PDC 1 - Algorithmes et structures de données pour l'indexation de grands volumes de données textuelles

BELFODIL Aimene
BENABBOU Maha
BENDIMERAD Ahmed Anes
EL-MOUTARAJI Hassan
INSA de Lyon

4 novembre 2015

Table des matières

Introduction

Nous connaissons tous qu'est ce qu'un moteur de recherche. Nous savons l'utiliser et nous connaissons bien à quoi il sert. Mais il est aussi intéressant de savoir comment il est développé, de savoir quels sont ses composants et comment il marche.

Un composant important dans les moteurs de recherche est l'index. Dans ce projet, l'objectif est d'implémenter un index et de développer des méthodes de recherche d'information en l'utilisant.

Le rapport est organisé comme suit. Tout d'abord, nous présenterons la conception du projet (diagramme de classe) en chapitre 1. Nous traiterons ensuite dans l'ordre, la lecture des documents depuis le corpus documentaire ainsi que leurs tokenisations en 2, la gestion du vocabulaire en chapitre 3, les méthodes d'indexations en chapitre 4. Deux méthodes de compressions sont ensuite explorées en chapitre 5. De même, deux méthodes de recherche sont détaillées en chapitre 6. Nous terminons par la vérification de la loi de Zipf ainsi que l'estimation de son paramètre en chapitre 7. Nous présentons, enfin, en chapitre 8 l'évaluation de la performance et la consommation mémoire centrale des différentes méthodes.

Le code développé durant ce projet est fourni dans le lien :
`https://github.com/mahaben/PDC01---Search-Indexing`.

Chapitre 1

Conception du projet

Dans ce projet, l'objectif est de construire tout d'abord un index inversé pour un ensemble de documents textuels, et ensuite l'exploiter pour la recherche d'information dans ces documents. Un index inversé est principalement composé des éléments suivants :

- Une collection de termes que l'on appelle **dictionnaire**. Cette collection contient les termes qui apparaissent dans les documents indexés.
- L'index contient pour chaque terme du dictionnaire une **posting list**. Celle-ci est une liste qui contient les identifiants de tous les documents dans lesquels le terme apparait. On peut aussi stocker d'autres informations dans les **posting list**, comme l'occurrence du terme dans chaque document, sa position, ... etc.

On peut illustrer une structure d'un index inversé dans la Table 1.1 :

Term	**Posting List**
t_1	d_1, d_3, d_5
t_2	d_2, d_3
t_3	d_1, d_5, d_6, d_8
t_4	d_1, d_2, d_3, d_4
t_5	d_2
t_6	d_4
t_7	d_7
t_8	d_5, d_6
t_9	d_2, d_4

TABLE 1.1 – Exemple : Représentation visuelle de l'Index

Dans ce qui suit, nous allons détailler en premier lieu les processus de construction et de chargement d'index, ainsi que la recherche, en spécifiant les différentes méthodes que nous avons implémenté. Ensuite, nous allons détailler la conception de notre projet via un diagramme de classe.

1.1 Les processus d'indexation

Notre projet peut être divisé en trois processus principaux. Le premier correspond à la construction d'index inversé à partir d'un ensemble de documents, et sa sauvegarde dans un fichier. Le deuxième est le chargement de l'index à partir du fichier. Le troisième consiste à l'opération de recherche d'informations en utilisant l'index.

1.1.1 Processus de construction d'index

Ce processus est schématisé dans la figure 1.1 :

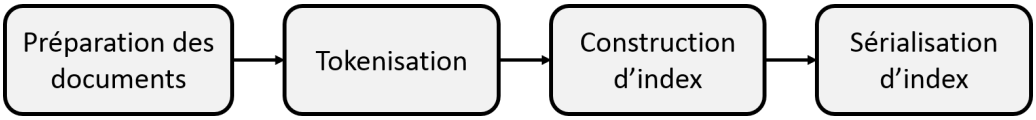

FIGURE 1.1 – Processus de construction d'index

1. **Préparation de documents :** Cette étape consiste à lire document par document à partir d'un ensemble de fichiers appartenant à un répertoire donné (corpus documentaire). Ensuite, on prépare ces documents pour les fournir à l'étape de tokenisation.
2. **Tokenisation :** Cette étape transforme chaque document en une liste de tokens dont chacun correspond à un terme du document. Nous avons implémenté deux méthode de tokenisation :
   - Dans la première, chaque terme est écrit en minuscule et ensuite retourné en tant que token. Celle-ci est implémentée complètement par nos soin.
   - La deuxième ajoute à la première deux fonctionnalités : La possibilité de configurer la liste des délimiteurs. Et la possibilité de configurer la liste des mots vides (Les *mots vides* sont des mots qui sont tellement communs qu'il est inutile de les indexer ou de les utiliser dans une recherche [5]). Cette deuxième méthode de tokenisation est implémentée à l'aide de la bibliothèque *StrTk*. Ces méthodes sont expliquées en détail dans le chapitre 2.
3. **Construction de l'index :** À ce stade, on construit l'index en utilisant les tokens récupérés à partir des documents à indexer. Nous avons développé deux méthodes de construction d'index :
   - La première construit directement l'index sur la mémoire centrale sans prendre en compte le risque de limitation de cette dernière.
   - La deuxième est basée sur la méthode **sort-based** (méthode basée mémoire secondaire).

   Ces deux méthodes sont détaillées dans le chapitre 4
4. **Sérialisation de l'index** Quand on construit l'index, on doit le sauvegarder sur le disque dur dans un fichier que l'on appelle fichier inversé. En effet, on ne peut pas toujours tenir l'index complètement sur la mémoire centrale car il est généralement volumineux. En plus, le fichier inversé permet de recharger directement l'index et éviter de le reconstruire à chaque fois qu'on ferme notre programme. Le fichier inversé contient l'ensemble des termes de l'index, les **posting list** (écrites d'une façon contigüe dans ce fichier), ainsi que d'autres informations. Nous avons développé trois manières pour la sérialisation. Elles sont différentes seulement dans la méthode de compression des **posting list** :

– La première méthode est d'écrire les **posting list** telles qu'elles sont dans le fichier (la seule contrainte est que chaque **posting list** soit contigüe).
– La deuxième méthode est de compresser les **posting list** en utilisant la méthode de compression **VByte**.
– La troisième méthode est de compresser les **posting list** en utilisant la méthode de compression **Gamma**.

Ces méthodes de compression sont détaillées dans le chapitre 5

1.1.2 Processus de chargement d'index

Ce processus correspond à l'opération du chargement de l'index à partir du fichier inversé produit lors d'une opération de construction d'index au préalable.

1.1.3 Processus de recherche d'information

Après la construction ou le chargement d'un index, on peut faire des requêtes de recherche d'information en utilisant un ou plusieurs mots (Des requêtes de type 'OU'). Cette opération consiste à trouver les **k** documents les plus pertinents par rapport à la requête donnée (**k** donné en paramètre). Pour ce faire, nous avons implémenté deux méthodes :

1. L'algorithme **Fagin**
2. L'algorithme **BM25**

Ces deux algorithmes sont détaillés dans le chapitre 6

1.2 Conception du projet : diagramme de classe

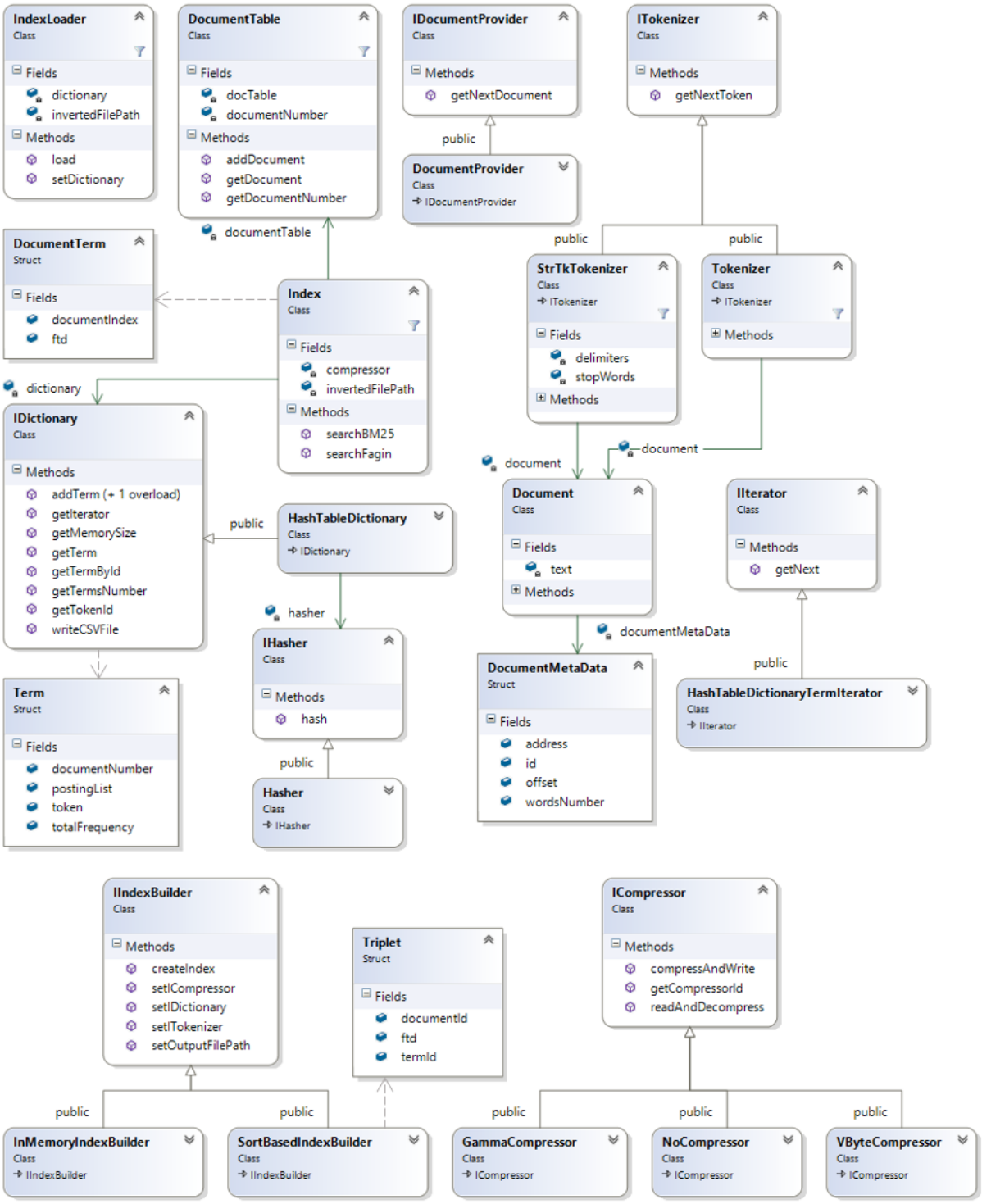

FIGURE 1.2 – Diagramme de classe du projet

Nous allons détailler, dans ce qui suit, les différents éléments qui apparaissent dans le diagramme de classe illustré dans la Figure 1.2.

1.2.1 Document

C'est une classe qui permet de contenir toutes les informations concernant un document. Elle est composée de deux éléments :

1. **documentMetadata :** c'est un attribut de type **DocumentMetaData**. Elle contient les informations suivantes (voir le chapitre 8 pour plus de détails sur la structure des documents indexés) :
   - L'identifiant du document (celui qui apparait dans la balise <DOCID>).
   - L'adresse du document, elle contient l'information présente dans la balise <DOCNO>.
   - L'offset du document dans le fichier qui le contient.
   - Le nombre de mots dans le document.
2. **text :** cet attribut est une chaine de caractères qui contient le texte du document (dans notre cas, on garde le texte qui apparait dans les balises suivantes : <HEADLINE>, <TEXT>, <GRAPHIC>).

1.2.2 IDocumentProvider

Cette interface permet de lire les documents à partir d'un ensemble de fichiers. Elle contient la méthode **getNextDocument()** qui permet de retourner le prochain document. S'il ne reste aucun document à retourner, cette méthode renvoie la valeur nulle.

Cette interface est implémentée par la classe **DocumentProvider** qui est développée spécifiquement pour lire les documents à partir de la base de données **latimes**. Elle a besoin seulement du chemin du répertoire qui contient les fichiers à indexer (ces fichiers doivent être décompressés).

1.2.3 ITokenizer

Cette interface lit les tokens (termes, mots) à partir d'un objet de type **Document**. Elle contient la méthode **getNextToken()** qui retourne le prochain token. S'il ne reste aucun token à retourner, cette méthode renvoie une chaine vide.

Cette interface est implémentée par deux classes :

1. **Tokenizer** : c'est un tokenizer simpliste qui permet de récupérer les termes après leur transformation en minuscule et suppression des balises.
2. **StrTkTokenizer** : celui-ci traite les mots-vides (la liste des mots vides étant paramétrable). Il permet aussi de spécifier la liste des délimiteurs.

1.2.4 DocumentTable

C'est une classe qui permet de sauvegarder et de gérer les méta-données des documents. Elle contient une collection de **DocumentMetaData** et quelques autres attributs secondaires. Elle est composée principalement des méthodes suivantes :

- **unsigned int addDocument(DocumentMetaData docMetaData) :** cette méthode ajoute le docMetaData à la collection, et retourne son indice dans cette collection.
- **DocumentMetaData* getDocument(unsigned int documentIndex) :** elle retourne les méta-données du document qui correspond à l'indice documentIndex.

1.2.5 IDictionary

C'est une interface qui permet d'implémenter un dictionnaire. Celui-ci est une collection de termes, ces termes sont stockés dans des structures de type **Term**. Ce dernier est composé des éléments ci-dessous :

- **token :** Le terme en chaine de caractère.
- **totalFrequency :** La nombre totale d'occurrences du terme dans le corpus.
- **documentNumber :** Le nombre de documents dans lesquels ce terme apparait.
- **postingList :** La posting list de ce terme. Celle-ci est un pointeur vers une liste d'éléments de type **DocumentTerm** dont chacun contient deux entiers : l'indice du document dans **DocumentTable**, l'occurrence du terme dans le document. Concrètement, cet attribut contient l'offset de la posting list du terme dans le fichier inversé.

Les méthodes les plus importantes de l'interface **IDictionary** sont :

- **Term* addTerm(string token) :** Elle crée un terme pour le token passé en paramètre, l'ajoute à la collection de termes, et le retourne.
- **Term* getTerm(string token) :** Elle retourne le terme qui correspond au token passé en paramètre. S'il n'existe pas dans la collection, la méthode retourne la valeur nulle.

Nous avons fait une implémentation pour l'interface **IDictionary**, c'est la classe **HashTableDictionary**. Ce dictionnaire stocke les termes dans une table de hachage. La fonction de hachage étant spécifiée via l'interface **IHasher** qui est un attribut de ce dictionnaire.

1.2.6 ICompressor

Cette interface est utilisée lors de la construction de l'index afin de compresser les posting list et les écrire sur le fichier inversé. Elle est utilisée aussi lors de la recherche d'information, afin de lire et décompresser les posting list à partir du fichier. Elle a trois implémentations :

1. **NoCompressor** : C'est une classe qui lit et écrit les posting list dans le fichier inversé sans aucune compression.
2. **VByteCompressor** : Elle utilise la méthode de compression VByte (compression bytewise).
3. **GammaCompressor** : Elle utilise la méthode de compression Gamma (compression bitwise).

1.2.7 Index

C'est la classe qui représente un index inversé dans la mémoire centrale. Elle est constituée des attributs suivants :

- **dictionary :** C'est le dictionnaire de l'index.
- **documentTable :** Elle contient les méta-données sur les documents indexés.

- **compressor :** c'est un objet de type **ICompressor** qui prend la responsabilité de lire et décompresser les posting list lors de l'opération de recherche.
- **invertedFilePath :** le chemin vers le fichier inversé de l'index.

Cette classe contient deux méthodes principales :
- **searchFagin(int topK,string query) :** Cette méthode recherche les **topK** documents les plus pertinents pour la requête **query** en utilisant l'algorithme **Fagin** (modèle vectoriel).
- **searchBM25(int topK, string query) :** Cette méthode recherche les **topK** documents les plus pertinents pour la requête **query** en utilisant l'algorithme **BM25** (modèle probabiliste).

1.2.8 IIndexBuilder

C'est une interface dont le rôle est de construire l'index. Elle contient les méthodes suivantes :
- **IIndexBuilder* setIDictionary(IDictionary *iDictionary) :** Cette méthode est utilisée pour spécifier le type de dictionnaire pour l'index à construire.
- **IIndexBuilder* setICompressor(ICompressor *iCompressor) :** Elle permet de choisir le compresseur que l'on va utiliser pour cet index.
- **IIndexBuilder* setITokenizer(int iTokenizerType) :** Elle spécifie le type de tokenizer.
- **IIndexBuilder* setOutputFilePath(string outputFilePath) :** Elle permet de préciser le chemin du fichier inversé à créer lors de la construction de l'index
- **Index* createIndex() :** Après spécification des paramètres en utilisant les méthodes expliquées précédemment, on fait appel à cette fonction qui crée l'index et le renvoie.

Cette interface est implémentée par les deux classes ci-dessous :

1. **InMemoryIndexBuilder :** Elle crée l'index directement sur la mémoire centrale sans prendre en considération le risque que la taille de l'index peut dépasser la taille de la mémoire.
2. **SortBasedIndexBuilder :** Elle crée l'index en utilisant la méthode **sort-based**. Elle se sert des structures de type **Triplet** (indice de document dans **DocumentTable**, identifiant du terme, fréquence du terme dans le document) pour effectuer sa tâche.

1.2.9 IndexLoader

Le rôle de cette classe est de charger l'index à partir d'un fichier inversé. Elle a besoin forcément du chemin vers le fichier inversé. Elle contient les méthodes suivantes :
- **IndexLoader * setDictionary(IDictionary * dictionary) :** Permet de préciser le type du dictionnaire utilisé dans l'index chargé.
- **Index * load() :** Cette méthode charge l'index et le renvoie.

Chapitre 2

Tokenisation

La première étape dans un processus d'indexation est la *tokenisation*, qui consiste en une transformation de chaque document en un séquence de tokens (termes, mots). À ce stade, la ponctuation est généralement éliminée et la casse est normalisée.

Une phase de suppression des *stop-words* est parfois rajoutée à ce stade. En fait, La présence de certains mots dans les documents, tels que les déterminants (the, a, this, etc.), les verbes de fonction (be, have, make, etc.), et conjonctions (that, and, etc.), est très commune et leurs indexation augmente considérablement la taille de l'index. En outre, ils ne sont pas informatif : une requête contenant "be" et "have" est susceptible de récupérer presque tous les documents (en anglais) du corpus. Il est alors commun à les ignorer dans les requêtes, et parfois dans l'index lui-même (même s'il est difficile de déterminer à l'avance si une information peut être utile ou pas).

Dans ce qui suit, nous allons tout d'abord présenter en détail la classe **DocumentProvider**. On présente, ensuite, deux version du tokenizer : une première version qui est basique et qui ne prend pas en considération la suppression des stop-words, et une deuxième version plus performante qui tient en compte la suppressions des stops-words.

2.1 Pré-traitement : DocumentProvider

Dans l'optique de rendre l'opération de tokenisation indépendante de la structure du corpus à indexer. Nous avons créé la classe **DocumentProvider** (qui est spécifique au corpus 'latimes' fournie).

En fait, le **DocumentProvider** prend en paramètre un chemin vers la répertoire contenant le corpus à indexer. Chaque fichier du corpus contient un ou plusieurs documents. Le **DocumentProvider** parcourt tous les fichiers et, sous la demande de l'*IndexBuilder*, renvoie le document suivant à traiter, jusqu'à ce qu'il parcourt tout le corpus.

Le document à renvoyer est de type **Document** contenant le texte à tokeniser, ainsi que ses méta données (**DocumentMetaData**).

2.2 Tokenisation basique

Le constructeur prend en paramètre le document à tokeniser, et à l'aide de la fonction **getNextToken()** renvoie, sous la demande de l'indexBuilder, le prochain token. Les tokens représentants des balises sont filtrés (ignorés).

2.3 Tokenisation avec StrSTK

StrTk (String Toolkit) est une librairie C++ libre. Elle se compose d'un ensemble d'algorithmes portables, génériques et robustes ainsi qu'un ensemble de procédures de traitement de chaîne de caractères.

Parmi les fonctionnalités qu'elle offre :

- Des routines de split rapides.
- Tokenisation et itérateurs de string génériques.
- L'intégration transparente avec STL et Boost.

L'idée derrière notre utilisation de la librairie StrTK est de pouvoir appliquer des filtres génériques pour extraire les tokens et les mettre dans une file. Deux filtres sont mis en place :

- **Le filtre des délimiteurs :** permettant de supprimer les balises et les caractères spéciaux. Une liste de délimiteurs est définie par défaut avec la possibilité de la personnaliser via la fonction **setDelimiters(string delimiters)**. Par exemple si nous souhaitons considérer les mots composés comme un seul mot (ex : "high-powered"), il suffit de définir une liste de délimiteurs qui ignore le caractère '-'.
- **Le filtre des stop-words :** Ce filtre permet d'éliminer les mots vides. On définit également une liste par défaut de stop-words avec la possibilité selon le contexte de la personnaliser.

La fonction **fillDequeWithTokens()** effectue tous les traitements nécessaires et génère une liste de tokens. Pour cela, nous avons utilisé la routine **strtk : :split** pour supprimer les espaces blancs. Par la suite, nous avons appliqué la routine **strtk : :parse** avec le filtre des délimiteurs pour construire la liste des tokens à partir du texte. Les mots vides sont ensuite supprimés de cette liste en utilisant le filtre des stop-words.

La fonction **getNextToken()**, sous la demande de l'indexBuilder, renvoie directement à partir de la liste construite le prochain token.

Chapitre 3

Dictionnaire

3.1 Les structures du dictionnaire

Étant donné un index inversé et une requête, notre première tâche est de déterminer pour chaque terme de la requête s'il existe existe dans le vocabulaire. Dans le cas affirmatif, on récupère sa **posting list**. Cette opération de recherche dans le vocabulaire utilise une structure de données classique appelée le *dictionnaire*. Plusieurs manières sont envisageables pour implémenter un dictionnaire, on cite : le hachage et les B-arbres [3].

3.1.1 Hachage

Le hachage est utilisé pour la recherche dans le dictionnaire dans certains moteurs de recherche. Chaque terme (clé) est haché en un entier sur un espace assez grand pour que les collisions de hachage soient peu probables. Cet entier représente l'indice initial du mot dans la table de hachage. les collisions sont résolues par des structures auxiliaires qui peuvent exiger des soins à maintenir.

La complexité de la recherche est de $\mathcal{O}(1)$. Cependant, ces structures ne permettent pas de récupérer tous les termes commençant par un préfixe donné dans un temps raisonnable.

3.1.2 B-Arbre

Une autre manière d'implémenter un dictionnaire est d'utiliser les B-arbres (arbres équilibrés). La recherche dans de telles structures est de complexité $\mathcal{O}(\log M)$ (avec M est le nombre de termes). Contrairement aux tables de hachage, ces structures permettent de récupérer rapidement tous les termes commençant par un préfixe donné.

3.2 Implémentation : HashTableDictionnary

Nous avons choisis d'implémenter le dictionnaire sous la forme d'une table de hachage. La classe **HashTableDictionnary** implémente ce concept dans le projet. Elle étend l'interface **IDictionary** qui définit la maquette à respecter indépendamment du choix de la structure du dictionnaire. Cette interface implémente deux méthodes importantes (voir Chapitre 1) : **getTerm(string token)** et **addTerm(string token)**.

Un **HashTableDictionnary** est une table de hachage avec chaînage externe. Il est défini par la taille de la table (qui doit être de préférence un nombre premier afin de minimiser les risques de collisions (par défaut 10009)) et une fonction de hachage (une implémentation de l'interface **IHasher**).

Durant la construction de l'index, à chaque fois qu'un terme est rencontré, la méthode **addTerm()** est appelée. Deux cas de figure se présentent : (i) Si le terme existe déjà dans le dictionnaire, il voit son nombre d'occurrence incrémenté. (ii) Sinon, le terme est inséré à la fin de la liste des collisions (dans le cas de non collision, la liste est créée contenant un seul élément).

Chapitre 4

Construction d'index

Nous avons présenté dans le chapitre 1 les différents composants d'un **Index**. Nous présentons par la suite les méthodes de construction d'index.

En général, les méthodes de construction d'index sont réparties en deux catégories :
- **Indexation basée mémoire centrale (in-memory index construction)** : Ces méthodes consistent à construire l'index entièrement en mémoire centrale. Ces méthodes ne peuvent être utilisées que si le corpus à indexer est petit.
- **Indexation basée mémoire secondaire (disk-based index construction)** : Ces méthodes consistent à construire l'index en utilisant la mémoire secondaire. Ces méthodes sont plus adaptées à l'indexation de grands corpus. Ils sont cependant moins rapide comparant aux méthodes en mémoire centrale. [2].

D'autres distinctions peuvent être faites sur les méthodes d'indexation. Par exemple on distingue entre : les méthodes de construction *statiques*, où la liste des documents à indexer est connu à l'avance. Et les méthodes de construction *dynamiques* qui traite les changements sur le corpus (ajout ou suppression de document).

Dans ce projet, nous avons choisi de se pencher sur les méthodes de construction d'index statiques. Plus précisément, nous avons implémenté une méthode d'indexation basée mémoire centrale et une autre basée mémoire secondaire (*sort-based*).

Ce chapitre est organisé comme suit. La section 4.1 développe les deux méthodes de constructions explorées. La section 4.2 traite le chargement d'index qui est sérialisé à la fin de la construction.

4.1 Méthodes de construction d'index

Dans la conception (diagramme de classe) de notre projet, l'interface **IIndexBuilder** abstrait la méthode de construction. Celle-ci contient la méthode **createIndex** qui permet de créer et sérialiser (*sauvegarder dans la mémoire secondaire*) l'index. Elle contient également d'autres méthodes qui permettent de configurer le dictionnaire (**IDictionary**), le compresseur (**ICompressor**) et la méthode de tokenization (**ITokenizer**).

L'index final dans la mémoire secondaire obtenu après construction est organisé comme suit :

1. **Dictionnary offset (4 octets) :** indique l'offset du dictionnaire dans le même fichier.
2. **Terms number (8 octets) :** le nombre de termes distincts dans l'index.
3. **Compressor Id (1 octet) :** identifiant du compresseur utilisé (0 pour non compression).
4. **Documents number (4 octets) :** le nombre de documents indexés.
5. **Document Table :** la table des méta-données des documents.
6. **Posting Lists :** l'ensemble des posting list de l'index.
7. **Dictionary :** la liste de tous les termes dans le dictionnaire. Chaque terme a pour forme *(taille du token, token, fréquence totale, nombre de documents contenant le terme, offset de la posting list associée dans le fichier).*

4.1.1 Inversion basée mémoire secondaire (sort-based)

De manière concise, l'indexation **sort-based** consiste à créer dans un premier temps dans la mémoire secondaire la liste de triplets $(tokenId, documentId, frequency)$. Puis, trier cette liste de triplets selon $tokenId$ et $documentId$. Et finalement, écrire dans le disque les *Posting list* de manière contigüe, créées à partir de la liste des triplets associée à chaque terme.

Chaque triplet garde pour chaque *token* (identifié de manière unique par $tokenId$) et pour chaque document (identifié par $documentId$) le nombre d'occurrence du terme *token* dans le document d'identifiant $documentId$.

Tandis que le $documentId$ peut être connu à l'avance, le $tokenId$ doit être calculé. Deux méthodes sont envisageables :

- **Identification en deux passes :** consistant à créer le mapping terme/identifiant dans la première passe sur le corpus.
- **Identification à la volée :** consistant à identifier le terme dès son apparition en assurant la bijection entre l'identifiant et le terme.

4.1.1.1 Étape de construction

La Figure 4.1 présente les étapes majeures de cette méthode. Un composant clé de cette méthode est le **buffer**. Il représente tous simplement une zone mémoire dans laquelle on stocke les triplets créés. Une fois le **buffer** rempli, il est vidé dans un fichier qu'on appellera **run**.

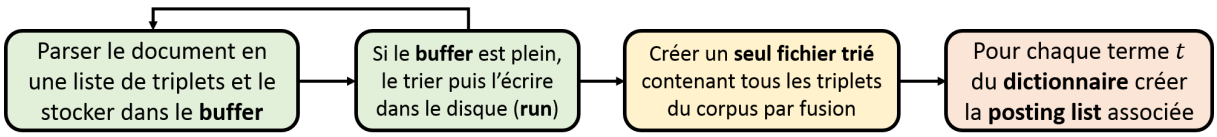

FIGURE 4.1 – Etapes de construction de l'index

1. Préparation de la construction

Dans notre projet, la classe responsable de cette méthode est **SortBasedIndexBuilder** qui implémente **IIndexBuilder**. On peut spécifier à cette classe une taille limite de la mémoire centrale [1]. Une partie de cette taille limite (50%) est réservé au **buffer des triplets** [2] .

Considérant *BufferSize* la taille en octet réservée au **buffer**. Ce buffer est découpé en k ($k \geq 3$) blocks (attribut *numberOfBlock* dans la classe). k est obtenue en divisant *BufferSize* par la taille optimal du block, qui est la taille d'un secteur dans le système d'exploitation (généralement égale à 4 KO). Chaque block peut contenir jusqu'à m triplets (attribut *numberOfTripletInBlock* dans la classe). Le nombre m est calculé en divisant la taille d'un block par la taille d'un triplet. Le nombre de triplets maximale dans le buffer est égale à $n = k * m$.

Avant de lancer la construction d'index, un dictionnaire vide est préparé. Ce dictionnaire regroupe à la fin le vocabulaire de l'index d'une part. D'autre part, il nous aidera a identifier les termes à la volée. Pour le dictionnaire **HashTableDictionary** de taille $size$, l'identifiant du terme est calculé comme suit : Considérant i la cellule occupée par le terme (obtenue par le haché du terme) et j la position du terme dans la liste des termes en collisions). L'identifiant $tokenId$ du terme est donné par :

$$tokenId = size * j + i \tag{4.1}$$

Cette identification nous aide à repérer rapidement le terme dans le dictionnaire à partir de son identifiant.

Exemple 1 : La Figure 4.2 montre le dictionnaire associé à une base de documents. L'ID du token **'fagin'** est $id_{fagin} = 5 * 2 + 3 = 13$.

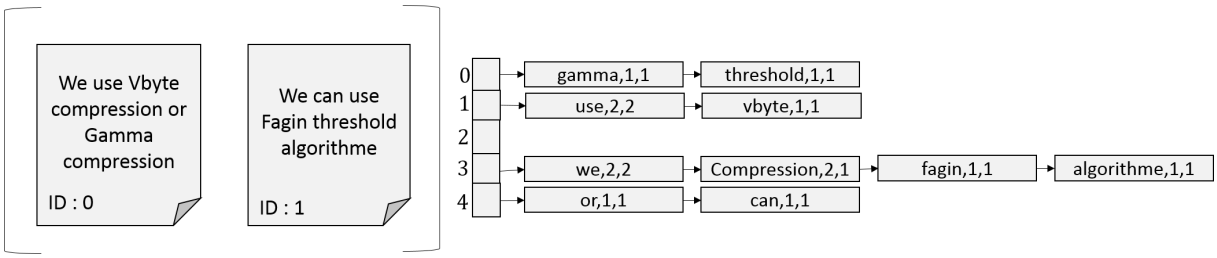

FIGURE 4.2 – Exemple : dictionnaire ($size = 5$)

1. On ne peux pas contrôler cependant les tailles de certains éléments tel que le dictionnaire et la table des documents dont les tailles sont en constante évolution
2. L'autre partie est réservée aux structures qui accompagnent la construction d'index

2. Préparation des Run

A cette étape, tout le corpus est parcouru document par document. Pour chaque document, nous calculons la liste des triplets $(tokenId, documentId, f)$ associée. Tous les triplets de cette liste sont ensuite stockés dans le buffer des triplets. A chaque fois que le buffer se remplisse (contient n triplets), celui-ci est trié selon $tokenId$ et $documentId$ puis vidé dans un fichier temporaire dit **run**. Le dictionnaire se construit au fur et à mesure les termes sont rencontrés.

Exemple 1 (suite) : La Figure 4.3 montre les deux run obtenus sur le corpus de l'exemple 1. Le *Run 1* contient 3 block contrairement au *Run 2* qui contient un unique block. Les triplets générés depuis le document 0 sont en **gras**. Le buffer utilisé contient 3 block dont chacun peut contenir jusqu'à 3 triplets.

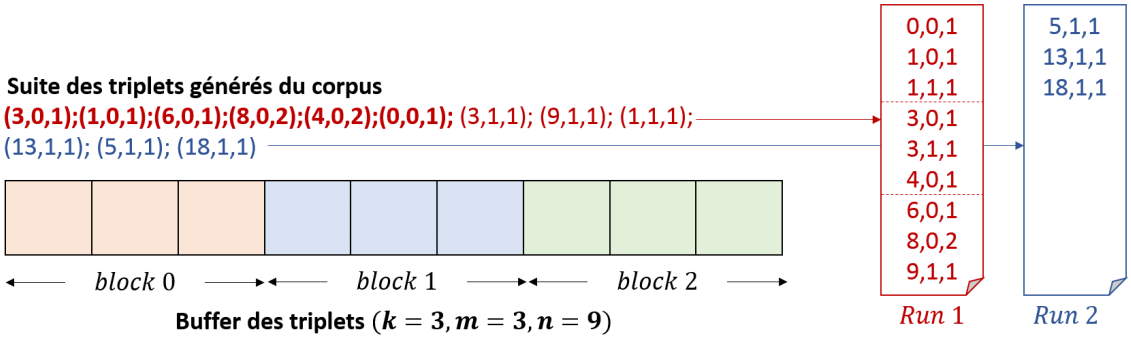

FIGURE 4.3 – Exemple : Préparation des Run

3. Fusion des différents Run

Le but de cette étape est de fusionner l'ensemble des **run** créés durant l'étape précédente en un seul fichier trié contenant tous les triplets du corpus. Pour cela, les run sont fusionnés $k-1$ par $k-1$ (le dernier block est réservé pour l'écriture du fichier résultant). De la même manière, les fichiers obtenus après la fusion de tous les run sont également fusionnés, et ainsi de suite jusqu'à obtention d'un seule fichier trié.

De manière brève, la fusion de $k-1$ fichier de triplets se fait de la manière suivante :

1. On charge le premier block de chaque fichier dans les $k-1$ premiers block du buffer, le $k^{\text{ème}}$ block est réservé à l'écriture et est initialement vide.
2. A chaque itération, on choisit le plus petit triplet (selon $tokenId$ et $documentId$). Ceci revient a choisir parmi les premiers triplets non encore traités de chaque fichier. Ce triplet est ensuite écrit dans le $k^{\text{ème}}$ block. A chaque fois que le $k^{\text{ème}}$ block est rempli, il est vidé dans le fichier de fusion. Inversement, à chaque fois qu'un block $j \in [[1, k-1]]$ est épuisé, on recharge le prochain block du fichier correspondant dans le block j du buffer. On termine une fois la totalité des fichiers à trier sont épuisées.

Exemple 1 (suite) : La Figure 4.4 montre la méthode utilisée pour fusionner les deux run. Le block 0 est réservé au premier run, le block 1 est réservé au deuxième run, le block 2 quant à lui est réservé à l'écriture du fichier final. Au premier temps les premiers block de chaque run sont chargés dans leurs block respectives. Cependant, le premier et le second block du run 1 sont épuisé avant même de prendre un triplet du block du run 2. A l'étape illustré dans la figure, le triplet (5,1,1) est choisi le premier (car plus petit que (6,0,1)) puis dans la même logique (6,0,1) et (8,0,2) sont choisis. Le buffer de l'écriture, étant plein, est vidé dans le fichier de fusion. Les deux petites flèches pointent sur les premiers triplets non encore traités de chaque block. On choisira à la prochaine étape parmi ces deux triplets.

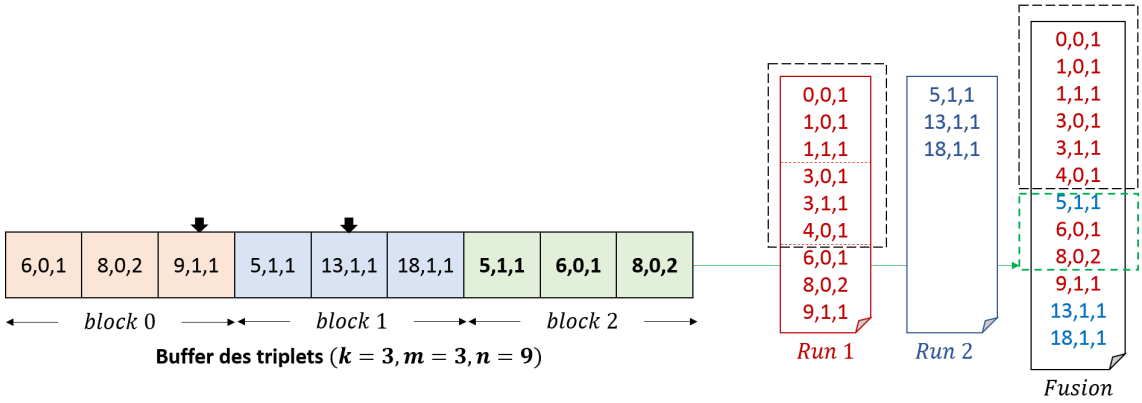

FIGURE 4.4 – Exemple : Fusion

4. Construction d'index

A cette étape, le fichier contenant l'ensemble des triplets du corpus généré à l'étape 3 est utilisé. Ce fichier est parcouru triplet par triplet. Une fois le terme ($tokenId$) change, la posting list de l'ancien terme est construite puis écrite après une éventuelle compression dans le fichier inversé (en respectant la structure détaillée au début de la section 4.1).

Exemple 1 (suite) : La Table 4.1 montre l'index obtenue à la fin (f_t représente la fréquence du terme dans le corpus, n_d^t représente le nombre de document où le terme apparaît). la Figure 4.5 montre la représentation réelle de l'index.

4.1.2 Inversion basée mémoire centrale (in-memory)

L'inversion basée mémoire centrale est une approche qui consiste à construire l'index entièrement en mémoire centrale. Cette approche est plus rapide comparant aux approches de construction basée mémoire secondaire. Cependant, cette approche est adapté au petits corpus. Elle ne supporte pas le passage à l'échelle.

Dans notre projet, la classe responsable de cette méthode est **InMemoryIndexBuilder** qui implémente **IIndexBuilder**. De manière concise, la méthode consiste à parcourir le corpus document par document. Chaque terme (token) rencontré est inséré dans le dictionnaire et voit sa **posting list** mis à jour (soit en incrémentant la fréquence dans le nœud du document actuel s'il existe, soit en insérant un nouveau nœud pour le document actuel).

Term			Posting List
Token	f_t	n_d^t	$(documentId, f)$
gamma	1	1	(0,1)
use	2	2	(0,1) ;(1,1)
we	2	2	(0,1) ;(1,1)
or	1	1	(0,1)
threshold	1	1	(1,1)
vbyte	1	1	(0,1)
compression	2	1	(0,2)
can	1	1	(1,1)
fagin	1	1	(1,1)
algorithme	1	1	(1,1)

TABLE 4.1 – Exemple : Représentation conceptuelle de l'Index

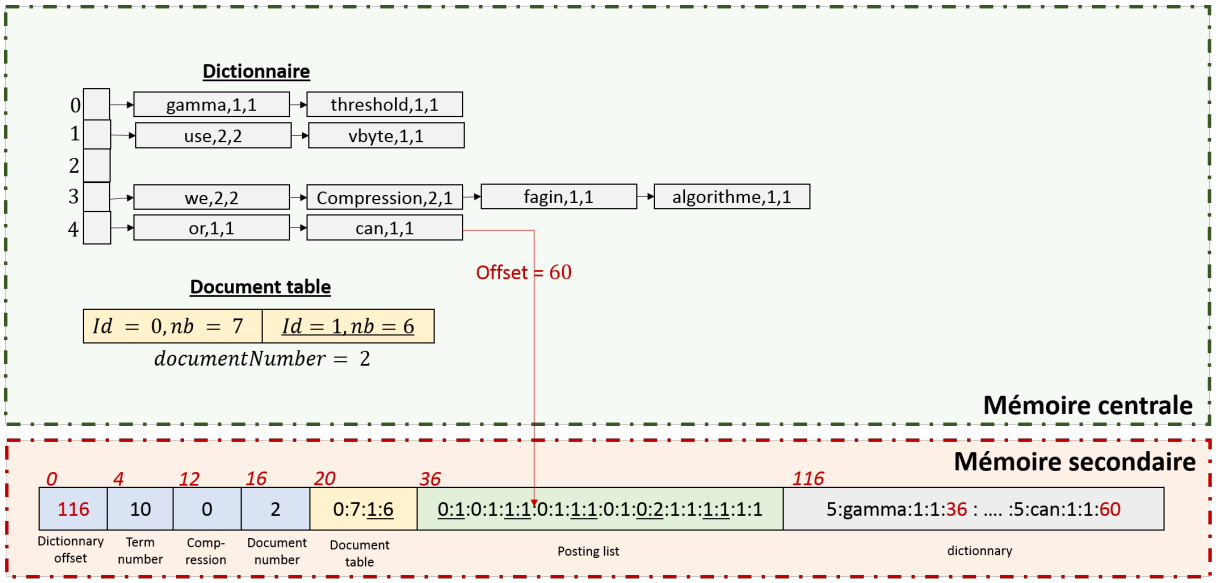

FIGURE 4.5 – Exemple : Représentation réelle de l'index

L'index est à la fin sérialisé (comme montré dans la Figure 4.5). Les posting list sont ensuite supprimées de la mémoire centrale.

4.2 Chargement d'index

Le chargement d'index consiste à charger dans la mémoire centrale une partie de l'index. Cette fonction est assurée par la méthode *load()* dans la classe **IndexLoader**. Le type de dictionnaire peut être par ailleurs configuré avant chargement de l'index. Ceci dit, qu'un index peut être construit en utilisant un dictionnaire sous la forme d'une table de hachage, et chargé avec un dictionnaire sous la forme d'un B-Arbre. Il est à noter que le chargement de l'index ne charge pas en mémoire centrale les posting list. En effet, les postings list seront chargés uniquement au besoin (la recherche), puisque c'est la partie la plus volumineuse de l'index.

Chapitre 5

Compression

La compression est une étape importante dans la construction d'index. En effet, la taille de l'index est généralement grande, donc il consomme beaucoup d'espace sur le disque dur. Afin de pallier à ce problème, on peut compresser quelques attributs de l'index ou le compresser entièrement. Cela veut dire que cette opération nous permet en premier lieu de réduire l'espace consommé dans le disque dur, mais elle peut aussi conduire parfois à un gain de temps dans l'exécution. En effet, c'est vrai que cette opération consomme un temps supplémentaire d'exécution pour les deux opérations de compression et décompression, mais le temps d'accès disque diminuera car la taille des données lues est plus petite.

Dans notre cas, nous compressons les posting list. Ces dernières sont des listes dont chaque élément est composée de deux entiers (de taille 4 octets pour chacun), ces entiers sont :

- **documentIndex :** L'indice du document dans la table documentTable.
- **ftd :** la fréquence du terme dans le document

Nous appliquons donc une compression sur les nombres entiers. Nous avons créé l'interface portant le nom **ICompressor**, celle-ci contient deux méthodes principales :

- **compressAndWrite :** elle reçoit en paramètre un fichier (ouvert en écriture) et une posting list (sous forme de liste chainée) à compresser. Cette méthode compresse la liste et écrit le résultat sur le fichier.
- **readAndDecompress :** à partir du fichier (ouvert en lecture) passé en paramètre, cette méthode lit une posting list de taille spécifiée par un paramètre "size" (le nombre de document où le terme apparaît), décompresse cette liste, et affecte le résultat à une variable "postingList" passée en paramètre.

Pour réaliser cette compression, nous allons minimiser l'espace occupé par chaque entier. Pour ce faire, nous allons nous articuler sur deux stratégies [1] :

1. Avec la compression, plus l'entier est petit moins il consomme de l'espace. Sachant que les posting list sont triées par indices de document (documentIndex), on peut garder la différence entre les indices au lieu des indices eux même. Autrement dit, supposons qu'on a la liste de documentIndex suivante :

   $$L = [80, 120, 150, 160, 210, 240]$$

   La liste de différence entre ces indices est :

   $$L_{\Delta} = [80, 40, 30, 10, 50, 30]$$

Il est clair qu'on peut retrouver L à partir de L_Δ. De plus, les entiers de L_Δ sont certainement inférieurs à leurs entiers correspondants dans L. Cela veut dire qu'avec compression, l'espace pris par L_Δ est inférieur à celui pris par L. Alors, dans nos implémentations des méthodes de compression, on compresse toujours la différence entre les indices de documents au lieu des indices eux même. Quant aux ftd (fréquence du terme dans le document), on les compresse tels qu'ils sont.

2. Ensuite, on compresse entier par entier. Pour cette étape, nous avons implémenté deux méthodes de compression. Elles sont détaillées dans ce qui suit.

5.1 Compression VByte

C'est une compression de type bytewise, ce type vise à écrire chaque entier sur un minimum d'octets. La méthode VByte code les entiers sur un nombre variable d'octets (selon sa taille). L'idée est de tester si l'entier d est inférieur à 128. Si c'est le cas, alors on code d sur les 7 derniers bits de l'octet, et on met le premier bit (le bit du poids fort) à 1 pour indiquer qu'on a fini. Sinon :

1. On prend le reste de la division de d par 128, et on le code sur les 7 derniers bits de l'octet courant. Ensuite, on met le premier bit à 0 pour indiquer qu'on n'a pas fini.
2. On applique récursivement la même procédure sur $d/128$

Exemple Supposons qu'on veut coder l'entier $x = 300$. Il est supérieur à 128, donc on ne peut pas le coder sur un seul octet.

1. On prend le reste de la division de 300 par 128 qui est 44 (en binaire c'est 0101100) et on le code sur le premier octet, en mettant le premier bit à 0. Cela donne :

$$0_1 = \underline{0}0101100$$

2. On applique la même procédure sur 300/128 qui est 2. L'entier 2 (en binaire c'est 0000010) est inférieur à 128, alors il peut tenir sur 7 bits. On l'écrit sur le deuxième octet et on met le premier bit à 1. Le résultat est :

$$0_2 = \underline{1}0000010$$

La représentation de 300 en code VByte est donc en deux octets : (O_1, O_2)

Quant au décodage, il suffit de lire les octets $b_n, ..., b_2$ qui commencent par 0, jusqu'à trouver l'octet b_1 qui commence par 1. L'entier x est :

$$x = b_n \times 128^{n-1} + ... + b_2 \times 128 + b_1$$

Dans notre projet, cette méthode a été développée dans la classe **VByteCompressor** qui implémente l'interface **ICompressor**. Sa fonction de compression parcourt la posting list élément par élément. Et pour chaque entier à compresser (la différence Δ pour **documentIndex**, et la valeur elle même pour ftd), elle fait appel à la fonction **compressUnsignedInt** qui compresse un entier selon la méthode VByte et l'écrit dans le fichier. De même pour la fonction de décompression, elle appelle la fonction **decompressUnsignedInt** pour chaque entier.

5.2 Compression Gamma

C'est une compression de type bitwise, ce type vise à écrire chaque entier sur un minimum de bits. La méthode de compression Gamma consiste à sauvegarder deux informations pour chaque entier x :

1. La longueur $L = \lfloor log_2(x) \rfloor + 1$ de l'entier x en bits.
2. L'entier x.

Exemple Supposons qu'on veut compresser le nombre $x = 19$ (en binaire c'est 10011) Sa longueur L est égale à 5, alors :

1. La première information va être représentée par une suite de $L - 1$ bits à 1 suivis d'un bit à 0. Donc c'est 11110.
2. L'entier 19 est en réalité égale à 10011 en binaire. Mais on sait déjà que le premier bit de tous les nombres non nuls est 1. Alors on peut optimiser en ne gardant que 0011 comme représentation de 19 (on n'écrit pas le premier bit, car il est certainement 1).

La représentation du 19 est la combinaison de ces deux informations, et c'est donc (11110,0011). Alors, la compression de cet entier est tenu sur 9 bits.

Remarque Seulement dans le cas où la taille du nombre en bits est égale à 1 (c'est possible pour l'entier 0 et 1), nous écrivons quand même le premier bit de ce nombre (qui est d'ailleurs le seul bit). Car dans ce cas, la longueur ne permet pas de savoir si c'est un 0 ou c'est un 1.

Quant au décodage, il suffit de lire tous les bits à 1 jusqu'à arriver à un bit à 0. Cela nous permet de déterminer la longueur L de l'entier. Dans notre exemple (de l'entier 19), on détermine que la longueur est égale à 5. Ensuite, on lit $L - 1$ bits et on les accomplit avec un bit à 1 à gauche, et cela donne l'entier recherché.

Dans notre projet, nous avons développé cette méthode dans la classe **GammaCompressor** qui implémente l'interface **ICompressor**. Sa fonction de compression parcourt tous les entiers de la posting list et les compresse un par un. Elle met le résultat dans un buffer de 8 bits. À chaque fois que ce dernier est plein, cette fonction le vide dans le fichier. De la même manière, la fonction de décompression utilise un buffer de 8 bits pour lire, à chaque fois qu'il est épuisé, elle le recharge en lisant un octet du fichier.

Chapitre 6

Recherche

Une fois les documents indexés, il est possible de rechercher ceux qui répondent au mieux à une requête donnée. Deux approches ont été utilisées dans ce projet :

- L'approche algébrique (ou vectorielle) qui considère que les documents et les requêtes font partie d'un même espace vectoriel.
- L'approche probabiliste qui essaie de modéliser la notion de pertinence.

6.1 Recherche par la méthode Fagin's threshold

6.1.1 Principe

Étant donnée une requête utilisateur Q et un nombre k, l'algorithme va retourner les k documents qui répondent au mieux à la requête sans nécessairement parcourir la totalité des posting list associées aux termes de la requête. Nous faisons l'hypothèse qu'on a un ensemble de listes triées par ordre décroissant de **TF-IDF**. [1]

Le **TF-IDF** (Term Frequency - Inverse Document Frequency) est une méthode de pondération souvent utilisée en recherche d'information. Cette mesure statistique permet d'évaluer l'importance d'un terme contenu dans un document, relativement à une collection ou un corpus. Le poids augmente proportionnellement au nombre d'occurrences du mot dans le document. Il varie également en fonction de la fréquence du mot dans le corpus.

$$\text{TF-IDF}_{i,j} = \text{TF}_{i,j}.\text{IDF}_i \tag{6.1}$$

$$\text{TF}_{i,j} = \frac{n_{i,j}}{N_j} \tag{6.2}$$

$$\text{IDF}_i = log_{10}\left(\frac{N}{|\{d_j : t_i \in d_j\}|}\right) \tag{6.3}$$

Avec :

- $n_{i,j}$: nombre d'occurrences du terme t_i dans le document d_j
- N_j : nombre de mots dans le document d_j
- N : nombre total de documents dans le corpus
- $|\{d_j : t_i \in d_j\}|$: nombre de documents d_j où le terme t_i apparaît (c'est-à-dire $n_{i,j} > 0$).

6.1.2 Implémentation

L'algorithme nécessite en entrée une requête Q contenant m termes, un nombre entier k et pour chaque terme t_i dans Q une liste de pairs $\langle documentIndex, \text{TF-IDF}\rangle$ triée par ordre décroissant de TF-IDF.

1. Soit R, la liste à retourner, initialement vide.
2. Pour i allant de 1 à m :
   (a) Récupérer le document $d^{(i)}$ en tête de la liste correspondante au terme t_i. C'est le document qui a le plus grand TF-IDF dans cette liste, vu que cette dernière est triée par ordre décroissant.
   (b) Calculer le score globale $g_{d^{(i)}}$ de ce document en parcourant les autres listes des autres termes et en récupérant à chaque fois le score $s(t_j, d^{(i)})$ du document $d^{(i)}$ avec $i \neq j$.
   (c) Si R contient moins de k documents, ajouter $d^{(i)}$ à R. Sinon, si $g_{d^{(i)}}$ est plus grand que le minimum de tous les scores des documents dans R, remplacer le document ayant le plus petit score dans R par $d^{(i)}$.
   (d) Supprimer le document $d^{(i)}$ de toute les listes où il apparaît.
3. $threshold = g(s(t_1, d^{(1)}), s(t_2, d^{(2)}), ..., s(t_m, d^{(m)}))$
4. Si R contient au moins k documents, et le score minimum des documents dans R est supérieur ou égal à $threshold$, ou, s'il ne reste aucune liste non vides, retourner R. Sinon, aller à 2.

Avec :
- $s(t_j, d^{(i)})$ représente le TF-IDF$_{i,j}$ (terme t_j et document $d^{(i)}$).
- $g_{d^{(i)}}$ représente la somme de tous les scores des termes $t_j \in d^{(i)}$ de la requête Q.
- $g(s(t_1, d^{(1)}), ..., s(t_m, d^{(m)}))$ représente $\sum_{i=1}^{m} s(t_i, d^{(i)})$

6.1.3 Appel de la fonction

- **Classe :** Index
- **Méthode :** searchFagin(int topK, string query)
- **Retour :** vector<pair<DocumentMetaData documentMetaData, double score>>

6.2 Recherche par la méthode BM25

6.2.1 Principe :

BM25 est une méthode de pondération utilisée en recherche d'information. Elle est une application du modèle probabiliste de pertinence qui exprime une estimation de la probabilité de pertinence d'un document par rapport à une requête, et ainsi classer une liste de documents dans l'ordre décroissant d'utilité probable pour l'utilisateur. BM25 est en fait une combinaison de fonctions de scoring, et de quelques paramètres.

Étant donné une requête Q contenant m mots $q_1, ..., q_m$, la fonction $BM25$ d'un document d est donnée par :

$$Score(d, Q) = \sum_{i=1}^{m} IDF(q_i).\frac{f(q_i, d).(k_1 + 1)}{f(q_i, d) + k_1.(1 - b + b * \frac{|d|}{avgdl})}$$

- $f(q_i, d)$: la fréquence du terme q_i dans le document d.
- $|d|$: le nombre de mots dans le document d.
- $avgdl$: la taille moyenne des documents dans le corpus.
- $k_1 \in [1.2, 2.0]$ fixé à 2 dans le projet.
- $b \in [0.5, 0.8]$ fixé à 0.75 dans le projet.
- $IDF(q_i) = log_{10}(\frac{N - n(q_i) + 0.5}{n(q_i) + 0.5})$
  - N : nombre total des documents dans le corpus
  - $n(q_i)$: nombre de documents contenant le terme q_i

Remarque : $(1 - b + b.\frac{|d|}{avgdl})$ utilisé dans la fonction BM25 est nommé **Length Normalized Component**. Le but étant de classer les documents d'une manière juste vu que les documents d'une grande taille seront avantagés en dépit des documents courts.

Critique de la méthode :

- Il faut noter que la formule d'IDF utilisée présente des inconvénients quand elle est utilisée pour des termes qui apparaissent dans plus de la moitié de l'ensemble des documents. La valeur d'IDF sera alors négative, ce qui veut dire que pour deux documents qui sont à peu près similaires et dont l'un contient le terme et l'autre ne le contient pas, celui qui ne le contient pas aura probablement un score plus haut que celui qui le contient. Ce qui signifie que les termes qui apparaissent dans plus de la moitié des documents influenceront de manière négative le score final d'un document.
- Le choix des valeurs des paramètres b et k_1 respectivement dans les intervalles $[0.5, 0.8]$ et $[1.2, 2.0]$ reste raisonnable dans la plupart des circonstances. Cependant, le modèle ne montre pas comment ceux-ci devront être choisis. Cela peut être considéré comme une limite du modèle.

6.2.2 Implémentation

1. Soit R, la liste à retourner, initialement vide
2. Calculer $avgdl$, la taille moyenne des documents dans le corpus
3. Pour i allant de 1 à m :
   (a) Pour chaque d contenant q_i :
      i. Calculer $score(q_i, d)$
      ii. Si R contient déjà d, ajouter $score(q_i, d)$ à l'ancien score, sinon ajouter une nouvelle entrée dans la liste $\langle d.documentIndex, score(q_i, d) \rangle$
4. Trier par ordre décroissant de score la liste R.
5. Retourner les k premiers éléments.

6.2.3 Appel de la fonction

- **Classe :** Index
- **Méthode :** searchBM25(int topK, string query)
- **Retour :** vector<pair<DocumentMetaData documentMetaData, double score>>

Chapitre 7

Divers

7.1 Estimation du paramètre de la loi de Zipf

7.1.1 Définition : Loi de Zipf

La loi de Zipf est une observation empirique concernant la fréquence des mots dans un text. Elle a pris le nom de son auteur, George Kingsley Zipf. Cette loi a d'abord été formulée par Jean-Baptiste Estoup et a été par la suite démontré (et généralisé) à partir des formules de Shannon par Benoît Mandelbrot (Loi de Zipf-Mandelbrot) [4].

Soit un texte contenant $N \in \mathbb{N}^*$ mots distincts. Soit un mot dont le rang est $k \in [[1, N]]$ (selon l'ordre décroissant des occurrences du mot) et soit $\alpha > 0$. La fonction de masse de la loi de Zipf est donnée par :

$$f : k \mapsto \frac{1}{H_{N,\alpha}} \frac{1}{k^\alpha} \tag{7.1}$$

Avec :

$$H_{N,\alpha} = \sum_{i=1}^{N} \frac{1}{i^\alpha} \tag{7.2}$$

Dans la plupart des langages existants, le paramètre α est proche de 1.

7.1.2 Estimation de α dans le corpus 'latimes'

Il existe plusieurs méthodes pour estimer le paramètre d'une fonction à partir d'une observation, on cite **la méthode des moindres carrés** et **la méthode maximum de vraisemblance**. Nous allons montrer dans ce qui suit de manière brève comment utiliser ces deux méthodes pour estimer le paramètre α de la loi de Zipf.

7.1.2.1 Méthodes d'estimations

Étant donné un texte contenant M mots et $N << M$ mots distincts. Ces mots sont ordonnées selon l'ordre décroissant de leurs occurrences. Soit $f_i = \frac{m_i}{M}$ la fréquence d'apparition du mot de rang $i \in [1, N]$ (dont l'occurrence est m_i).

Utilisation de la méthode des moindres carrés (LS) : Cette méthode essaye de minimiser la somme quadratique des déviations observées entre les fréquences observées et

les fréquences données par la loi de Zipf (la variable étant α). Autrement dit, la méthode des moindres carrés, dans ce cas, tente de minimiser la fonction :

$$LS(\alpha) = \sum_{i=1}^{N} [log(f_i) + \alpha log(i) + log(H_{N,\alpha})]^2 \tag{7.3}$$

Utilisation de la méthode maximum de vraisemblance (MLE) : Cette méthode essaye de maximiser la vraisemblance (en utilisant comme modèle la loi de Zipf). Autrement dit, la méthode maximum de vraisemblance, dans ce cas, tente de maximiser la fonction :

$$MLE(\alpha) = -\alpha \sum_{i=1}^{N} f_i log(i) - log(H_{N,\alpha}) \tag{7.4}$$

7.1.2.2 Estimation de α dans le corpus 'latimes'

Nous allons utiliser les deux méthodes d'estimations énoncées ci-dessus pour estimer α de la loi de Zipf dans tous le corpus **'latimes'** (131896 documents, 441688 termes distincts). La méthode des moindres carrés a donné $\alpha_{LS} = 1.2860$ (en rouge). Cependant la méthode maximum de vraisemblance a donné $\alpha_{MLE} = 1.0632$ (en vert). La Figure 7.1 donne les fréquences observées et données par la loi de Zipf dans les échelles log-log et normale respectivement.

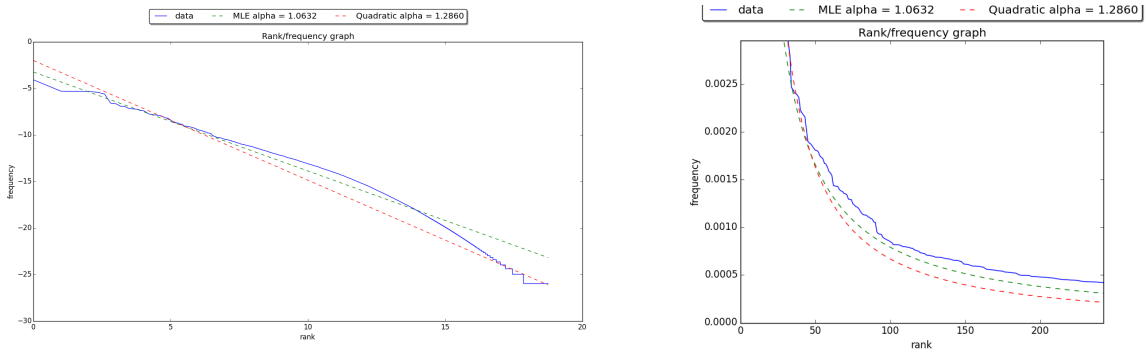

FIGURE 7.1 – Estimation du paramètre s de la loi de zipf (Echelle **log-log** à gauche, Echelle **normale** à droite)

Chapitre 8

Tests de performance

8.1 Jeu de données

- **Nombre de fichiers :** 730 (Tout le corpus)
- **Nombre de documents :** 131 896 documents.
- **Taille totale :** 475 Mo
- **Nombre de termes différents dans le dictionnaire :** 441 688 termes [1]
- **Structure d'un document :**
  - **<DOC> :** représente le début d'un document
  - **<DOCNO> :** identifie le document de manière unique
  - **<DOCID> :** représente l'ID du document
  - **<DATE> :** représente la date de publication
  - **<HEADLINE> :** représente le titre du document
  - **<TEXT> :** représente le corps du document
  - **<GRAPHIC> :** représente la description d'une image

8.2 Tests

Les tests ont été effectués dans une machine de Processeur I3 (2.53 GHz) et 3 GO de RAM.

8.2.1 Construction d'index

La construction de l'index dépend de plusieurs paramètres : (i) la méthode d'indexation, (ii) la méthode de compression et (iii) la méthode de tokenisation. Nous avons utilisé pour les tests un dictionnaire de type table de hachage contenant **10 009 cellules**. La Table 8.1 donne les performances en utilisant le tokenizer *SimpleTokenizer*. La Table 8.2 donne les performances en utilisant le tokenizer *StrtkTokenizer*. La Table 8.3 donne la taille de l'index dans le disque selon la méthode de compression avec l'utilisation de *SimpleTokenizer*. La Table 8.4 donne la taille de l'index dans le disque selon la méthode de compression avec l'utilisation de *StrtkTokenizer*.

1. En utilisant *SimpleTokenizer*

Méthode	Compression	Temps (s)	UC (%)	Memoire (Mo)
In-Memory	No	693 s	26 %	870 Mo
	VByte	690 s	26 %	870 Mo
Sort-based (64 KO)	No	1170 s	24 %	45 Mo
	Gamma	1080 s	25 %	47 Mo
Sort-based (2048 KO)	Gamma	648 s	25 %	48 Mo

TABLE 8.1 – Tests : Construction d'index (avec *SimpleTokenizer*)

Méthode	Compression	Temps (s)	UC (%)	Memoire (Mo)
In-Memory	No	298s	24%	709 Mo
	VByte	315s	24%	712 Mo
Sort-based (64 KO)	No	597s	25%	41 Mo
	Gamma	604s	25%	40 Mo
Sort-based (2048 KO)	Gamma	465 s	25 %	40 Mo

TABLE 8.2 – Tests : Construction d'index (avec *StrtkTokenizer*)

Compression	Taille (Ko)
No	261 665 Ko (53.8% du corpus)
VByte	82 128 Ko (16.9% du corpus)
Gamma	58 513 Ko (12.0% du corpus)

TABLE 8.3 – Tests : Taille d'index dans le disque (avec *SimpleTokenizer*)

Compression	Taille (Ko)
No	227 852 Ko (46.8% du corpus)
VByte	68 791 Ko (14.2% du corpus)
Gamma	13 280 Ko (2.8% du corpus)

TABLE 8.4 – Tests : Taille d'index dans le disque (avec *StrtkTokenizer*)

8.2.2 Chargement d'index

Le chargement d'index ne dépend pas de la méthode de compression, étant donné que les posting list ne sont pas chargées durant la phase de chargement. La Table 8.5 résume les informations concernant les performances du chargement. Le dictionnaire utilisé est une table de hachage contenant **10 009 cellules**. Dans ces tests, le tokenizer utilisé est *SimpleTokenizer*.

Temps (s)	UC (%)	Memoire (Mo)
14 s	24 %	31 Mo

TABLE 8.5 – Tests : chargement d'index

8.2.3 Recherche dans l'index

Les performances de la recherche dépend de deux facteurs : (i) *la méthode de compression* : car la décompression des *positing list* se fait durant la phase initiale de la recherche et (ii) *la méthode de recherche*. La Table 8.6 résume les performances de la recherche de la requête = *the broadway chicago castro* dans l'index (avec topk=10). Dans ces tests, le tokenizer utilisé est *SimpleTokenizer*.

Méthode	**Compression**	**Temps (s)**	**UC (%)**	**Memoire (Mo)**
Fagin	No	3 s	14 %	5 Mo
	VByte	1 s	15 %	4 Mo
	Gamma	1 s	15 %	4 Mo
BM25	No	2 s	19 %	8 Mo
	VByte	4 s	24 %	7 Mo
	Gamma	4 s	24 %	7 Mo

TABLE 8.6 – Tests : Recherche dans l'index

Conclusion

Nous avons pu à l'issue de ce projet explorer diverses méthodes d'indexation des corpus documentaires. Nous avons aussi découvert diverses approches de compression (VByte, Gamma) ainsi que des méthode de recherche en utilisant un index (modèle vectoriel : algorithme de Fagin, modèle probabiliste : algoritme BM25). Nous retenons dans cette conclusion deux leçons assez générales :

- Tenir compte de la consommation de la mémoire centrale pour n'importe quelle méthode qui doit répondre au problème du passage à l'échelle.
- Penser toujours à utiliser la mémoire secondaire pour diminuer la charge sur la mémoire centrale et diminuer considérablement le nombre de défauts de pages qui ont un impacte dévastateur sur la performance de l'algorithme.

Bibliographie

[1] S. Abiteboul, I. Manolescu, P. Rigaux, M.-C. Rousset, and P. Senellart. *Web data management*. Cambridge University Press, 2011.

[2] S. Büttcher, C. L. Clarke, and G. V. Cormack. *Information retrieval : Implementing and evaluating search engines*. Mit Press, 2010.

[3] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[4] Wikipédia. Loi de zipf — wikipédia, l'encyclopédie libre, 2015. [En ligne ; Page disponible le 27-octobre-2015].

[5] Wikipédia. Mot vide — wikipédia, l'encyclopédie libre, 2015. [En ligne ; Page disponible le 3-novembre-2015].