Université de Bourgogne

UFR Sciences et Techniques

Licence Informatique – Projet Info4B

20 avril 2026

Conception et Implémentation d'un Moteur de Recherche Local de Haute Performance

Analyse Fonctionnelle et Architecture en Couches

Auteurs :
Antoine RAGOT – RAILLAT
Arnaud LELAURE
Benjamin BAREILLE

Encadrant :
M. Eric LECLERCQ

Module :
Info4B

Résumé

Ce rapport détaille la conception, l'architecture logicielle et l'implémentation d'un moteur de recherche local "Full-Text" fondé sur une architecture client-serveur en Java.

Le noyau du projet repose sur un **Index Inversé** couplé à un tri par score **TF-IDF**. L'application garantit une cohérence des données via un cache **LRU** (Least Recently Used) et une journalisation transactionnelle **WAL** (Write-Ahead Logging). La gestion multi-threadée assure l'indexation, la surveillance récursive du système de fichiers (NIO.2), et la communication par Sockets TCP permettant la recherche complexe, l'extraction de métadonnées et le téléchargement de fichiers binaires.

Table des matières

1. Analyse Fonctionnelle du Sujet

1.1 Compréhension du Problème et Objectifs

Le cahier des charges impose de développer une application capable de classer et de retrouver des fichiers par mots-clés. L'outil doit aller au-delà d'un explorateur de fichiers classique (limité au nom et à la taille) en lisant le contenu textuel profond des documents.

De plus, l'application doit être conçue comme un service d'arrière-plan permanent (Serveur) accessible par un ou plusieurs terminaux distants (Clients) via le réseau, garantissant un fonctionnement ininterrompu.

1.2 Vulgarisation : La Métaphore du Bibliothécaire

Pour appréhender la logique du système, on peut utiliser l'analogie du "Bibliothécaire Automatisé".

Imaginez une bibliothèque où, chaque fois qu'un nouveau livre arrive, un bibliothécaire (le **Serveur**) le lit intégralement, identifie les mots les plus importants, et met à jour un grand registre alphabétique (l'**Index Inversé**). Ce registre ne classe pas les livres par titre, mais par contenu : à chaque mot correspond une liste de livres et la fréquence d'apparition du mot.

L'utilisateur (le **Client**), au lieu de parcourir tous les rayons, se contente de crier un mot au bibliothécaire. Ce dernier consulte son registre et répond instantanément : "Le livre A parle beaucoup de votre sujet (score élevé), le livre B un peu moins". C'est cette orchestration qui constitue le cœur de notre application.

1.3 Découpage en Sous-Problèmes

Pour maîtriser la complexité du projet, le système a été décomposé en cinq sous-problèmes :

1. **Collecte de données variées** : Naviguer de manière autonome dans les répertoires et extraire le texte depuis des formats divers (Texte brut, PDF, Images via métadonnées EXIF, Archives DOCX, et Web HTML).
2. **Structuration des Données** : Transformer les chaînes de caractères en structures interrogeables rapidement, tout en filtrant certains mots (Stop-Words).
3. **Algorithmique de Pertinence** : Évaluer le lien entre la recherche de l'utilisateur et les documents pour les classer par ordre d'importance.
4. **Résilience et Persistance** : Assurer que l'index ne soit pas perdu en cas de fermeture brutale du processus, sans impacter les performances d'écriture sur le disque.

5. **Communication Réseau** : Établir un protocole fiable permettant l'interrogation de la base et le téléchargement de fichiers de A à Z.

1.4 Règles de Fonctionnement de l'Application

Le système obéit à des règles définies lors de notre phase de conception :

— **Identification des fichiers identiques.** : L'application calcule une empreinte (Hash) des fichiers. Deux fichiers identiques avec des noms différents peuvent être détectés comme des clones exacts.

— **Mise à jour intelligente des mots-clés** : Si l'utilisateur modifie la liste des Stop-Words (mots ignorés), le système réindexe dynamiquement pour répercuter ce changement.

— **Mise à jour automatique et instantanée** : L'utilisateur n'a jamais à forcer l'indexation. Toute création, modification ou suppression d'un fichier dans le dossier surveillé met à jour l'Index Inversé instantanément, sans que le serveur ne cesse de répondre aux clients.

1.5 Inventaire des Fonctionnalités Client (CLI)

Le client propose une grammaire de commandes permettant de piloter l'intégralité du serveur à distance.

— `-h` : Affiche l'aide et la liste des commandes disponibles.

— `-l` : Liste tous les fichiers indexés.

— `-t <message>` : Envoie un message au serveur et reçoit un écho (test de communication).

— `-q` : Coupe la connexion du client proprement.

— `-s <mot1,mot2> [- <motExclu>]` : Recherche standard avec calcul du score TF-IDF. Permet d'exclure des termes via l'opérateur `-`.

— `-m <chemin>` : Affiche les métadonnées enregistrées (poids, date, nombre de mots). Possède les sous-commandes :
  — `-rn` (renommer)
  — `-rm` (supprimer) pour agir physiquement sur le disque du serveur.

— `-tag <action> <chemin> <mot>` : Annoter des fichiers :
  — `-add` (ajouter un/des tag(s))
  — `-l` (afficher le/les tag(s) actuel(s))
  — `-rm` (enlever un/des tag(s)) pour la recherche sémentique.

— `-d <chemin1> <chemin2>` : Détecte si deux fichiers sont des copies exactes (comparaison par empreinte MD5).

— `-r <chemin>` : Affiche le contenu textuel d'un fichier.

— `-ar <mot1 et/ou/sauf mot2>` : Recherche avancée. Effectue des opérations ensemblistes sur les listes de résultats. Les opérateurs sont insensibles à la casse.

— `-exif <chemin_image>` : Extraction des métadonnées EXIF d'une image (modèle de l'appareil, date de prise de vue).
— `-exif -set <chemin> <texte>` : Modifie le contenu de la métadonnée "description".
— `-dl <chemin>` : Téléchargement binaire du serveur vers le client.
— `-clean` : Déclenche la compaction du journal (WAL).
— `-reindex` : Réindexation totale depuis zéro.
— `-sw -add / -rm / -l <mot>` :
  — `-add` (ajouter un/des stop-word(s))
  — `-rm` (enlever un/des stop-word(s))
  — `-l` (lister les stop-words))
  Les modifications déclenchent une réindexation automatique.

1.6 Documentation et Site Web

En complément de l'interface en ligne de commande, Antoine en grande partie avec l'aide légère de Benjamin a développé un **site web de documentation technique responsive** accessible à l'adresse https://searchengine.antoineragot.com. Réalisé en HTML, CSS et Bootstrap, il présente de manière visuelle et interactive l'ensemble des commandes disponibles, leurs options et les comportements attendus, avec des exemples de terminaux simulés. Ce travail supplémentaire illustre la démarche d'ingénierie logicielle complète du trinôme, au-delà du seul code fonctionnel.

2. Fondements Théoriques et Algorithmiques

2.1 Le Modèle de Recherche d'Information

La Recherche d'Information ne se résume pas à une simple comparaison de chaînes de caractères. Elle consiste à modéliser mathématiquement le contenu des documents pour répondre à une intention de recherche.

2.1.1 Prétraitement et Normalisation (Tokenisation)

Avant d'être indexé, le texte brut subit plusieurs transformations dans `Main.indexerFichier()`. Le contenu est séparé en mots individuels grâce à l'expression régulière Java suivante, qui découpe sur tout caractère qui n'est ni une lettre ni un chiffre (Unicode inclus) :

```
texteMinuscule.split("[^\\p{L}\\p{N}]+")
```

`p{L}` désigne toute lettre Unicode (y compris les caractères accentués comme "é" ou "ü"), et `p{N}` tout caractère numérique. Le `^` en début de classe inverse la sélection : on coupe sur tout ce qui n'est *pas* une lettre ou un chiffre, c'est-à-dire les espaces, la ponctuation, les tirets, etc. Le `+` évite de produire des tokens vides entre deux séparateurs consécutifs.

2.1.2 L'Index Inversé

Dans un système de fichiers classique, la recherche d'un terme impose un parcours séquentiel. Si N est le nombre de documents et M la taille moyenne d'un document, la complexité est de $O(N \times M)$. Cette approche est inenvisageable pour un système temps réel.

L'indexe inversé repose sur une approche totalement différente. Il s'agit d'un dictionnaire qui associe chaque **Token** (mot extrait) à une liste d'occurrences.

Formalisation mathématique :
Soit $\mathcal{D} = \{d_1, d_2, ..., d_n\}$ l'ensemble de documents et $\mathcal{T} = \{t_1, t_2, ..., t_m\}$ l'ensemble des termes uniques (le lexique). L'index inversé est une application I telle que :

$$I(t_i) = \{(d_j, f_{i,j}) \mid t_i \in d_j\} \tag{2.1}$$

où $f_{i,j}$ représente la fréquence du terme t_i dans le document d_j. Cette structure permet un accès en temps presque constant $O(1)$ grâce au hachage.

2.2 Calcul de la pertinence des résultats : Le Score TF-IDF

Le classement des résultats repose sur une analyse chiffrée du texte : chaque document est converti en une série de scores permettant de mesurer sa pertinence avec la requête.

2.2.1 Le facteur TF (Term Frequency)

La fréquence d'un terme mesure l'effectif local du mot dans un document donné. Pour éviter de favoriser injustement les documents très longs, nous utilisons une normalisation :

$$\mathrm{TF}(t, d) = \frac{\text{fréquence brute de } t \text{ dans } d}{\text{nombre total de mots dans } d} \tag{2.2}$$

2.2.2 Le facteur IDF (Inverse Document Frequency)

L'IDF mesure l'importance globale du terme dans tout le groupe de documents. Un terme qui apparaît dans peu de documents est très discriminant :

$$\mathrm{IDF}(t, \mathcal{D}) = \ln\left(\frac{|\mathcal{D}|}{|\{d \in \mathcal{D} : t \in d\}|}\right) \tag{2.3}$$

L'utilisation du logarithme évite qu'un mot trop fréquent n'écrase les autres. Théoriquement, si un mot est présent dans tous les documents, son IDF devient ln(1)=0, ce qui neutralise son influence.

Cependant, dans notre implémentation, nous avons choisi d'ajouter une constante à ce calcul (typiquement +1). Cette modification garantit que même un mot présent partout conserve un score strictement positif, permettant au moteur de toujours proposer un classement aux utilisateurs au lieu d'annuler purement et simplement le résultat.

2.2.3 Calcul du Score Final

Le score de pertinence d'un document d pour une requête Q est la somme des produits TF-IDF pour chaque terme de la requête :

$$\text{Score}(Q, d) = \sum_{t \in Q} \text{TF}(t, d) \times \text{IDF}(t, \mathcal{D}) \tag{2.4}$$

3. Architecture Logicielle Détaillée

3.1 Conception en Couches Fonctionnelles (Modèle OS)

L'architecture d'un système d'exploitation est décrite comme une pile de **couches fonctionnelles** où chaque couche repose sur les services de la précédente, sans connaître les détails internes de sa voisine (principe de la *boîte noire*). La liste est, de bas en haut : le **matériel**, la **gestion des E/S** (pilotes, interruptions), la **gestion de la mémoire**, la **gestion du processeur** (ordonnancement), la **gestion des fichiers**, et enfin l'**interface utilisateur** (CLI).

Notre application reproduit fidèlement cette topologie, adaptée au domaine du moteur de recherche. Le tableau suivant établit la correspondance directe entre les différentes couches et les composants de notre projet :

Couche OS	**Couche Projet**	**Classes implémentant la couche**
Matériel / Gestion E/S	Couche I/O	`SurveillanceTempsReel`, `ProcessBuilder`, `Doublon`
Gestion de la mémoire	Couche Noyau/Données	`IndexInverse`, `CacheLRU`, `Journal`, `IdVersChemin`
Gestion des fichiers	Couche Services	`Main` (serveur), `Recherche`, `ExtracteurTexte`, `StopWord`
Interface utilisateur (CLI)	Couche Application	`Client.Main`, `Main.server()`

3.1.1 Couche 1 : Matériel et Gestion des E/S (I/O Layer)

C'est la couche la plus basse. Elle interagit directement avec le disque dur et le système d'exploitation hôte. Les périphériques d'E/S communiquent avec le noyau par **interruptions** : au lieu de scruter en permanence l'état du disque (attente active), le noyau est *notifié* par le matériel lorsqu'un événement survient.

- **Responsabilité** : Lecture des octets, détection des interruptions matérielles (E/S), exécution de processus enfants natifs.
- **Classes et Composants impliqués** :
  - `SurveillanceTempsReel` : Utilise l'API `WatchService`, qui est une abstraction Java des notifications d'événements du système de fichiers du noyau Linux. Notre thread se bloque sur `watchKey.pollEvents()` et n'est réveillé que lorsqu'un événement réel survient sur le disque, évitant toute attente active.
  - `ProcessBuilder` : Lance des exécutables natifs (`pdftotext`, `exiv2`) et récupère leur sortie standard via des *Pipes* (tubes), un mécanisme de communication interprocessus.
  - `Doublon` : Accède aux blocs de fichiers pour générer les empreintes cryptographiques MD5
  - `Écriture et persistance système` : En pilotant l'outil *exiv2* via un `ProcessBuilder`, le système modifie directement les métadonnées des fichiers images. Cette action déclenche une interruption traitée par le `WatchService`, forçant une réindexation automatique du fichier modifié.

3.1.2 Couche 2 : Noyau de Gestion des Données (Kernel/Data Layer)

Cette couche gère la mémoire vive (RAM) et la persistance. Elle joue le rôle du gestionnaire de mémoire d'un SE.

- **Responsabilité** : Maintenir l'intégrité de l'Index en RAM, gérer l'optimisation mémoire (LRU), et assurer la sécurité transactionnelle sur disque (WAL).
- **Classes impliquées** :
  - `IndexInverse` : Structure de données principale en RAM (`ConcurrentHashMap` imbriquée).
  - `StockagesDocuments` et `CacheLRU` : Gestionnaire d'optimisation mémoire basé sur un `LinkedHashMap`. Limite le nombre de documents en mémoire à 100 pour éviter la saturation du tas JVM.
  - `Journal` : Implémente le système de *Write-Ahead Logging*.
  - `IdVersChemin` : Table de correspondance virtuelle inspirée du concept d'**inode** des systèmes Unix. Un inode associe un identifiant numérique unique aux métadonnées d'un fichier ; notre `IdVersChemin` associe un entier (`int`) à un chemin absolu, permettant à l'`IndexInverse` de ne stocker que des entiers plutôt que des chaînes de caractères longues.

3.1.3 Couche 3 : Couche Services (Service Layer)

L'équivalent des appels système dans un SE. Cette couche contient la logique applicative complexe.

- **Responsabilité** : Transformer une donnée brute en information. Effectuer les calculs algorithmiques et le filtrage.
- **Classes impliquées** :
  - `Main` (méthodes statiques) : Ordonnanceur principal supervisant le `Files.walk().parallel()`.
  - `Recherche` : Exécute l'algorithme TF-IDF et la logique booléenne (ET/OU/SAUF).
  - `ExtracteurTexte` : ExtracteurTexte : Uniformise les contenus de différents formats (PDF, DOCX, JPG) en texte brut pour permettre leur traitement par le moteur.
  - `StopWord` : Filtre d'analyse sémantique, chargé depuis le fichier `stopword.txt`.

3.1.4 Couche 4 : Interface Réseau et Utilisateur (Application Layer)

La couche supérieure assure l'interface avec l'utilisateur, à l'image du Shell dans un système d'exploitation.

- **Responsabilité** : Regroupement de plusieurs connexions en une seule, dialogue TCP, formatage des réponses.
- **Classes impliquées** :
  - `Client.Main` : Interface terminal-utilisateur, gestion de la barre de progression, décodeur du protocole de téléchargement binaire.
  - `Main.server()` : Gestionnaire du `ServerSocket` et boucle d'écoute TCP multi-clients. Chaque connexion entrante génère un nouveau thread dans un `ThreadGroup` dédié.

3.2 Diagramme d'Architecture Logicielle

Le schéma ci-dessous représente l'empilement des couches. Chaque couche expose ses services à la couche supérieure uniquement, sans que celle-ci ait besoin de connaître les détails d'implémentation.

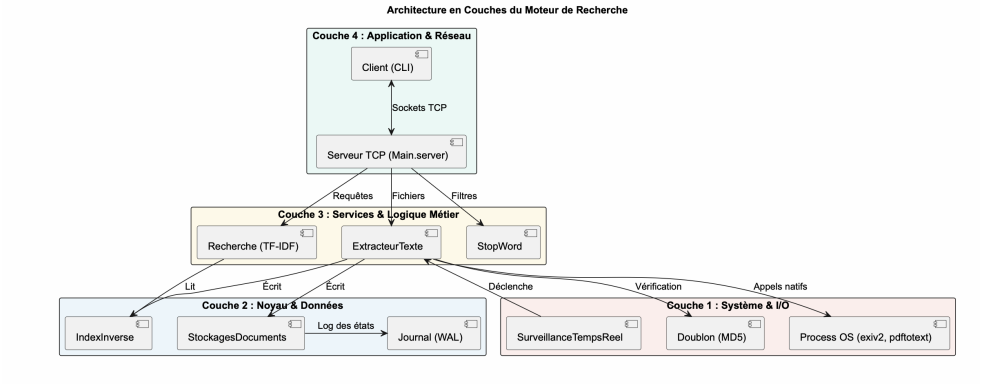

FIGURE 3.1 – Architecture en couches fonctionnelles de l'application

4. Spécification des Classes et Structures de Données

4.1 Choix des Structures de Données Centrales

L'optimisation des résultats du moteur repose sur le choix de structures adaptées à la concurrence (multi-threading) et permettant un accès en complexité presque constante $O(1)$.

4.1.1 L'Index Inversé (`ConcurrentHashMap` imbriquée)

La classe `IndexInverse` est l'élément central de la recherche. Les collections Java standard ne sont pas *thread-safe* par défaut. C'est pourquoi nous utilisons l'imbrication de maps concurrentes :

Listing 4.1 – Structure et methodes principales de l'Index Inverse

```
1  public class IndexInverse {
2      private final ConcurrentHashMap<String, ConcurrentHashMap<Integer,
           Integer>> indexInverse = new ConcurrentHashMap<>();
3
4      public void indexerMot(String mot, int idDocument) {
5          // Crée sous dico si nouveau mot
6          ConcurrentHashMap<Integer, Integer> sousDico = indexInverse.
               computeIfAbsent(mot, k -> new ConcurrentHashMap<>());
7          // Fait +1 à la fréquence si mot déjà connu dans le fichier sinon
               init à 1.
```

```
8           sousDico.merge(idDocument, 1, Integer::sum);
9       }
```

— **La Clé Primaire (`String`)** : Le mot normalisé (en minuscules).
— **La Valeur Principale** : Un sous-dictionnaire concurrent.
— **Le Sous-Dictionnaire** : Associe l'ID interne d'un document (`Integer`) à la fréquence absolue du mot dans ce document (`Integer`).

L'utilisation stricte de `ConcurrentHashMap` permet à de multiples threads (lors du scan initial `parallel()`) d'écrire simultanément dans l'index sans verrouiller la structure entière.

4.1.2 La Table d'Identifiants Virtuels (`IdVersChemin`)

Inspirée du concept d'**inode** Unix, cette classe associe un entier interne au chemin absolu de chaque fichier indexé. L'implémentation utilise une simple `ArrayList<String>` : l'identifiant d'un document est son indice dans la liste. Cette approche garantit un accès en $O(1)$ et évite de dupliquer les chaînes de caractères dans l'index.

Listing 4.2 – Structure d'IdVersChemin (table d'inodes virtuelle)

```
1  public class IdVersChemin {
2      private final ArrayList<String> idVersChemin = new ArrayList<>();
3      private int idCourant = -1;
4
5      public IdVersChemin() {
6      }
7
8      public synchronized void addPath(String path) {
9          if (!idVersChemin.contains(path)) {
10             idVersChemin.add(path);
11             idCourant = idVersChemin.size() - 1;
12         }
13     }
14
15     public int getIdCourant() {
16         return idCourant;
17     }
18
19     public synchronized String getChemin(int id) {
20         if (id >= 0 && id < idVersChemin.size()) {
21             return (String) idVersChemin.get(id);
22         }
23         return null;
24     }
```

Remarque : la classe MetaDataDocument contient désormais un `HashSet<String>tags`. L'utilisation d'un HashSet garantit l'unicité des tags pour un même fichier et permet une vérification d'existence en $O(1)$.

4.1.3 Le Cache LRU (`LinkedHashMap`)

Stocker les métadonnées de milliers de fichiers peut saturer la mémoire allouée à la JVM. La classe `CacheLRU` est un système de nettoyage automatique qui surcharge le `LinkedHashMap` de Java :

Listing 4.3 – Implementation du Cache a eviction automatique (CacheLRU.java)

```
public class CacheLRU<K, V> extends LinkedHashMap<K, V> {
    private final int capaciteMax; // Création de ma limite (modifiable
        depuis DocumentStore)

    public CacheLRU(int capacite) {
        // Le (super) c'est le constructeur de LinkedHashMap
        // Le true est pour l'ordre d'accès
        super(capacite, 0.75f, true);
        this.capaciteMax = capacite;
    }

    @Override // Réécriture
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        return size() > capaciteMax;
    }
}
```

Le paramètre `accessOrder = true` déplace dynamiquement un document consulté en fin de liste. Lorsque la `capaciteMax` (fixée à 100 dans `StockagesDocuments`) est atteinte, le système remplace automatiquement la donnée la moins récemment consultée, à la manière d'un algorithme de remplacement de pages.

4.2 Spécification des Classes Métier

4.2.1 Classe `ExtracteurTexte`

Agissant comme un traducteur universel, cette classe sélectionne dynamiquement la méthode de lecture appropriée selon l'extension du document (PDF, DOCX, JPG...). Deux méthodes d'extraction sortent du lot :

Listing 4.4 – Extrait d'ExtracteurTexte.java

```
switch (this.extension) {
    case "txt", "csv", "md", "json", "xml" -> {
        return new String(Files.readAllBytes(Paths.get(this.cheminFichier)))
            ;
    }
    case "pdf" -> {
        // Communication par Pipe avec le processus pdftotext du SE
        ProcessBuilder pb = new ProcessBuilder("pdftotext",
            this.cheminFichier, "-");
```

```
9           pb.redirectErrorStream(true);
10          Process process = pb.start();
11          String texte = new String(process.getInputStream().readAllBytes());
12          process.waitFor();
13          return texte;
14      }
15      case "html", "htm" -> {
16          String html = new String(Files.readAllBytes(Paths.get(this.
                cheminFichier)));
17          // Suppression des balises <script>, <style> puis de toutes les
                balises HTML
18          return html.replaceAll("(?s)<script.*?</script>", " ")
19                     .replaceAll("(?s)<style.*?</style>", " ")
20                     .replaceAll("<[^>]+>", " ");
21      }
22  }
23
24  (...)
```

Ce code illustre la communication par *Pipes* avec l'OS pour les PDF et les images, et le traitement des fichiers "bruts" pour extraire le contenu des archives DOCX.

4.2.2 Classe `StockagesDocuments`

`StockagesDocuments` est le registre central des métadonnées de tous les fichiers indexés. Elle encapsule un `CacheLRU` limitant à 100 entrées en mémoire, assurant le remplacement automatique des documents les moins consultés.

Listing 4.5 – Methodes principales de StockagesDocuments.java

```
1   public class StockagesDocuments {
2       private final CacheLRU<String, MetaDataDocument> stockagesDocuments =
3           new CacheLRU<>(100);
4
5       public synchronized void ajouterDocument(int id, String chemin,
6               long poids, long dateModification, long nombreTotalMots) {
7           stockagesDocuments.put(chemin,
8               new MetaDataDocument(id, chemin, poids, dateModification,
9                   nombreTotalMots)); // Ajouter les metaData d'un document au
                        stockage de tous les documents
10      }
11
12      // Acces par chemin (O(1) grace au hachage)
13      public synchronized MetaDataDocument getMetaData(String chemin) {
14          return stockagesDocuments.get(chemin);
15      }
16
17      // Acces par identifiant numerique interne (parcours lineaire O(n))
18       public MetaDataDocument getMetaDataById(int id) {
```

```
        for (MetaDataDocument metaData : stockagesDocuments.values()) {
            if (metaData.getId() == id) {
                return metaData;
            }
        }
        return null;
    }
}
```

Les méthodes `ajouterDocument`, `getMetaData` et `supprimerDocument` sont déclarées `synchronized` car elles sont appelées depuis plusieurs threads concurrents (thread d'indexation, thread du WatchService, thread client). `getMetaDataById` n'est pas synchronisée car elle est uniquement appelée depuis `Recherche`, dont l'accès est déjà sérialisé par la couche applicative.

4.2.3 Classe `StopWord`

La classe `StopWord` gère le dictionnaire des mots ignorés lors de l'indexation. Elle s'appuie sur un `HashSet<String>` pour garantir l'unicité des entrées et des vérifications d'appartenance en $O(1)$.

Listing 4.6 – Structure et methodes de StopWord.java

```
public class StopWord {
    // HashSet : unicite garantie + acces en O(1)
    private HashSet<String> words = new HashSet<>();

    public StopWord() {
        try {
            BufferedReader reader = new BufferedReader(new FileReader("src/
                main/java/Serveur/stopword.txt"));
            String line = reader.readLine();

            while (line != null) {
                this.words.add(line.trim().toLowerCase()); // Permet de
                    mettre les stop-word en minuscule (sinon ils seront ignor
                    és)
                line = reader.readLine();
            }

            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void addMot(String[] words) throws IOException {
        FileWriter writer = new FileWriter("src/main/java/Serveur/stopword.
            txt", true);
        for (String word : words) {
```

```
24              String motPropre = word.trim().toLowerCase(); // Passe tout en
                    minuscule et enlève les espaces inutiles
25              writer.write(motPropre + "\n");
26              this.words.add(motPropre);
27          }
28          writer.close();
29      }
30
31        public void removeMot(String[] words) throws IOException {
32          for (String word : words) {
33              this.words.remove(word.trim().toLowerCase());
34          }
35
36          FileWriter writer = new FileWriter("src/main/java/Serveur/stopword.
                txt", false);
37          for (String motRestant : this.words) {
38              writer.write(motRestant + "\n");
39          }
40          writer.close();
41      }
42  }
```

L'ajout d'un mot-clé s'effectue en mode `append(true)`, ce qui permet une écriture rapide en fin de fichier sans modifier les données existantes. À l'inverse, la suppression impose une réécriture intégrale du fichier (`append(false)`) afin d'en retirer physiquement l'entrée, suivie d'une réindexation totale pour maintenir la cohérence du système.

4.2.4 Le Journal Transactionnel : Modèle Producteur-Consommateur

La logique **Producteur-Consommateur** est l'un des grands modèles de coordination entre processus concurrents. Dans ce modèle, un *buffer borné* (bounded-buffer) sert de zone tampon entre des threads qui produisent des données et un thread qui les consomme. La synchronisation est assurée par les primitives `wait()` et `notifyAll()` sur l'objet partagé (moniteur Java).

Notre classe `Journal` applique exactement ce modèle : Le système de persistance repose sur un modèle Producteur-Consommateur : les threads d'indexation (Producteurs) insèrent les opérations dans une file d'attente en mémoire (`fileOperations`), tandis qu'un thread dédié, le `Thread-Journal-Ecrivain` (Consommateur), assure l'écriture physique dans le fichier `journal.csv`. Ce mécanisme utilise un tampon borné (fixé par `CAPACITE_MAX = 100`) pour réguler le flux entre la mémoire vive et le stockage disque.

Ce thread est configuré en mode **démon** (`setDaemon(true)`) et avec une priorité minimale (`setPriority(Thread.MIN_PRIORITY)`). Un thread démon ne bloque pas l'arrêt de la JVM : si le programme se termine, ce thread s'arrête automatiquement sans nécessiter de terminaison explicite.

Listing 4.7 – Mecanisme Producteur-Consommateur dans Journal.java

```java
// --- Côté PRODUCTEUR (threads d'indexation) ---
private void ajouterOperation(String operation) {
    synchronized (fileOperations) {
        // Attente passive si le buffer est plein (evite le debordement)
        while (fileOperations.size() >= CAPACITE_MAX && actif) {
            try { fileOperations.wait(); } catch (InterruptedException e) {
               return; }
        }
        if (!actif) return;
        fileOperations.add(operation);
        fileOperations.notifyAll(); // Reveille le thread Consommateur
    }
}

// --- Côté CONSOMMATEUR (Thread-Journal-Ecrivain) ---
synchronized (fileOperations) {
    while (fileOperations.isEmpty() && actif) {
        try { fileOperations.wait(); } catch (InterruptedException e) {
           return; }
    }
    if (!actif && fileOperations.isEmpty()) break;
    operation = fileOperations.remove(0);
    fileOperations.notifyAll();
}
// Ecriture sur le disque
if (operation != null) {
    try {
        synchronized (Journal.this) {
            writer.write(operation);
            writer.newLine();
            writer.flush();
            }
        } catch (IOException e) {
            e.printStackTrace();
            }
    }
```

Les appels `wait()` et `notifyAll()` assurent une exclusion mutuelle parfaite sans attente active (boucle *while(true)* vide). L'écriture physique sur le disque est effectuée en dehors du bloc `synchronized` pour ne pas bloquer les producteurs pendant une opération I/O potentiellement longue.

Pourquoi un fichier CSV ? Le format CSV présente deux avantages concrets. Il est **lisible directement** dans n'importe quel éditeur de texte, ce qui facilite le débogage après un crash. Il est aussi **indépendant du code** : le fichier reste exploitable quelle que soit l'évolution des classes Java, sans risque de corruption lié à un changement de structure.

Gestion de la persistance et intégrité du TF-IDF : L'intégration de l'annotation manuelle a révélé un défi technique lié à la compaction du journal (WAL). Lors de la restauration classique, réinsérer un mot dans l'index augmentait artificiellement sa fréquence absolue, faussant le score TF-IDF après plusieurs redémarrages. Pour garantir la robustesse mathématique du moteur, nous avons conçu une architecture de journalisation à deux états : la commande TAG (qui agit sur le calcul sémantique en direct) et la commande `TAG_VISUEL` (générée uniquement lors de la compaction, qui restaure l'interface utilisateur et les étiquettes de métadonnées sans déclencher de ré-indexation).

4.2.5 Diagramme de Classes UML

Le diagramme ci-dessous présente les principales relations entre les classes du projet. `Main` (Serveur) orchestre la création de tous les composants. `Recherche` et `SurveillanceTempsReel` tiennent des références directes sur les structures de données partagées.

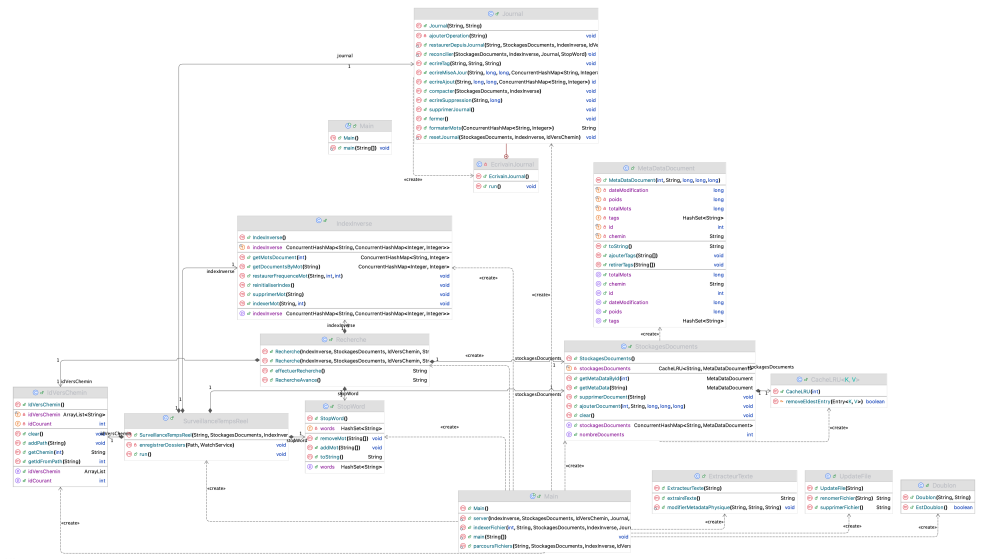

FIGURE 4.1 – Diagramme de classes UML simplifié

4.3 Justification de l'Utilisation des `static`

4.3.1 Le Point d'Entrée Java (obligatoire)

— **Où ?** : `public static void main(String[] args)`

— **Pourquoi ?** : La JVM doit lancer le programme avant qu'aucun objet n'existe en mémoire. La méthode de démarrage appartient donc à la classe elle-même, et non à une instance.

4.3.2 Les Constantes (économie mémoire)

— **Où ?** : `ANSI_BLEU`, `ANSI_VERT`, `ANSI_RESET` dans plusieurs classes ; `CAPACITE_MAX = 100` dans `Journal`.

— **Pourquoi ?** : Comme vu dans le cours, chaque instance d'un objet occupe sa propre zone dans le tas (heap) de la JVM. Sans `static`, chaque objet `Recherche` créé contiendrait sa propre copie des constantes de couleur ANSI. Le mot-clé `static` rattache la variable à la **classe elle-même** : une seule copie existe en mémoire, partagée par toutes les instances. C'est le même principe que la zone de données partagée entre un processus père et ses threads fils (Chapitre 2).

4.3.3 Les Méthodes Utilitaires Sans État

— **Où ?** : `Main.parcoursFichiers()`, `Main.indexerFichier()`, `Main.server()`.

— **Pourquoi ?** : La classe `Main` n'est jamais instanciée avec `new Main()`. Ces méthodes n'ont pas d'état interne propre : elles reçoivent tout ce dont elles ont besoin en paramètre (`indexInverse`, `journal`, etc.) et travaillent dessus. Les déclarer `static` reflète cette absence d'état.

4.3.4 Les Méthodes Appelées avant Toute Instanciation

— **Où ?** : `Journal.restaurerDepuisJournal()`, `Journal.reconcilier()`, `Journal.resetJournal()`.

— **Pourquoi ?** : Ces méthodes sont invoquées au démarrage du programme, avant que l'objet `Journal` ne soit construit avec `new Journal()`. En termes de systèmes d'exploitation, c'est le même principe qu'une mise en route du noyau qui s'exécute avant que les processus utilisateurs puissent démarrer. Une méthode qui doit être appelée avant la naissance de son objet doit obligatoirement être `static`.

4.3.5 Le Booléen d'Affichage pour le Débogage

— **Où ?** : `public static boolean DEBUG = false;` dans `Main.java`.

— **Pourquoi ?** : C'est un interrupteur global de débogage. Le rendre `static` (sans `final`) permet de l'activer ou de le désactiver depuis n'importe quel endroit du programme, sans passer le booléen en paramètre dans chaque fonction. Son absence de `final` est intentionnelle : il peut être modifié dynamiquement.

5. Description des Algorithmes Principaux

5.1 Algorithme de Recherche et de Tri (`-s`)

La commande de recherche représente le point de convergence entre les structures de données (Index Inversé) et les calculs de pertinence (TF-IDF). Son rôle est de transformer une requête textuelle en une liste ordonnée de fichiers.

5.1.1 Traitement de la Requête et Calcul du Score

Lorsqu'une recherche est lancée, le système décompose la chaîne de caractères en mots-clés. Pour chaque document contenant au moins un de ces mots, un *score de pertinence* est calculé :

- Le moteur interroge l'Index Inversé pour récupérer le poids numérique de chaque mot-clé dans le document cible.
- Ces poids sont cumulés pour former le « profil numérique » du document vis-à-vis de la requête. Plus un mot rare (IDF élevé) est présent fréquemment (TF élevé) dans un fichier, plus ce dernier remonte dans le classement.

5.1.2 Optimisation par Cache (LRU)

Pour garantir une haute performance, la classe `Recherche` interagit avec le `CacheLRU` avant tout calcul intensif.

- **Lecture :** Si la requête a déjà été effectuée récemment, les résultats sont instantanément extraits de la mémoire vive sans recalculer les scores.
- **Écriture :** Chaque nouvelle recherche est mémorisée dans la RAM. Si la capacité maximale est atteinte, l'algorithme *Least Recently Used* supprime la recherche la plus ancienne pour libérer de la place.

5.1.3 Ordonnancement et Affichage

Une fois les scores obtenus, les documents sont triés par ordre décroissant de pertinence. Le système utilise un comparateur personnalisé pour traiter les listes de résultats. Enfin, le chemin absolu de chaque fichier est renvoyé au client, accompagné de son score, permettant à l'utilisateur d'identifier immédiatement les documents les plus proches de son besoin.

5.2 L'Algorithme d'Évaluation Sémantique : TF-IDF

La classe `Recherche` ne se contente pas d'un filtrage binaire (présence/absence). Elle mesure la qualité des résultats via la formule TF-IDF.

5.2.1 Modélisation et Implémentation

Soit N le nombre total de documents. Pour chaque mot de la requête, on calcule :

$$\text{IDF}(t) = \ln\left(\frac{N}{\text{Nombre de documents contenant } t}\right) \tag{5.1}$$

$$\text{TF}(t, d) = \frac{\text{Occurrences de } t \text{ dans } d}{\text{Nombre total de mots de } d} \tag{5.2}$$

Listing 5.1 – Algorithme TF-IDF dans effectuerRecherche() (Recherche.java)

```
// Calcul IDF
int nbDocsAvecMot = indexDuMot.size();
double idf = Math.log((double) totalDocs / nbDocsAvecMot);

for (Integer id : indexDuMot.keySet()) {
    boolean aExclure = false;
    for (String motExclu : motsNonRecherches) {
        if (indexInverse.getDocumentsByMot(motExclu).containsKey(id)) {
            aExclure = true; break;
        }
    }
    if (!aExclure) {
        MetaDataDocument metaData = stockagesDocuments.getMetaDataById(id);
        if (metaData == null) continue;
        double tf = (double) indexDuMot.get(id) / metaData.getTotalMots();
        double score = tf * idf;


        String chemin = idVersChemin.getChemin(id);
        scoresParChemin.merge(chemin, score, Double::sum);
    }
}
List<Map.Entry<String, Double>> listeTriee =
    new ArrayList<>(scoresParChemin.entrySet());
listeTriee.sort(Map.Entry.<String, Double>comparingByValue().reversed());
```

Le tri final est réalisé par `sort()` avec un comparateur inversé sur les valeurs, et non par `TreeMap`. Ce choix est délibéré : un `TreeMap` trierait par clé (le chemin) et non par valeur (le score). On utilise donc une `ArrayList` triée, plus flexible.

L'opérateur fonctionnel `merge(..., Double::sum)` garantit une fusion inséparable et mathématiquement correcte des scores des différents termes d'une requête multi-mots.

5.3 L'Algorithme de Recherche Avancée

Pour répondre aux requêtes complexes de type `et`, `ou`, `sauf`, nous avons implémenté un algorithme en utilisant des `HashSet`.

Listing 5.2 – Algorithme de Recherche Avancée (Recherche.java)

```
public String RechercheAvance() {
    HashSet<Integer> resultat = new HashSet<>();
    HashSet<Integer> resultatFinal = new HashSet<>();
    String operateur = "ou";
    boolean premierMot = true;

    for (String mot : motsRecherches) {
        if (mot.equals("et") || mot.equals("ou") || mot.equals("sauf")) {
            operateur = mot;
            continue;
        }
        if (premierMot) {
            resultatFinal.addAll(indexInverse.getDocumentsByMot(mot).keySet
                ());
            premierMot = false;
        } else {
            resultat.addAll(indexInverse.getDocumentsByMot(mot).keySet());
            if (operateur.equals("et"))   resultatFinal.retainAll(resultat);
            if (operateur.equals("ou"))   resultatFinal.addAll(resultat);
            if (operateur.equals("sauf")) resultatFinal.removeAll(resultat);
            operateur = "ou";
            resultat.clear();
        }
    }
    // ... Formatage et renvoi de la reponse
}
```

Remarque importante : Les opérateurs sont traités en minuscules dans le code (`et`/`ou`/`sauf`). Lors de la saisie d'une commande `-ar`, les mots passent par `toLowerCase()` dans le constructeur de `Recherche`, ce qui rend la recherche insensible à la casse. Ainsi, `"-ar java ET reseau"` et `"-ar java et reseau"` produisent le même résultat.

Cet algorithme est particulièrement performant car il manipule uniquement des identifiants numériques (`Integer`) en RAM, sans jamais lire les fichiers sur le disque.

5.4 Détection de Doublons (via un checksum)

La classe `Doublon` détecte si deux fichiers sont des copies identiques grâce au hachage **MD5**.

Listing 5.3 – Generation d'une empreinte MD5 (Doublon.java)

```
public boolean EstDoublon() throws IOException, NoSuchAlgorithmException {
```

```
    byte[] dataFile1 = Files.readAllBytes(Paths.get(f1));
    byte[] hash1 = MessageDigest.getInstance("MD5").digest(dataFile1);
    String checksum1 = new BigInteger(1, hash1).toString(16);

    byte[] dataFile2 = Files.readAllBytes(Paths.get(f2));
    byte[] hash2 = MessageDigest.getInstance("MD5").digest(dataFile2);
    String checksum2 = new BigInteger(1, hash2).toString(16);

    return checksum1.equals(checksum2);
}
```

La probabilité de collision du MD5 est négligeable dans un contexte local. L'égalité stricte entre deux `checksum` garantit que deux documents sont des copies exactes.

Limite identifiée et assumée : L'implémentation actuelle utilise `Files.readAllBytes()` qui charge l'intégralité du fichier en mémoire. Pour des fichiers de taille modérée (documents texte, PDF, images), ce comportement est acceptable dans le cadre de notre application. Une évolution serait d'utiliser un `DigestInputStream` couplé à un `BufferedInputStream` pour calculer le hash par flux sans jamais charger plus qu'un tampon en RAM, conformément au modèle de flux présenté dans le cours.

5.5 Surveillance Temps Réel et Gestion du Debouncing

La classe `SurveillanceTempsReel` implémente l'interface `Runnable` et tourne dans un thread démon de priorité minimale. Son algorithme central repose sur le blocage passif du thread jusqu'à réception d'un événement :

Listing 5.4 – Boucle principale du WatchService (SurveillanceTempsReel.java)

```
while (true) {
    WatchKey info = watchservice.take(); // Boucle infinie, mais avec une mé
        thode bloquante

    for (WatchEvent<?> action : info.pollEvents()) {
        WatchEvent.Kind<?> typeAction = action.kind(); // Type d'évènements
        if (typeAction == StandardWatchEventKinds.OVERFLOW) continue;

        Thread.sleep(150);
        //(...)
        if (typeAction == StandardWatchEventKinds.ENTRY_CREATE) { /* ... */
            }
        else if (typeAction == StandardWatchEventKinds.ENTRY_MODIFY) {
            int vraiId = meta.getId();
            ConcurrentHashMap<String, Integer> anciensMots =
                indexInverse.getMotsDocument(vraiId);
            for (String mot : anciensMots.keySet()) {
                indexInverse.getDocumentsByMot(mot).remove(vraiId);
            }
```

```
18              Main.indexerFichier(vraiId, cheminComplet, ...);
19          }
20          else if (typeAction == StandardWatchEventKinds.ENTRY_DELETE) { /*
                ... */ }
21      }
22      info.reset();
23  }
```

5.6 Distinction entre Métadonnées Internes et Métadonnées Physiques

Le moteur propose deux types d'accès aux métadonnées d'un fichier, qui répondent à des besoins fondamentalement différents.

5.6.1 Métadonnées internes du moteur (`-m`)

Lors de l'indexation d'un fichier, la méthode `indexerFichier()` calcule et stocke dans `MetaDataDocument` un ensemble de métriques propres au moteur :

Listing 5.5 – Stockage des metadonnees internes lors de l'indexation

```
1  stockagesDocuments.ajouterDocument(
2      id,
3      cheminFichier,
4      file.length(),
5      file.lastModified(),
6      nbMots
7  );
```

Ces données n'existent que dans deux endroits : la RAM du serveur (via `StockagesDocuments`) et le fichier `journal.csv`. Elles sont **calculées par le moteur** au moment de l'indexation et ne sont pas inscrites dans le fichier lui-même.

5.6.2 Métadonnées physiques du fichier (`-exif`)

La commande `-exif` délègue entièrement la lecture à l'outil système `exiv2` via un `ProcessBuilder`. Ces métadonnées sont inscrites dans le fichier image lui-même selon les standards **EXIF** (données techniques de l'appareil photo) et **IPTC** (données éditoriales comme la légende ou les mots-clés) :

Listing 5.6 – Lecture des metadonnees physiques via exiv2

```
1  case "jpg", "jpeg", "png" -> {
2      ProcessBuilder processBuilder = new ProcessBuilder("exiv2", "-pa",
3          this.cheminFichier);
4      processBuilder.redirectErrorStream(true);
```

```
5       Process process = processBuilder.start();
6       String texteExtrait = new String(process.getInputStream().readAllBytes()
            );
7       process.waitFor();
8       return texteExtrait;
9   }
```

Ces métadonnées existent **indépendamment du moteur** : elles sont lisibles par n'importe quel logiciel compatible (Lightroom, Preview, ExifTool). La commande `-exif -set` permet même d'écrire physiquement dans ces métadonnées via
`exiv2 -M "set Iptc.Application2.Caption..."`.

5.6.3 Tableau récapitulatif

	`-m`	`-exif`
Source	RAM du moteur (`MetaDataDocument`)	Fichier physique (EXIF/IPTC)
Données	Id interne, taille, date OS, nb mots	Appareil, résolution, GPS, légende
Persistance	RAM + `journal.csv`	Dans le fichier lui-même
Dépendance	Propre au moteur	Indépendant du moteur
Modification	Non modifiable via CLI	Modifiable via `-exif -set`

5.7 Algorithme d'Annotation Dynamique (Commande `-tag`)

La commande `-tag` répond à un besoin spécifique du cahier des charges : enrichir des fichiers qui ne contiennent pas de texte exploitable, comme des images `.jpg`. L'index inversé ne peut extraire aucun mot signifiant du contenu binaire de ces fichiers. L'annotation manuelle permet à l'utilisateur de pallier cette limite en attachant des mots-clés arbitraires directement aux métadonnées internes d'un document. La commande accepte trois sous-commandes :

- `-tag -add <chemin> <mot1,mot2>` : associe des mots-clés au fichier et les injecte dans l'index.
- `-tag -rm <chemin> <mot1,mot2>` : retire des mots-clés et les supprime de l'index.
- `-tag -l <chemin>` : liste les tags actuellement associés au fichier.

5.8 Algorithme d'Annotation et Intégrité du TF-IDF

La gestion de la persistance des tags a nécessité une attention particulière pour préserver l'intégrité de l'Index Inversé.

Commande TAG : Utilisée lors d'une action utilisateur en direct. Elle enregistre le mot dans le journal et incrémente sa fréquence dans l'index.

Commande TAG_VISUEL : Générée exclusivement lors de la compaction du journal (`compacter()`). Elle permet de restaurer l'affichage des étiquettes au redémarrage sans ré-indexer le mot, évitant ainsi de doubler artificiellement la fréquence du terme et de fausser les scores de pertinence.

6. Protocole Réseau et Interface Client

6.1 La Topologie Client-Serveur et les ThreadGroups

Le moteur écoute sur le port TCP 12345. La méthode `Main.server()` instancie un `ServerSocket` bloquant. À chaque connexion entrante (`accept()`), un nouveau thread anonyme est créé. Ces threads sont regroupés dans un `ThreadGroup` nommé `"Clients-connectes"`, un mécanisme de gestion collective des threads qui permet notamment de connaître le nombre de clients connectés via `clientsGroup.activeCount()`.

Listing 6.1 – Creation d'un thread par client avec ThreadGroup

```
1 ThreadGroup clientsGroup = new ThreadGroup("Clients-connectes");
2 new Thread(clientsGroup, () -> {
3     // Traitement du client...
4 }, "Client-" + clientsGroup.activeCount()).start();
```

Ce principe permet au serveur de gérer plusieurs terminaux clients simultanément, de manière préemptive.

6.2 Conception des commandes textuelles (CLI)

Peu importe la puissance du moteur, ce qui compte pour l'utilisateur, c'est que l'application réponde vite et soit simple à utiliser.

6.2.1 La Bibliothèque JLine 3

L'intégration de **JLine 3** transforme une console statique en un terminal intelligent.

— **Historique et navigation** : Les flèches directionnelles permettent de naviguer dans les anciennes requêtes.

— **Lecture non-bloquante** : JLine facilite l'interruption d'une commande par `Ctrl+C` sans faire planter le socket réseau.

6.3 Le Téléchargement de Fichier (`-dl`)

Le téléchargement de fichier est le défi majeur de cette couche. Un flux TCP textuel (`DataInputStream.readUTF()`) ne peut pas transmettre un fichier `.pdf` ou `.jpg` sans corrompre l'encodage.

Pour pallier cela, nous avons implémenté un protocole de "Bascule de Mode" piloté par le serveur :

1. Le serveur envoie un signal de contrôle contenant la taille exacte du fichier : `"File_incomming... <taille> <nom>"`
2. Le client abandonne temporairement sa boucle de lecture texte (`readUTF()`) et capte le flux brut `InputStream`.
3. Les octets sont reçus par fragments de 4 Ko.

Listing 6.2 – Bascule vers le flux binaire coté Client (Client/Main.java)

```
} else if (reponse.startsWith("File_incomming...")) {
    String[] donnees = reponse.split(" ", 3);
    long taille_fichier = Long.parseLong(donnees[1]);
    String nomFichier = donnees[2];

    System.out.println(ANSI_VERT + "Début du téléchargement : " + ANSI_BLEU
        + nomFichier + ANSI_RESET);
    String racine = System.getProperty("user.home"); // permet de récupérer
        le dossier racine de l'utilisateur
    File dossier = new File(racine, "Downloads"); // on se met ensuite sur
        le dossier des téléchargements
    dossier.mkdirs();
    String chemin_sauvegarde = dossier.getAbsolutePath()
        + File.separator + nomFichier; // Construit le chemin absolu final
            avec File.separator pour gérer les \ ou /

    FileOutputStream ecriture = new FileOutputStream(chemin_sauvegarde);
    byte[] buffer = new byte[4096]; // Chunks de 4 Ko
    long total_lu = 0;
    int quantite_lu;

    // Boucle bornee par la taille exacte declaree par le serveur
    while (total_lu < taille_fichier
        && (quantite_lu = in.read(buffer, 0,
            (int) Math.min(buffer.length, taille_fichier - total_lu))) !=
                -1) {
        ecriture.write(buffer, 0, quantite_lu);
        total_lu += quantite_lu;
        // Affichage de la barre de progression
    }
    ecriture.close();
}
```

La technique de fragmentation par chunks de 4 Ko protège la mémoire vive : le client peut télécharger des fichiers de plusieurs gigaoctets avec seulement quelques Ko de RAM consommés à tout instant. `File.separator` assure la portabilité entre Linux (/) et Windows (\).

7. Jeu de Tests et Validation

7.1 Environnement de Test

Le répertoire expérimental (`src/testIndexed`) a été créé avec une forte hétérogénéité pour couvrir tous les cas d'usage :

- **Fichiers .txt** : Modifiés en temps réel par des éditeurs de texte (Vim, Notepad++).
- **Fichiers .docx / .pdf** : Documents volumineux sollicitant la décompression ZIP et les *Pipes*.
- **Fichiers .jpg** : Photographies numériques avec métadonnées EXIF.

7.2 Traces d'Exécution et Résultats

7.2.1 Test 1 : Cohérence du Calcul TF-IDF

Objectif : S'assurer que le moteur de pertinence discrimine correctement les mots rares des mots communs.

- **Requête** : `-s coucou`
- **Console Client** :

```
> -s coucou
Document: src/testIndexed/coucou2.txt, Score TF-IDF: 0.11507282898071235
Document: src/testIndexed/coucou.txt, Score TF-IDF: 0.09589402415059362
Document: src/testIndexed/test.txt, Score TF-IDF: 0.019178804830118724
Document: src/testIndexed/millionsWords.txt, Score TF-IDF: 5.99356380828E-7
END_OF_MESSAGE
```

Interprétation d'un score tendant vers zéro : En conclusion, un score TF-IDF très proche de zéro indique une pertinence marginale du document vis-à-vis de la requête. Sur le plan analytique, cette situation se présente sous deux conditions : soit le terme recherché est dilué dans un document excessivement long (facteur TF très faible), soit il s'agit d'un mot extrêmement banal présent dans la quasi-totalité du corpus (facteur discriminant IDF proche de zéro). Néanmoins, la préservation et l'affichage de cette valeur infinitésimale sont fondamentaux. Contrairement à une annulation pure et simple du score, cela garantit au moteur de recherche la capacité de maintenir un ordonnancement strict et complet des résultats, offrant ainsi une réponse déterministe à l'utilisateur, même pour les requêtes les plus génériques.

7.2.2 Test 2 : Maintien de l'État après Crash

Objectif : Vérifier la tolérance aux pannes du serveur.

— **Action** : Arrêt brutal (`kill -9`) du processus Serveur.

— **Vérification du fichier** `journal.csv` :

```
PATH;src/testIndexed
AJOUT;src/testIndexed/Test_Exiv.jpg;1774599052324;1127703;jpg:1,image:2,
    thumbnail:1,test:1,src:1,mime:1,testindexed:1,none:1,type:1,exiv:1,file
    :2,size:2,2323:1,1200:1,bytes:1,x:1,name:1,jpeg:1,1127703:1
AJOUT;src/testIndexed/test.txt;1774599052312;85;a:1,coucou:1,capte:1,il:1,c
    :1,documents:1,est:1,tester:1,mot:1,voir:1,si:1,y:1,quand:1,plusieurs:1
...
```

— **Redémarrage** : Relance de `Main.java`.

— **Console Serveur** : `"Restauration depuis journal.csv : 8 documents rechargés, pas de réindexation"`

Conclusion : Succès. Le buffer `IdVersChemin` et les tables de l'Index Inversé sont remontés en RAM en quelques millisecondes, sans relire physiquement les fichiers sur le disque.

7.2.3 Test 3 : Recherche Avancée Booléenne

Objectif : Vérifier le fonctionnement des opérateurs ensemblistes.

```
 > -ar java ET coucou
src/testIndexed/millionsWords.txt

 > -ar coucou SAUF java
src/testIndexed/test.txt
src/testIndexed/coucou.txt
src/testIndexed/coucou2.txt
```

Conclusion : Succès. L'intersection (`et`) et la différence (`sauf`) fonctionnent correctement. Note : les opérateurs sont insensibles à la casse grâce à la normalisation en minuscules dans le constructeur de `Recherche`.

7.2.4 Test 4 : Téléchargement Binaire (`-dl`)

Objectif : Vérifier l'intégrité d'un fichier binaire après transfert.

```
-dl src/testIndexed/CC-MATH4B-mars25.pdf
Début du téléchargement : CC-MATH4B-mars25.pdf
Progression : [################] 100%
Téléchargement terminé ! (394409 octets en 5 ms)
```

Conclusion : Succès. Le fichier téléchargé est identique à l'original (vérification par comparaison MD5 avec la commande `-d`).

7.2.5 Test 5 : Annotation Dynamique et Recherche par Tags (`-tag`)

Objectif : Vérifier qu'un fichier image, non indexable par son contenu binaire, devient trouvable après annotation manuelle, et qu'il disparaît des résultats après retrait du tag.

```
> -tag -add src/testIndexed/Test_Exiv.jpg evaluation
Tags ajoutés avec succès !

 > -s evaluation
Document: src/testIndexed/Test_Exiv.jpg, Score TF-IDF: 0.09902102579427789 [
    Tags: evaluation]
Document: src/testIndexed/millionsWords.txt, Score TF-IDF:
    1.4441017543463758E-6


 > -tag -rm src/testIndexed/Test_Exiv.jpg evaluation
Tags retirés avec succès !

 > -s evaluation
Document: src/testIndexed/millionsWords.txt, Score TF-IDF:
    2.1661526315195636E-6
```

Analyse des résultats :

- **Après** `-tag -add` : `Test_Exiv.jpg` apparaît en première position avec un score TF-IDF de 0.073. Ce score non nul confirme que le mot "evaluation" a bien été injecté dans l'`IndexInverse` et qu'il est présent dans peu de documents (IDF élevé).
- `millionsWords.txt` **à 2.1661526315195636 E -6** : le fichier contient le mot "evaluation" mais dans une proportion infime par rapport à sa taille totale (plusieurs millions de mots), ce qui donne un TF quasi nul. C'est le comportement attendu de la normalisation TF.
- **Après** `-tag -rm` : `Test_Exiv.jpg` disparaît complètement des résultats. La suppression ciblée de l'identifiant dans le sous-dictionnaire du mot a bien fonctionné, sans affecter les autres documents qui contiendraient ce terme.

Conclusion : Succès. Ce test valide l'ensemble de la chaîne d'annotation : résolution de l'identifiant interne, injection dans l'index, et cohérence immédiate des résultats de recherche.

7.2.6 Test 6 : Modification Physique des Métadonnées EXIF (`-exif -set`)

Objectif : Vérifier que la commande `-exif -set` inscrit physiquement une description dans le fichier image, et qu'elle est immédiatement lisible via `-exif`.

```
> -exif -set src/testIndexed/Test_Exiv.jpg "Notes Info4A"
Description incrustee physiquement dans l'image avec succes !

> -exif src/testIndexed/Test_Exiv.jpg
--- Donnees EXIF/IPTC de l'image ---
```

```
Exif.Image.Orientation      Short     1   top, left
Exif.Image.XResolution      Rational  1   72
Exif.Image.YResolution      Rational  1   72
Exif.Image.ResolutionUnit   Short     1   inch
Exif.Photo.PixelXDimension  Long      1   1200
Exif.Photo.PixelYDimension  Long      1   2323
Iptc.Application2.Caption   String   12   Notes Info4A
```

Analyse : La ligne `Iptc.Application2.Caption` confirme que la valeur "Notes Info4A" a bien été écrite dans le fichier image via `exiv2`. Cette modification est **persistante** : elle survit à un redémarrage du serveur et reste visible dans n'importe quel logiciel de visualisation d'images compatible IPTC.

Distinction importante entre `-m` et `-exif` : Ces deux commandes donnent des informations sur un fichier, mais leur nature est fondamentalement différente :

- `-m <chemin>` retourne les métadonnées **internes au moteur**, stockées dans `MetaDataDocument` lors de l'indexation (identifiant interne, taille en octets, date de dernière modification, nombre de mots indexés). Ces données n'existent que dans la mémoire du serveur et dans `journal.csv` — elles sont propres au moteur.
- `-exif <chemin>` lit les métadonnées **physiquement inscrites dans le fichier** (données EXIF/IPTC via `exiv2`). Ces informations existent indépendamment du moteur, dans le fichier lui-même, et sont exploitables par n'importe quel autre logiciel.

Conclusion : Succès. La chaîne de modification est complète : appel `ProcessBuilder` → `exiv2` → écriture physique dans le fichier → lecture immédiate confirmée. Le `WatchService` détecte également la modification du fichier et déclenche une ré-indexation automatique, mettant à jour les métadonnées internes du moteur en conséquence.

7.2.7 Test 7 : Persistance des Tags après Compaction du Journal (`-clean`)

Objectif : Vérifier que les tags survivent à une compaction du journal WAL. C'est un test de cohérence critique : la commande `-clean` réécrit entièrement `journal.csv` en ne gardant que l'état actuel de chaque document. Si les tags ne sont pas correctement sérialisés lors de cette réécriture, ils seraient perdus définitivement.

```
 > -tag -add src/testIndexed/Test_Exiv.jpg chat
Tags ajoutés avec succès !

 > -s chat
Document: src/testIndexed/Test_Exiv.jpg, Score TF-IDF: 0.09902102579427789 [
   Tags: chat]
Document: src/testIndexed/millionsWords.txt, Score TF-IDF:
   1.4441017543463758E-6


 > -clean
```

```
Journal compacté avec succès.

 > -s chat
Document: src/testIndexed/Test_Exiv.jpg, Score TF-IDF: 0.09902102579427789 [Tags: chat]
Document: src/testIndexed/millionsWords.txt, Score TF-IDF: 1.4441017543463758E-6
```

Conclusion : Succès. Ce test confirme que l'annotation manuelle est une fonctionnalité pleinement résiliente : elle résiste non seulement aux crashes (Test 2, WAL Replay) mais également aux opérations de maintenance courantes comme la compaction du journal.

8. Répartition des Tâches et Organisation du Travail

8.1 Méthodologie et Outils Collaboratifs

Le projet a été mené par un trinôme dont l'effectif a nécessité une organisation rigoureuse pour justifier la dérogation accordée par M. Leclercq. L'ensemble du code source a été hébergé sur un dépôt **Git** privé (GitHub). Nous avons adopté un workflow structuré : chaque développeur travaillait sur une branche dédiée à sa fonctionnalité (`feature/journal-wal`, `feature/recherche-avancee`, `feature/watchservice`, etc.) avant de soumettre une *pull request* relue par un autre membre du groupe.

Cette contrainte organisationnelle, plus exigeante qu'en binôme, nous a confrontés à des conflits de fusion réels sur les fichiers partagés comme `IndexInverse.java` et `Main.java`. Ces conflits, résolus manuellement en équipe, ont représenté une charge de travail supplémentaire authentiquement formatrice.

Recherche documentaire et fondements théoriques La réalisation de ce projet s'est prioritairement appuyée sur les enseignements dispensés dans le cadre du module Info4B. Les supports de cours et de travaux dirigés ont constitué notre base de connaissances principale, notamment pour la maîtrise des concepts de synchronisation (modèle Producteur-Consommateur, moniteurs Java), la gestion avancée des flux d'entrées/sorties (NIO.2) et la programmation réseau (Sockets TCP).

Bien que la documentation officielle Java (Javadoc) et diverses ressources sur Internet (forums de développement, articles techniques) aient été consultées de manière complémentaire — principalement pour appréhender la syntaxe de bibliothèques externes comme `JLine 3` ou pour résoudre des erreurs d'implémentation spécifiques —, l'architecture globale et les choix algorithmiques découlent directement des paradigmes étudiés en cours. Cette démarche nous a permis

de garantir une cohérence stricte entre la théorie académique enseignée et notre implémentation technique.

8.2 Tableau de Répartition par Fonctionnalité

Fonctionnalité	**Description technique**	**Responsable**
Architecture globale	Conception des 4 couches, intégration de tous les composants, configuration Maven, script `run.sh`	Antoine
Interface Client (CLI)	`Client/Main.java` : intégration JLine 3, protocole de bascule binaire pour `-dl`, barre de progression, gestion de toutes les commandes	Antoine et Arnaud et Benjamin
Gestion réseau	`Client/Main.java` : Initialisation de la `Socket`, flux `DataInput/OutputStream`, protocole de téléchargement de fichiers avec barre de progression dynamique et gestion du buffer	Antoine et Arnaud
Surveillance temps réel	`SurveillanceTempsReel.java` WatchService (NIO.2), enregistrement récursif des sous-dossiers	Arnaud
Extraction multi-format	`ExtracteurTexte.java` : extraction du texte pour TXT/CSV/MD/JSON/XML/PDF/-DOCX/HTML/JPG, communication par Pipes avec `pdftotext` et `exiv2`	Antoine et Arnaud

Index Inversé	`IndexInverse.java` : `ConcurrentHashMap` imbriquée, méthodes `indexerMot()`, `getMotsDocument()`, `restaurerFrequenceMot()`	Antoine
Table d'identifiants	`IdVersChemin.java` : gestion de la correspondance ID interne ↔ chemin absolu (inodes virtuelles)	Antoine
Métadonnées documents	`MetaDataDocument.java`, `StockagesDocuments.java`, `UpdateFile.java` `Développement du système d'étiquetage dynamique (-tag) et de l'affichage contextuel des tags lors des recherches`	Antoine
Gestion des Stop-Words	`StopWord.java` : lecture/écriture du fichier `stopword.txt`, déclenchement de la réindexation	Antoine
Site web de documentation	Interface HTML/CSS/-Bootstrap de la documentation, hébergée sur `searchengine.antoineragot.com`	Antoine

Journalisation WAL	`Journal.java` : conception et implémentation intégrale du système WAL, modèle Producteur-Consommateur (`wait()`/`notifyAll()`), thread démon, restauration depuis `journal.csv`, compaction	Benjamin
Gestion de la concurrence	Utilisation de structures thread-safe (`ConcurrentHashMap`), synchronisation fine avec `synchronized`, verrous et mécanismes de signalisation `wait/notifyAll` pour l'accès aux ressources partagées	Arnaud et Benjamin
Moteur de recherche TF-IDF	`Recherche.java` : calcul TF-IDF, tri par score, gestion des mots exclus (-), retour des résultats formatés	Benjamin
Recherche avancée booléenne	Méthode `RechercheAvance()` : opérateurs ET/OU/-SAUF via opérations ensemblistes sur `HashSet`	Arnaud
Vérification Stop-Words	Détection dans les résultats si un mot recherché est un Stop-Word	Benjamin
Cache LRU	`CacheLRU.java` : surcharge de `LinkedHashMap` avec `accessOrder = true` et remplacement automatique	Arnaud

Commandes serveur	Implémentation d'une partie des commandes dans `Main.server()` : `-m`, `-exif`, `-d`, `-r`, `-dl`, `-reindex`, `-sw`, `-tag`, `-exif -set` et gestion des cas d'erreur	Arnaud et Antoine et Benjamin
Détection de doublons	`Doublon.java` : calcul de l'empreinte MD5 via `MessageDigest`	Antoine
Tests, correction et validation	Conception du banc de tests, rédaction des scénarios, vérification des cas aux limites (Edge Cases), traces d'exécution, correction et recherche de potentiels bugs	Benjamin, Arnaud
Rapport LaTeX	Rédaction, mise en page, schémas d'architecture, relecture croisée	Benjamin, Arnaud, Antoine

8.3 Bilan de la Répartition

Antoine a pris en charge l'intégration globale et la majorité de l'architecture (couches basses, réseau, client), ce qui représente la partie la plus transversale et la plus complexe à coordonner. Benjamin s'est concentré sur les mécanismes de fiabilité et d'algorithmique (WAL, TF-IDF), les fonctionnalités les plus intellectuellement exigeantes. Arnaud a couvert les structures de gestion mémoire, la recherche avancée et assuré la validation par les tests.

Cette répartition a évolué naturellement au fil du projet selon les compétences de chacun, tout en restant cohérente avec nos engagements initiaux.

9. Conclusion et Perspectives

9.1 Bilan de l'Application

Ce projet a permis de développer une solution complète couvrant les concepts fondamentaux du cours Info4B : processus et threads (gestion multi-threadée de l'indexation et du réseau),

synchronisation (modèle Producteur-Consommateur dans le journal), gestion de la mémoire (cache LRU), gestion des E/S et interruptions (WatchService, Pipes), communication inter-processus (Sockets TCP), et persistance des données.

L'architecture en couches stricte garantit un faible couplage : on peut remplacer la stratégie d'extraction de texte (`ExtracteurTexte`) sans toucher à l'index, et on peut faire évoluer le protocole réseau sans toucher à la logique de recherche.

9.2 Perspectives d'Évolution

Plusieurs évolutions peuvent rapprocher ce prototype d'une solution industrielle :

— **Recherche par famille de mots** : Actuellement, "programme" et "programmation" sont deux entrées distinctes dans l'index. Un algorithme de réduction sémantique augmenterait massivement le taux de rappel.

— **Gestion de la priorité des opérateurs booléens** : Dans `-ar`, l'expression `a ou b et c` n'évalue pas `b et c` en premier. Un analyseur syntaxique (parser) avec un arbre d'expression résoudrait cette limite.

— **Externalisation de la Base de Données** : Si le nombre de documents dépasse la centaine de milliers, l'Index Inversé en RAM devra être migré vers un système clé-valeur persistant optimisé (Redis, RocksDB).

10. Utilisation des Modèles Génératifs (LLM)

Conformément aux consignes du sujet, nous déclarons ici l'ensemble des portions de code ayant bénéficié d'une assistance par modèle de langage (Gemini), ainsi que les prompts utilisés. Dans tous les cas, le code généré a été relu, compris, adapté et intégré manuellement par le membre du groupe concerné.

10.1 Portions de Code Générées ou Assistées

10.1.1 Restauration depuis le journal (Journal.java)

Le bloc de parsing du fichier `journal.csv` lors du redémarrage a été généré en partie par LLM.

Listing 10.1 – Portion generee par LLM dans restaurerDepuisJournal()

```
ConcurrentHashMap<String, Integer> frequence = new ConcurrentHashMap<>();
if (champs.length > 4 && !champs[4].isEmpty()) {
    for (String motFreq : champs[4].split(",")) {
```

```
        String[] paire = motFreq.split(":");
        if (paire.length == 2) {
            int freq = Integer.parseInt(paire[1]);
            frequence.put(paire[0], freq);
            indexInverse.restaurerFrequenceMot(paire[0], docId, freq);
        }
    }
}
```

Prompt utilisé : *"J'ai un fichier CSV où chaque ligne contient des mots et leurs fréquences sous la forme mot1 :2,mot2 :5. Comment je fais pour lire ça en Java et reconstruire une structure en mémoire qui associe chaque mot à son nombre d'occurrences ? J'utilise ConcurrentHashMap."*

10.1.2 Mécanisme Producteur-Consommateur (Journal.java)

La structure du pattern Producteur-Consommateur avec `wait()`/`notifyAll()` sur le buffer borné a été co-développée avec l'aide d'un LLM pour assurer la correction de la synchronisation.

Listing 10.2 – Structure Producteur-Consommateur assistee par LLM

```
synchronized (fileOperations) {
    while (fileOperations.isEmpty() && actif) {
        try { fileOperations.wait(); }
        catch (InterruptedException e) {
            Thread.currentThread().interrupt(); return;
        }
    }
    if (!actif && fileOperations.isEmpty()) break;
    operation = fileOperations.remove(0);
    fileOperations.notifyAll();
}
```

Prompt utilisé : *"Est-ce que mon implémentation du pattern producteur-consommateur est correcte ? J'ai une ArrayList partagée entre plusieurs threads, un thread consommateur qui attend avec wait() que la liste soit non vide, et les producteurs qui font notifyAll() après chaque ajout. Le bug c'est que parfois le consommateur ne se réveille pas. Voici mon code : [code joint]"*

10.1.3 Calcul TF-IDF et tri des résultats (Recherche.java)

L'implémentation du calcul TF-IDF avec accumulation via `merge()` et le tri final par `comparingByValue().reversed()` a bénéficié d'une assistance LLM.

Listing 10.3 – TF-IDF et tri assistes par LLM

```
double idf = Math.log((double) totalDocs / nbDocsAvecMot);
double tf = (double) indexDuMot.get(id) / metaData.getTotalMots();
scoresParChemin.merge(chemin, tf * idf, Double::sum);

```

```java
// Tri par score decroissant
List<Map.Entry<String, Double>> listeTriee =
    new ArrayList<>(scoresParChemin.entrySet());
listeTriee.sort(Map.Entry.<String, Double>comparingByValue().reversed());
```

Prompt utilisé : *"Je calcule un score TF-IDF par document et je stocke les résultats dans une ConcurrentHashMap<String, Double> où la clé c'est le chemin du fichier. Le problème c'est que quand l'utilisateur tape plusieurs mots, je veux additionner les scores plutôt que d'écraser. Et ensuite je veux trier par score décroissant. Comment faire ça proprement en Java ?"*

10.1.4 Cache LRU (CacheLRU.java)

L'utilisation de `LinkedHashMap` avec `accessOrder = true` et la surcharge de `removeEldestEntry()` a été suggérée par LLM.

Listing 10.4 – Cache LRU suggere par LLM

```java
public class CacheLRU<K, V> extends LinkedHashMap<K, V> {
    private final int capaciteMax;
    public CacheLRU(int capacite) {
        super(capacite, 0.75f, true); // true = access-order
        this.capaciteMax = capacite;
    }
    @Override
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        return size() > capaciteMax;
    }
}
```

Prompt utilisé : *"Je cherche à limiter le nombre d'entrées dans une Map Java à 100 maximum, en supprimant automatiquement la moins récemment utilisée quand on dépasse la limite. J'ai vu qu'on peut faire ça avec LinkedHashMap mais je ne comprends pas bien le paramètre accessOrder et la méthode removeEldestEntry. Tu peux m'expliquer ?"*

10.1.5 Extraction MD5 pour la détection de doublons (Doublon.java)

La chaîne `MessageDigest` → `BigInteger` → `toString(16)` pour produire un checksum lisible a été générée par LLM.

Listing 10.5 – Calcul MD5 genere par LLM

```java
byte[] hash1 = MessageDigest.getInstance("MD5").digest(dataFile1);
String checksum1 = new BigInteger(1, hash1).toString(16);
```

Prompt utilisé : *"Pour détecter si deux fichiers sont identiques, je veux calculer leur hash MD5 en Java avec MessageDigest. J'arrive à obtenir un tableau de bytes mais comment le convertir en une chaîne hexadécimale lisible pour pouvoir comparer les deux résultats avec equals() ?"*

10.1.6 WatchService et enregistrement récursif (SurveillanceTempsReel.java)

La méthode `enregistrerDossiers()` utilisant `Files.walk()` pour enregistrer récursivement tous les sous-dossiers auprès du `WatchService` a été assistée par LLM.

Listing 10.6 – Enregistrement recursif WatchService assiste par LLM

```
private void enregistrerDossiers(Path racine,
        WatchService watchService) throws IOException {
    Files.walk(racine)
        .filter(Files::isDirectory)
        .forEach(dossier -> {
            try {
                dossier.register(watchService,
                    StandardWatchEventKinds.ENTRY_CREATE,
                    StandardWatchEventKinds.ENTRY_MODIFY,
                    StandardWatchEventKinds.ENTRY_DELETE);
            } catch (IOException e) {
                System.err.println("Erreur sur : " + dossier);
            }
        });
}
```

Prompt utilisé : *"Mon WatchService fonctionne mais il ne surveille que le dossier racine, pas les sous-dossiers. Si je crée un fichier dans un sous-dossier, rien ne se passe. Comment enregistrer récursivement tous les sous-dossiers existants au démarrage, et aussi les nouveaux dossiers créés pendant l'exécution ?"*

10.1.7 Extraction DOCX (ExtracteurTexte.java)

Le traitement d'un fichier `.docx` comme une archive ZIP pour en extraire `word/document.xml` a été suggéré par LLM.

Listing 10.7 – Extraction DOCX suggeree par LLM

```
java.util.zip.ZipFile zipFile = new java.util.zip.ZipFile(cheminFichier);
java.util.zip.ZipEntry documentXML = zipFile.getEntry("word/document.xml");
String contenuXML = new String(zipFile.getInputStream(documentXML).
    readAllBytes());
String textePur = contenuXML.replaceAll("<[^>]+>", " ");
```

Prompt utilisé : *"Est-ce qu'il existe un moyen d'extraire le texte d'un fichier .docx en Java sans ajouter de dépendance externe comme Apache POI ? Je veux éviter d'alourdir le projet avec des bibliothèques supplémentaires."*

10.1.8 Connexion Client Interactive (Client.java)

La mise en place de la lecture utilisateur pour configurer dynamiquement la destination de la socket a été générée par LLM pour fournir une interface console robuste.

Listing 10.8 – Saisie utilisateur et initialisation de la Socket

```
Scanner scanner = new Scanner(System.in);
System.out.print("Adresse IP du serveur > ");
String ip = scanner.nextLine().trim();
System.out.print("Port > ");
String port = scanner.nextLine().trim();
Socket socket = new Socket(ip, Integer.parseInt(port));
```

Prompt utilisé : *"En Java, comment puis-je demander à l'utilisateur de saisir une adresse IP et un port dans la console, puis utiliser ces informations pour ouvrir une connexion Socket ?"*

10.1.9 Récupération de l'adresse IP locale (Serveur.java)

L'astuce technique consistant à ouvrir une `DatagramSocket` vers une IP externe (Google DNS) pour identifier l'interface réseau active du serveur sans "hardcoder" d'adresse a été suggérée par un LLM.

Listing 10.9 – Détection automatique de l'IP du serveur

```
String ipServeur;
try (final DatagramSocket socketIP = new DatagramSocket()) {
    socketIP.connect(InetAddress.getByName("8.8.8.8"), 10002);
    ipServeur = socketIP.getLocalAddress().getHostAddress();
} catch (Exception e) {
    ipServeur = "Impossible de récupérer l'IP";
}
System.out.println("Server is running...");
System.out.println("Adresse IP du serveur : " + ipServeur + ":" + server.
   getLocalPort());
```

Prompt utilisé : *"Mon code Java renvoie souvent 127.0.0.1 quand je demande l'adresse IP locale du serveur. Comment obtenir l'adresse IP réelle de la machine sur le réseau local, celle que les clients doivent utiliser pour se connecter, de manière fiable ?"*

10.1.10 Aide à la conception et choix des structures de données

Au-delà de l'implémentation, un LLM a été consulté lors de la phase de conception pour orienter le choix des structures de données les plus adaptées aux contraintes de performance et de concurrence du projet.

— **Analyse des performances :** Comparaison des structures pour l'index inversé (choix d'une `ConcurrentHashMap` imbriquée pour garantir la rapidité de recherche et la sécurité entre les threads).

— **Optimisation de la recherche :** Suggestions sur l'utilisation des `HashSet` pour optimiser les opérations ensemblistes (ET, OU, SAUF) lors des recherches avancées.
— **Recherche de bibliothèques :** Orientation vers `JLine 3` pour la gestion du terminal interactif et vers des outils natifs comme `WatchService` pour éviter les dépendances lourdes.

10.1.11 Révision et correction du rapport

Un LLM a été utilisé pour parfaire la qualité rédactionnelle du présent document, garantissant ainsi une présentation claire et professionnelle des travaux réalisés.

— **Correction orthographique et grammaticale :** Une vérification systématique a été effectuée pour éliminer les coquilles et les fautes d'accord.
— **Amélioration du registre de langue :** Reformulation de certains passages pour adopter un ton plus académique et précis, adapté à un rapport de projet technique.
— **Structuration logique :** Assistance dans l'articulation des idées pour assurer une transition fluide entre les différentes sections (architecture, implémentation et tests).

10.2 Bilan

L'architecture globale, les choix de structures de données, la conception des couches fonctionnelles, la logique des commandes serveur, l'intégration des composants et la rédaction du rapport ont été réalisés manuellement par le trinôme. Le recours au LLM s'est concentré sur des passages techniques précis (algorithmes de synchronisation, manipulation de collections Java, appels système) pour lesquels chaque membre du groupe a su expliquer et justifier chaque ligne produite lors de la relecture.

Fin du rapport.