l'adapter en fonction de la taille des résumés. Enfin ce composant s'interface directement avec le système de stockage et l'index qui lui permettent de stocker et de rechercher les URIs des documents en utilisant les filtres de Bloom.

3.4.2 DataStore

C'est un système de stockage distribué de type clé-valeur qui repose sur une *DHT*. Les URIs et les résumés des documents publiés sont enregistrés sur une table distribuée en utilisant les

filtres de Bloom comme clé de stockage. L'insertion et la récupération d'une URI s'effectue en utilisant les primitives *put* et *get* de la *DHT*.

3.4.3 Index

C'est le troisième composant de FreeCore. Il comprend deux modules : un système de stockage distribué (indexStore) reposant sur une *DHT* et une logique d'indexation qui définit la manière d'indexer les filtres de Bloom.

3.4.3.1 Indexing System

Outre l'interfaçage avec Query Handler, ce module effectue principalement des traitements sur les filtres de Bloom et définit le format des données à maintenir pour faciliter la recherche des sur-ensembles d'un filtre de Bloom. Ce module comprend aussi un ensemble de routines permettant de gérer l'index et de coordonner les recherches de sur-ensembles. Selon le schéma d'index utilisé, ce module peut disposer de fonctionnalités supplémentaires.

3.4.3.2 IndexStore

L'indexStore est un système de stockage distribué de type clé-valeur qui repose sur une *DHT* qui permet de stocker l'index. Les données maintenues par l'indexStore dépendent du schéma d'index utilisé mais sont toujours fournies sous forme de couples (clé, valeur). L'insertion d'un couple ou la récupération des valeurs associées à une clé s'effectuent en utilisant les primitives *put* et *get* de la *DHT*.

3.5 Architecture distribuée de FreeCore

FreeCore est une application pair-à-pair construite au dessus d'une *DHT* qui lui offre un service de routage basé sur des clés – *Key Based Routing (KBR)*. La Figure 3.6 schématise l'architecture distribué de FreeCore. Nous pouvons y distinguer des pairs comprenant chacun deux couches : la couche FreeCore et la couche KBR.

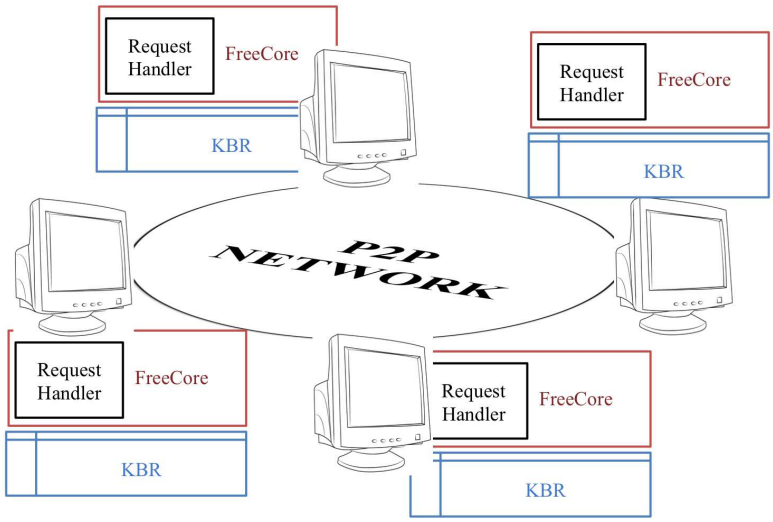


FIGURE 3.6 – Architecture distribuée de FreeCore

Chaque pair dispose de l'application FreeCore qui lui offre les services de publication de contenu et de recherche par mots-clés. Au sein de cette application nous avons le module QueryHandler qui gère les demandes de service effectuées par le pair.

La deuxième couche (KBR) est un service de routage basé sur clés qui permet au pair d'acheminer des messages à ses voisins contenus dans une table de routage.

3.6 Jeux de données

Pour évaluer nos propositions, nous considérons un jeu de données réel composé de 4.636.000 résumés de pages *Wikipédia* dont les caractéristiques sont présentées dans le tableau 3.1. Dans chaque résumé, nous avons supprimé les mots de liaisons et les articles et nous avons lemmatisé les mots en les ramenant à leur radical.
Pour les résumés ayant plus de 100 mots, nous sélectionnons pour chacun d'eux les 100 termes les plus importants. Pour cela, nous utilisons la méthode de pondération *TF-IDF* pour déterminer l'importance d'un terme. Cette méthode nous permet de calculer l'importance d'un terme contenu dans un résumé, relativement au jeu de données et consiste à déterminer le produit de la fréquence du terme et de la fréquence inverse de document.
La fréquence du terme (*tf*) est le nombre d'occurrences de ce terme dans le résumé considéré, tandis que la fréquence inverse de document (*idf*) est le logarithme (en base 10) de l'inverse de la proportion de documents du jeu de données qui contiennent le terme.

Ensuite nous avons associé à chaque résumé l'URL permettant de le récupérer à partir de la collection *DbPedia*. Nous obtenons au final un corpus de 4.090.885 résumés associés chacun à un URL unique. Dans la suite nous utiliserons le terme document pour désigner le résumé et l'identifiant qui lui est associé.

Enfin, nous avons découpé le corpus en 6 jeux de données dont les caractéristiques sont résumés dans la tableau 3.3 ci-dessous.

Nom	Nombre de mots des résumés	Taille (nombre de documents)
wiki1	1 <= nb mots < 10	481380
wiki2	10 <= nb mots < 20	997581
wiki3	20 <= nb mots < 40	1297593
wiki4	40 <= nb mots < 60	618164
wiki5	60 <= nb mots < 80	336373
wiki6	80 <= nb mots <= 100	359794

Tableau 3.3 – Découpage du jeu de données

3.7 Conclusion

Dans ce chapitre, nous avons présenté les principes de la recherche par mots-clés avec des filtres de Bloom représentant des résumés de documents. Nous avons décrit une méthode efficace d'insertion et de récupération des documents représentés par des filtres de Bloom dans un index distribué sur une *DHT*. La résolution d'une recherche par mots-clés représentée par un filtre de Bloom constitue cependant une opération difficile : il faut un mécanisme de localisation des filtres de Bloom de la descendance qui représentent des documents stockés dans la *DHT*. Pour cela, nous allons indexer les filtres de Bloom des documents stockés dans la *DHT*.

4 FragIndex : Un substrat d'indexation de filtres de Bloom basé sur des listes inversées de mots binaires

Dans ce chapitre, nous présentons `FragIndex`, un schéma d'index de filtres de Bloom qui associe à chaque filtre un ensemble de mots-binaires et qui fait correspondre à chaque mot-binaire la liste de filtres de Bloom qui le contiennent.

`FragIndex` repose sur les trois piliers suivants que nous détaillons au fil du chapitre :

- la notion de filtre de Bloom fragmenté qui permet de caractériser chaque filtre de Bloom par un ensemble de mots-binaires,
- la notion de liste inversée de mots-binaires qui permet de construire un index de filtre de Bloom exploité par la suite pour la recherche de descendants d'un filtre,
- la notion d'inclusion partielle de filtres de Bloom qui permet de réduire le coût du calcul de la descendance d'un filtre de Bloom.

La Section 4.1 introduit la notion de filtre de Bloom fragmenté. Ensuite la Section 4.2 définit la relation d'inclusion entre fragments d'une part, et d'inclusion partielle de filtres de Bloom d'autre part. La Section 4.3 présente la procédure d'indexation de filtres fragmentés dans une DHT, alors que la Section 4.4 se focalise sur le protocole de recherche de sur-ensembles d'un filtre stockés dans `FragIndex`. La Section 4.5 discute les performances de ce schéma d'index tandis que la section 4.6 revient sur certains choix de conception et de configuration et discute des alternatives ou des pistes d'amélioration. Enfin, la Section 4.7 conclut ce chapitre.

4.1 Construction et caractérisation des filtres de Bloom fragmentés

4.1.1 Principe de fragmentation

Pour simplifier, nous considérons d'abord que la longueur m des filtres de Bloom est telle qu'il existe deux entiers λ et c tels que $m{=}\lambda \times c$. Ainsi, nous assimilons un filtre de Bloom à la

concaténation de λ mots-binaires de longueur c bits chacun. Par convention, ces mots-binaires sont numérotés de 1 à λ en commençant par la gauche.

Ensuite, à chaque mot-binaire distinct, nous associons la liste de toutes ses positions d'occurrences dans le filtre de Bloom et nous désignons par *fragment*, tout couple (v, p) formé d'un mot-binaire v de longueur c et de l'ensemble p de ses positions d'occurrences dans le filtre de Bloom. Par commodité, nous dirons qu'un fragment est de longueur c qui renvoie à la longueur du mot binaire v du fragment.

Nous obtenons enfin une suite d'au maximum λ fragments correspondant à la décomposition du filtre de Bloom en λ mots binaires.

Exemple 4.1. Soit F_1=1001 0010 0101 1111 0010 0010 0101 1111, un filtre de Bloom de 32 bits. F_1 est la concaténation de 8 mots binaires de 4 bits chacun (c = 4) et comprend quatre fragments. Le filtre de Bloom F_1 peut être réécrit comme suit :
F_1= {f_1, f_2, f_3, f_4 } avec f_1=(1001, {1}), f_2=(0010, {2,5,6}), f_3=(0101, {3,7}) et f_4=(1111, {4,8}).

La fragmentation consiste alors à réécrire le filtre de Bloom sous la forme d'un ensemble de fragments. C'est un processus réversible car nous reconstituons aisément le filtre de Bloom initial par la concaténation successive des mots-binaires contenus dans les fragments, et ce, dans l'ordre de leurs positions.

4.1.2 Représentations fragmentées d'un filtre de Bloom

Soit F un filtre de Bloom de longueur m, λ, c et k trois entiers tels que m=$\lambda \times c$ et $k \leq \lambda$. Tout ensemble $\{(v_1, p_1), \ldots, (v_k, p_k)\}$ est une représentation fragmentée de F de paramètre c si les conditions suivantes sont satisfaites :

- $\forall i \in \{1, \ldots, k\}, v_i$ est un mot-binaire de longueur c et $p_i \subset \{1, \ldots, \lambda\}$.
- $(1 \leq k \leq \lambda) \wedge (\bigcup\{p_i\}_{1 \leq i \leq k} = \{1, \ldots, \lambda\})$
- Si un filtre de Bloom F est fragmenté en λ morceaux de longueur c, numérotés de 1 à λ à partir du morceau le plus à gauche, alors $\forall j \in \{1, \ldots, \lambda\}$, $(\exists i, 1 \leq i \leq k)$ tel que $((j \in p_i) \wedge (v_i == F[j]))$, où $F[j]$ désigne le j^{ième} morceau de F.

Nous notons $\mathcal{R}(F, c)$ une représentation fragmentée de F de paramètre c.

Soit $\mathcal{R}(F, c)$ = $\{(v_1, p_1), \ldots, (v_k, p_k)\}$ une représentation fragmentée d'un filtre F de longueur m. Nous disons que :

- $\mathcal{R}(F, c)$ est canonique si elle comprend autant de fragments que de positions, c'est-à-dire k = λ.
- $\mathcal{R}(F, c)$ est dite factorisée si $\forall i, j \in 1, \ldots, k, (i \neq j) \Rightarrow (v_i \neq v_j)$

4.1.3 Caractérisation d'un filtre par un ensemble de mots-binaires

Un filtre peut avoir plusieurs représentations fragmentées de paramètre c. Toutefois, pour une longueur de fragment donnée, toutes les représentations fragmentées utilisent le même ensemble de mots binaires.

Nous appelons description binaire de F de taille c, notée $\mathcal{D}_b(F,c)$, l'ensemble de mots-binaires distincts utilisés par toutes les représentations fragmentées de F en fragments de longueur c. Si F est de longueur m, $\mathcal{D}_b(F,c)$ contient au plus λ mots avec $\lambda = \frac{m}{c}$.

Soit id_F, l'identifiant du filtre de Bloom F et $\mathcal{R}(F,c)$ une représentation fragmentée de F. En associant à chaque fragment de la représentation l'identifiant du filtre de Bloom, nous obtenons un ensemble de couples contenant chacun un fragment et un identifiant. Nous désignons cet ensemble par *Filtre de Bloom Fragmenté (FBF)* et nous disons que $fbf(F)$ est le filtre de Bloom fragmenté de F

Exemple 4.2. Si id_1 identifie le filtre F_1 de l'exemple 4.1 et $\mathcal{R}(F_1,4) = \{f_1,f_2,f_3,f_4\}$ sa représentation fragmentée, alors l'ensemble $FBF_1 = \{(f_1,id_1);(f_2,id_1);(f_3,id_1);(f_4,id_1)\}$ est le filtre de Bloom fragmenté construit à partir du filtre F_1.

Dans la suite, nous admettons qu'il existe une fonction `split(`F`)` (respectivement `merge(`FBF`)`) qui permet de fragmenter un filtre de Bloom F (respectivement de fusionner les fragments du FBF) passé en paramètre. De même qu'il existe une fonction `creerFBF(`$\mathcal{R}(F,c)$`,` id_F`)` qui construit un filtre de Bloom fragmenté à partir d'une représentation fragmentée et d'un identifiant de filtre.

Ainsi, considérant toujours notre exemple, nous avons :
`split(`F_1`)` donne $\mathcal{R}(F_1,4) = \{f_1,f_2,f_3,f_4\}$, `creerFBF(`$\mathcal{R}(F_1,4)$`,` id_1`)` donne $FBF_1 = \{(f_1,id_1);(f_2,id_1);(f_3,id_1);(f_4,id_1)\}$ et
`merge(`FBF_1`)` donne F_1.

4.2 Relations d'inclusion partielle et calcul de descendances

Cette section définit une relation d'inclusion entre fragments et quelques relations d'inclusion partielle entre filtres de Bloom, puis propose une condition nécessaire et suffisante pour qu'un filtre contienne un autre ainsi qu'un moyen de déterminer la descendance d'un filtre à partir de celles de ses fragments.

4.2.1 Relation d'inclusion entre les fragments

Soient $f_1 = (v_1,p_1)$ et $f_2 = (v_2,p_2)$, deux fragments de filtres de longueur c. Nous disons que f_1 contient f_2, noté $f_2 \subset f_1$, si et seulement si les deux conditions suivantes sont satisfaites :

1. $v_2 \rightsquigarrow v_1$, c'est-à-dire $\forall i, 0 \leq i < c, v_2[i] = 1 \Rightarrow v_1[i] = 1$,
2. $p_1 \cap p_2 = p_2$

4.2.2 Relations d'inclusion partielle

Soient F et Q deux filtres de longueur m ; $\mathcal{R}(F,c) = \{f_1,\ldots,f_k\}$ une représentation fragmentée de F avec $\forall i \in \{1,\ldots,k\}, f_i = (v_i,p_i)$; $\mathcal{R}(Q,c) = \{q_1,\ldots,q_\lambda\}$ la représentation fragmentée canonique de Q ; et $p \subset \{1,\ldots,\lambda\}$.

Nous disons que *F* contient partiellement *Q* relativement aux fragments situés aux positions contenues dans p, si et seulement si : $\forall j \in p, \exists i \in \{1, \ldots, k\}$ tel que $q_j \subset f_i$.

Nous notons $S_p(Q, c)$ l'ensemble de tous les filtres de longueur m qui contiennent partiellement*Q* relativement aux fragments situés aux positions contenues dans p.

Propriété 4.1. Soient Q un filtre de m bits, $\mathcal{R}(Q, c) = \{q_1, \ldots, f_\lambda\}$ la représentation fragmentée canonique de *Q*, et p_z l'ensemble des positions des fragments de $\mathcal{R}(Q, c)$ de valeur nulle. Tout filtre *F* de m bits contient partiellement *Q* relativement à ses fragments de valeur égale zéro.

Démonstration : *Considérons* $\{f_1, \ldots, f_\lambda\}$ *la représentation fragmentée canonique de* F. *Posons* $\forall j \in \{1, \ldots, \lambda\}, q_j = (w_j, s_j), f_j = (v_j, p_j)$. *On a* $\forall i \in p_z$ *on a : (i)* $s_i \cap p_i = s_i = \{i\}$ *; et* $w_j == 0$.
L'implication $\forall l, 0 \leq l \leq c-1, (w_j[l] == 1) \Rightarrow (v_j[l] == 1)$ *est donc satisfaite. On en déduit que* $\forall j \in p_z, q_j \subset f_j$. *Donc* F *contient partiellement* Q *relativement à chacun de ses fragments de valeur nulle.*

4.2.3 Calcul de la descendance d'un filtre

Soient *F* et *Q* deux filtres de longueur m, $\mathcal{R}(F, c) = \{f_1, \ldots, f_k\}$ une représentation fragmentée de longueur c de *F* avec $\forall i \in \{1, \ldots, k\}, f_i = (v_i, p_i)$, et $\mathcal{R}(Q, c) = \{q_1, \ldots, q_\lambda\}$ la représentation fragmentée canonique de longueur c de *Q*. La propriété suivante donne une condition nécessaire et suffisante pour qu'un filtre de Bloom F contienne un filtre Q.

Propriété 4.2.

$$(Q \rightsquigarrow F) \Leftrightarrow (\forall j \in \{1, \ldots, \lambda\}, \exists i \in \{1, \ldots, k\}, q_j \subset f_i) \tag{4.1}$$

Démonstration :
(i) *Supposons que* $Q \rightsquigarrow F$. *Soit* $\{f_1, \ldots, f_k\}$ *avec* $\forall i \in \{1, \ldots, k\} f_i = (v_i, p_i)$ *une représentation fragmentée de* F *avec des fragments de longueur c. Les ensembles* $p_1, \ldots, p_k$ *forment une partition de l'ensemble de positions de fragments* $\{1, \ldots, \lambda\}$.

Ainsi $\forall j \in \{1, \ldots, \lambda\}, (\exists i \in \{1, \ldots, k\} \wedge j \in p_i)$. *Le fragment* q_j *de* Q *a pour valeur la suite de bits* $Q[(j-1)*c] \ldots Q[(j*c)-1]$ *de Q et comme* $j \in p_i$, *la suite* $F[(j-1)*c] \ldots F[(j*c)-1]$ *est la valeur* v_i *du fragment* f_i. *Puisque* $Q \rightsquigarrow F$, *on en déduit que* $\forall \alpha \in \{(j-1)*c, \ldots, (j*c)-1\}, (Q[\alpha] == 1) \Rightarrow (F[\alpha] == 1)$. *Comme* $\{j\} \subset p_i$, *on conclut que* $q_j \subset f_i$.
(ii) *Supposons que* $\forall j \in \{1, \ldots, \lambda\}, ((\exists i \in \{1, \ldots, k\}) \wedge (q_j \subset f_i))$. *Soit* r *un entier compris entre* 0 *et* $m-1$, *on veut montrer que* $(Q[r] == 1) \Rightarrow (F[r] == 1)$.

En effet, lorsqu'on fragmente un filtre de longueur m *en fragments de* c *bits et qu'ensuite on numérote les fragments de gauche à droite à partir de 1, le bit* r *appartient au fragment de position* $\alpha = (r/c) + 1$.

Ainsi on a $1 \leq \alpha \leq \lambda, avec \lambda = m/c$. *D'après l'hypothèse, on en déduit que :* $\exists \eta \in \{1, \ldots, k\}$ *tel que* $(\alpha \in p_\eta) \wedge (q_\alpha \subset f_\eta)$. *Or* $((q_\alpha \subset f_\eta) \wedge (\alpha \in p_\eta)) \Rightarrow (\forall l \in \{(\alpha-1)*c, \ldots, (\alpha*c)-1\}, (Q[l] == 1) \Rightarrow (F[l] == 1))$. *Par construction,* $(\alpha-1)*c \leq r \leq (\alpha*c)-1$. *Ainsi, on a montré que* $\forall r, 0 \leq r \leq (m-1), (Q[r] == 1) \Rightarrow (F[r] == 1)$, *c'est-à-dire que* $Q \rightsquigarrow F$.

Enfin, la propriété suivante permet de déterminer la descendance d'un filtre à partir de celles de ses fragments.

Propriété 4.3. Soient Q un filtre de longueur m et $\mathcal{R}(Q,c) = \{q_1,\ldots,q_\lambda\}$, avec $\forall j \in \{1,\ldots,\lambda\}, q_j = (v_j, s_j)$, la représentation fragmentée canonique de Q avec des fragments de longueur c. Soit $\mathcal{D}(Q)$ l'ensemble des descendants du filtre Q.

$$\mathcal{D}(Q) = \bigcap S_{s_j}(Q,c)_{1\leq j\leq\lambda} \tag{4.2}$$

En d'autres termes, l'ensemble de descendants d'un filtre Q est égal à l'intersection des ensembles de filtres de même longueur qui le contiennent partiellement relativement aux fragments de sa représentation fragmentée canonique.

Démonstration :
(i) *Supposons que* F *est un descendant de* Q. *Soit* $\mathcal{R}(F,c) = \{f_1,\ldots,f_k\}$ *une représentation fragmentée de* F *avec des fragments de longueur* c *et telle que* $\forall i \in \{1,\ldots,k\}, f_i = (v_i, p_i)$. *D'après la propriété 4.1,* $\forall j \in \{1,\ldots,\lambda\}, ((\exists i \in \{1,\ldots,k\}) \wedge (q_j \subset f_i))$. *C'est-à-dire que* F *contient partiellement* Q *relativement à chacun des fragments de sa représentation fragmentée canonique. Autrement dit,* F *appartient à chaque ensemble de filtres de longueur* m *bits qui contiennent partiellement* Q *relativement à chacun des fragments de longueur* c *bits de sa représentation fragmentée canonique. Formellement* $F \in \bigcap S_{s_j}(Q,c)_{1\leq j\leq\lambda}$.
(ii) *Supposons maintenant que* $\mathrm{F} \in \bigcap S_{s_j}(Q,c)_{1\leq j\leq\lambda}$ *et considérons* $\mathcal{R}(F,c) = \{f_1,\ldots,f_k\}$ *une représentation fragmentée de* F *avec des fragments de longueur* c *telle que* $\forall i \in \{1,\ldots,k\}, f_i = (v_i, p_i)$.

Soit $r, 0 \leq r \leq (m-1)$ *tel que* $Q[r] == 1$. *Le bit* r *appartient au fragment* q_α *de la représentation fragmentée canonique de* Q *avec* $\alpha = (r/c) + 1$. *D'après l'hypothèse, nous avons* $\mathrm{F} \in S_{\{\alpha\}}(Q,c)$. *Or* $(\mathrm{F} \in S_{\{\alpha\}}(Q,c)) \Rightarrow (\forall l \in \{(\alpha-1)*c,\ldots,(\alpha*c)-1\}, (Q[l] == 1) \Rightarrow (F[l] == 1))$. *Par construction,* $r \in \{(\alpha-1)*c,\ldots,(\alpha*c)-1\}$. *On en déduit que* $\forall r \in 0,\ldots,m-1, (Q[r] == 1) \Rightarrow (F[r] == 1)$, *c'est-à-dire que* $Q \rightsquigarrow F$.

4.3 Indexation des filtres de Bloom dans FragIndex

FragIndex associe un identifiant unique à chaque filtre qu'il indexe. Soit c un paramètre de configuration qui fixe la taille des fragments des filtres de Bloom résumant les documents et les requêtes. FragIndex associe à chaque mot-binaire v de longueur c une liste de couples (p, fid), où p est l'ensemble des positions occupées par v dans le filtre de Bloom identifié par fid.

L'indexation d'un filtre de Bloom (*c.f.* Algorithme 4.1) consiste d'une part à le mettre sous forme d'un ensemble de fragments et d'autre part à répartir les fragments non vides (*i.e* les fragments ayant au moins un bit 1) sur une *Distributed Hash Table (DHT)* en utilisant le principe des listes inversées.

Pour ce faire, FragIndex détermine une représentation fragmentée factorisée pour chaque filtre de Bloom et associe aux fragments de la représentation l'identifiant du filtre de Bloom (Ligne 2 à Ligne 4).

Algorithme 4.1 indexation d'un filtre de Bloom

Input: F, c

```
1:  function add(F)
2:      R(F, c) ← split(F);
3:      fid ← timestamp();
4:      FBF ← creerFBF(R(F, c), fid);
5:      for each (f, fid) ∈ FBF do
6:          if (f ≠ 0) then // cas des fragments vides
7:              δ ← (fid, p(f));
8:              DHT.put(hash(v(f)), δ);
9:          end if
10:     end for
11:     return;
12: end function
```

Ainsi, pour chaque filtre de Bloom à indexer, `FragIndex` construit un *FBF* qui est un ensemble de couples $\{C_i = (f_i, fid), i \in \{1, \ldots, k \leq \lambda\}\}$. Chaque élément de cet ensemble est un couple C_i composé d'un fragment f_i qui désigne un mot-binaire ($v(f_i)$) accompagné de ses positions d'occurrence ($p(f_i)$) et de l'identifiant fid du filtre.

Après la création des *FBFs* correspondant aux filtres de Bloom des documents publiés, `FragIndex` fabrique un index inversé avec les mots-binaires des fragments et répartit les entrées de cet index sur la *DHT*.

Pour cela, les couples $\{C_i = (f_i, fid), i \in \{1, \ldots, k \leq \lambda\}\}$ sont regroupés en fonction de leur mot-binaire $b = v(f_i)$. Puis, à chaque mot-binaire b, `FragIndex` fait correspondre la liste de couples $(p(f_i), fid)$ où $p(f_i)$ est l'ensemble de positions de fragment de valeur b du filtre identifié par *fid* (Ligne 5 à Ligne 10. Il faut noter que pour réduire la quantité de données indexées, les fragments vides (*i.e.* composés uniquement de 0) ne sont pas indexés (Ligne 6).

Désignons par δ, la structure de donnée formée par le couple $(p(f_i), fid)$, δ comprend les positions d'un fragment f_i ($p(f_i)$) et l'identifiant (fid) du filtre contenant ce fragment. Ainsi, nous associons à chaque mot-binaire une liste de données δ. Le Tableau 4.2 suivant montre un exemple d'index de listes inversées de mots-binaires construit à partir des filtres de Bloom utilisés pour stocker des documents.

Comme dans la solution de [Joung et al. 07], `FragIndex` dispose d'un ensemble d'entrées sous forme de mots-binaires de longueur c. Dès lors, la suite du processus d'indexation des filtres de Bloom consiste à utiliser ces mots-binaires comme des clés d'indexation pour distribuer ces entrées sur la *DHT*. Cela se fait en associant chaque mot-binaire à un identifiant de serveur d'indexation (Figure 4.1).

Soient $\mathcal{F}$ l'ensemble des filtres de Bloom résumant les documents à indexer, $\mathcal{B}$ l'ensemble des mots-binaires de longueur c contenus dans les représentations fragmentées des filtres de $\mathcal{F}$ ($\mathcal{B} = \cup_{F \in \mathcal{F}} \mathcal{D}_b(F, c)$) et $\mathcal{K}$ l'espace des identifiants des serveurs sur la *DHT*. `FragIndex`

Filtre utilisé comme clé de stockage	fid
10100100 10001010 00000100 11001001	Id_1
10100111 10010010 00011100 10010010	Id_2
10100011 10011010 10100100 10101110	Id_3
00101010 10011110 01100000 10001011	Id_4
00101110 10010110 10010110 00101010	Id_5

Tableau 4.1 – Exemple de filtres utilisés pour stocker des documents.

Entrée	Enregistrement
10100100	({1}, Id_1) ; ({3}, Id_3)
10001010	({2}, Id_1)
00000100	({3}, Id_1)
11001001	({4}, Id_1)
10100111	({1}, Id_2)
10010010	({2,4}, Id_2)
00011100	({3}, Id_2)
10100011	({1}, Id_3)
10011010	({2}, Id_3)
10101110	({4}, Id_3)
00101010	({1}, Id_4) ; ({4}, Id_5)
10011110	({2}, Id_4)
01100000	({3}, Id_4)
10001011	({4}, Id_4)
00101110	({1}, Id_5)
10010110	({2,3}, Id_5)

Tableau 4.2 – Exemple d'index inversé de mots-binaires.

utilise une fonction de hachage $hash$ dont le rôle est de faire correspondre chaque mot-binaire à un seul serveur de la *DHT*. Une telle fonction peut être définie comme suit :

$$hash\colon \forall b \in \mathcal{B},\ \exists !\, key \in \mathcal{K} \mid key = hash(b)$$

Sur chaque serveur d'index, `FragIndex` stocke ainsi des méta données servant à reconstruire les filtres de Bloom utilisés pour stocker les documents. Comparée à la solution de [Joung et al. 07] qui utilise beaucoup d'espace pour stocker la totalité des mots-clés sur les serveurs d'index, la solution `FragIndex` parvient à atténuer la charge d'indexation des serveurs.

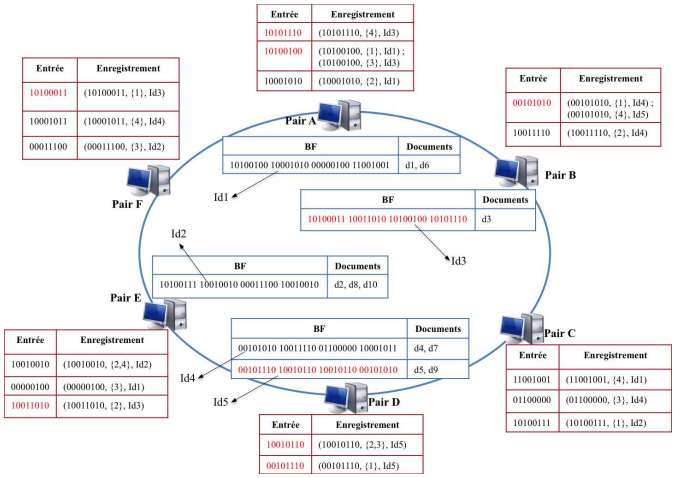


FIGURE 4.1 – Indexation des filtres de Bloom dans une DHT

4.4 Recherche de sur-ensembles d'un filtre

Soit $\mathcal{F}$ l'ensemble des filtres de Bloom indexés par `FragIndex`. Le but du protocole de recherche est de retrouver tous les éléments de $\mathcal{F}$ qui sont descendants du filtre passé en paramètre. Afin d'atteindre cet objectif, `FragIndex` exploite la relation d'inclusion partielle des filtres de Bloom et implémente un protocole de recherche de filtres qui contiennent partiellement le filtre de Bloom donné. D'après la propriété 4.3, l'ensemble de sur-ensembles du filtre de la requête sera l'intersection de tous les ensembles de filtres qui le contiennent partiellement.

Pour faciliter la compréhension du protocole de recherche de sur-ensembles d'un filtre, nous présentons d'abord à la section 4.4.1 une version centralisée du protocole de recherche de sur-ensembles. La recherche est réalisée par un coordonnateur qui fait des appels à distance successivement à tous les serveurs d'index susceptibles de stocker des fragments de filtres qui contiennent partiellement la requête. Ensuite la section 4.4.2 expose la version distribuée du même protocole de recherche. Cette version permet de réduire le trafic réseau en demandant à chaque serveur visité de relayer la recherche au serveur d'index suivant.

4.4.1 Recherche centralisée par un coordonnateur

L'Algorithme 4.2 présente la procédure de recherche exécutée par le coordonnateur, c'est-à-dire tout serveur d'index qui reçoit une demande contenant un filtre de Bloom Q dont nous recherchons les sur-ensembles et l'identifiant $clientID$ du pair émetteur de la requête.

Cette procédure de recherche centralisée comprend trois phases : la fragmentation du filtre de Bloom de la requête et la détermination de l'ordre d'exploration des différents fragments, la récupération de tous les fragments de filtres stockés dans `FragIndex` et contenant partiellement Q, et la reconstruction des filtres sur-ensemble de Q à partir de leurs fragments récupérés. Nous expliquons ces trois phases dans les paragraphes 4.4.1.1, 4.4.1.2, et

Algorithme 4.2 Recherche de sur-ensembles par un coordonnateur

Input: filtre de Bloom : Q
Input: Identifiant pair émetteur : $clientID$
1: **function** superSetSearch($Q, clientID$)
2: $\mathcal{R}(Q,c) \leftarrow$ split(Q) ; // *détermination de la représentation fragmentée de Q*
3: sort($\mathcal{R}(Q,c)$) ; // *Ordonne les fragments selon le nombre de bits 1 présents*
4: fbfCont $\leftarrow$ collectSuperSetFrags($\mathcal{R}(Q,c)$) ; // *Retourne la liste des fbf sur-ensembles de $\mathcal{R}(Q,c)$*
5: bfCont = traiterFinCollecte(fbfCont) ;
6: send(bfCont, $clientID$) ; // *Retourne la liste des FB sur-ensembles de Q à l'initiateur de la recherche*
7: **return** ;
8: **end function**

4.4.1.3. Une fois l'ensemble de filtres sur-ensembles de la requête reconstitué, cet ensemble est retourné au client qui a initié la requête.

4.4.1.1 Phase 1 : fragmentation du filtre de la requête et détermination de l'ordre d'exécution

La première phase de l'Algorithme 4.2 consiste à fragmenter le filtre de Bloom de la requête de mots-clés. La fonction `split(`Q`)` (Ligne 2) utilise le même paramètre λ que lors de l'indexation pour fragmenter le filtre de la requête et on obtient au maximum λ fragments.

Ensuite, la fonction `sort(`$\mathcal{R}(Q,c)$`)` ordonne les fragments obtenus en commençant par celui qui dispose du plus grand nombre de bits 1 (Ligne 3). Par exemple, pour la requête représentée par le filtre de Bloom $Q = 00100010\,10010010\,00000100\,00100010$, nous obtenons 3 fragments à l'issu de la fragmentation (Figure 4.2) qui constitue chacun une étape de recherche. L'ordre d'exécution de cette requête est déterminé en ordonnant les fragments selon le nombre de bits 1 qu'ils contiennent. Ici, il s'agira d'abord de traiter le fragment qf_2 puis le fragment qf_1 et enfin le fragment qf_3 qui contient le moins de bits 1.

FIGURE 4.2 – Phase 1 du protocole de recherche

4.4.1.2 Phase 2 : récupération des fragments qui contiennent partiellement le filtre de la requête

La deuxième phase de l'Algorithme 4.2 (Ligne 4) consiste à récupérer les fragments des filtres qui contiennent partiellement le filtre de Bloom Q de la requête.

Dans cette phase de la recherche, l'Algorithme explore les nœuds d'index responsables des mots-binaires appartenant aux descendances des fragments de la requête. Ainsi, avec la représentation fragmentée $\mathcal{R}(Q,c)$ du filtre de Bloom de la requête, le coordonnateur exécute

la procédure décrit par l'Algorithme 4.3 qui parcourt l'ensemble des descendants de chaque fragment de $\mathcal{R}(Q, c)$.

Cette procédure itère sur tous les fragments de la représentation $\mathcal{R}(Q, c)$ en considérant à chaque itération celui qui a le plus de bits 1. Pour chaque fragment, la procédure détermine sa descendance composée d'un ensemble mbs de mots-binaires qui sont potentiellement associés à des serveurs indexant des données δ.

Le coordonnateur interroge ensuite les serveurs d'index responsables des mots-binaires de la descendance des fragments $\mathcal{R}(Q, c)$ puis sélectionne les données δ associées aux mots-binaires et récupère les fragments de filtres sur-ensembles de la requête.

Algorithme 4.3 Collecte des fragments des sur-ensembles de la requête

Input: $\mathcal{R}(Q, c)$

```
1:  function collectSuperSetFrags(R(Q, c))
2:      continuer ← true ; // Continuer la recherche tant que ce boolean est true
3:      fbfs ← ∅ ; // Conteneur de fbf satisfaisant la requête.
4:      vfrs ← ∅ ; // Conteneur des fragments déjà explorés.
5:      mbs ← ∅ ; // Conteneur de mots binaires à explorer.
6:      vmbs ← ∅ ; // Conteneur de mots binaires déjà visités.
7:      for each step ∈ R(Q, c) do
8:          mbs ← descendants(v(step)) ;
9:          for each b ∈ mbs do
10:             if ((b ≠ 0) ∧ (b ∉ vmbs)) then // tous les mots binaires sauf celui composé de 0 uniquement
11:                 records ← Index.getSuperSetFrags(R(Q, c), b) ; // Appel distant de la méthode
    du serveur d'index.
12:                 for each fb ∈ records do
13:                     if ((fbfs == ∅) ∨ (vfrs == ∅) ∨ (fb.fid ∈ fbfs)) then
14:                         fbfs.add(fb) ;
15:                     end if
16:                 end for
17:                 vmbs.add(b) ;
18:             end if
19:         end for
20:         vfrs.add(step) ;
21:         prune(fbfs, vfrs) ;
22:     end for
23:     return fbfs ; // Retourne le conteneur des FBF qui contiennent partiellement la requête.
24: end function
```

À la fin de chaque itération traitant un fragment de $\mathcal{R}(Q, c)$, le coordonnateur élimine tout filtre qui ne contient pas partiellement Q relativement à tous les fragments déjà traités. C'est le rôle de la procédure $prune(fbfs, vfrs)$ (c.f Algorithme 4.4) qui élimine tout élément F de $fbfs$ qui ne contient pas partiellement la requête Q relativement aux positions des fragments déjà traités. Le fonctionnement de cette procédure est simple et consiste à vérifier si chaque

Algorithme 4.4 épuration des réponses

Input: $fbfs$ // *conteneur de* fbf
Input: G // *groupe de fragments déjà explorés*
1: **function** prune($fbfs, G$)
2: **for each** $F \in fbfs$ **do**
3: **if** ($\exists q \in G, q = (v, p)$ tel que $F \not\subset_p (Q, c)$ **then** // *q est un fragment de Q déjà traité*
4: supprimer F dans $fbfs$;
5: **end if**
6: **end for**
7: **return** $fbfs$; // *retourne les* fbf *restant*
8: **end function**

fbf récupéré contient partiellement Q, le filtre de la requête, relativement à tous les fragments déjà visités. Si tel n'est pas le cas, le fbf en question est enlevé de la liste des réponses.

La collecte des sur-ensembles partiels décrit par l'Algorithme 4.3 se termine quand tous les fragments de la description $\mathcal{R}(Q, c)$ sont traités. L'Algorithme retourne alors un conteneur de triplets (v, p, fid) où v est un mot-binaire de longueur c, p un ensemble de positions de fragments, et fid un identifiant de filtre indexé par `FragIndex`.

Les triplets sont regroupés en fonction des identifiants de filtres fid et tous les triplets ayant le même identifiant forment un seul filtre de Bloom fragmenté. Ainsi, le coordonnateur dispose à la fin de la procédure de tous les fbf qui peuvent être des sur-ensembles du filtre de Bloom de la requête.

Détermination de fragments de sur-ensembles par chaque serveur d'index
Afin de déterminer les réponses à retourner au coordonnateur, chaque serveur d'index exécute la procédure *getSuperSetFrags()* de l'Algorithme 4.5. Pour chaque donnée δ (avec $\delta = (p, fid)$) associée au mot-binaire donné, la procédure construit le fragment correspondant et vérifie si ce dernier contient partiellement la requête. Si oui, le fragment et l'identifiant du filtre le contenant sont ajoutés sur la liste des réponses à retourner au coordonnateur.

4.4.1.3 Phase 3 : reconstitution des sur-ensembles de la requête

À la fin de l'exécution de l'Algorithme 4.3, la structure *fbfCont* retournée contient tous les fragments non vides des filtres de Bloom fragmentés qui sont des sur-ensembles du filtre de Bloom $\mathcal{R}(Q, c)$ de la requête.

Le coordonnateur utilise alors la procédure $traiterFinCollecte()$ en lui passant le conteneur de fragments retourné par la procédure de collecte. Pour chaque identifiant de filtre présent dans *fbfCont*, cette procédure récupère l'ensemble de fragments associés à cet identifiant, puis insère les différents mots-binaires aux positions indiquées par chaque fragment.

Les positions pour lesquelles des fragments n'existent pas sont complétées avec des fragments de valeur constituée des bits 0. Pour cela, nous admettons l'existence de la fonction $isCompleted(fbf)$ qui vérifie si un fbf dispose d'un fragment à toutes les λ positions ou

Algorithme 4.5 Retourne la liste de fragments de sur-ensembles indexés de la requête

Input: $\mathcal{R}(Q, c)$
Input: v
1: **function** getSuperSetFrags($\mathcal{R}(Q, c), v$)
2: FBF[ ] list $\leftarrow \emptyset$
3: $\Delta \leftarrow$ localIndexStore.get(v) ; // $\Delta = \{\delta_0, \delta_n\}$: *ensemble de couples (p,fid)*
4: **for each** $\delta \in \Delta$ **do**
5: isSuperSetFrag = true ;
6: Fragment fr $\leftarrow$ creerFragment(v,δ.p)
7: **for each** j $\in \delta$.p **do**
8: q $\leftarrow$ getFragment($R(Q, c), j$) // *Retourne le fragment de Q à la position j.*
9: **if** ($q \not\subset fr$) **then**
10: isSuperSetFrag = false ;
11: break ;
12: **end if**
13: **end for**
14: **if** (isSuperSetFrag == true) **then**
15: list.add(δ.fid, fr) ;
16: **end if**
17: **end for**
18: **return** (list)
19: **end function**

non et nous considérons la procédure $complete(fbf)$ (*c.f.* Algorithme 4.6) qui associe des fragments vides aux positions n'ayant pas de fragments.

4.4.2 Recherche répartie de sur-ensembles

Comme la version centralisée du protocole de recherche de sur-ensembles, l'objectif du protocole de recherche répartie est de retrouver les sur-ensembles du filtre de la requête qui sont stockés dans `FragIndex`. La différence entre la version centralisée et la version distribuée réside dans la procédure de collecte de sur-ensembles partiels du filtre de la requête, c'est-à-dire la phase 2 du protocole de recherche de sur-ensembles. Les phases 1 et 3 restent inchangées.

Pour la version centralisée, la collecte est réalisée par un coordonnateur, alors que pour la version distribuée la collecte est réalisée par un agent qui visite successivement tous les serveurs d'index pertinents et collecte progressivement les fragments des sur-ensembles partiels de la requête.

Ainsi, comparée à sa version centralisée, la version distribuée du protocole de recherche de sur-ensembles vise à réduire la latence de la recherche, la consommation de bande passante, et le trafic réseau. De plus la solution distribuée répartit la charge du traitement entre les serveurs d'index concernés plutôt que de centraliser tout au niveau du serveur coordonnateur.

Le principal challenge de la procédure de collecte distribuée de sur-ensembles partiels du filtre de la requête est de déterminer un algorithme de parcours des serveurs d'index en

Algorithme 4.6 Rajout des fragments vides et reconstitution des filtres

```
Input: fbfCont // Conteneur de fragments des sur-ensembles
1:  function traiterFinCollecte(msg)
2:      // Exécutée à la fin de collecte de fragments de sur-ensembles de la requête.
3:      bfCont = createBloomFilterConteneur();
4:      for each fbf ∈ fbfCont do
5:          if (isCompleted(fbf)==false) then
6:              complete(fbf);
7:          end if
8:          F ← merge (fbf); // Reconstruit le filtre à partir des fragments du fbf.
9:          bfCont.add(F);
10:     end for
11:     return bfCont;
12: end function
13: function complete(fbf)
14:     p_z ← {1 ... λ} \ {∪_{p(f),∀f∈fbf}};
15:     if (p_z ≠ ∅) then
16:         fragment f;
17:         p(f) ← p_z;
18:         v(f) ← 0; // fragment vide
19:         ajouter f dans fbf;
20:     end if
21:     return fbf; // retourne le fbf complet
22: end function
```

charge des listes associées aux descendants des fragments de ce filtre. La Section 4.4.2.1 présente l'algorithme de construction de l'arbre de descendants d'un fragment. La construction des arbres de descendance de fragments est la première brique de base de la procédure de collecte distribuée. Ensuite la Section 4.4.2.2 présente un algorithme de parcours des arbres de descendance des fragments d'une requête. Enfin, la Section 4.4.2.3 présente la procédure de collecte distribuée qui exploite l'algorithme de parcours de descendance d'une chaîne de bits pour parcourir tous les serveurs indexant des fragments qui contiennent partiellement le filtre de la requête.

4.4.2.1 Construction de l'arbre de descendance d'une chaîne de bits

Soit Q un filtre de m bits et $\mathcal{R}(Q, c)$ une représentation fragmentée de Q avec des fragments de c bits. Soit (q_v, q_p) un élément de $\mathcal{R}(Q, c)$, q_v est une chaîne de c bits et q_p un sous-ensemble de l'ensemble de positions de fragments de Q.

Soit $\mathcal{D}(q_v)$ l'ensemble de descendants de q_v. Soit $P = \{0, \ldots, c-1\}$. Nous définissons la relation de *précédence immédiate* entre les descendants de q_v, notée $\prec$, comme suit : $\forall v, w \in \mathcal{D}(q_v), (v \prec w)$ si et seulement si il existe $l \in P$ tel que les 3 relations suivantes sont satisfaites

1. $(v[l] == 0) \wedge (w[l] == 1)$
2. $\forall j \in P, (j \neq l) \Rightarrow (v[j] == w[j])$
3. $\forall j \in P, (j \geq l) \Rightarrow (v[j] == q_v[j])$

Soit $v, w \in \mathcal{D}(q_v)$ tel que $v \prec w$, nous étiquetons l'arc (v, w) avec l'unique valeur $l \in P$ tel que $v[l] \neq w[l]$. Par construction, l est la longueur du plus long préfixe commun aux chaînes de bits v, w.

Le graphe de descendance d'une chaîne de bits ainsi défini est un arbre valué qui a pour racine la chaîne de bits considérée. Tout arc est étiqueté par la longueur du plus long préfixe commun aux deux sommets qu'il relie.

La Figure 4.3 donne le graphe de descendance de la chaîne "10010001". Cette chaîne ayant 5 bits 0, on a au total 2^5 sommets dans cet arbre de descendance.

Propriétés

1. Chaque nœud de l'arbre de descendance d'une chaîne de bits est complètement défini par la donnée de son prédécesseur par la relation de "*précédence immédiate*" et la connaissance de la longueur du préfixe commun entre le nœud et son prédécesseur. En effet, chaque nœud de l'arbre est obtenu en changeant de 0 à 1 le bit de son prédécesseur de rang égal à la longueur du préfixe commun entre les deux nœuds.

   Par convention, on caractérisera la racine de l'arbre par le couple (racine, -1).

2. La connaissance du prédécesseur d'un nœud par la relation de "*précédence immédiate*" permet de déterminer les frères de droite de ce nœud. En effet, les frères de droite auront le même prédécesseur et sont caractérisés par les positions de bit 0 du prédécesseur supérieures à la position qui caractérise le nœud courant.

Ces deux propriétés de l'arbre de descendance représentant le graphe de la relation de *précédence immédiate* entre descendants d'une chaîne permettent de définir une procédure récursive de parcours de cet arbre.

4.4.2.2 Parcours de l'arbre de descendance d'un fragment

Comme l'illustre la Figure 4.3, un nœud de l'arbre de descendance d'une chaîne de bits peut avoir plusieurs fils. Les fils d'un nœud sont rangés de gauche à droite par ordre croissant de la longueur du préfixe commun de chacun avec le nœud père.

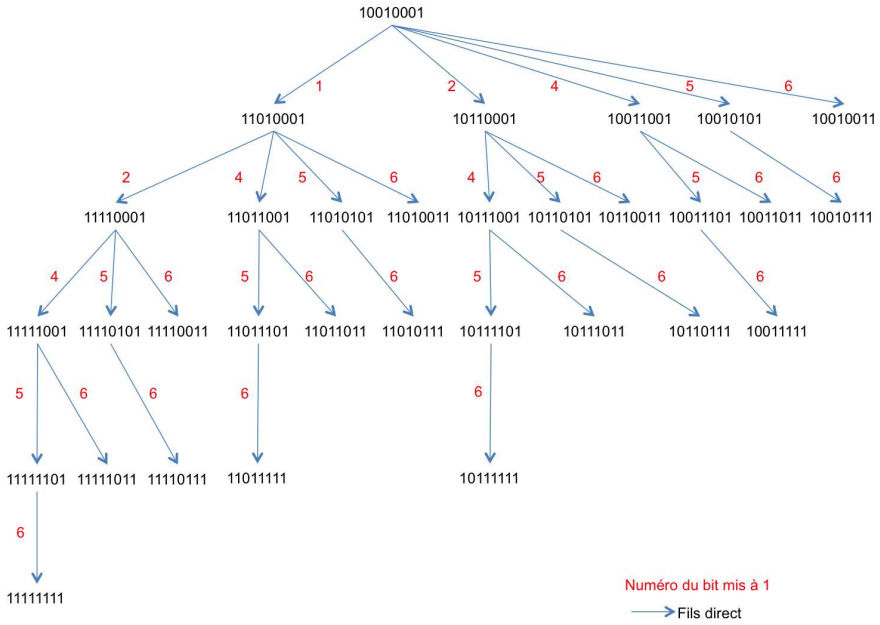


FIGURE 4.3 – Arbre de descendance du fragment 10010001

Pour traverser l'arbre de descendance d'une chaîne, nous utilisons l'algorithme de parcours en profondeur en suivant l'ordre préfixe : *on visite d'abord le nœud, puis ses fils en commençant par le fils le plus à gauche et en terminant par celui qui est le plus à droite.*

L'Algorithme 4.7 présente les principales fonctions utilisées pour réaliser ce parcours.

Algorithme 4.7 Parcours préfixe de l'arbre de descendance

```
1:  /*****************************************************************************
2:  Primitives génériques utilisées quelle que soit la nature de la collecte
3:  *****************************************************************************/
4:  // Traitement générique à la réception d'un msg contenant une requête de collecte.
5:  function onReceiveCollectRequest(msg)
6:      Node visitedNode ← getVisitedNode(msg);
7:      traiterRequete(msg, visitedNode);
8:      sv ← prochaineDestination(msg);
9:      envoyerTo(msg, sv);
10: end function
11: // Initialise la pile msg.stack pour le parcours de l'arbre de descendance de rootStr.
12: function initParcours(msg, rootStr) // Initialisation de la pile de nœuds à visiter
13:     Node node;
14:     node.pred ← rootStr;
15:     node.cplen ← -1;
16:     msg.stack.push(node);
17: end function
18: // Retourne le descendant immédiat de nodeId correspondant au 0 de rang plen
19: function succ(nodeId, plen)
20: // nodeId : chaîne de bits contenant potentiellement des 0 ; plen : position du bit à mettre à 1 si le bit correspondant est 0
21:     if (plen == -1) then return (nodeId); // Ceci désigne le nœud lui-même
22:     else
23:         if ((plen ≥ nodeId.length()) ∨ (nodeId[plen] == 1)) then return (NULL);
24:         else
25:             ch ← nodeId;
26:             ch[plen] ← 1;
27:             return (ch);
28:         end if
29:     end if
30: end function
31: // Retourne le node visité et met à jour la pile pour le parcours préfixe
32: function getVisitedNode(msg)
33:     node ← msg.stack.pop();
34:     Bit[] current ← succ(node.pred, node.cplen);
35:     // Détermination des 2 fils le plus à gauche du nœud visité et les ajouter à la pile
36:     lp_fg ← pzero(current, node.cplen + 1);
37:     if (lp_fg < current.length) then
38:         lp_fd ← pzero(current, lp_fg + 1);
39:         if (lp_fd < current.length) then
40:             Node rnode ← createNode();
41:             rnode.pred ← current;
42:             rnode.cplen ← lp_fd;
43:             stack.push(rnode);
44:         end if
45:         Node lnode ← createNode();
46:         lnode.pred ← current;
47:         lnode.cplen ← lp_fg;
48:         stack.push(lnode);
49:     end if
50:     return node;
51: end function
```

```
52: // Retourne l'identifiant du serveur en charge de la prochaine étape du parcours.
53: function prochaineDestination(msg)
54:     if (msg.stack.notEmpty()) then
55:         Node top ← msg.stack.pop() ;
56:         nextNodeId ← succ(top.pred, top.cplen) ;
57:         sv_suivant ← nodeServer(nextNodeId) ;
58:         msg.stack.push(top) ;
59:     else
60:         sv_suivant ← nodeServer(msg.cid) ;
61:         msg.type ← NotificationFinCollecte ;
62:     end if
63:     return sv_suivant ;
64: end function
65: /*****************************************************************************
66: Méthodes spécifiques
67: *****************************************************************************/
68: // Démarre le parcours de l'arbre de descendance racine demandé par clientId.
69: function parcoursDescendance(racine, clientId)
70:     RequeteParcoursDescendance msg ;
71:     msg.cid ← clientId ;
72:     msg.type ← CollectRequest ;
73:     msg.stack ← createStack() ;
74:     msg ← initParcours(racine) ;
75:     // Démarrage du parcours de l'arbre de la descendance
76:     sv ← prochaineDestination(msg) ;
77:     envoyerTo(msg, sv) ;
78: end function
79: // Traitement d'un msg de collecte avec msg.type = CollectRequest.
80: function traiterRequete(msg, cnode)
81:     msg.repCont.push(succ(cnode.pred, cnode.cplen)) ; // Enregistre le nœud visité dans le conteneur.
82: end function
83: // Traitement d'un msg de collecte msg.type = NotificationFinCollecte.
84: function onReceiveNotificationFinCollecte(msg)
85:     afficher(msg.repCont) ;
86: end function
```

Nous supposons que chaque nœud de l'arbre est géré par un serveur et admettons l'existence des trois primitives suivantes :

1. $nodeServer(nodeId)$ permet de déterminer l'identifiant du serveur en charge du nœud connaissant sa valeur $nodeId$;
2. $envoyerTo(msg, dest)$ permet d'envoyer un message de type $MessageParcoursDescendance$ au serveur donné par le deuxième argument ;

3. $pzero(ch, nb)$ retourne la position du premier bit 0 dans *ch* après *nb* bits ; sinon la longueur de *ch*.

La fonction *parcoursDescendance()* est le point d'entrée. Elle prend en argument la chaîne *racine* dont on veut parcourir l'arbre de descendants, ainsi que l'identifiant du client qui a initié cette demande parcours.

En premier, la fonction *parcoursDescendance()* crée un message de parcours, une structure de type $MessageParcoursDescendance$ qui comprend les champs suivants :

type : le type du message ($CollectRequest$ ou $NotificationFinParcours$) ;

cid : identifiant du pair de *FreeCore* responsable de l'initiateur de la requête ;

stack : pile de descendants, ainsi que leurs descendances et frères de droite à visiter ;

repCont : conteneur de réponses issues des visites.

Une fois la requête correctement initialisée, la procédure détermine le serveur responsable du nœud racine et lui envoie la requête.

À la réception d'un message msg de type $MessageParcoursDescendance$ avec le champ $msg.type$ initialisé à $CollectRequest$, le système sous-jacent invoque la procédure *onReceiveCollectRequest()*. Cette procédure exécute quatre actions :

1. *getVisitedNode()* implémente la stratégie de parcours, c'est-à-dire le parcours en profondeur dans l'ordre préfixe. Premièrement, on retire le nœud au sommet de la pile de nœuds *msg.stack*, c'est le nœud courant à traiter. Ensuite on met à jour la pile de parcours en y rajoutant les 2 fils les plus à gauche du nœud qui était au sommet et ce, en commençant par le plus à droite des deux.
2. *traiterRequete()* réalise le traitement permettant de collecter les informations recherchées concernant le nœud visité. Le traitement de base, spécifié par l'Algorithme 4.7 consiste juste à l'ajout du nœud visité dans le conteneur de réponses.
3. *prochainDestinataire()* détermine le prochain serveur à qui on relaie la suite de la collecte. Si la pile *msg.stack* n'est pas vide, le serveur en charge du nœud au sommet de la pile actuelle est le prochain destinataire. Si la pile est vide, le prochain destinataire est le client qui a initialisé la collecte. Dans ce dernier cas, le type du message est mis à jour afin de marquer la fin de la collecte.
4. Envoie du message au prochain serveur qui devra continuer la requête ou traiter la fin du parcours si la collecte est terminée.

À la réception d'un message de type $MessageParcoursDescendance$ ayant son champ type égal à $NotificationFinCollecte$, le système exécute la méthode $onReceiveNotificationFinCollecte()$. La seule action de cette méthode est d'afficher la liste des nœuds visités.

Appliqué à la Figure 4.3, le parcours de cet arbre collecte les identifiants des 32 nœuds et les affiche à la fin du parcours dans l'ordre des visites :
10010001, 11010001, 11110001, 11111001, 11111101, 11111111, 11111011, 11110101, 11110111, 11110011, 11011001, 11011101, 11011111, 11011011, 11010101, 11010111, 11010011, 10110001, 10111001, 10111101, 10111111, 10111011, 10110101, 10110111, 10110011, 10011001, 10011101, 10011111, 10011011, 10010101, 10010111, 10010011

4.4.2.3 Collecte distribuée de sur-ensembles d'une requête

La procédure de collecte est initialisée par un nœud de *FragIndex* qui a reçu la demande de recherche émise par un nœud client de *FreeCore*. Cette procédure démarre au terme de la première étape de traitement de toute requête de recherche de sur-ensembles d'un filtre de requête (étape décrite à la section 4.4.1.1). Au total, la procédure de collecte prend en entrée la liste ordonnée (par le nombre de bits 1) des fragments du filtre de la requête, ainsi que l'identifiant du pair responsable de l'utilisateur ayant initié cette requête.

Comme pour le protocole centralisé (*c.f.* section 4.4.1), la collecte distribuée des sur-ensembles d'un filtre de Bloom nécessite de visiter tous les serveurs d'index identifiés par les descendants des fragments de ce filtre et de collecter les sur-ensembles partiels stockés par chacun des serveurs visités. Ceci revient à prendre chaque fragment de la requête et à parcourir sa descendance selon l'algorithme présenté à la section 4.4.2.2 en adaptant notamment les actions effectuées par les procédures $traiterRequete()$ et $onReceiveNotificationFinCollecte()$.

Concrètement, le point de départ de la collecte distribuée est l'appel à la procédure $distCollectionOfSupFrags()$ de l'Algorithme 4.8 en lui passant la liste ordonnée des fragments du filtre dont on cherche les sur-ensembles, ainsi que l'adresse du pair *FreeCore* à qui retourner la réponse de la recherche. Cette procédure initialise un message de type $CollectSupFragsRequest$, une structure qui étend $RequeteParcoursDescendance$ en ajoutant trois champs :

frags : liste ordonnée (en fonction du nombre de bit 1) de l'ensemble de fragments du filtre de la requête ;

step : le fragment dont on explore la descendance ;

vfrags : conteneur destiné à maintenir la liste de fragments déjà explorés ;

Une fois le message de collecte correctement initialisé, il est envoyé au premier serveur d'index susceptible de stocker des sur-ensembles partiels (*c.f* Algorithme 4.8).

Algorithme 4.8 Collecte distribuée de sur-ensembles partiels

Input: $lfrags$: liste ordonnée de fragments de la requête
Input: $clientId$: Id du pair FreeCore responsable de l'utilisateur initiateur de la requête

```
1:  function distCollectionOfSupFrags(lfrags, clientId)
2:  // Initialisation de la procédure de collecte répartie de sur-ensemble
3:      RequeteCollecteFrags msg ;
4:      msg.type ← RequeteParcours ;
5:      msg.repCont ← createFragConteneur() :
6:      msg.cid ← clientId ;
7:      msg.frags ← lfrags ;
8:      msg.vfrags ← createEmptyList() ;
9:      msg.step ← msg.frags.pop() ;
10:     msg.stack ← createStack() ;
11:     initParcours(msg, msg.step.v) ; // Démarrage du parcours de l'arbre de la descendance
12:     serveur ← prochaineDestination(msg) ;
13:     envoyerTo(msg, serveur) ;
14: end function
15: function traiterRequete(msg, visitedNode)
16: // Collecte les fragments des FBF qui contiennent la requête et les ajoute dans msg
17:     topNode ← msg.stack.pop() ;
18:     msg.stack.push() ;
19:     entryId ← succ(topNode.pred, topNode.cplen) ;
20:     fbfs ← getSuperSetFrags((msg.frags ∪ msg.vfrags), entryId) ;
21:     for each fb ∈ fbfs do
22:         if ((msg.frags == ∅) ∨ (msg.vfrags == ∅) ∨ (fb.fid ∈ msg.repCont)) then
23:             msg.repCont.add(fb) ;
24:         end if
25:     end for
26: // Mise à jour de msg si l'étape courante est terminée, c-à-d msg.stack vide.
27:     if (msg.stack==∅) then
28:         prune(msg.repCont, msg.vfrags) ;
29:         msg.vfrags.add(msg.step) ;
30:         if (msg.frags ≠ ∅) then
31:             msg.step ← msg.frags.pop() ;
32:             initParcours(msg, msg.step.v) ;
33:         end if
34:     end if
35: end function
36: // Traitement générique à la réception d'un msg de notification de fin de collecte.
37: function onReceiveNotificationFinCollecte(msg)
38:     traiterFinCollecte(msg) ;
39: end function
```

Lors de la réception d'un message de type $CollectSupFragRequest$ avec le champ *type* contenant la valeur $CollectRequest$, le système appelle la méthode $onReceiveCollectRequest()$ de l'Algorithme 4.7. Le traitement spécifique réalisé localement par chaque serveur est défini la procédure $traiterRequete()$ de l'Algorithme 4.8. Cette fonction détermine l'ensemble de fragments stockés sur ce serveur et qui contiennent partiellement la requête, puis les ajoute dans le conteneur de réponses. De plus, si le parcours de la descendance du fragment courant est terminé, elle initialise le parcours de la descendance du fragment suivant.

À la réception d'un message de type $CollectSupFragRequest$ avec le champ *type* contenant la valeur $NotificationFinCollecte$, le système appelle la méthode $onReceiveNotificationFinCollecte()$ de l'Algorithme 4.8. Le traitement spécifique d'une notification de fin de collecte est en fait la procédure $traiterFinCollecte()$ de l'Algorithme 4.6 de l'étape 3 de l'algorithme centralisé (voir Section 4.4.1.3).

4.5 Évaluation des performances

Nous avons développé un prototype expérimental de `FragIndex` (*c.f.* Chapitre6) en java et avons utilisé le simulateur peersim et la DHT Pastry pour évaluer les performances de notre solution.

Dans la suite, nous présentons les résultats de l'évaluation expérimentale effectuée pour démontrer la faisabilité du système `FragIndex`. L'évaluation concerne principalement deux aspects à savoir l'étude du coût d'indexation des données et l'étude du coût de la recherche d'un sur-ensemble. Nous comparons aussi les performances de la solution `FragIndex` avec celle de la solution proposée par [Joung et al. 05] que nous désignons par `Hypercube`.

4.5.1 Coût d'indexation des données

Pour évaluer le cout de l'indexation de `FragIndex`, nous étudions les deux aspects qui sont le nombre d'accès effectués sur la DHT lors de l'indexation d'une donnée et la répartition des données au niveau des serveurs d'indexation. Les résultats présentés dans cette section sont obtenus avec la configuration par défaut donnée par le tableau 4.3 ci-dessous

.

Paramètre	valeur	Signification
m	1024	longueur des filtres de Bloom
λ	128	nombre de fragments
c	8	longueur des fragments
r	8	longueur des r-bits (solution `Hypercube`)

Tableau 4.3 – Paramètres de simulation

Dans un premier temps, nous considérons les jeux données wiki 1, wiki 2, wiki 3 et wiki 4 présentés dans le tableau 3.3. Pour chaque jeux de données, nous indexons 100 000 documents

en utilisant les paramètres du tableau 4.3 et nous mesurons le nombre de messages DHT-Put envoyés pour ajouter un document dans `FragIndex`. Sur les courbes de la Figure 4.4, nous

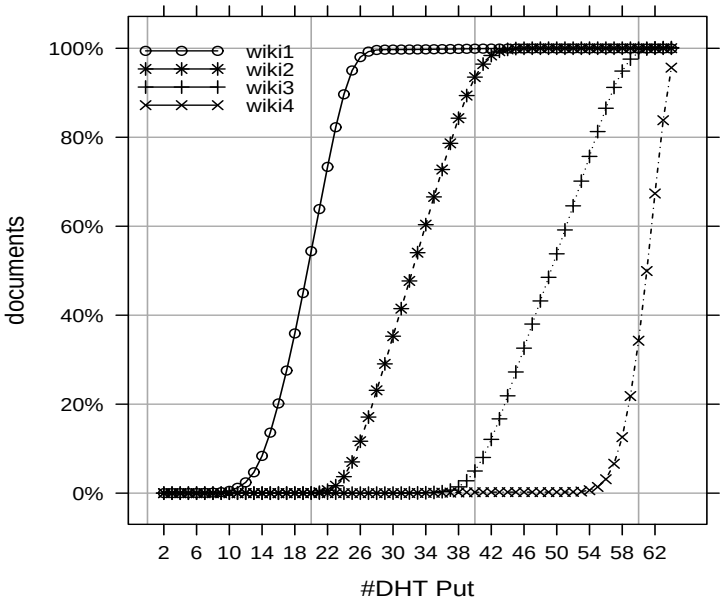


FIGURE 4.4 – cout indexation des documents

observons que le cout de l'indexation dépend du nombre de mots-clés dans les documents. Cela se traduit par le fait que le nombre de messages DHT-Put envoyés est plus important pour les jeux de données contenant des documents composés d'un grand nombre de mots-clés. Par exemple, avec le jeux de données wiki 1 contenant des documents de 1 à 10 mots-clés, 80% des documents sont indexés avec moins de 22 messages. Tandis que pour le jeu de données wiki 4 ayant des documents de 40 à 60 mots-clés, ont 80% des documents qui sont indexés avec plus de 58 messages DHT-Put. En conclusion, nous disons que pour optimiser le nombre d'accès DHT lors de l'indexation, il est nécessaire d'avoir des filtres de Bloom creux. Comparée à la solution `Hypercube` qui utilise un seul message DHT-Put pour indexer un document, la solution `FragIndex` est plus couteuse car elle utilise plusieurs clés d'indexation contrairement à la solution `Hypercube` qui fabrique une seule clé d'indexation.

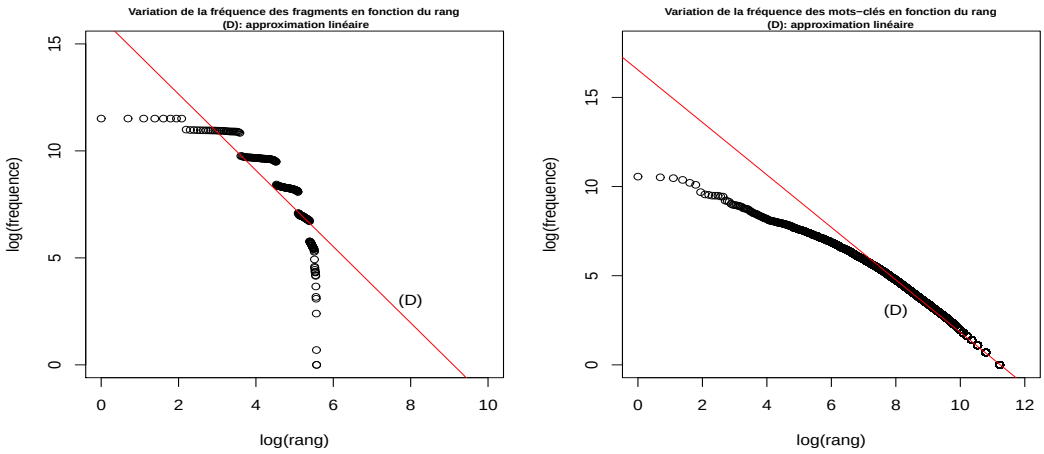
En deuxième lieu, nous nous intéressons à la répartition de la charge au niveau des serveurs d'index. Pour cela, nous étudions d'abord, la popularité des fragments pour voir si la fragmentation permet d'obtenir des fragments plus fréquents que les autres.

Ainsi, nous indexons 100 000 documents du jeu de donnée wiki 4 en se servant des paramètre du tableau 4.3 et nous déterminons la fréquence d'apparition des fragments (voir Figure 4.5a). Parallèlement, nous calculons les fréquences des mots-clés dans les documents

utilisés pour construire les filtres de Bloom indexés puis nous représentons sur la Figure 4.5b la variation des fréquences en fonction du rang dans un repère logarithmique.

La comparaison des courbes de la Figure 4.5 nous permet de vérifier si les fréquences d'apparition des fragments obéit à la loi de zipf au même titre que les fréquences des mots-clés.

La loi de Zipf stipule que le produit entre le rang d'un mot et sa fréquence d'apparition est constant, ainsi la fréquence des mots est inversement proportionnelle à leur rang. Dans ce cas, les fréquences d'apparition des fragments représentées par des nuages de points de coordonnées (log(rang) ; log(frequence)) alignés. Or, en superposant les nuages de points aux droites de régression simple, nous constatons sur la Figure 4.5a qu'il y a une nette inflexion pour les grandes et les petites valeurs de fréquence. Sur la Figure 4.5b, l'inflexion concerne seulement les grandes valeurs de fréquence et la droite de régression a une forte pente de 0,992 qui montre que la majeure partie des points sont proches de la droite. Ce résultat va dans le sens des observations connues qui stipulent que les fréquences d'apparition des mots-clés suivent la loi Zipf. Pour les fragments, la pente de la droite de régression est de 0,682, cet observation montre que les fréquences d'apparition des fragments ne varient pas inversement en fonction du rang. Ainsi, nous pouvons dire que la loi de Zipf ne s'applique pas sur la fréquence des fragments et par conséquent, aucun serveur d'index ne devrait être responsable d'un fragment populaire.



(a) Popularité des fragments

(b) Popularité des mots-clés

FIGURE 4.5 – Fréquence des mots binaires vs fréquence des mots-clés

Afin de mieux visualiser la répartition de la charge au niveau des serveurs d'index, nous étudions comparativement les systèmes `FragIndex` et `Hypercube` en indexant dans chacun d'eux 100.000 documents. Nous déterminons la charge de chaque serveur en calculant le nombre d'éléments stockés dans un serveur. Nous résumons les valeurs observées sur les tableaux 4.4 et 4.5 puis nous représentons ces valeurs sur la Figure 4.6 qui montre respectivement la distribution de la charge au niveau des systèmes `FragIndex` et

Hypercube.

Il est intéressant de noter que pour FragIndex, chaque document étant associé à un filtre de Bloom qui est découpé en λ fragments pour obtenir plusieurs clés d'indexation tandis que pour Hypercube, chaque document est associé à une seule clé d'indexation. Ainsi, le nombre de données indexées par FragIndex est largement supérieur à celui de Hypercube. Malgré cette inégalité sur la quantité de données indexés par les deux systèmes, nous voyons que les deux systèmes se comportent différemment selon la variation du nombre de mots-clés contenus dans les documents indexés. Ainsi, nous constatons que la

Jeu de données	Min.	1er Qu.	Médiane	Moyenne	3em Qu.	Max.	Données indexées
Wiki 1	1	13	37,5	5 808	607	98 150	1 161 601
Wiki 2	1	25	262	7 887	3 165	99 710	1 940 185
Wiki 3	1	370,5	2 215	13 270	13 060	100 000	3 384 886
Wiki 4	67	2 548	9 093	21 020	30 200	99 770	5 359 665

Tableau 4.4 – Distribution charge des serveurs : solution FragIndex

Jeu de données	Min.	1er Qu.	Médiane	Moyenne	3em Qu.	Max.	Données indexées
Wiki 1	1	86	224	391,9	529	2 421	100 000
Wiki 2	1	4,25	34,50	561,9	251	20 710	100 000
Wiki 3	1	1	9	1 353	68,25	68 940	100 000
Wiki 4	1	3	10	4174	197.8	94310	100 000

Tableau 4.5 – Distribution charge des serveurs : solution hypercube

solution Hypercube répartit mieux la charge quand il s'agit du jeu de données wiki 1 qui comporte des documents ayant 10 mots-clé au maximum. pour les autres jeux de données, nous voyons que la majeure partie des données sont indexées au niveau d'un petit nombre de serveur. Par exemple, avec le jeu de données wiki 4, le tableau 4.5 montre que 50% des serveurs utilisés contiennent au plus 10 données parmi les 100.000 alors que le serveur le plus chargé contient 94310 données. La Figure 4.6b permet de visualiser le déséquilibrage de décharge qui survient si les documents contiennent un grand nombre de mots-clés. À l'occurrence, pour le jeu de données wiki 4, nous voyons que tous les 100.000 document se trouvent dans moins 10% des serveurs.

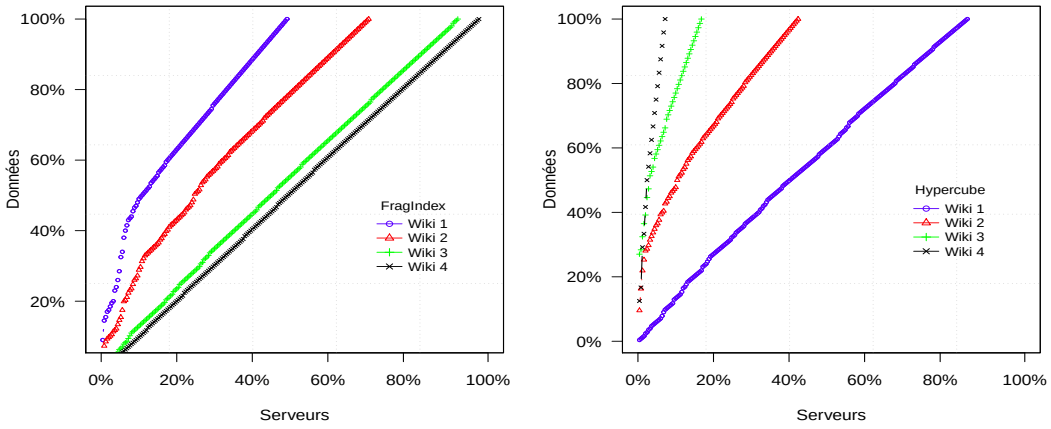
Le scénario inverse est observé pour le système FragIndex qui répartit mieux la charge quand il s'agit des jeux de données composés de documents ayant un plus grand nombre de mots-clés tels que Wiki 2, wiki 3 ou wiki 4.

La répartition de la charge dans les systèmes FragIndex et Hypercube est fortement influencée par la longueur des mots-binaires utilisés comme des clés d'indexation et du nombre de mots-clés qu'ils représentent. En effet, la solution Hypercube associe chaque ensemble de mots-clés à un mot-binaire (rbits) de longueur fixe. Cette association s'effectue à l'aide d'une fonction de hachage qui fait correspondre chaque mot à une position du rbits. Si le nombre de mots à représenter est supérieur à la longueur des rbits, il y a possibilité de saturation du

rbits qui entraine que toutes ses position soit mises à 1. Ainsi, avec des rbits de 8 bits et des documents contenant plus de 10 mots-clés, nous sommes dans le cas où tous les document sont pratiquement représentés par des rbits sont saturés c'est-à-dire composés uniquement de 8 bits 1. Ce qui explique le fait que 94310 documents parmi 100 000 sont indexés sur un seul serveur.

La solution apportée par `FragIndex` permet d'indexer des documents contenant plus de mots-clés. Ainsi, avec des filtres de Bloom de 1024 bits optimisés pour représenter des documents contenant 64 mots clés, on obtient des mots-binaires (fragments) majoritairement creux. Ainsi nous voyons sur la Figure 4.6, qu'avec des documents du jeu de données wiki 4, tous les serveurs d'indexation sont utilisés. De plus le serveur le plus chargé contient seulement moins de 2% des données soit 99 770 sur les 5 359 665 données indexées.

Puisque les filtres de Bloom utilisés sont configurés pour des documents contenant 64 mots-clés, la charge du système `FragIndex` est mal répartie si les documents contiennent moins de 10 mots-clés. En effet, avec moins de 10 mots-clés, nous avons des filtres de Bloom de 1024 bits avec beaucoup de positions à 0, et en fragmentant les filtres on se retrouve avec des fragments ayant peu de bits 1 (généralement un seul bit 1). De ce fait seuls les serveurs identifiés par des fragments ayant un bit 1 sont occupés et les autres sont inutilisés.



(a) solution FragIndex (b) Solution Hypercube

FIGURE 4.6 – Répartition de la charge des serveurs d'index

4.5.2 Cout de la recherche de sur-ensembles

Pour le cout de la recherche des sur-ensembles d'un filtre de Bloom, nous étudions deux critères de performance à savoir le trafic et la latence. Le trafic est la quantité d'information échangée entre les serveurs lors de la résolution d'une requête de sur-ensemble. La latence correspond à la durée de traitement d'une requête.

Afin de mesurer la latence, nous indexons dans un premier temps 20.000 documents du jeux de données wiki 1 sur les systèmes `FragIndex` et `Hypercube` et des documents de

wiki 4 sur FragIndex. Nous soumettons à chaque système 10 requêtes en variant le nombre de mots-clés recherchés. La Figure 4.7a montre les résultats obtenus. Nous pouvons constater qu'avec les jeux de données utilisés, le système FragIndex prend plus de temps pour satisfaire une requête. Nous voyons aussi que la solution Hypercube prend pratiquent 20 ms pour trouver les réponses quand le nombre de mots recherchés est supérieur à 8 et enfin, nous observons que plus le nombre de mots recherché est grand, moins les systèmes prennent du temps pour trouver les réponses.

L'explication est que si le nombre de mots recherché est petit, les fragments et les rbits contiennent peu de bits 1, ainsi, le nombre de descendants à explorer augmente et le temps de recherche est plus long. Pour la solution Hypercube, chaque mot recherché met un bit 1 sur le rbit, de ce fait, le nombre de serveurs à explorer peut être exprimé en fonction du nombre N de mots recherchés comme suit : $nbServ = 2^{r-N}$. Alors que pour FragIndex, le nombre de mots recherchés impacte sur le nombre d'étapes à réaliser pour résoudre la requête (le tableau 4.6 indique le nombre d'étapes de recherche à réaliser pour chaque taille de requête de notre bench).

Nombre de mots recherchés	1	2	3	4	5	6	7	8	9	10
Nombre d'étapes de recherche	6	7	9	10	12	13	13	14	14	17

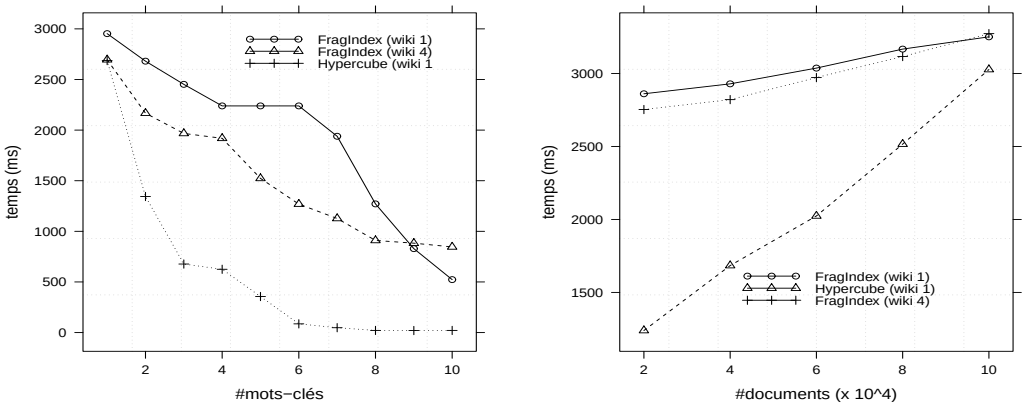
Tableau 4.6 – Nombre d'étapes de recherche par taille de requête

Malgré que l'augmentation du nombre de mots recherchés engendre plus d'étapes de recherche, le nombre de descendants visités par étape diminue quand le nombre de mots recherchés est grand et par conséquent, le temps de recherche diminue.

Avec un index contenant des documents de wiki 4, le temps de recherche est moins long qu'avec des documents de wiki 1. La raison est que dans le premier cas, le temps de recherche est allongé par la durée de traitement des données au niveau des serveurs surchargés.

Dans un deuxième temps, nous avons cherché à connaître l'impact du nombre de documents indexés sur la latence des recherches de sur-ensemble. Pour cela, nous testons les systèmes FragIndex et Hypercube en variant le nombre de documents indexés et soumettons à chaque fois 10 requêtes de 5 mots-clés puis nous calculons la latence moyenne des recherches effectuées. La courbe sur la Figure4.7b montre le temps moyen de la résolution d'une requête en fonction du nombre de documents indexés.

Nous constatons sans surprise que généralement la latence d'une recherche augmente quand les documents indexés augmentent. Par ailleurs, en comparant l'allure des courbes, nous voyons que la latence au niveau du système Hypercube croit plus rapidement que celle obtenue avec FragIndex. Cela est du au fait que Hypercube compare une requête avec toutes les descriptions des documents stockés sur un serveur alors que FragIndex effectue des tests bit-à-bit entre les fragments d'une requête et ceux des listes inversées stockés sur un serveur. En conséquence, nous disons que FragIndex passe mieux à l'échelle que la solution Hypercube.



(a) Latence en fonction du nombre de mots recherchés (b) Latence en fonction du nombre de documents

FIGURE 4.7 – Latence d'une requête FragIndex vs Hypercube

Enfin, pour mesurer le trafic, nous considérons le nombre de fragments retournés par chaque serveur visité. Nous prenons ici le ratio du nombre de fragment retourné par un serveur sur le nombre total de fragments stockés par le serveur.

Nous voyons sur la Figure 4.8 qu'au début de l'évaluation d'une requête, 0,4% des fragments sont envoyés à travers le réseau. Mais ce pourcentage diminue rapidement au fur et à mesure de l'évaluation pour atteindre 0% à la fin de la recherche. Ce résultat est attendu car l'épuration des réponses permet d'éliminer progressivement les *FBF* qui ne satisfont pas les requêtes.

4.6 Discussions

Cette section discute les choix effectués pour la conception du système `FragIndex`. Nous commençons par dégager les pistes d'améliorations de la solution et enfin, nous abordons les limites de l'approche suivie.

4.6.1 Amélioration de la solution `FragIndex`

En découpant les filtres de Bloom en k morceaux ($k \leq \lambda$), nous fixons le coût d'indexation d'un document qui devient constant quel que soit le nombre de termes contenus dans le document. En terme de communication, le coût d'indexation maximal de notre solution est de λ messages envoyés sur la *DHT*.

Mais cette option contraint `FragIndex` à limiter le nombre de serveurs d'indexation à 2^c serveurs. Pour augmenter ce nombre, nous avons implémenté une version de `FragIndex` qui utilise les fragments (couples (mots-binaire, position)) comme des entrées

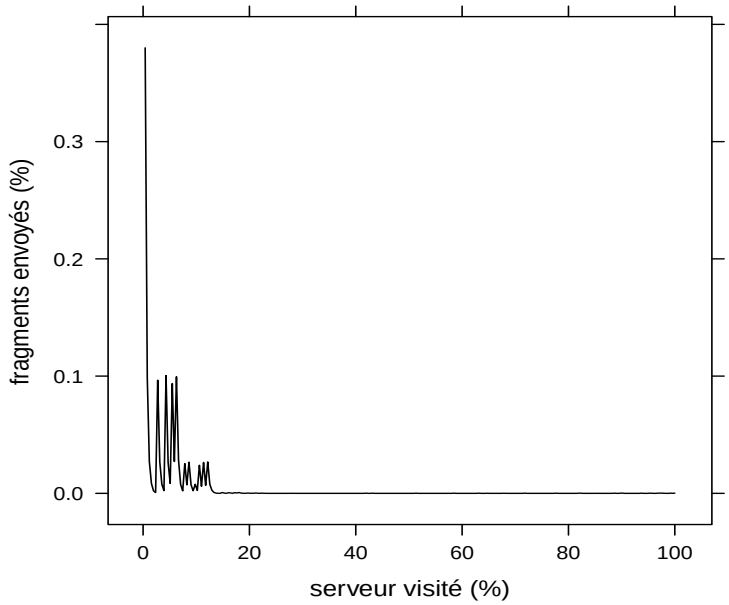


FIGURE 4.8 – Trafic généré par une requête

de l'index. Concrètement, nous convertissons les positions en des mots-binaires de $log(\lambda)$ bits que nous concaténons aux morceaux obtenus suite à la fragmentation.

Cette amélioration nous a permis de répartir les fragments sur un plus grand nombre de serveurs, mais sa faiblesse est que les coûts d'indexation et de recherche sont plus élevés. En effet, chaque fragment doit être indexé autant de fois qu'il a de positions et lors de la recherche, l'exploration de la descendance d'un fragment est effectué autant de fois que le fragment contienne de positions.

Nous avons aussi implémenté une version de `FragIndex` parallélisant la recherche de sur-ensembles. Dans cette version, il s'agit de programmer des points de rendez-vous (à l'occurrence les serveurs responsables des mots-binaires de $\mathcal{R}(Q, c)$) où différents processus de collecte de fragments renvoient les fbf trouvés.

Cette amélioration nous permet de réduire la latence des recherches des sur-ensembles mais elle est complexe à mettre en œuvre.

4.6.2 Limite de l'approche

Nous avons conçu `FragIndex` pour répondre à la problématique de recherche de sur-ensembles d'un filtre de Bloom décrivant une requête de mots-clés (*c.f.* Section 3.2.1). Comme nous l'avons vu, pour retrouver les sur-ensembles d'un filtre de longueur m ayant z bits 0,

une exploration naïve consiste à envoyer 2^z messages pour récupérer tous les sur-ensembles recherchés.

Pour réduire le nombre de messages, une solution était de réduire la longueur des filtres de Bloom indexés afin de diminuer le nombre d'éléments de la descendance. Mais cela conduirait à l'augmentation de la probabilité de faux positif, ce qui impacterait négativement sur la précision des réponses et donc sur les performances de la recherche.

Nous avons choisi d'adapter l'approche des listes inversées de mots-clés à des filtres de Bloom afin de rendre efficace la recherche de sur-ensembles. Nous avons ainsi introduit les listes inversées de mots-binaires pour pouvoir indexer des filtres de Bloom et les retrouver facilement à partir de requêtes de sur-ensembles.

Certes cette approche permet de réduire le nombres de messages mais elle est confrontée à deux problèmes que nous nous posons dans les deux paragraphes qui suivent.

4.6.2.1 Problème de calibrage des paramètres

Le premier facteur bloquant de cette approche est la difficulté de trouver un bon compromis entre les paramètres du système `FragIndex`. En effet, `FragIndex` fragmente les filtres de Bloom pour les indexer et propose une méthode de recherche de sur-ensembles consistant à retrouver et à regrouper les descendants des fragments des requêtes.

La fragmentation nécessite de fixer le paramètre de configuration c correspondant à la taille des fragments. Ainsi, en fonction de la valeur de c, on a des fragments de grande longueur (donc le nombre de fragment $\lambda = \frac{m}{c}$ est très petit) ou des fragments de petite longueur (λ est un grand nombre dans ce cas). Aucun de ces deux cas de figure ne convient à `FragIndex` dans la mesure où dans le premier cas (des fragments de grande longueur), il y a beaucoup de descendants pour un fragment du fait de la nature creuse des filtres de Bloom. De ce fait, le coût de la recherche de sur-ensembles est élevé. Pour le deuxième cas, avoir des fragments de petite longueur augmente considérablement la taille de l'index. Cela conduit à avoir le même effet que les mots populaires dans les listes inversées de mots-clés et contraint `FragIndex` à utiliser un petit nombre de serveurs pour indexer tous les filtres.

Un autre problème lié au paramétrage est la difficulté d'utiliser `FragIndex` avec des environnements de nature différente. Puisque le paramétrage est fait en fonction de la taille m des filtres de Bloom, `FragIndex` ne peut être configuré que pour des filtres de Bloom d'une même longueur. Cela pose le problème d'hétérogénéité des documents représentés par les filtres de Bloom.

Mais pour des documents homogènes (même nombre de mots-clés) `FragIndex` peut convenir si la longueur des filtres de Bloom utilisés n'est pas trop grande. Nos résultats obtenus avec de petits filtres de Bloom sont d'ailleurs très satisfaisants [Makpangou et al. 14].

4.6.2.2 Problème d'indexation

La deuxième difficulté de l'approche explorée se trouve au niveau conceptuel et reste intrinsèque à la solution. En effet, les performances de `FragIndex` dépendent de la structure de données et non des données elles-mêmes. Cela est dû au fait que pour retrouver

les sur-ensembles, la procédure de recherche doit tenir compte du filtre de Bloom de la requête. Ainsi, l'exploration des descendances des fragments est générale quelque soit les données recherchées.

4.7 Conclusions

Dans ce chapitre, nous avons présenté `FragIndex`, une solution d'indexation des filtres de Bloom permettant de retrouver les documents stockés dans une *DHT*. Nous avons ciblé la problématique liée à la répartition des données stockées sur les serveurs d'indexation et du passage à l'échelle au niveau de la solution de [Joung et al. 07]. Notre solution `FragIndex` permet de stocker les documents sur un plus grand nombre de serveurs, de répartir efficacement les données et d'indexer plus de documents sans ralentir les recherches de sur-ensembles.

La solution `FragIndex` est néanmoins confrontée à deux difficultés intrinsèques à sa conception, cela nous conduit à changer l'approche d'indexation et à explorer l'approche des arbres de préfixes.

5 | PMT– Prefix Matching Tries

Ce chapitre présente *Prefix Matching Tree (PMT)*, un index de filtres de Bloom conçu pour la recherche de sur-ensembles. La première section revient sur la problématique de la recherche par mots-clés. La deuxième section décrit la solution de l'index *PMT* ; elle abordera notamment la construction d'un arbre de préfixe et présentera les algorithmes d'indexation et de recherche qui permettent à *PMT* de retrouver efficacement les sur-ensembles d'un ensemble de mots. La troisième section présentera l'évaluation expérimentale de *PMT* et montrera ses performances sur la recherche de sur-ensembles.

5.1 Problématique de recherche par mots-clés

Étant donnée une requête sous forme d'un ensemble de mots, nous nous intéressons à la recherche des documents caractérisés par des sur-ensembles de l'ensemble des mots de la requête.

Pour cela, nous résumons chaque ensemble de mots d'un document ou d'une requête par un filtre de Bloom de longueur m choisie de sorte que le filtre de Bloom ait une faible probabilité de faux positif. Ainsi, la recherche de sur-ensembles d'une requête revient à retrouver tous les filtres de Bloom de documents qui contiennent le filtre de Bloom de la requête.

En d'autres termes, si q est un ensemble de mots et F_q le filtre de Bloom résumant cet ensemble, la recherche de sur-ensembles de q consiste à trouver tout document d résumé par un filtre de Bloom F_d tel que :

$$\forall i, 0 \leq i < m, F_q[i] = 1 \Rightarrow F_d[i] = 1.$$

Nous utilisons le filtre de Bloom caractérisant un document comme sa clé de stockage dans une table de hachage distribuée – *Distributed Hash Table (DHT)* puis nous construisons un index de filtres de Bloom capable de localiser tous les documents dont les clés de stockage sont des sur-ensembles d'un filtre de Bloom donné.
Les arbres de préfixes [Ramabhadran et al. 04, Caron et al. 06, Tang et al. 10, Cortés et al. 16] sont des structures d'index conçues pour traiter des requêtes complexes et pourraient être utilisés pour effectuer des recherches de sur-ensembles [Ngom and Makpangou 17].

Cependant, en présence de filtres de Bloom creux au niveau desquels les bits 0 sont plus nombreux que les bits 1, les données indexées (les filtres de Bloom) sont mal réparties.
Or, dans les arbres de préfixes, une mauvaise répartition des données indexées entraine le

déséquilibrage de charge sur les serveurs d'indexation et l'augmentation de la latence de recherche.

En effet, si les données sont représentées par des préfixes ayant plus de 0 que de 1, les index de préfixes se présentent sous formes d'arbre déséquilibré dans lequel certains nœuds sont très utilisés pour le traitement des requêtes et le stockage des données tandis que d'autres sont sous utilisés.

5.2 Prefix Matching Trie – PMT

PMT, est une structure d'indexation arborescente conçue pour le traitement des requêtes de sur-ensembles, c'est-à-dire, la recherche des clés de stockage des documents qui contiennent (*matchent*) le filtre de Bloom d'une requête de mots-clés.

C'est un arbre binaire composé de deux types de nœuds, les nœuds internes qui ont des fils et les nœuds feuilles qui ne possèdent pas de fils. Chaque nœud interne a exactement deux fils désignés par fils gauche et fils droit.

À un nœud donné, *PMT* associe un label construit de la manière suivante :

- Chaque nœud interne est relié à son fils gauche (respectivement fils droit) par un arc étiqueté par 0 (respectivement 1)
- le label d'un nœud est la concaténation des étiquettes des arcs qui le relient à la racine.
- Par convention, le label de la racine de l'arbre est "/".

Ainsi, à tout nœud de l'arbre *PMT*, nous assignons le label unique issu du chemin qui le relie à la racine de l'arbre. Nous référons ce label à l'identifiant de ce nœud.
Par exemple, le fils gauche (respectivement droit) de la racine est identifié par "/0" (respectivement "/1").

5.2.1 Construction et maintenance de PMT

PMT est conçu pour indexer des filtres de Bloom de même longueur m en tenant compte de la distribution asymétrique des données à indexer. Pour cela, il propose une translation (*mapping*) intelligente des données sur les identifiants des nœuds.

5.2.1.1 Mapping des filtres de Bloom sur les identifiants de nœuds

Nous associons une clé d'indexation à chaque filtre de Bloom et nous indexons des couples *(key, fb)* où ***fb*** est un filtre de Bloom de longueur m et ***key*** est la clé d'indexation de longueur D associée au filtre *fb*.

Pour associer des clés d'indexation aux filtres de Bloom, nous procédons comme suit :

1. Nous fixons deux paramètres de configuration de *PMT*, c et k tels que m est un multiple de c et k est une valeur inférieure à c.
2. Nous fragmentons chaque filtre de Bloom en fragments de c bits chacun. Par exemple, si m vaut 1024 et c=8, on aura 128 fragments.

3. Nous remplaçons chaque fragment du filtre par 0 (respectivement 1) si la valeur du fragment est inférieure (respectivement supérieure ou égale) à 2^k.

Idéalement, nous aimerions une valeur de k qui permet de répartir tout ensemble de fragments de filtres de même rang en deux partitions quasi-égales. La détermination d'une telle valeur s'avère difficile parce que d'une part, on ne dispose pas à l'avance des filtres de Bloom à indexer et d'autre part les bonnes valeurs de k peuvent différer d'un ensemble de fragments à un autre. Ainsi, nous proposons deux heuristiques pour fixer le paramètre k.

Si on dispose d'un échantillon E représentatif des filtres de Bloom à indexer, on choisit un rang i ($0 \leq i < \frac{m}{c}$) quelconque puis on détermine k de sorte que l'ensemble des filtres de Bloom de E soit réparti dans deux partitions :

$$p_0 = \{fb \in E | f_i < 2^k\} et\, p_1 = \{fb \in E | f_i \geqslant 2^k\};$$

où f_i désigne le fragment de rang i du filtre de Bloom *fb*.

Dans le cas où un échantillon n'est pas disponible, on fixe une valeur statique à k par exemple une valeur comprise entre $p/2 - 2$ et $p/2 + 2$, où $p = \frac{m}{c}$.

Une fois le paramètre k déterminé, nous l'utilisons pour associer chaque filtre de Bloom à une clé d'indexation. L'Algorithme 5.1 présente en pseudo-code le calcul d'une clé d'indexation pour un filtre de Bloom fb.

Algorithme 5.1 Calcul des clés d'indexation

Input: fb : Filtre de Bloom
Output: key

```
1:  function computekey(BloomFilter fb)
2:      cursor ← 0
3:      length ← fb.length()
4:      key ← "" ;
5:      while (cursor < length) do
6:          frag ← fb.substring(cursor, cursor+p)
7:          if (frag.intValue() < 2^k) then
8:              key ← key ⊙ 0          ▷ ⊙ désigne la concaténation
9:          else
10:             key ← key ⊙ 1
11:         end if
12:         cursor ← cursor+p
13:     end while
14:     return key
15: end function
```

5.2.1.2 Insertion et suppression de filtres de Bloom

Comme dans *Prefix Hash Tree (PHT)*, *PMT* utilise seulement les nœuds feuilles pour stocker les données. De même, la capacité de stockage d'un nœud feuille est limitée à B données au maximum.

Le processus d'insertion (respectivement de suppression) d'un filtre de Bloom fb consiste d'abord à déterminer la clé d'indexation associée à fb en utilisant l'algorithme 5.1, puis à localiser la feuille chargée de stocker le couple (key, fb) et à ajouter (respectivement supprimer) le couple.

Pour localiser la feuille chargée de stocker fb, *PMT* utilise une procédure de recherche *PMT-Lookup*, qui retrouve l'identifiant de la feuille qui est un préfixe de la clé associée au filtre de Bloom fb. Une fois la feuille localisée, *PMT* ajoute (ou supprime) le couple.

Lors de l'insertion d'une donnée (voir algorithme 5.2), il peut arriver qu'un nœud soit plein c'est-à-dire contient plus de B éléments, dans ce cas, la feuille pleine doit être divisée en deux nouvelles feuilles et les couples qu'elle stocke sont redistribués sur les deux nouvelles feuilles créées.

Algorithme 5.2 Insertion d'un filtre de Bloom

Input: fb : Filtre de Bloom
Input: node : nœud feuille exécutant l'opération

```
1:  function insert(fb)
2:      key ← computekey(fb)
3:      l ← node.level
4:      label ← node.label
5:      node.contents ← (key, fb)
6:      if (node.contents >= B) then
7:          if (l >= D) then
8:              return
9:          else
10:             split(node)
11:         end if
12:     end if
13: end function
```

Concrètement, soit un couple *(key,bf)* stocké sur une feuille pleine se trouvant au niveau l tel que $l < D$, ce couple est assigné au fils gauche si $key[l] = 0$, sinon on l'assigne au fils droit. Puisque la redistribution des données tient compte du bit l de la clé d'indexation, un cas particulier survient quand tous les D bits de la clé sont utilisés. L'arbre atteint alors sa profondeur maximale et ne peut plus grandir. Dans ce cas, la feuille pleine acquiert un statut spécial désigné par "*feuille terminale*" et sera autorisée à stocker plus de B données.

Concernant la suppression d'un filtre de Bloom, le scénario inverse peut se produire. Ainsi, il peut arriver que le total des données dans deux feuilles sœurs soit inférieur à B. Afin de

maintenir la cohérence de l'arbre, ces deux feuilles doivent être supprimées et leurs contenus agrégés au niveau de leur parent commun de niveau $l-1$.

Nous préconisons un mécanisme similaire à celui de LIGHT [Tang et al. 10] qui consiste à définir un seuil minimal sm sous lequel, une feuille interroge sa sœur. Si le résultat de cette interrogation montre que le contenu de la feuille interrogée est aussi inférieur à sm, une procédure de fusion est enclenchée (*c.f.* Algorithme 5.3). En pratique, pour limiter les vérifications inutiles, le seuil minimal peut être fixé à une valeur inférieure ou égale $B/2$.

La fusion consiste d'abord à déterminer l'identifiant de la sœur en inversant le dernier bit du label de la feuille et de demander à la sœur la taille de son index local (Ligne 5 à Ligne 6). Ensuite, si la taille cumulée des index locaux des deux sœurs est inférieure à B, l'algorithme détermine leur parent de niveau inférieur et lui assigne le contenu des deux feuilles. Enfin, *PMT* supprime les deux sœurs et met à jour le statut de leur parent qui devient une feuille (Lignes 7 à 16).

Algorithme 5.3 Fusion de deux feuilles sœurs

Input: $node$: nœud Feuille

```
1:  function merge(node)
2:      if (node.content < sm) then
3:          l ← node.level
4:          label ← node.label
5:          labelSoeur ← linvertLastBit(label)
6:          soeurContent ← poll(labelSoeur)
7:          if (node.content + soeurContent < B) then
8:              labelParent ← label.subString(0, l-1)
9:              soeur ← DHT-get(labelSoeur)
10:             parent ← DHT-get(labelParent)
11:             parent.content ← node.content + soeur.content
12:             update(parent)
13:             delete(node)          ▷ supprime le nœud de l'index
14:             delete(soeur)         ▷ supprime la sœur de l'index
15:             DHT-put(labelParent, parent)
16:         end if
17:     end if
18: end function
```

5.2.1.3 Localisation de la feuille responsable d'un filtre de Bloom

Soit fb le filtre de Bloom dont on souhaite localiser la feuille de *PMT* responsable et key la clé d'indexation qui lui est associée. La méthode *PMT-Lookup* retourne le label de l'unique feuille chargée du stockage du couple *(key, fb)*.

Pour rendre cette recherche efficace, *PMT* exploite le fait que les clés d'indexation sont souvent creuses (c'est-à-dire qu'elles contiennent un grand nombre de bits 0) et implante une procédure de localisation hybride qui combine la localisation linéaire et la localisation binaire.

En effet, *PMT* utilise des clés de grande taille par exemple 128 ou 256 bits. Lorsque les arbres *PMT* sont de hauteur très inférieure à la longueur des clés, une recherche binaire accède à plusieurs nœuds externes avant de trouver un nœud existant. À l'inverse, lorsque la hauteur de l'arbre augmente, une localisation linéaire visite de plus en plus de nœuds internes.

L'intuition derrière ce protocole de localisation hybride est de limiter à la fois les accès aux nœuds externes (inhérents à l'approche binaire) et aux nœuds internes (inhérents à l'approche linéaire), et ce, quel que soit la taille de l'arbre *PMT* utilisé pour stocker les données indexées.

PMT-lookup commence par faire des recherches linéaires avec des préfixes de plus en plus longs. À la différence du protocole de localisation linéaire de *PHT*, l'augmentation des longueurs des préfixes testés peut être supérieure à 1. Lorsque le dernier incrément ajouté fait désigner un nœud inexistant, on applique une recherche binaire limitée à cette portion de la clé.

Concrètement à partir d'un préfixe p_r et un suffixe s de la clé, *PMT* calcule un incrément de longueur supérieure ou égale à 1 c'est-à-dire il détermine un incrément x en sautant les bits 0 de la clé jusqu'au premier bit 1 suivant le préfixe p_r. Le nouveau préfixe à tester est obtenu en concaténant le préfixe p_r à l'incrément déterminé par la fonction `next-increment` notée $\mathcal{I}nc(s,l)$ et définie par l'équation 5.1 ci-dessous.

$$\mathcal{I}nc(s,l) = \begin{cases} z1, & si\ s = z1[0|1]*\ et\ len(z) < l \\ z_l, & si\ s = z[0|1]*\ et\ len(z) \geq l \\ 1, & sinon \end{cases} \tag{5.1}$$

où z désigne une suite de bit 0, $[0|1]*$ désigne une suite quelconque de bits, z_l désigne une suite de l bits 0 et len(z) désigne le nombre de bits 0.

La fonction $\mathcal{I}nc(s,l)$ procède de manière simple.
Si le suffixe s débute par une suite de z bits 0 suivie d'un bit 1 et d'une suite quelconque de bits, l'incrément x est la concaténation des z bits 0 et du premier bit 1 quand sa longueur ne dépasse pas l.
Si le suffixe s débute par une suite de z bits 0 de longueur supérieure ou égale à l, c'est-à-dire len(z) >= l, les l premiers bits sont retournés. Enfin, si le suffixe ne débute pas par un bit 0, la fonction retourne le bit 1.

L'algorithme *PMT-Lookup* (algorithme 5.4) exploite la fonction $\mathcal{I}nc(x,l)$ et effectue une recherche quasi-binaire qui consiste à considérer un préfixe initial et d'augmenter de manière itérative la longueur du préfixe de départ en tenant compte de la partie restante dans la clé d'indexation *key*.

Au début, *PMT-Lookup* est invoqué en fournissant la clé correspondant à la donnée recherchée et un préfixe de départ pr_i (par exemple celui de la racine) permettant d'initialiser le processus de localisation.

Soit pr_c, le préfixe courant identifiant un nœud interne, *PMT-Lookup* détermine un préfixe suivant à tester (pr_n) en concaténant le préfixe pr_c à l'incrément déterminé par la fonction $\mathcal{I}nc(x,l)$, puis vérifie si le nouveau préfixe pr_n correspond au label d'un nœud de l'arbre ou non (Lignes 5 à 7). Nous distinguons trois cas :

(i) pr_n ne correspond pas à un nœud de l'arbre. Dans ce cas, *PMT-Lookup* divise l'incrément par deux et réitère l'opération avec un préfixe plus court jusqu'à ce qu'on trouve le label d'un nœud existant (Lignes 8 à 14).

(ii) pr_n est le label d'un nœud interne, alors *PMT-Lookup* poursuit la recherche à partir de ce nœud mais en considérant uniquement le reste de la clé (Ligne 17).

(iii) le préfixe pr_n correspond au label de la feuille recherchée.

Algorithme 5.4 PMT-Lookup (pr_i, key)

```
1:  pr_c ← pr_i ; len ← key.length()
2:  reste ← key.substring(pr_c.length() -1, key.length())
3:  trouver ← false
4:  while (trouver = false) do
5:      inc ← Inc(reste, len)
6:      pr_n ← pr_c ⊙ inc
7:      node ← DHT-get (pr_n)
8:      while (node n'existe pas) do // Diminuer la moitié supérieure de l'incrément.
9:          mid ← inc.length() / 2
10:         len ← inc.length()
11:         inc ← inc.substring(0, mid)
12:         pr_n ← pr_c ⊙ inc
13:         node = DHT-get (pr_n)
14:     end while
15:     if (node est un nœud Parent) then // Mise à jour du préfixe courant déjà parcouru et du suffixe restant
        à examiner
16:         reste ← reste.substring(inc.length(), reste.length())
17:         pr_c ← p_n
18:     else
19:         trouver ← true
20:     end if
21: end while
22: return pr_n
```

Ainsi, on commence par faire des recherches linéaires avec des préfixes de plus en plus long. À la différence du protocole linear-lookup de *PHT*, l'augmentation des longueurs des préfixes testés n'est pas de 1 à chaque fois. L'incrément peut être de longueur supérieure ou égale à 1. Lorsque le dernier incrément ajouté fait désigner un nœud inexistant, on applique une recherche binaire limitée à cette portion de la clé.

5.2.2 Recherche de sur-ensembles d'un filtre de Bloom

L'objectif de *PMT* est de fournir un mécanisme efficace de localisation de sur-ensembles de filtres de Bloom. Concrètement, à partir d'une requête de filtre de Bloom q de longueur

m, *PMT* peut retrouver efficacement les filtres de Bloom qui possèdent des bits 1 aux mêmes positions que la requête, c'est-à-dire, tous les filtres de Bloom f de même longueur tels que : $\forall i, 0 \leq i < m, q[i] = 1 \Rightarrow f[i] = 1$. Nous appelons *superset search* la recherche de tous les filtres de Bloom qui contiennent un filtre donné.

Pour réduire la latence d'une *superset search*, nous adoptons un mécanisme similaire à la résolution des requêtes d'intervalle (*range query*) qui est prise en compte par les systèmes PHT [Ramabhadran et al. 04], LIGHT [Tang et al. 10], DST [Zheng et al. 06] et ECHO [Hidalgo et al. 16]. La différence principale entre la résolution d'une *range query* et une *superset search* est que dans le premier cas, la requête fixe explicitement les bornes minimale et maximale des données à retrouver tandis que dans une *superset search*, la requête ne fixe pas explicitement la borne supérieure des valeurs. Ainsi une *superset search* peut être considérée comme la recherche de tous les ensembles (filtres de Bloom) contenant un ensemble minimal fixé par le filtre de Bloom de la requête. Cela revient tout simplement à déterminer les filtres de Bloom appartenant à la descendance $\mathcal{D}(filtre\,requête)$

Grâce à la translation (*mapping*) des filtres de Bloom en des clés d'indexation, *PMT* stocke les filtres de Bloom sur des feuilles de labels partiellement ordonnés allant d'un minimum (/000...) à un maximum (/111...).

Ainsi, ayant une requête de filtre de Bloom qfb associée à une clé d'indexation qKey, une *superset search* de qfb consiste de manière intuitive à retrouver la feuille de label l_q responsable de la valeur qKey et à parcourir l'index *PMT* jusqu'à la feuille de label l_{max} responsable du descendant maximal.

Par construction, toute feuille de label l_f appartenant à une branche de l'arbre *PMT* située entre la feuille de label l_q et celle de label l_{max}, a un parent de label l_{pr} qui contient (*matche*) le préfixe de qKey de même longueur. En désignant par $len(l_{pr})$ la longueur du label l_{pr}, nous distinguons deux cas :

1. Le bit de qkey à la position $len(l_{pr}) + 1$ est "1", alors soit le label l_f de la feuille termine par "1" et que la feuille contient des couples (key, fb) dont les filtres de Bloom peuvent matcher la requête qfb soit c'est sa sœur qui termine par "1" et est considérée comme la bonne feuille à explorer. En d'autres termes, seule la fille qui se trouve à droite du parent sera explorée (voir figure 5.1).
2. Si le bit de qkey à la position $len(l_{pr}) + 1$ est "0", dans ce cas, les deux feuilles de labels $l_{pr}0$ et $l_{pr}1$ doivent être explorer car elles peuvent contenir chacune des filtres de Bloom qui matchent la requête qfb.

Donc une *superset search* revient à parcourir toutes les branches de l'arbre *PMT* situées entre les deux feuilles de label l_q et l_{max}, à examiner les contenus des feuilles dont les labels matchent des préfixes de la clé recherchée et à prendre tous les couples dont les filtres de Bloom fb appartiennent à la descendance $\mathcal{D}(qbf)$ du filtre de Bloom de la requête qfb.

Le traitement d'une *superset search* (*c.f.* algorithme 5.5) s'effectue en répétant successivement trois phases qui sont : (i) la localisation de la feuille la plus à gauche de l'arbre PMT, non encore explorée et susceptible de contenir de bonnes réponses ; (ii) la récupération des filtres de Bloom qui matchent la requête au niveau de la feuille localisée ; (iii) la

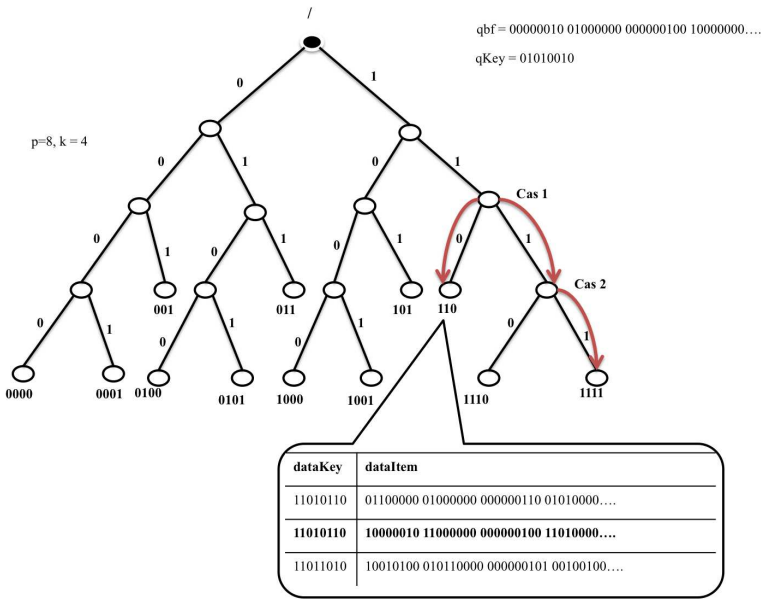


FIGURE 5.1 – Recherche superset.
Cas 1 : le $3^{ième}$ bit de "01010010" après la longueur du label "11" est "0", donc les 2 sœurs "110" et "111" sont à explorer.
Cas 2 : le $4^{ième}$ bit de "01010010" après la longueur du label "111" est "1", seule la feuille droite est à explorer

détermination de la racine de la prochaine branche de droite de *PMT* dont les feuilles peuvent contenir des données qui satisfont la requête.

Grâce à l'algorithme *PMT-Lookup*, la première phase localise à chaque itération la feuille la plus à gauche qui peut contenir des réponses (ligne 6). Par construction, *PMT-Lookup* part d'un préfixe initial et retourne l'unique feuille accessible à partir du préfixe initial responsable de la donnée recherchée.

Pour la deuxième phase, il s'agit d'un accès à la *DHT* qui peut être effectué en parallèle. Nous supposons l'existence d'une fonction *getIfSuperset* qui permet de récupérer uniquement les filtres de Bloom qui matchent le filtre de Bloom de la requête (Ligne 7). De cette manière, la quantité de données transférée dans le réseau est limitée à la quantité nécessaire pour avoir les bonnes réponses. De plus, comme la requête est résolue à partir d'un préfixe, les feuilles localisées indiquent seulement qu'elles contiennent des filtres de Bloom qui matchent le préfixe de la requête et que le reste des bits peuvent ne pas correspondre avec les filtres de Bloom indexés. La fonction `getIfSuperset` permet de réaliser le matching complet de la requête au niveau du nœud distant et de récupérer uniquement les bonnes réponses ; ce qui réduit du coup, le bruit dans les réponses.

Puisqu'il est pratiquement impossible de parcourir toutes les feuilles jusqu'à la borne maximale, le traitement d'une *superset search* peut être interrompu quand un nombre suffisant de réponses est trouvé (Ligne 10) (nous prévoyons un paramètre t qui fixe le seuil du nombre de réponses attendues).

Algorithme 5.5 Recherche des sur-ensembles d'un filtre de Bloom

```
Input: qfb : requête de filtre de Bloom
Input: : t : nombre de filtres à retourner
Output: supersets : ensemble contenant les sur-ensembles de fb
 1: function supersetSearch(qfb, t)
 2:     qkey ← computekey(qfb)
 3:     supersets ← ∅
 4:     bid ← "/"
 5:     while (la branche bid existe) do
 6:         label ← PMT-Lookup(bid, qkey) // récupération des filtre de Bloom fb qui match qfb
 7:         res ← DHT-getIfSuperset(label, qfb)
 8:         supersets ← supersets ∪ res
 9:         if (superset.size >= t) then
10:             return supersets
11:         end if
12:         bid = nextBranch(label)
13:     end while
14:     return supersets
15: end function
```

Pour la troisième phase, après chaque itération, l'algorithme continue le traitement de la *superset search* en déterminant une nouvelle branche à partir de laquelle, les deux phases de la recherche seront effectuées. Avec l'aide d'une fonction très pratique appelée `nextBranch` qui détermine soit le label du frère de droite de la feuille courante, soit le label du frère du premier ancêtre gauche de cette feuille (*c.f.* Figure 5.2), l'algorithme poursuit la recherche jusqu'à épuiser toutes les branches qui mènent vers des feuilles contenant des réponses.

La fonction *nextBranch* est définie comme suit :

$$nextBranch(l) = \begin{cases} p1, & si\ l = p0[1]* \\ null, & sinon \end{cases} \tag{5.2}$$

où p désigne un préfixe quelconque.

La Figure 5.2 schématise le fonctionnement de *nextBranch*. Nous pouvons y voir que la fonction `nextBranch` retourne à chaque fois, le nœud racine de la première branche à la droite de la feuille courante.

La procédure de *superset search* présenté ci-dessus peut être amélioré pour réduire le nombre d'accès à la *DHT* sous-jacente. En effet la fonction *nextBranch* peut retourner une feuille. Dans ce cas, la *superset search* va utiliser l'algorithme *PMT-Lookup* qui va à sont tour effectuer des accès à la *DHT* pour localiser la feuille déjà connue.

Nous pouvons réduire ce surplus d'accès *DHT* en adaptant l'algorithme de localisation pour qu'il vérifie le statut du nœud avant d'entamer la procédure normale. Nous appelons *PMT-LookAhead* l'algorithme de localisation adapté qui retourne directement le préfixe de départ

s'il correspond au label d'une feuille, auquel cas, la feuille est localisée en effectuant un seul accès *DHT*.

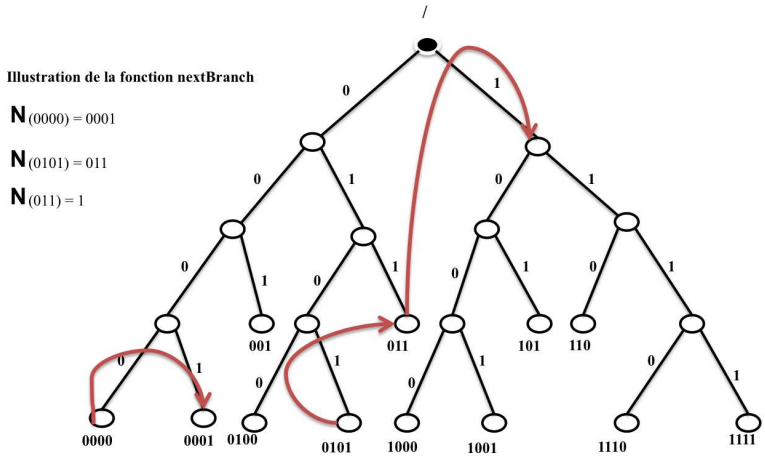


FIGURE 5.2 – Fonction next branch

5.3 Évaluation de l'index PMT

Pour évaluer *PMT*, nous utilisons le simulateur Peersim [Montresor and Jelasity 09] et le paquetage Pastry disponible sur le site de Peersim [1]. Pour stocker un document, nous appliquons la fonction de hachage SHA1 [Eastlake and Jones 01] sur son filtre de Bloom et nous stockons le couple (fb, url) sur le pair responsable de la valeur trouvée.

Dans la suite, nous allons d'abord étudier les performances structurelles de la solution *PMT*. Nous nous intéresserons principalement au taux d'occupation des feuilles stockant les données en fonction du paramètre k. Ensuite, nous mesurerons les performances de la méthode de localisation de *PMT* (*PMT-Lookup*) et la comparons à celle de l'index *PHT*. Nous terminerons par l'évaluation du protocole de recherche de sur-ensembles.

5.3.1 Impact du paramètre k

Une des principales nouveautés de *PMT* est l'introduction d'une fonction de répartition qui divise successivement une ensemble de clés de données en deux parties. *PMT* est un index conçu pour des applications de recherche dans lesquels la fréquence des bits 0 et 1 dans les clés est disproportionnée. Dans de pareilles situations, un index d'arbre binaire classique se retrouve très déséquilibré conduisant certaines feuilles à stocker plus de données que d'autres. De plus, puisque l'index *PMT* prévoit des nœuds terminaux si la longueur d'un label dépasse la longueur maximale des clés (D), un défaut d'équilibrage conduirait à plus de nœuds terminaux

1. http ://peersim.sourceforge.net/

situés dans la partie où l'arbre va pencher et les serveurs correspondants à ses nœuds seraient plus sollicités que les autres serveurs.

Le partitionnement successif de l'ensemble des données en utilisant le paramètre k permet d'équilibrer l'arbre *PMT* et de répartir les données équitablement sur les serveurs responsables des feuilles de *PMT*. Mais la difficulté réside sur la détermination du bon paramètre k qui peut varier d'un ensemble de données à un autre.

Pour étudier l'impact du paramètre k, nous avons considéré différents jeux de données du tableau 3.3 et pour chaque jeu de données, nous avons construit un dataset synthétique contenant des filtres de Bloom générés de manière aléatoire. Nous effectuons une série de test en indexant à chaque fois 300.000 filtres de Bloom pris aléatoire dans un même jeu de données.

Pour chaque ensemble de filtres de Bloom indexés, nous déterminons une valeur appelée $médiane$. Pour cela, on fragmente les filtres de Bloom et on considère la valeur entière du premier fragment de chaque filtre de Bloom. Ensuite, nous calculons la valeur qui divise l'ensemble des valeurs entières obtenues en deux parties égales.

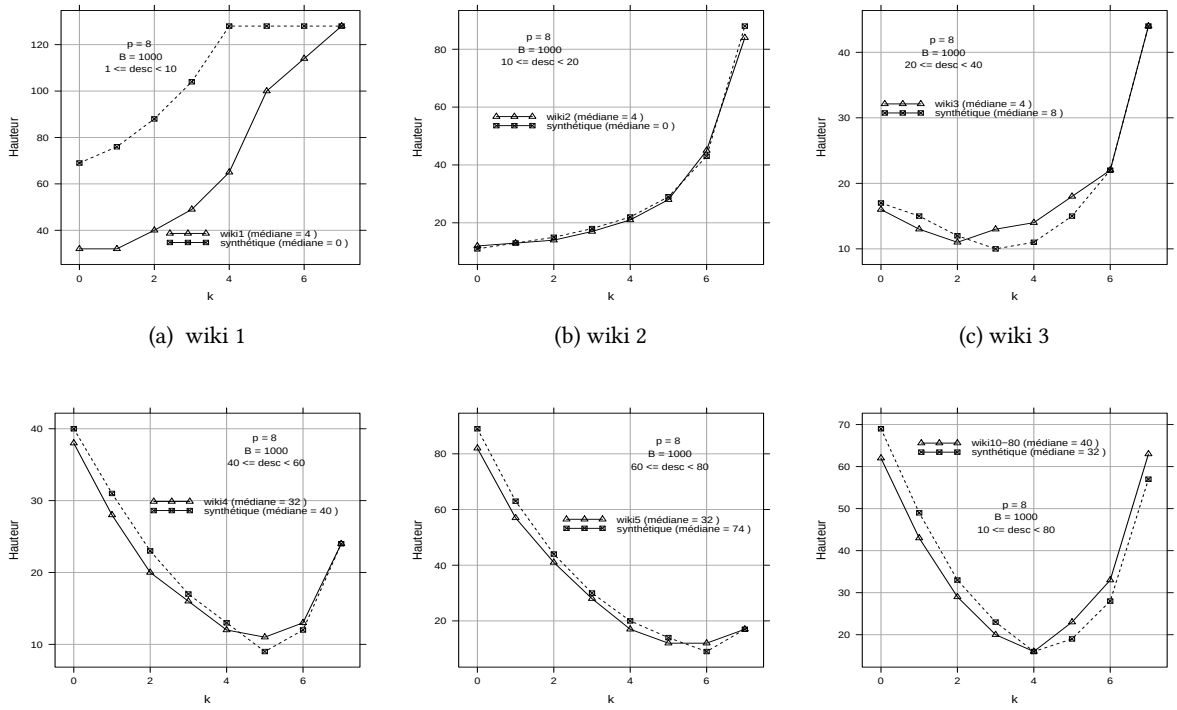
Nous mesurons deux métriques étroitement liées aux choix de k à savoir le taux d'occupation et la hauteur de l'arbre *PMT*. Nous définissons le taux d'occupation comme le ratio $\frac{nombre\ de\ clés}{B}$ correspondant au niveau de chargement d'un feuille.

S'agissant de la hauteur de l'arbre *PMT*, nous constatons sur les figures 5.3c, 5.3d, 5.3e, 5.3f que la hauteur minimale de l'arbre est obtenue quand la valeur de k est égale au logarithme de la valeur médiane ($log_2(mediane)$). En s'éloignant de cette valeur, on observe une augmentation de la hauteur de l'arbre synonyme d'un arbre déséquilibré qui penche vers un des deux cotés.

Dans le cas des jeux de données composés d'ensembles de mots de petite taille (entre 1 et 10 mots) tel que wiki 1 (Figure 5.3a), la valeur médiane ne correspond pas à la valeur de k qui donne la plus petite hauteur. Cette situation s'explique par le sur-dimensionnement du filtre de Bloom utilisé pour représenter ces ensembles. En effet, un filtre de Bloom de longueur 256 bits suffit amplement pour représenter un ensemble de 10 mots. Représentés avec un filtre de 1024 construit avec 5 fonctions de hachage, les 10 mots vont mettre au plus 50 bits à 1 et le restant des 1024 bits du filtre de Bloom restent à 0. La fragmentation par morceaux de 8 bits donnera au plus 50 fragments ayant un bit 1 et donc les 3/5 des 128 fragments sont nuls (*i.e.* ils ont une valeur entière égale à 0). Dans ce cas plus de la moitié des fragments du corpus sont nuls et aucune valeur ne permet pas diviser l'ensemble en deux parties. Néanmoins, la valeur k pourra être utilisé pour avoir un arbre de hauteur minimal donc déséquilibré.

Nous poussons cette étude en calculant l'écart entre la hauteur maximale et la hauteur minimale des feuilles de l'arbre. L'écart entre ces deux valeurs nous donne des indications sur la structure de l'arbre obtenu.

La figure 5.4 confirme les conclusions avancées ci-dessus. En effet, les valeurs de k égales aux logarithmes des médianes donnent les plus faibles écarts et nous permet de dire qu'on a des arbres plus équilibrés. Le constat fait avec le jeu de donnée wiki1 est aussi confirmé car nous voyons que la courbe correspondant à ce jeu de donnée est croissante à partir de k.



(a) wiki 1 (b) wiki 2 (c) wiki 3

(d) wiki 4 (e) wiki 5 (f) wiki 10 à 80 mots

FIGURE 5.3 – Hauteur de l'arbre obtenu avec différentes valeur de k

Le taux d'occupation est aussi une métrique dont le comportement est lié au paramètre k. Pour s'en apercevoir, nous indexons toujours 300.000 filtres de Bloom en fixant la limite de stockage d'une feuille à $B = 1000$, et nous mesurons le ratio $\frac{nombre\,de\,clés}{B}$ sur chaque feuille en fonction des valeurs de k [2].

Idéalement, une feuille doit contenir à sa création, la moitié des données se trouvant sur son parent qui a atteint la limite de B données, le taux d'occupation initial d'une feuille doit alors tourner autour de la valeur 0,5. Mais puisqu'il est difficile d'avoir un paramètre de répartition parfait, nous considérons qu'un taux d'occupation supérieur à 0,4 est une bonne répartition et qu'un taux d'occupation inférieur à 0,4 crée une charge (de plus 60% des données) sur l'une des feuilles nouvellement créées. L'allocation de plus des 60% de données à une feuille lors de sa création et l'insertion de nouvelles données accélèrent le remplissage de la feuille et peuvent engendrer la création des nouvelles feuilles en cascade. L'arbre se retrouve alors penché vers le coté du nœud surchargé laissant les feuilles opposées sous occupées.

Nous voyons ce comportement avec les jeux de données dont les filtres de Bloom sont sur-dimensionnés et très creux. Sur les figures 5.5a et 5.5b, on voit qu'aucune valeur de k ne

2. Pour des raisons de lisibilité des figures, nous représentons seulement les valeurs de k significatives

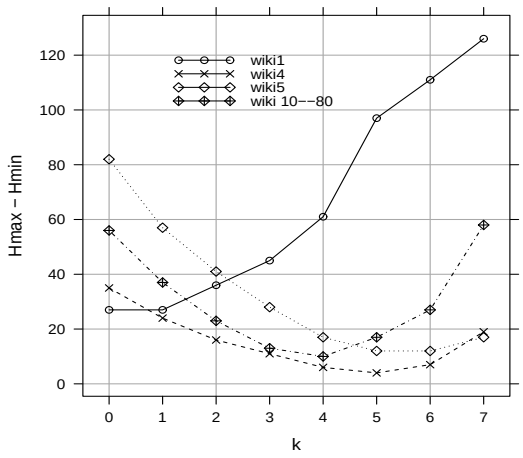
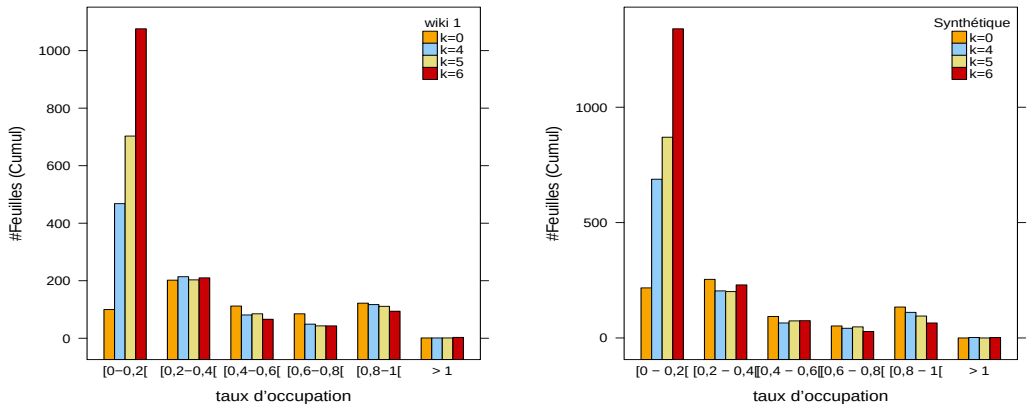


FIGURE 5.4 – Équilibrage de l'arbre PMT

permet d'avoir une bonne répartition. Par contre, dans les autres scénarios, le choix d'une valeur de k proche de la valeur médiane donne une bonne répartition des données. Ainsi, la majeur partie des feuilles ont un taux d'occupation supérieur à 0,4.

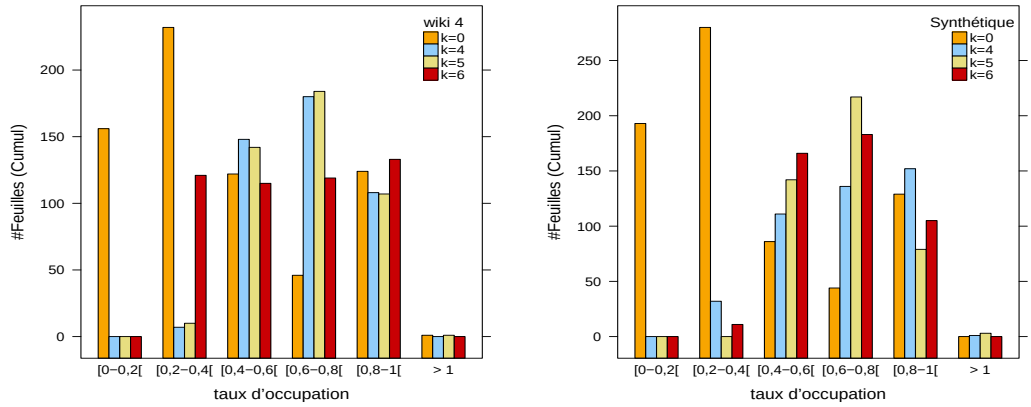


(a) wiki 1 (b) synthétique 1

FIGURE 5.5 – Taux de répartition des filtres sur-dimensionnés

Avec le jeu de donnée wiki 4 par exemple, en fixant la valeur de k à 4 ou 5, on obtient plus de 95% des nœuds ayant un taux d'occupation supérieur à 0,4 (Figures 5.6a et 5.6b). Un comportement similaire est observé avec le jeu de données wiki 3 où 87, 3% des feuilles ont

une charge supérieure à 400 données soit 40% de la capacité initiale B.



(a) wiki 4

(b) synthétique 4

FIGURE 5.6 – Taux de répartition des filtres bien dimensionnés

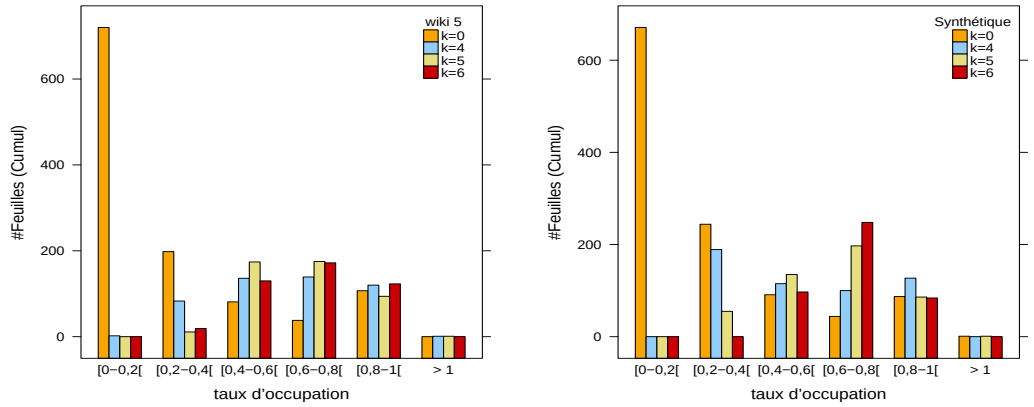
Avec des filtres de Bloom sous-dimensionnés, le même comportement est observé sur la figure 5.7 où nous voyons que la valeur de k correspondant au logarithme de la médiane est $k = 5$ pour le jeu de donnée wiki5 et $k = 6$ pour le dataset synthétique. Nous voyons qu'avec le jeu de données wiki 5, $k = 5$ permet d'avoir 97,5% des feuilles ayant un taux supérieur à 0,4 (figure 5.7a) et $k = 6$ donne 100% des feuilles ayant un taux d'occupation supérieur à 0,4 (figure 5.7b). Ce bon résultat s'explique par le fait que les filtres sont légèrement sous-dimensionnés.

Mais le sous-dimensionnement des filtres augmente la valeur médiane qui devient plus grande et se rapproche de la valeur de c. Le phénomène inverse des filtres de Bloom sur-dimensionnés peut se reproduire quand la valeur médiane est égale à p. Cela veut dire que les fragments pleins sont plus nombreux que les fragments creux et, dans ce cas, une grande quantité de données sera toujours transférée vers le coté droit de l'arbre.

Dans le cas d'un jeu de données contenant des filtres de Bloom mixtes c'est-à-dire des filtres de Bloom contenant entre 10 et 80 mots, la valeur $k = 5$ correspondant au logarithme de la médiane donne une meilleure répartition avec des données synthétiques (figure 5.8b), la valeur $k = 4$ donne une meilleure répartition pour les données de Wikipédia avec 85% de feuilles ayant plus de 400 données (figure 5.8a).

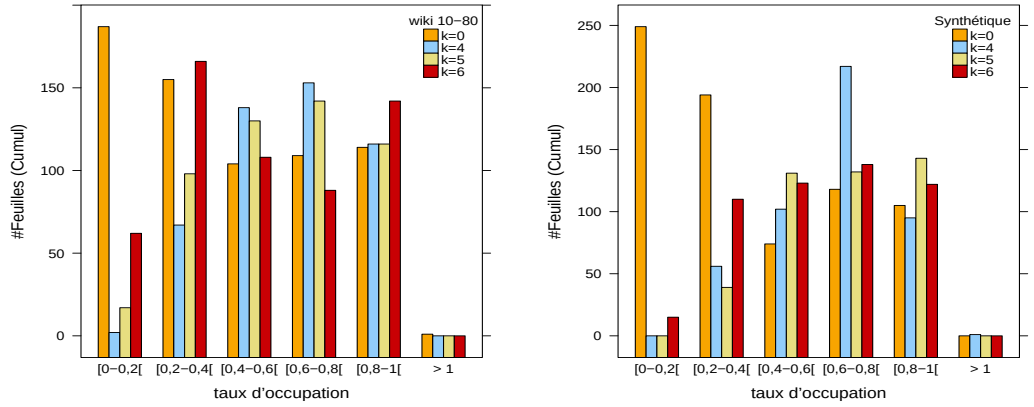
L'étude de la répartition des données confirme qu'une valeur de k proche de $log_2(mediane)$ permet d'obtenir un meilleur équilibrage et montre l'importance du paramètre de partitionnement k.

Pour terminer cette section, nous avançons que si nous disposons d'un échantillon des



(a) wiki 5 (b) synthétique 5

FIGURE 5.7 – Taux de répartition des filtres sous-dimensionnés



(a) wiki 10 à 80 mots (b) synthétique 10 à 80 mots

FIGURE 5.8 – Taux de répartition des filtres de Bloom mixtes

données à indexer ou des filtres de Bloom synthétiques représentant les caractéristiques des données à indexer, alors on peut configurer *PMT* avec un paramètre k égale à la valeur médiane qui répartit l'ensemble des valeurs de fragments en deux parties.

À défaut d'échantillon ou de filtres de Bloom synthétiques, le choix de k peut s'effectuer en fonction de la taille des fragments c. Si les filtres de Bloom ne sont pas sur-dimensionnés, toutes nos mesures montrent que la valeur de k est comprise entre $\frac{p}{2} - 2$ et $\frac{p}{2} + 2$. Nous voyons qu'avec un jeu de données comportant des documents avec un nombre de mots variés (de 10 à

80 mots) pris équitablement sur les 4 jeux de données wiki 2, wiki 3, wiki 4 et wiki 5, $k = p/2$ donne une meilleure répartition (*c.f.* figures 5.8a et 5.8b).

5.3.2 Performances de la localisation de filtres de Bloom

Dans cette section, nous comparons les algorithmes de localisation des feuilles dans les solutions *PMT* et *PHT*. Les algorithmes de localisation permettent de trouver les labels des feuilles responsables d'un préfixe donné. Ainsi, les performances de ces méthodes impactent sur le coût de maintenance de l'index. Si la localisation d'une feuille responsable du préfixe d'une donnée effectue beaucoup d'accès sur la *DHT*, alors le coût (en latence) de l'insertion (et/ou suppression) de données devient plus grand et constitue un sur-coût ajouté pour rendre le service souhaité.

La solution *PHT* propose une méthode de localisation binaire qui divise la longueur de la clé en deux et commence par rechercher la moitié inférieure. Si cette recherche retourne un nœud interne, le processus recommence avec un nouveau préfixe égale au milieu de la moitié supérieure. Si le préfixe correspond à un nœud externe, la moitié supérieure de l'intervalle est supprimée et la recherche recommence avec la moitié inférieure.

Cette méthode de localisation ne profite pas des cas où une recherche linéaire est plus efficace. Par exemple si la longueur maximale des préfixes est très longue et que l'arbre est peu profond, cet algorithme effectue plusieurs vérifications avant de trouver un n'ud de l'arbre.

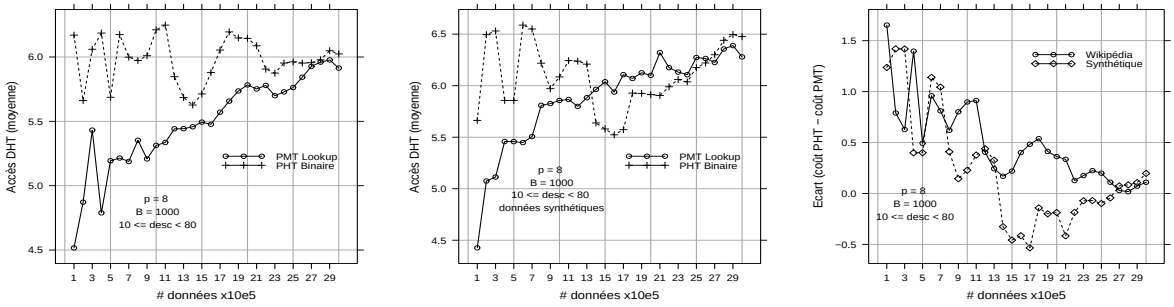
Par contre, la solution *PMT* combine la recherche binaire et la recherche linéaire avec des incréments de longueur variable. Cela lui permet de considérer seulement les bits 1 pour atteindre la feuille recherchée tout en éliminant des données qui ne satisfont pas une requête lors d'une *superset search*.

Pour comparer ces deux méthodes, nous indexons entre 100.000 et 3.000.000 de documents et nous exécutons 1000 requêtes de localisation des feuilles chargées de stocker des documents choisis de manière aléatoire. Nous calculons ensuite le nombre moyen d'accès *DHT* effectué par chaque algorithme. La figure 5.9 montre la variation de cette moyenne en fonction de la taille du jeu de données.

Sur les figures 5.9a et 5.9b, la variation du coût obtenu avec *PHT* n'est pas régulière tandis qu'avec la solution *PMT*, nous voyons que le nombre d'accès moyen augmente quand le nombre de documents indexés augmente.

De manière générale, l'algorithme PMT-lookup est plus performant et permet d'obtenir un certain gain positif. Sur la figure 5.9c, nous montrons la variation de la différence des coûts (coût PHT - coût PMT) des deux algorithmes. Pour les données de Wikipédia, nous obtenons un écart positif qui permet d'économiser jusqu'à deux accès *DHT* en moyenne.

Cependant, le gain de performance peut être réduit si la hauteur de *PMT* est très grande. Sur la figure 5.10a, représentant les mesures effectuées avec un arbre de hauteur 15, nous voyons que près de 750 requêtes effectuent au plus de 7 accès *DHT* avec l'algorithme PMT-Lookup contre 500 requêtes pour PHT-Binaire, ce qui fait une différence de 250 requêtes traitées efficacement avec PMT-Lookup. Par contre, avec un arbre de hauteur plus grande,



(a) wiki 10 à 80 mots (b) Synthétique (c) Gain/Perte

FIGURE 5.9 – Comparaison de PMT vs PHT : coût de la localisation d'une feuille

cette différence est réduite, voir par exemple la figure 5.10d où l'arbre a une hauteur de 26 où seulement 600 requêtes effectuent moins de 7 accès *DHT*.

5.3.3 Évaluation de la recherche superset

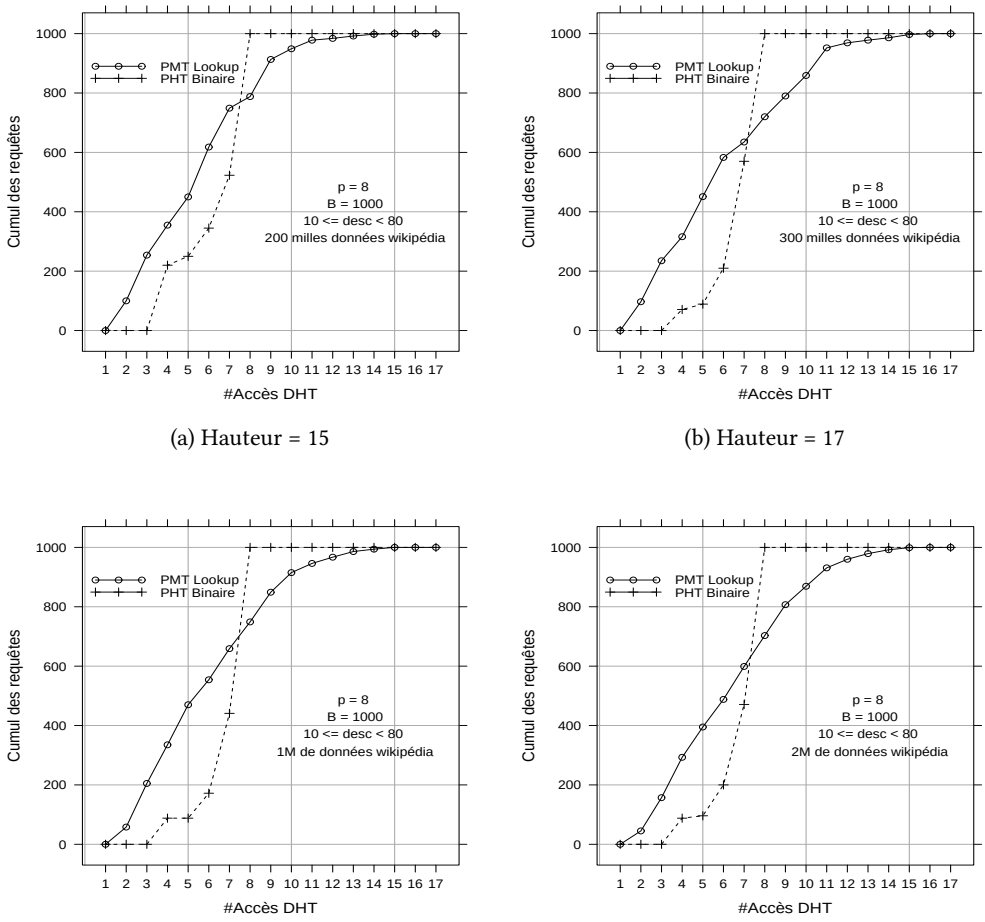
Dans cette section, nous fixons la valeur de B à 1000 et k à 5. Nous utilisons 300.000 documents issus du jeu de données wiki 4 et nous fabriquons un jeu de données synthétiques similaire contenant le même nombre de documents.

La résolution d'une *superset search* s'effectue en deux phases : la localisation d'une feuille susceptible de contenir de bonnes réponses et l'accès à la *DHT* pour récupérer les filtres de Bloom satisfaisants (Algorithme 5.5). Nous mesurons donc les performances de cet algorithme en étudiant le coût de ces deux phases.

Pour le coût de la localisation, nous prenons la latence comme étant le nombre de fois que *PMT* accède à la *DHT* sous-jacente pour localiser toutes les feuilles à explorer. Dans nos mesures nous considérons que *PMT* récupère toutes les réponses disponibles (c'est-à-dire nous ne considérons pas de seuil pour arrêter le processus de résolution d'une *superset search*). Nous testons donc le cas extrême où la *superset search* parcourt l'arbre *PMT* à partir du label minimal contenant des réponses jusqu'au label maximal.

Les figures 5.11a et 5.11b montrent le nombre d'accès *DHT* effectués par des *superset search* de requêtes contenant entre 10 à 50 mots. Nous voyons que le nombre total d'accès *DHT* effectués par une *superset search* diminue quand le nombre de mots recherché augmente.

Pour le coût de la deuxième phase, *PMT* utilise une méthode *DHT-GET-IF* qui récupère seulement les filtres de Bloom qui matchent la requête, nous considérons alors que la latence de cette deuxième phase équivaut au temps nécessaire pour accéder les feuilles localisées (*c.f.* figures 5.12a et 5.12b).



(a) Hauteur = 15

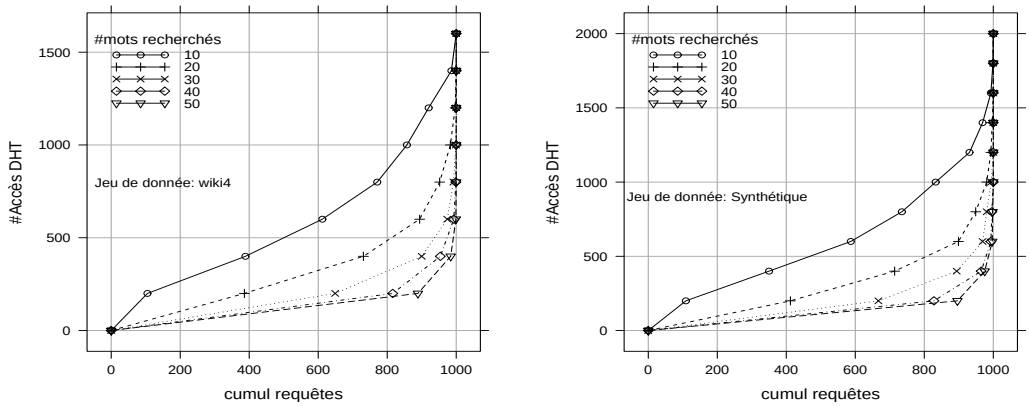
(b) Hauteur = 17

(c) Hauteur = 22

(d) Hauteur = 26

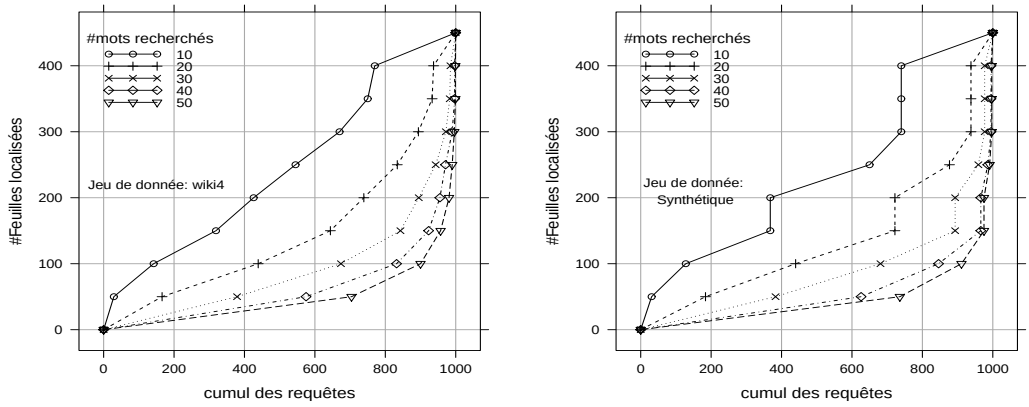
FIGURE 5.10 – Courbes cumulative du coût de PMT-Lookup et PHT-Binaire

Nos mesures montrent que la localisation des feuilles consomme plus d'accès quand les requêtes contiennent un petit nombre de mots-clés.
D'une part, ces observations s'expliquent par le fait qu'avec des requêtes creuses, les filtres de Bloom des requêtes sont moins précis et beaucoup de données indexées sont considérées comme satisfaisantes.
D'autre part, la transformation des filtres de Bloom des requêtes creuses en des clés d'indexation donne aussi des clés creuses. Les labels minimaux correspondant à ces clés creuses sont majoritairement proches du label minimal de *PMT* (c'est-à-dire le label 0000...), ainsi, les requêtes creuses explorent presque tout l'arbre. Sur la figure 5.12a, nous avons un arbre de hauteur 11 contenant 444 feuilles et nous voyons que 205 requêtes de 10 mots sur 1000 requêtes testées explorent tout l'arbre *PMT* tandis que seulement 100 requêtes de 50



(a) wiki 4 (b) Synthétique

FIGURE 5.11 – Performance de la recherche superset : Latence de la recherche



(a) wiki 4 (b) Synthétique

FIGURE 5.12 – Performance de la recherche superset : Nombre de Feuilles trouvées

mots explorent plus de 100 feuilles.

Nous avons aussi comparé les performances de la résolution d'une *superset search* en utilisant les deux méthodes de localisation *PMT-Lookup* et *PMT-LookAhead* (*c.f.* figure 5.13). Sans surprise, nous voyons que l'utilisation de *PMT-LookAhead* réduit le nombre d'accès *DHT*. L'écart entre les deux algorithmes est plus grand quand les requêtes sont creuses et s'explique par le fait qu'on explore presque tout l'arbre, et qu'un coup sur deux, la fonction *nextBranch* tombe sur une feuille.

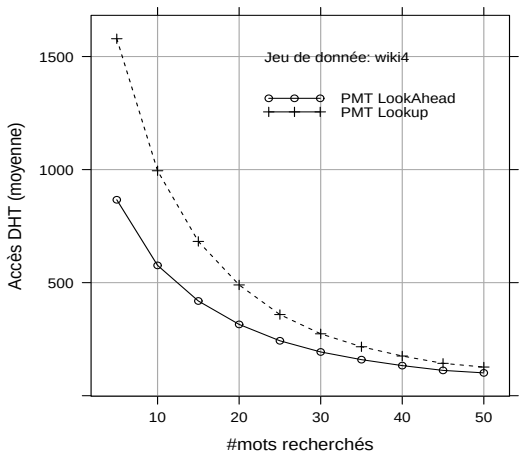


FIGURE 5.13 – PMT-Lookup vs PMT-LookAhead

5.4 Conclusion

Dans ce chapitre, nous avons présenté Prefix Matching Trie (*PMT*) : un schéma d'indexation et de recherche de sur-ensembles. *PMT* propose un support pour le traitement efficace des requêtes de sur-ensembles en utilisant un arbre de préfixe. *PMT* exploite la distribution des données et propose une fonction de répartition paramétrable permettant d'indexer les données sur les feuilles d'un arbre binaire équilibré. Cette fonction offre à *PMT* la capacité de répartir efficacement les données sur l'ensemble des feuilles, en effet, plus de 80% des serveurs utilisent au moins 40% de leur capacité de stockage. En plus, *PMT* propose une méthode de localisation efficace des feuilles chargées du stockage d'une donnée. Comparé à la méthode de location binaire de *PHT*, *PMT-Lookup* permet d'économiser jusqu'à deux accès *DHT*. Enfin, *PMT* propose une technique de recherche de sur-ensembles capable de récupérer toute les réponses disponibles en effectuant un nombre limité d'accès à la *DHT*. Nous considérons qu'avec toutes ces propriétés, *PMT* est une base solide pour la recherche de sur-ensembles sur les *over-DHT index*.

6 | Implémentation d'un prototype expérimentale de FreeCore

Pour valider nos propositions, nous avons développé un prototype expérimental en utilisant l'environnement de simulation peersim. Pour cela, nous schématisons sur la Figure 6.1 le système que nous avons mis en œuvre pour simuler FreeCore.

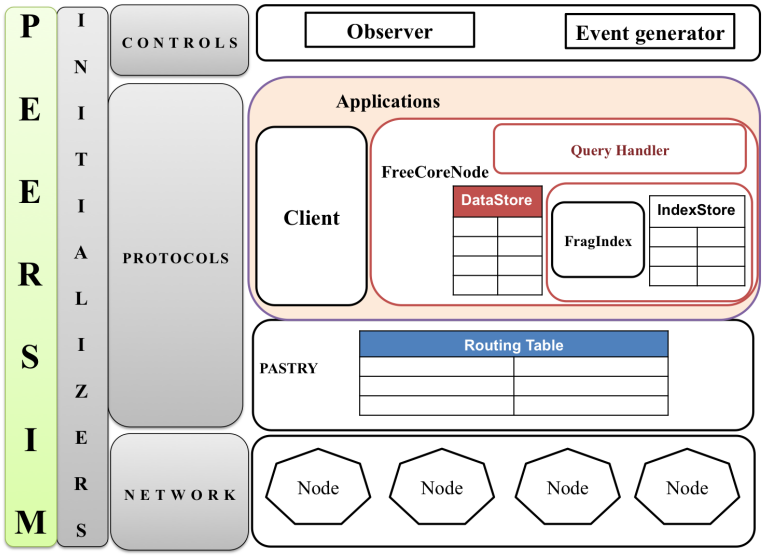


FIGURE 6.1 – Architecture distribuée de FreeCore

La suite du chapitre comprend la section 6.1 qui décrit l'environnement peersim et la section 6.2 qui détaille l'implémentation d'un prototype de FreeCore.

6.1 Le simulateur peersim

Peersim est un simulateur de réseau pair-à-pair développé à l'université de Bologne et distribué sous licence GNU/LGPL. Son code source [1] écrit en Java lui confère un caractère

1. disponible sur internet sur le site http ://peersim.sourceforge.net

modulaire et extensible. Peersim est évolutif car il permet de simuler jusqu'à plus d'un million de nœuds et il supporte le caractère dynamique des nœuds d'un système pair-pair.

Peersim dispose de deux modes de simulation : la simulation par cycles et la simulation par évènements. Dans la simulation par cycles, tous les nœuds du réseau sont sélectionnés tour à tour et chacun d'eux exécute un comportement (protocole) à chaque cycle. La simulation par évènements tient compte de la couche transport et permet à un nœud d'exécuter un comportement suite à l'arrivée d'un évènement (message) parmi un ensemble de messages défini. L'échange de messages entre les nœuds est effectué en considérant une couche transport fournie par peersim.

Afin de réaliser une simulation avec peersim, les développeurs ont mis à notre disposition un ensemble de composants qui nous permettent de créer un réseau pair-à-pair virtuel. Nous avons :

— Le réseau (Network) : ce composant représente le réseau physique qui est est composé d'un ensemble de nœuds (node). Il s'agit d'un tableau de taille fixe contenant les nœuds du système à simuler.

— Les nœuds ou pairs : ce sont des abstractions des entités physiques du réseau. Ils sont composés de piles de protocoles et possèdent chacun un identifiant unique.

— Les protocoles (Protocols) : Ils permettent de définir les comportements qu'un nœud doit exécuter. Selon le type de simulation utilisé, les comportements définis sur un protocole sont exécutés tour à tour par tous les nœuds (simulation par cycles) ou bien par les nœuds qui ont sont sollicités (simulation événementielle).

— Les contrôleurs (Controls) : ce sont des composants qui s'exécutent à des moments précis de la simulation, ils permettent de modifier les comportements des nœuds ou des liens qui existent entre les nœuds.

— Les initiateurs (Initializers) : Ils sont exécutés une seule fois au début de la simulation et permettent d'initialiser le système.

Peersim est entièrement paramétrable car tous les paramètres d'une simulation sont fournis au système à travers un fichier de configuration. C'est un fichier texte qui contient des variables définissant l'état du réseau, des nœuds et des contrôles. Dans ce fichier on y trouve par exemple la taille du réseau, le temps de simulation, les moments d'exécuter les contrôles et les protocoles à exécuter par les nœuds.

6.2 Implémentation du prototype

La figure 6.1 schématise notre prototype expérimental construit sur l'environnement peersim. Sur cette figure, nous avons trois niveaux : le niveau physique, le niveau routage et le niveau application.

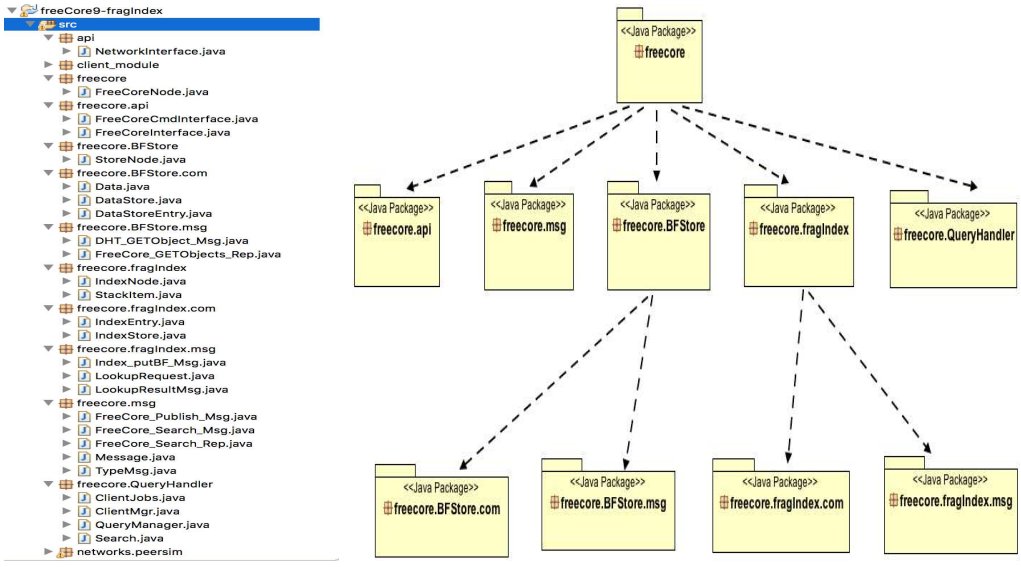
Sur le niveau physique, nous avons le composant Network de peersim qui comprend un ensemble de nœuds. L'implémentation de ce niveau est réalisé à travers un contrôleur qui est exécuté au lancement de la simulation et qui configure chaque nœud du système. La configuration d'un nœud consiste à y installer des instances des protocoles des niveaux routage et applicatif.

Sur le niveau routage, nous utilisons le package Pastry [2] comme système de routage basé sur clés (KBR). Nous implémentons par la suite un protocole fonctionnant sur le principe de simulation évènementielle qui effectue le routage et l'acheminement des messages vers le niveau application.

Sur le niveau application, nous avons construit deux modules indépendants de la structure du niveau routage. Nous avons d'une part un module freeCoreNode sur lequel on a implémenter toute les fonctionnalité de FreeCore et d'autre part, un module client qui implémente les fonctionnalités d'un utilisateur de FreeCore. La section suivante explique la mise en œuvre de l'application FreeCore.

6.3 Mise en œuvre de l'application FreeCore

Nous avons implémenté le système FreeCore en utilisant 26 classes réparties en 11 paquets (voir figure 6.2). Les paquets sont structurés de manière à hiérarchiser les composants de FreeCore. Ainsi, nous avons un grand paquet *freecore* qui contient les sous paquets *freecore.api*, *freecore.com*, *freecore.msg*, *freecore.BFStore* et *freecore.fragIndex*. Nous détaillons dans les paragraphes suivants les rôles des différents paquets mais au paravent, nous expliquons brièvement le fonctionnement d'un nœud FreeCore.



(a) capture écran du code (b) Organisation en package

FIGURE 6.2 – Structuration de la mise en œuvre

Sur la figure 6.3, nous présentons un diagramme de classe simplifié du fonctionnement d'un nœud de FreeCore. Un nœud de freeCore comprend principalement quatre composants : un

2. disponible sur internet sur http ://peersim.sourceforge.net

composant index, un composant DataStore, un composant gestion des clients et un composant gestionnaire des recherches. Il présente deux interfaces d'accès : une aux clients qui soumettent des taches de publication et de recherche et une autre pour ses propres composants.

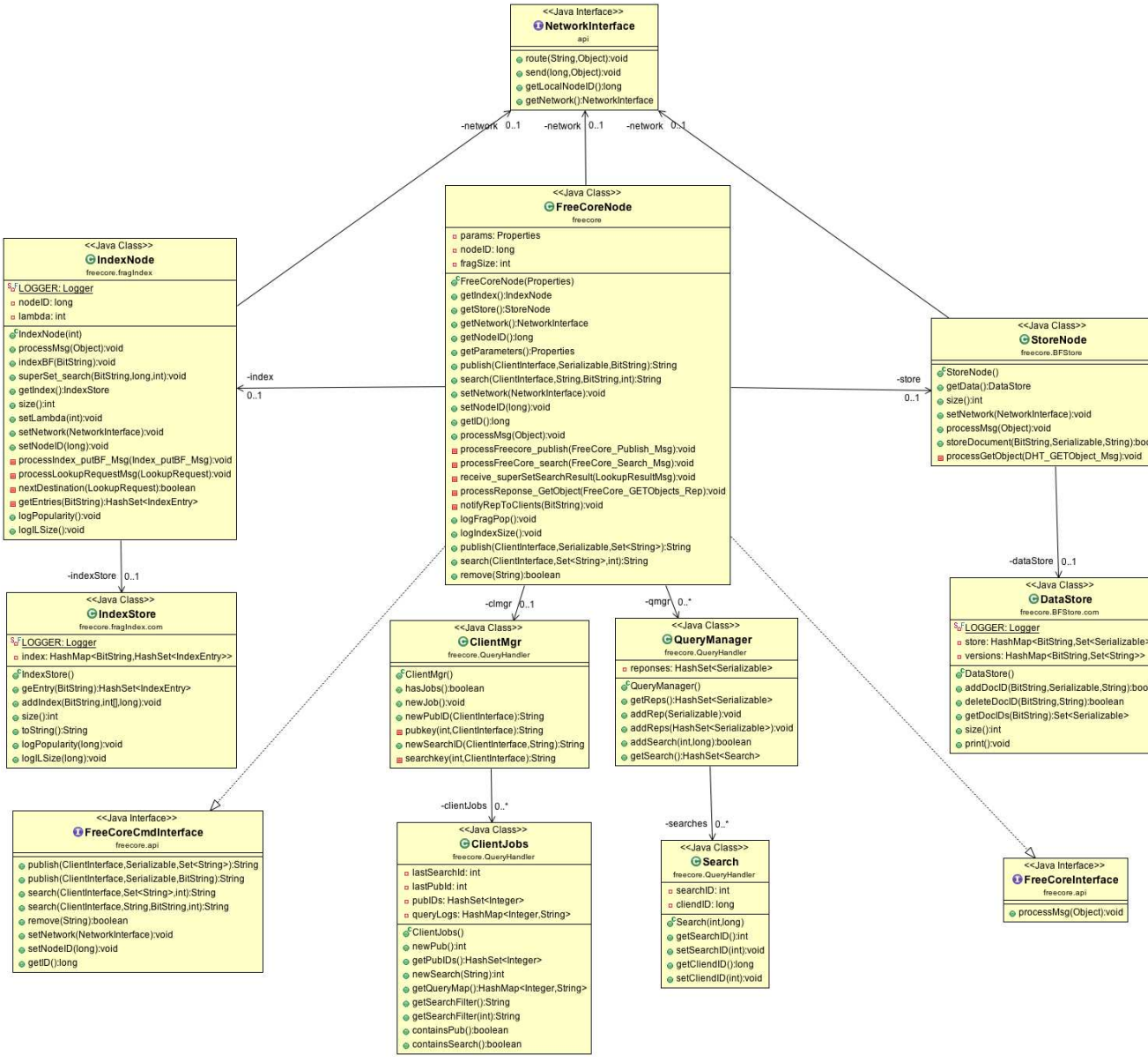


FIGURE 6.3 – Diagramme de classe de FreeCore

Le paquet freecore.api

Ce paquet contient les deux interfaces d'appel de FreeCore à savoir l'interface d'appel ascendant (upcall) et l'interface d'appel descendant (down call). Ces interfaces définissent les méthodes par lesquelles les utilisateurs passent pour accéder à FreeCore et sont implémentées par la classe *FreeCoreNode.java*.

Le paquet freecore

Outre les sous paquets qu'il contient, on trouve la classe *FreeCoreNode.java* qui implémente l'API de système FreeCore. Dans cette classe, nous avons trois méthodes importantes qui sont :

- La méthode *search* appelée par les utilisateurs de FreeCore pour réaliser une recherche par mots-clés,
- La méthode *publish* appelée par les utilisateurs de FreeCore pour publier un résumé et une URI,
- La méthode *processMsg* appelée par les composants Index et DataStore de FreeCore.

On note ici, qu'on a simplifié le fonctionnement de la classe et des méthodes *search* et *publish* qui prennent des filtres de Bloom en entrée et non des ensembles de mots-clés. Ainsi, la génération des filtres de Bloom est réalisée en amont de l'application FreeCore.

Le paquet freecore.QueryHandler

Ce paquet regroupe les classes qui permettent d'implémenter la composante QueryHandler du système. Il comprend deux autres classes *ClientMgr.java* et *ClientJobs.java* qui facilitent la gestion des requêtes soumises par les clients et deux autres classes *QueryManager.Java* et *Search.java* qui gèrent les recherches de sur-ensembles de filtres envoyées au composant index.

Le paquet freecore.msg

Dans ce paquet nous avons les classes représentant les messages destinés à FreeCore. Il s'agit des messages de publication et de recherche par mots-clés qui sont envoyés par les utilisateurs de FreeCore et des réponses que le composant index adresse à FreeCore.

Le paquet freecore.BFStore

Ce paquet regroupe les classes qui implémentent la composante DataStore de FreeCore. Il contient la classe *StoreNode.java* et deux sous paquets. La classe *StoreNode.java* représente l'API de la composante BFStore et déclenche les traitements des messages reçus par le composant. En plus de cette classe, nous avons le sous paquet *freecore.BFStore.msg* qui regroupent les classes représentant les messages destinés au composant BFStore et le sous paquet *freecore.BFStore.com* contenant la classe *DataStore.java* qui représente la base de donnée locale du nœud.

Le paquet freecore.fragIndex

Dans ce paquet nous avons les classes qui implémentent le composant Index de FreeCore. Nous présentons ici l'implémentation de l'index fragIndex réalisée à l'aide de la classe *IndexNode.java* et de deux sous paquets. La classe *IndexNode.java* représente l'API de fragIndex et permet de déclencher les traitements des messages reçus par l'index. Dans fragIndex, nous avons aussi le sous paquet *freecore.fragIndex.msg* qui contient les classes représentant les messages destinés à fragIndex et le sous paquet *freecore.fragIndex.com* contenant la classe *Index.java* qui représente l'index local du nœud.

6.4 Conclusion

Ce chapitre était consacré à la mise en œuvre de FreeCore. Dans la première section, nous y avons donné l'architecture fonctionnelle qui permet de comprendre le fonctionnement de FreeCore. La deuxième section a présenté l'architecture distribuée du protocole et a permis de distinguer les différentes couches qui compose le système FreeCore. Enfin, la dernière section est consacré à l'implémentation d'un prototype de FreeCore sur l'environnement de simulation peersim.

Conclusion et perspectives

Conclusions

Motivée par le retard de la recherche scientifique dans les pays en voie de développement, cette thèse propose un système de stockage, d'indexation et de notification appelé FreeCore. En effet, pour ces pays, FreeCore permet aux chercheurs de suivre l'actualité de la recherche au plan national et d'avoir une vision globale sur les résultats scientifiques obtenus par les unités de recherche locales. En conséquence, un chercheur peut retrouver à tout moment les productions scientifiques locales et peut aussi partager les résultats de ses recherches. La spécification de FreeCore a été l'objet de notre papier [Makpangou et al. 12] dans lequel nous avons spécifié les différents services offert par le système FreeCore.

Ainsi, au cœur du travail effectué durant cette thèse, figure en grande partie la conception d'un système de recherche d'informations dans un environnement caractérisé par des centaines de milliers d'utilisateurs disposant des capacités informatiques réduites. Ces caractéristiques expliquent le choix des systèmes distribués de type *pair-à-pair (P2P)* et principalement l'utilisation des tables de hachage distribuées – *Distributed Hash Table (DHT)* pour la confection de FreeCore.

De ce fait, nous avons abordé dans ce manuscrit la problématique de l'indexation et de la recherche des résumés de documents scientifiques et nous avons les contributions suivantes.

Dans un premier temps, nous avons proposé un système de stockage distribué des résumés de documents en se basant sur leur contenu. Pour cela nous avons utilisé les filtres de Bloom pour représenter les résumés de documents et nous avons décrit une méthode efficace d'insertion et de récupération des documents représentés par des filtres de Bloom dans un index distribué sur une *DHT*.

Le stockage des documents avec l'aide des filtres de Bloom présente un double avantage, il permet de regrouper les documents similaires dans un seul endroit afin de les retrouver plus rapidement et en même temps, il permet de retrouver les documents en faisant des recherches par mots-clés en utilisant un *Filtre de Bloom (FB)* représentant les mots recherchés. Cependant, la résolution d'une requête par mots-clés représentée par un filtre de Bloom constitue une opération difficile, il faut un mécanisme de localisation des filtres de Bloom de la descendance qui représentent des documents stockés dans la *DHT*.

C'est ainsi que nous avons proposé dans un deuxième temps, des index de filtres de Bloom distribués sur des *DHTs*. Nous en avons proposé deux que nous pouvons résumer comme suit.

Le premier système d'index proposé durant cette thèse est intitulé `fragIndex`. Il combine les principes d'indexation basée sur contenu et de listes inversées et répond à la problématique liée à la grande quantité de données stockée au niveau des index basés sur contenu tel que celui proposé par [Joung et al. 07]. En effet, l'utilisation des filtres de Bloom de grande longueur permet à notre solution `fragIndex` de stocker les documents sur un plus grand nombre de serveurs et de les indexer en utilisant moins d'espace. Nous avons présenté cette solution dans notre article [Makpangou et al. 14] où l'on a vu qu'avec la solution, les recherches peuvent être coûteuse en latence et en consommation de bande passante car il faut visiter un nombre important de serveurs avant de répondre à une requête de mots-clés. Pour cette raison, nous avons étendu cette solution afin de réduire le coût des recherches par mots-clés.

Le deuxième système d'index que nous avons proposé est *Prefix Matching Tree (PMT)* qui supporte efficacement le traitement des requêtes de sur-ensembles en utilisant un arbre de préfixes. *PMT* exploite la distribution des données et propose une fonction de répartition paramétrable permettant d'indexer les documents avec un arbre binaire équilibré. Ce qui permet à *PMT* de répartir efficacement les documents sur les serveurs d'indexation. En outre, nous avons proposé une méthode efficace de localisation des documents contenant un ensemble de mots-clés donnés. Comparé aux solutions de même catégorie, *PMT* permet d'effectuer des recherches de sur-ensembles en un moindre coût et constitue est une base solide pour la recherche de sur-ensembles sur les *over-DHT index*.

Nous avons aussi proposé *Summary Prefix Tree (SPT)* [Ngom and Makpangou 17], une autre version de *PMT* qui conserve le caractère déséquilibré des arbres de préfixes obtenu avec des données dont les distributions sont mal réparties tout en permettant d'effectuer des recherches de sur-ensembles efficaces.

Enfin, pour valider les travaux de cette thèse, nous avons implémenté les différentes solutions proposées en java et nous avons utilisé le simulateur peersim pour tester leurs performances. Nous avons présenté un prototype expérimental de FreeCore dans le chapitre 6 qui implémente le système FreeCore en utilisant l'index `fragIndex`.

Perspectives

FreeCore comprend en dehors du moteur de recherche un système de notification chargé de gérer les abonnements des utilisateurs et de notifier à ces derniers de l'apparition des nouveaux documents. La perspective qui suit naturellement ce travail, c'est la réalisation d'un module pour la gestion des notifications. Mais nous savons déjà que sa conception doit faire face au défi de passage à l'échelle dans un environnement comprenant des milliers d'utilisateurs disposant de ressources limitées (connexion intermittente et faible débit de bande passante).

Puisque le moteur de recherche de FreeCore permet de sélectionner les informations pertinentes à partir d'une liste de mots-clés, une question essentielle est de savoir comment coupler le gestionnaire de notifications avec l'index de sorte qu'un utilisateur reçoit immédiatement les documents pertinents dès leur publication. Nous devrons aussi définir la

manière d'envoyer les informations pertinentes aux utilisateurs qui, nous l'avons dit plutôt, sont mal connectés.

Par ailleurs, un des problèmes rencontrés avec l'utilisation des filtres de Bloom comme identifiant de stockage d'un document est qu'il occupe beaucoup d'espace. Cela influe sur le volume d'informations échangées pour stocker et rechercher un document. Nous avons commencé à définir un mécanisme de compression qui permettra de réduire l'espace utilisé pour stocker un document tout en gardant les propriétés classiques des filtres de Bloom. À la différence des techniques de compression des filtres de Bloom proposées par [Mitzenmacher 01] et [Putze et al. 10], notre méthode de compression tient compte des positions des bits 0 et 1 et permet de comparer deux filtres de Bloom compressés.

Enfin, dans un futur immédiat, nous envisageons de déployer un prototype du moteur de recherche de FreeCore au Sénégal afin de voir le comportement des différents protocoles proposés.

Bibliographie

[Abiteboul et al. 08] Abiteboul S., Manolescu I., Polyzotis N., Preda N., and Sun C. (2008). "xml processing in dht networks". In *24th IEEE International Conference on Data Engineering* , (pp. 606–615).

[Ahmed and Boutaba 11] Ahmed R. and Boutaba R. (2011). "a survey of distributed search techniques in large scale distributed systems". *IEEE Communications Surveys and Tutorials*, 13(2), (pp. 150–167).

[Barroso et al. 03] Barroso L. A., Dean J., and Holzle U. (2003). "web search for a planet : The google cluster architecture". *IEEE Micro*, 23(2), (pp. 22–28).

[Bially 67] Bially T. (1967). *"A class of dimension changing mapping and its application to bandwidth compression"*. PhD thesis, Polytechnic Institute of Brooklyn.

[Bloom 70] Bloom B. H. (1970). "space/time trade-offs in hash coding with allowable errors". *Commun. ACM*, 13(7), (pp. 422–426).

[Bonifati et al. 04] Bonifati A., Matrangolo U., Cuzzocrea A., and Jain M. (2004). "xpath lookup queries in p2p networks". In *6th annual ACM international workshop on Web information and data management*, WIDM '04 , (pp. 48–55). New York, NY, USA : ACM.

[Caron et al. 16] Caron E., Datta A. K., Petit F., and Tedeschi C. (2016). Self-stabilizing prefix tree based overlay networks. *International Journal of Foundations of Computer Science*, 27(05), (pp. 607–630).

[Caron et al. 06] Caron E., Desprez F., and Tedeschi C. (2006). Dynamic prefix tree for service discovery within large scale grids. In *Sixth IEEE International Conference on Peer-to-Peer Computing (P2P'06)* , (pp. 106–116).

[Chen et al. 12] Chen H., Jin H., Chen L., Liu Y., and Ni L. M. (2012). "optimizing bloom filter settings in peer-to-peer multikeyword searching". *IEEE Transactions on Knowledge and Data Engineering*, 24(4), (pp. 692–706).

[Chen et al. 08] Chen H., Jin H., Wang J., Chen L., Liu Y., and Ni L. M. (2008). "efficient multi-keyword search over p2p web". In *17th International Conference on World Wide Web*, WWW '08 , (pp. 989–998). New York, NY, USA : ACM.

[Cortés et al. 16] Cortés R., Bonnaire X., Marin O., Arantes L., and Sens P. (2016). Geotrie : A scalable architecture for location-temporal range queries over massive geotagged data sets.

In *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)* , (pp. 10–17).

[de Berg et al. 97] de Berg M., van Kreveld M., Overmars M., and Schwarzkopf O. (1997). *"Computational Geometry : Algorithms and Applications"*. Secaucus, NJ, USA : Springer-Verlag New York, Inc.

[Deerwester et al. 90] Deerwester S. C., Dumais S. T., Landauer T. K., Furnas G. W., and Harshman R. A. (1990). "indexing by latent semantic analysis". *JASIS*, 41(6), (pp. 391–407).

[Eastlake and Jones 01] Eastlake D. and Jones P. (2001). "us secure hash algorithm 1 (sha1)".

[Efthimiadis 08] Efthimiadis E. N. (2008). "how do greeks search the web ? : A query log analysis study". In *2nd ACM Workshop on Improving Non English Web Searching*, iNEWS '08 , (pp. 81–84). New York, NY, USA : ACM.

[Fan et al. 00] Fan L., Cao P., Almeida J., and Broder A. Z. (2000). "summary cache : A scalable wide-area web cache sharing protocol". *IEEE/ACM Trans. Netw.*, 8(3), (pp. 281–293).

[Fu et al. 11] Fu Y., Hu R., Chen J., Wang Z., and Tian G. (2011). "an improved lookup algorithm on over-dht paradigm based p2p network". In *International Conference on Web Information Systems and Mining*, volume Part I of *WISM'11* , (pp. 200–207). Berlin, Heidelberg : Springer-Verlag.

[Guo et al. 06] Guo D., Wu J., Chen H., and Luo X. (2006). "theory and network applications of dynamic bloom filters". In *25th IEEE International Conference on Computer Communications (INFOCOM)* , (pp. 1–12).

[Harvey et al. 03] Harvey N. J. A., Jones M. B., Saroiu S., Theimer M., and Wolman A. (2003). "skipnet : A scalable overlay network with practical locality properties". In *4th USENIX Conference on Internet Technologies and Systems*, volume 04 of *USITS'03* , (pp. 9–9). Berkeley, CA, USA : USENIX Association.

[Hidalgo et al. 11] Hidalgo N., Arantes L., Sens P., and Bonnaire X. (2011). "a tabu based cache to improve latency and load balancing on prefix trees". In *17th IEEE International Conference on Parallel and Distributed Systems (ICPADS)* , (pp. 557–564). Tainan, Taiwan.

[Hidalgo et al. 16] Hidalgo N., Arantes L., Sens P., and Bonnaire X. (2016). Echo. *J. Parallel Distrib. Comput.*, 88(C), (pp. 31–45).

[Hmedeh et al. 12] Hmedeh Z., Kourdounakis H., Christophides V., du Mouza C., Scholl M., and Travers N. (2012). "subscription indexes for web syndication systems". In *International Conference on Extending Database Technology (EDBT)* , (pp. 311–322). Berlin, Germany.

[Hofmann 99] Hofmann T. (1999). "probabilistic latent semantic indexing". In *22nd ACM SIGIR Annual International Conference on Research and Development in Information Retrieval*, SIGIR '99 , (pp. 50–57). New York, NY, USA : ACM.

[Jamard et al. 07] Jamard C., Gardarin G., and Yeh L. (2007). "indexing textual xml in p2p networks using distributed bloom filters". In *12th international conference on Database systems for advanced applications*, DASFAA'07 , (pp. 1007–1012). Berlin, Heidelberg : Springer-Verlag.

[Jan 06] Jan M. (2006). *"JUXMEM : un service de partage transparent de données pour grilles de calcul fondé sur une approche pair-à-pair"*. PhD thesis, Université de Rennes 1.

[Jin et al. 06] Jin X., Yiu W.-P. K., and Chan S.-H. G. (2006). "supporting multiple-keyword search in a hybrid structured peer-to-peer network". In *IEEE International Conference on Communications (ICC)* , (pp. 42–47). : IEEE Computer Society.

[Joung et al. 05] Joung Y.-J., Fang C.-T., and Yang L.-W. (2005). "keyword search in dht-based peer-to-peer networks". In *25th IEEE International Conference on Distributed Computing Systems*, ICDCS '05 , (pp. 339–348). Washington, DC, USA : IEEE Computer Society.

[Joung et al. 07] Joung Y.-J., Yang L.-W., and Fang C.-T. (2007). "keyword search in dht-based peer-to-peer networks". *IEEE Journal on Selected Areas in Communications*, 25(1), (pp. 46–61).

[Jun and Ahamad 06] Jun S. and Ahamad M. (2006). "feedex : collaborative exchange of news feeds". In *15th international conference on World Wide Web*, WWW '06 , (pp. 113–122). New York, NY, USA : ACM.

[Koloniari and Pitoura 04] Koloniari G. and Pitoura E. (2004). "content-based routing of path queries in peer-to-peer systems". In *9th International Conference Extending Database Technology (EDBT)*.

[Liu et al. 04] Liu L., Ryu K. D., and Lee K.-W. (2004). "supporting efficient keyword-based file search in peer-to-peer file sharing systems". In *IEEE Global Telecommunications Conference (GLOBECOM '04)*, volume 2 , (pp. 1259–1265).

[Makpangou et al. 12] Makpangou M., Ngom B., and Ndiaye S. (2012). "friticores : A rss feed monitoring and dissemination system". In *4th International IEEE EAI Conference on e-Infrastructure and e-Services for Developing Countries*, AFRICOM 12.

[Makpangou et al. 14] Makpangou M., Ngom B., and Ndiaye S. (2014). "freecore : Un substrat d'indexation des filtres de bloom fragmentés pour la recherche par mots clés". In *ComPAS'2014*.

[Mitra et al. 97] Mitra M., Buckley C., Singhal A., and Cardie C. (1997). "an analysis of statistical and syntactic phrases". In *Computer-Assisted Information Searching on Internet*, RIAO '97 , (pp. 200–214). Paris, France, France : Centre des Hautes Etudes Internationale d'Informatique Documentaire.

[Mitzenmacher 01] Mitzenmacher M. (2001). "compressed bloom filters". In *20th ACM Annual Symposium on Principles of Distributed Computing*, PODC '01 , (pp. 144–150). New York, NY, USA : ACM.

[Montresor and Jelasity 09] Montresor A. and Jelasity M. (2009). "peersim : A scalable p2p simulator simulator". In *9th International Conference on Peer-to-Peer (P2P'09)* , (pp. 99–100). Seattle, WA.

[Ngom and Makpangou 17] Ngom B. and Makpangou M. (2017). "summary prefix tree : An over dht indexing data structure for efficient superset search". In *16th IEEE International Symposium on Network Computing and Applications (NCA 2017)*.

[Ngom et al. 10] Ngom B., Sarr I., Ndiaye S., and Gueye M. (2010). "routage distribué de flux rss". In *10th African Conference on Research in Computer Science and Applied Mathematics* Yamoussoukro, Côte d'Ivoire.

[Papapetrou et al. 10] Papapetrou O., Siberski W., and Nejdl W. (2010). "pcir : Combining dhts and peer clusters for efficient full-text p2p indexing". *Comput. Netw.*, 54, (pp. 2019–2040).

[Putze et al. 10] Putze F., Sanders P., and Singler J. (2010). "cache-, hash-, and space-efficient bloom filters". *J. Exp. Algorithmics*, 14, (pp. 4 :4.4–4 :4.18).

[Ramabhadran et al. 04] Ramabhadran S., Ratnasamy S., Hellerstein J. M., and Shenker S. (2004). "brief announcement : Prefix hash tree". In *23rd ACM Symposium on Principles of Distributed Computing* St. John's, Newfoundland, Canada.

[Rao and Chen 11] Rao W. and Chen L. (2011). "a distributed full-text top-k document dissemination system in distributed hash tables". *World Wide Web*, 14, (pp. 545–572).

[Ren et al. 10] Ren Z.-J., Chen K., Shou L.-D., Chen G., Bei Y.-J., and Li X.-Y. (2010). "haps : Supporting effective and efficient full-text p2p search with peer dynamics". *J. Comput. Sci. Technol.*, 25(3), (pp. 482–498).

[Reynolds and Vahdat 03] Reynolds P. and Vahdat A. (2003). "efficient peer-to-peer keyword searching". In *ACM/IFIP/USENIX International Conference on Middleware*, Middleware '03 , (pp. 21–40). New York, NY, USA : Springer-Verlag New York, Inc.

[Salton 89] Salton G. (1989). *"Automatic Text Processing : The Transformation, Analysis, and Retrieval of Information by Computer"*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc.

[Salton et al. 75] Salton G., Wong A., and Yang C. S. (1975). "a vector space model for automatic indexing". *Commun. ACM*, 18(11), (pp. 613–620).

[Sandler et al. 05] Sandler D., Mislove A., Post A., and Druschel P. (2005). "feedtree : sharing web micronews with peer-to-peer event notification". In *4th international conference on Peer-to-Peer Systems*, IPTPS'05 , (pp. 141–151). Berlin, Heidelberg : Springer-Verlag.

[Schmidt and Parashar 04] Schmidt C. and Parashar M. (2004). "enabling flexible queries with guarantees in p2p systems". *IEEE Internet Computing*, 8(3), (pp. 19–26).

[Schütze et al. 95] Schütze H., Hull D. A., and Pedersen J. O. (1995). "a comparison of classifiers and document representations for the routing problem". In *18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '95 , (pp. 229–237). New York, NY, USA : ACM.

[Sharma and Jain 15] Sharma D. and Jain S. (2015). "context-based weighting for vector space model to evaluate the relation between concept and context in information storage and retrieval system". In *International Conference on Computer, Communication and Control (IC4)* , (pp. 1–5).

[Shi et al. 04] Shi S., Yang G., Wang D., Yu J., Qu S., and Chen M. (2004). "making peer-to-peer keyword searching feasible using multi-level partitioning". In *3rd International Conference on Peer-to-Peer Systems*, IPTPS'04 , (pp. 151–161). Berlin, Heidelberg : Springer-Verlag.

[Silverstein et al. 99] Silverstein C., Marais H., Henzinger M., and Moricz M. (1999). "analysis of a very large web search engine query log". *SIGIR Forum*, 33(1), (pp. 6–12).

[Singh and Singh 14] Singh V. and Singh A. K. (2014). "nearest keyword set search in multi-dimensional datasets". *CoRR*, abs/1409.3867.

[Skobeltsyn et al. 07] Skobeltsyn G., Luu T., Žarko I. P., Rajman M., and Aberer K. (2007). "query-driven indexing for scalable peer-to-peer text retrieval". In *2nd International Conference on Scalable Information Systems*, InfoScale '07 , (pp. 14 :1–14 :9). ICST, Brussels, Belgium, Belgium : ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[Szekeres et al. 11] Szekeres A., Baranga S. H., Dobre C., and Cristea V. (2011). "a keyword search algorithm for structured peer-to-peer networks". *Int. J. Grid Util. Comput.*, 2, (pp. 204–214).

[Tang et al. 03] Tang C., Xu Z., and Mahalingam M. (2003). "psearch : Information retrieval in structured overlays". *SIGCOMM Comput. Commun. Rev.*, 33(1), (pp. 89–94).

[Tang et al. 10] Tang Y., Zhou S., and Xu J. (2010). "light : A query-efficient yet low-maintenance indexing scheme over dhts". *IEEE Trans. on Knowl. and Data Eng.*, 22, (pp. 59–75).

[Tigelaar et al. 12] Tigelaar A. S., Hiemstra D., and Trieschnigg D. (2012). "peer-to-peer information retrieval : An overview". *ACM Trans. Inf. Syst.*, 30(2), (pp. 9 :1–9 :34).

[Wang et al. 09] Wang M.-F., Zhang D.-F., Tian X.-M., xia-an Bi, and Zeng B. (2009). "multi-keyword search over p2p based on counting bloom filter". In *2nd International Conference on Networking and Information Technology (IPCSIT)*.

[Witten et al. 99] Witten I. H., Moffat A., and Bell T. C. (1999). *"Managing Gigabytes (2nd Ed.) : Compressing and Indexing Documents and Images"*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc.

[Yang and Ho 06] Yang K.-H. and Ho J.-M. (2006). "proof : A dht-based peer-to-peer search engine". In *IEEE/WIC/ACM International Conference on Web Intelligence*, WI '06 , (pp. 702–708). Washington, DC, USA : IEEE Computer Society.

[Yu and Shoon 11] Yu X. and Shoon A. C. T. (2011). "a time/space efficient xml filtering system for mobile environment". In *12th IEEE International Conference on Mobile Data Management*, volume 01 of *MDM '11* , (pp. 184–193). Washington, DC, USA : IEEE Computer Society.

[Zheng et al. 06] Zheng C., Shen G., Li S., and Shenker S. (2006). "distributed segment tree : Support of range query and cover query over dht". In *Electronic publications of the 5th International Workshop on Peer-to-Peer Systems (IPTPS)*.

[Zhou et al. 06] Zhou Y., Chen X., and Wang C. (2006). "a self-organizing search engine for rss syndicated web contents". In *22nd International Conference on Data Engineering Workshops* , (pp.5̃2).