Projet 3 DAAR 2025

Breton Noé

n°21516014

Boudrouss Réda

n°28712638

Durbin Deniz Ali

n°21111116

Sorbonne Université
France

Table des matières

1 Introduction

Lien vers la vidéo de présentation : **https://www.youtube.com/watch?v=nyT_8-T6X7Y**

Il est peut-être possible de trouver une instance en ligne à l'adresse **https://daar.rboud.com**. Cependant, il est possible que l'instance soit down à tout moment, et il est préférable de lancer le code localement.

1.1 Contexte

Nous presentons une application web de recherche textuelle de livres de la bibliothèque du projet Gutenberg. Nous avons cherché a reduire le nombre de librairie externes utilisées, en implementant nous-memes les algorithmes clefs de recherche et de classement, par exemple, nous n'utilisons aucune librairie de recherche textuelle preexistante (Lucene, ElasticSearch, etc…).

1.2 Architecture Technique

L'application est structurée en trois couches distinctes :

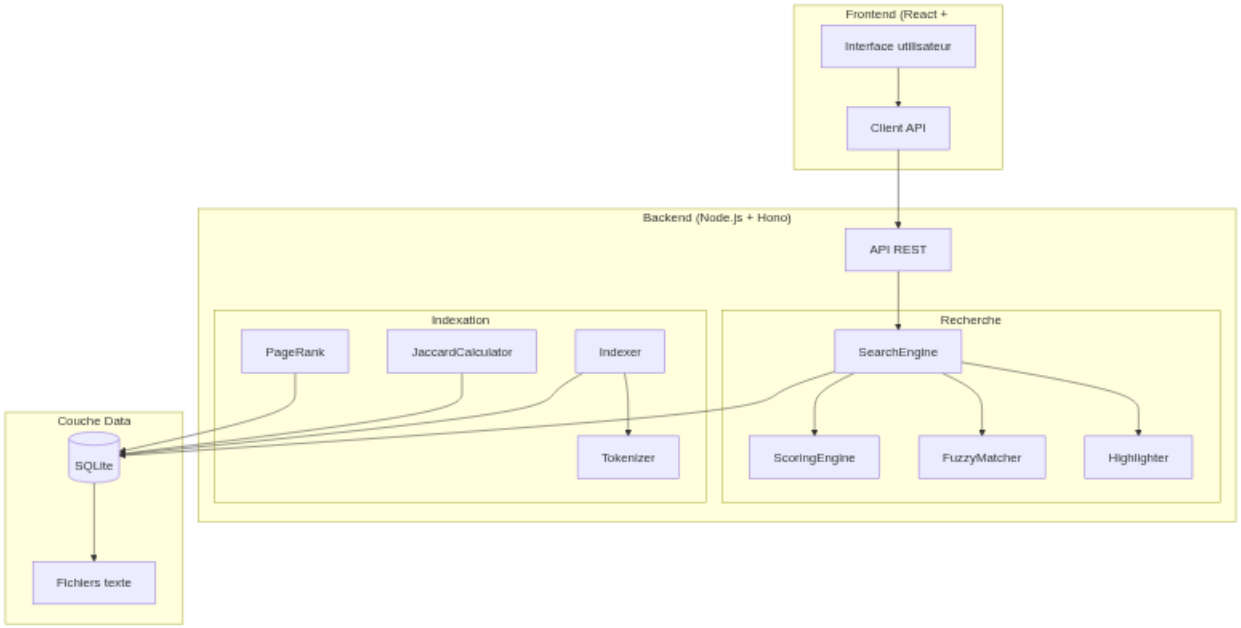

FIGURE 1 –

— **Couche Data** (`Backend/data/`) : fichiers textuels des livres, images de couverture, et base de données SQLite contenant l'index inversé, le graphe de Jaccard et les scores PageRank.
— **Backend** (`Backend/`) : serveur Node.js utilisant le framework Hono, exposant une API REST pour la recherche, l'indexation et l'administration.
— **Frontend** (`Frontend/`) : application React avec TanStack Router pour la navigation.

1.3 Contributions et Choix de Conception

Fonctionnalité explicite de Recherche :

— Tokenization puis indexation dans une reverse index
— Plusieurs options de tokenization (stop words, casse, longueur)

Recherche avec Regex :

— Support de regex en utilisant un NFA avec cache DFA (expliqué dans le projet 1)

— Analyse le pattern regex pour générer une requête SQL optimisée qui filtre les termes candidats

Classement :

— Graphe de Jaccard pondéré par IDF
— PageRank sur le graphe de Jaccard
— Score hybride BM25 + PageRank
— Bonus de proximité et de titre

Suggestion :

— Livres similaires basées sur jaccard et pagerank (dans la page de détail d'un livre)
— Suggestions de livres les plus cliqués dans la page d'accueil
— Suggestions de livre proches des livres populaires dans la page d'accueil

Ajouts :

— Recherche floue (Levenshtein)
— Highlighting des résultats
— Interface Admin pour configurer tout les paramètres de l'application

1.4 Execution du code

Il est recommendé d'utiliser docker compose pour lancer le serveur avec la commande `docker compose up` (ou `doker-compose up`).

Sinon il faut :

— Installer Node.js avec NPM si vous ne l'avez pas déjà.
— Installer pnpm. Généralement il faut juste `npm install -g pnpm`.

— Lancer `pnpm install` dans le root du projet pour installer les dépendances.

— Lancer `pnpm build` dans le root du projet pour build le projet.

— Entrer dans le dossier Backend et lancer `pnpm start` ou `pnpm run dev`pour lancer le serveur.

— Aller sur http://localhost:3000 pour accéder à l'application.

Commencez par aller sur l'interface Admin sur **http://localhost:3000/admin** pour importer les livres et configurer l'application.
Pensez à ajouter un livre, avant d'en rajouter plusieurs.
Utilisez le mot de passe `admin` pour vous connecter.
Faites attention, à partir d'un certain point, la construction du graphe de jaccard peut prendre plusieurs minutes. (Il est conseillé de commencer avec un `JACCARD_MAX_TERM_FREQUENCY` haut ~50% pour des petites quantités de livres (< 100), et de le baisser drastiquement à 5%)

2 Structures de Données

2.1 Index Inversé

2.1.1 Définition

Pour la recherche textuelle, nous utilisons un index inversé avec positions. il associe chaque terme à la liste des documents le contenant, avec la fréquence et les positions d'occurrence. (Les positions seront utiles pour le highlighting et le calcul de bonus de proximité, nous expliquerons cela plus tard.)

Formellement, pour un corpus $D = \{d_1, d_2, ..., d_n\}$ et un vocabulaire V, l'index inversé I est défini par :

$$I : V \to \mathcal{P}(D \times \mathbb{N} \times \mathbb{N}^*)$$

où chaque entrée contient le document, la fréquence du terme, et les positions d'occurrence.

2.1.2 Implémentation

Notre index inversé est stocké dans SQLite avec le schéma suivant :

```
CREATE TABLE inverted_index (
  term TEXT NOT NULL,
  book_id INTEGER NOT NULL,
  term_frequency INTEGER NOT NULL,
  positions TEXT,  -- JSON array: [12, 45, 89, ...]
  PRIMARY KEY (term, book_id)
);

CREATE INDEX idx_term ON inverted_index(term);
CREATE INDEX idx_book_id ON inverted_index(book_id);
```

Le champ `positions` stocke les positions en caractères de chaque occurrence, permettant le highlighting et le calcul de bonus de proximité.

2.1.3 Complexité

— Recherche d'un terme : $O(1)$ grâce à l'index SQL sur `term`
— Insertion d'un terme : $O(\log n)$ pour la mise à jour de l'index B-tree
— Espace : $O(|V| \times \bar{d})$ où $\bar{d}$ est le nombre moyen de documents par terme

2.2 Statistiques des Termes

Pour le calcul des scores BM25 et IDF, nous maintenons des statistiques globales :

```
CREATE TABLE term_stats (
  term TEXT PRIMARY KEY,
  document_frequency INTEGER NOT NULL,  -- df(t)
  total_frequency INTEGER NOT NULL      -- Σ tf(t,d)
);
```

Ces statistiques sont mises à jour lors de l'indexation et permettent notamment de calculer l'IDF :

$$IDF(t) = \log\left(\frac{N}{df(t)}\right)$$

où N est le nombre total de documents et $df(t)$ le nombre de documents contenant le terme t.

2.3 Graphe de Jaccard

2.3.1 Représentation

Le graphe de similarité de Jaccard modélise les relations entre documents. Chaque livre est un sommet, et une arête pondérée relie deux livres.

```
CREATE TABLE jaccard_edges (
  book_id_1 INTEGER NOT NULL,
  book_id_2 INTEGER NOT NULL,
  similarity REAL NOT NULL,
```

```
  PRIMARY KEY (book_id_1, book_id_2)
);
```

2.3.2 Propriétés du Graphe

Le graphe est non-orienté (stocké avec $book\_id\_1 < book\_id\_2$) et pondéré. Pour optimiser l'espace, seules les k meilleures arêtes par sommet sont conservées (Top-K).

Avec n livres et $k = 50$ voisins maximum par livre, le nombre d'arêtes est borné par $O(n \times k)$.

2.4 Scores PageRank

Les scores PageRank pré-calculés sont stockés pour éviter un recalcul à chaque requête :

```
CREATE TABLE pagerank (
  book_id INTEGER PRIMARY KEY,
  score REAL NOT NULL
);
```

2.5 Tokenizer

Le tokenizer transforme le texte brut en termes indexables. Nous avons implémenté un tokenizer simple qui découpe le texte en mots, en ignorant la ponctuation et les caractères spéciaux. Voici les paramètres configurables et leurs valeurs par défaut :

Paramètre	Valeur	Description
`minWordLength`	2	Longueur minimale d'un terme
`removeStopWords`	true	Filtrage des mots vides
`caseSensitive`	false	Normalisation en minuscules
`keepPositions`	true	Conservation des positions

3 Similarité de Jaccard

3.1 Définition

3.1.1 Jaccard Classique

L'indice de Jaccard mesure la similarité entre deux ensembles. Pour deux documents A et B représentés par leurs ensembles de termes :

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Cette mesure est comprise entre 0 (aucun terme commun) et 1 (ensembles identiques).

3.1.2 Jaccard Pondéré par IDF

Le Jaccard classique traite tous les termes de manière égale. Or, les termes rares sont plus discriminants que les termes fréquents. Nous utilisons donc une version pondérée par l'IDF :

$$J_{IDF}(A, B) = \frac{\sum_{t \in A \cap B} IDF(t)}{\sum_{t \in A \cup B} IDF(t)}$$

où $IDF(t) = \log(N/df(t))$.

Cette pondération favorise les documents partageant des termes rares, qui sont généralement plus significatifs pour déterminer la similarité thématique.

3.2 Construction du Graphe

3.2.1 Algorithme

La construction naïve du graphe de Jaccard a une complexité $O(n^2)$ comparaisons. Pour un corpus de 1664 livres, cela représente environ 1,4 million de paires à évaluer.

Nous appliquons plusieurs optimisations pour réduire ce coût :

1. **Incrémental** : le graphe est construit progressivement lors de l'ajout de nouveaux livres, en comparant seulement avec les livres existants
2. **Filtrage des termes trop fréquents** : les termes présents dans plus de 70% des documents sont ignorés (stop words dynamiques)
3. **Minimum de termes partagés** : seules les paires ayant au moins 5 termes communs sont considérées
4. **Pré-calcul des candidats** : une requête SQL identifie les paires potentiellement similaires avant le calcul exact. On peut perdre de potentielles paires, mais cela réduit drastiquement le nombre de comparaisons.

Plus on réduit le pourcentage ou on augmente le nombre de termes partagés requis, plus on perd de paires, mais plus on gagne en performance. L'algorithme étant quadratique, il est important de limiter au maximum le nombre de paires à comparer.

3.2.2 Pseudo-code

```
fonction buildJaccardGraph():
    charger les fréquences documentaires
    pour chaque batch de livres:
        charger les termes avec IDF
        identifier les candidats (termes partagés >= seuil)
        pour chaque paire candidate:
            calculer similarité pondérée IDF
            si similarité >= threshold:
                ajouter à la liste des voisins
    appliquer le filtre Top-K
    insérer les arêtes dans la base
```

3.3 Paramètres de Configuration du calcul de similarité

Paramètre	Valeur	Description
`similarityThreshold`	0.1	Seuil minimum de similarité
`topK`	50	Nombre maximum de voisins par livre
`maxTermFrequency`	0.7	Fréquence maximum d'un terme (70%)
`minSharedTerms`	5	Minimum de termes partagés

3.4 Complexité

— **Temps** : $O(n \times c \times \bar{t}^2)$ où n est le nombre de livres, c le nombre moyen de candidats par livre, et $\bar{t}$ le nombre moyen de termes par livre
— **Espace** : $O(n \times k)$ arêtes stockées avec le filtre Top-K

4 Algorithme PageRank

Nous avons choisi d'utiliser pagerank car pour un projet de l'UE AAGA, nous avons rendu un rapport où nous avons étudié l'algorithme.

4.1 Définition

4.1.1 Principe

PageRank est un algorithme développé par Larry Page et Sergey Brin pour classer les pages web. Il modélise un "surfeur aléatoire" qui navigue de page en page en suivant les liens. La probabilité stationnaire de se trouver sur une page définit son score PageRank.

Dans notre contexte, les "pages" sont les livres et les "liens" sont les arêtes du graphe de Jaccard. Un livre avec un PageRank élevé est un livre central, connecté à de nombreux autres livres importants.

4.1.2 Formule Itérative

Le score PageRank d'un sommet v est défini par :

$PR(v) = \frac{1-d}{N} + \frac{d \cdot S}{N} + d \sum_{u \in \text{In}(v)} \frac{PR(u)}{\deg^+(u)}$

avec

$S = \sum_{u \text{dangling}} \frac{PR(u)}{N}$

où : - d est le facteur d'amortissement (typiquement 0.85) - N est le nombre total de sommets - $\text{In}(v)$ est l'ensemble des sommets pointant vers v - $\deg^+(u)$ est le degré sortant de u - Les sommets "dangling" sont ceux sans liens sortants

$d - 1$ représente la probabilité de téléportation vers une page aléatoire, assurant que le surfeur ne reste pas bloqué. S redistribue le score des sommets sans liens sortants.

Ces ajout permettent permettent de gérer les sommets sans liens sortants et certain cas de boucles qui accumuleront un score.

4.2 Implémentation

4.2.1 Algorithme Itératif (Power Iteration)

```
fonction computePageRank(graph, d, maxIter, tolerance):
    N ← nombre de sommets
    incomingEdges, outDegrees, danglingNodes ← prétraiter(graph)
    r ← vecteur de taille N initialisé à 1/N
    pour i de 1 à maxIter:
        danglingSum ← somme des r[u] pour u dans danglingNodes
        baseRank ← (1 - d)/N + d × (danglingSum / N)
        r_new ← vecteur vide de taille N
        pour chaque sommet v de 0 à N-1:
            r_v ← baseRank
            pour chaque u dans incomingEdges[v]:
                r_v ← r_v + d × (r[u] / outDegrees[u])
            r_new[v] ← r_v
        diff ← somme des |r_new[v] - r[v]| pour tous les v
        r ← r_new
        si diff < tolerance:
            retourner r
    retourner r
```

4.2.2 Paramètres

Paramètre	Valeur	Description
`damping (d)`	0.85	Facteur d'amortissement
`maxIter`	100	Nombre maximum d'itérations
`tolerance`	10^{-6}	Seuil de convergence

4.3 Complexité

— **Temps** : $O(k \times \bar{d})$ où k est le nombre d'itérations et $\bar{d}$ le degré moyen entrant, qui est directement limité par le Top-k du graphe de Jaccard, donc au maximum 50 dans notre cas.
— **Espace** : $O(|V|)$ pour stocker les scores

5 Système de Scoring

5.1 Score BM25

Notamment utilisé pour le classement des résultats de la recherche textuelle.

5.1.1 Définition

BM25 (Best Matching 25) est une fonction de scoring probabiliste issue du modèle de pertinence BM. C'est une évolution du TF-IDF qui intègre une normalisation par la longueur du document et une saturation de la fréquence des termes.

Pour une requête Q composée des termes $q_1, ..., q_n$ et un document D :

$$BM25(D, Q) = \sum_{i=1}^{n} IDF(q_i) \cdot \frac{tf(q_i, D) \cdot (k_1 + 1)}{tf(q_i, D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{avgdl}\right)}$$

où : - $tf(q_i, D)$ est la fréquence du terme q_i dans le document D - $|D|$ est la longueur du document (en mots) - $avgdl$ est la longueur moyenne des documents du corpus - k_1 et b sont des paramètres de tuning

5.1.2 Composantes

IDF (Inverse Document Frequency) :

$$IDF(t) = \log\left(\frac{N - df(t) + 0.5}{df(t) + 0.5} + 1\right)$$

Cette formule, légèrement différente du IDF classique, évite les valeurs négatives pour les termes très fréquents.

Saturation TF : Le terme $(k_1 + 1)$ au numérateur et k_1 au dénominateur créent une saturation, augmenter la fréquence d'un terme au-delà d'un certain point a un effet décroissant sur le score.

Normalisation de longueur : Le paramètre b contrôle l'impact de la longueur du document. Avec $b = 0$, pas de normalisation ; avec $b = 1$, normalisation complète.

5.1.3 Paramètres

Paramètre	Valeur	Description
k_1	1.2	Saturation de la fréquence
b	0.75	Normalisation par la longueur

5.2 Score Hybride (BM25 + PageRank)

Nous combinons le score BM25 avec le score PageRank pour obtenir un score hybride. Cela nous permet d'exploiter à la fois la pertinence textuelle et l'importance structurelle des documents.

$$Score_{hybride} = w_{BM25} \cdot BM25(D, Q) + w_{PR} \cdot PR(D) \cdot N$$

où : - $w_{BM25} = 0.6$ est le poids du score BM25 - $w_{PR} = 0.4$ est le poids du PageRank - N est le nombre total de documents (pour normaliser PageRank)

5.2.1 Bonus de Proximité

Un bonus multiplicatif est appliqué lorsque les termes de la requête apparaissent proches les uns des autres dans le document. La proximité est calculée à partir des positions stockées dans l'index inversé.

Il est de 1 par défaut, de 3 si il existe une phrase exacte, et de manière croissante entre 1 et 3 dépendant de la proximité des termes.

5.2.2 Bonus de Titre

Auquel est ajouté un bonus si les termes de la requête apparaissent dans le titre du document. Il est de 2 si et seulement si tous les termes apparaissent dans le titre.

5.3 Scoring des Suggestions

Pour les suggestions de livres similaires, nous utilisons :

$$Score_{suggestion} = 0.6 \times Jaccard(D, D_{ref}) + 0.4 \times PR(D) \times 100$$

où D_{ref} est le document de référence (celui consulté par l'utilisateur).

5.3.1 Recommandations Basées sur les Clics

Sur la page d'accueil, nous recommandons des livres populaires et des livres proches des livres populaires.

```
pour chaque livre figurant parmi les livres les plus cliqués:
    récupérer les voisins Jaccard
    pour chaque voisin:
        score ← similarité × (clics / max_clics)
        agréger les scores des voisins multi-sources
```

5.4 Configuration

Les poids du scoring sont configurables dynamiquement via l'interface d'administration :

Paramètre	Défaut	Description
`bm25Weight`	0.6	Poids du score BM25
`pageRankWeight`	0.4	Poids du PageRank
`enableProximityBonus`	true	Activer le bonus de proximité

6 Recherche Avancée par Regex

6.1 Problématique

Nous voulons supporter des recherches avancées via des expressions régulières. Cependant, appliquer un automate à chaque terme de l'index est coûteux. Si un vocabulaire atteint 500 000 termes (proche de notre base de donnée), une approche naïve serait impraticable.

6.2 Architecture de la Solution

Notre approche combine deux niveaux de filtrage :

1. **Pré-filtrage SQL** : réduire le nombre de candidats en exploitant la structure du pattern
2. **Validation NFA** : appliquer l'automate uniquement aux termes candidats si nécessaire

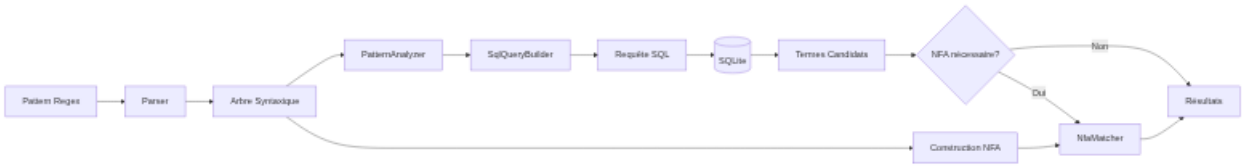

FIGURE 2 –

6.3 Analyse du Pattern

Le module `PatternAnalyzer` examine l'arbre syntaxique pour déterminer le type de pattern :

Type	Exemple	Optimisation SQL
exact	cat	term = 'cat'
alternation	cat|dog	term IN ('cat', 'dog')
prefix	cat.*	term LIKE 'cat%'
suffix	.*ing	term LIKE '%ing'
contains	.*cat.*	term LIKE '%cat%'
sql_pattern	c.t	term LIKE 'c_t'
match_all	.*	1=1 (tous)
complex	(ab)*c*	

6.3.1 Contraintes de Longueur

Pour les patterns complexes, l'analyseur essaie de calculer les bornes de longueur minimale et maximale. Par exemple, le pattern `a..b` a une longueur exacte de 4, permettant d'ajouter `LENGTH(term) = 4` à la requête SQL.

6.4 Construction de la Requête SQL

Le module `SqlQueryBuilder` génère une clause WHERE optimisée :

```
interface SqlQuery {
  whereClause: string;      // Clause WHERE
  parameters: any[];        // Paramètres bindés
  needsNfaFiltering: boolean; // NFA encore nécessaire?
}
```

6.4.1 Indicateur NFA

L'attribut `needsNfaFiltering` indique si le filtrage SQL est suffisant : - `false` pour `exact`, `alternation`, `sql_pattern` : le SQL est exact - `true` pour les autres : validation NFA requise

6.5 Pipeline Complete

```
function searchRegex(pattern: string): SearchResult[] {
  // 1. Parser le pattern
  const syntaxTree = parseRegex(pattern);
  const nfa = nfaFromSyntaxTree(syntaxTree);

  // 2. Analyser pour optimisation SQL
  const analysis = analyzeSqlPattern(syntaxTree);
  const sqlQuery = buildSqlQuery(analysis, "term");

  // 3. Récupérer les candidats
  const candidates = db.prepare(`
    SELECT DISTINCT term FROM term_stats
    WHERE ${sqlQuery.whereClause}
  `).all(...sqlQuery.parameters);

  // 4. Filtrer avec NFA si nécessaire
  let matchingTerms: string[];
  if (sqlQuery.needsNfaFiltering) {
    const matcher = new NfaMatcher(nfa);
    matchingTerms = candidates
      .filter(t => matcher.match(t.term));
  } else {
    matchingTerms = candidates.map(t => t.term);
  }

  // 5. Récupérer les livres contenant ces termes
  return findBooksWithTerms(matchingTerms);
}
```

7 Recherche Floue et Highlighting

7.1 Recherche Floue (Fuzzy Search)

7.1.1 Motivation

Les utilisateurs font fréquemment des fautes de frappe ou d'orthographe. Fuzzy Search permet de trouver des résultats même lorsque la requête ne correspond pas exactement aux termes indexés.

7.1.2 Distance de Levenshtein

La distance de Levenshtein (ou distance d'édition) entre deux chaînes est le nombre minimum d'opérations élémentaires pour transformer l'une en l'autre : - **Insertion** d'un caractère - **Suppression** d'un caractère - **Substitution** d'un caractère

7.1.3 Algorithme

Pour deux chaînes a de longueur m et b de longueur n, on construit une matrice D de taille $(m+1) \times (n+1)$:

$$D[i][j] = \begin{cases} i & \text{si } j = 0 \\ j & \text{si } i = 0 \\ D[i-1][j-1] & \text{si } a[i] = b[j] \\ 1 + \min(D[i-1][j], D[i][j-1], D[i-1][j-1]) & \text{sinon} \end{cases}$$

La distance finale est $D[m][n]$.

7.1.4 Implémentation

```typescript
function levenshteinDistance(a: string, b: string): number {
  const matrix: number[][] = [];

  // Initialisation
  for (let i = 0; i <= b.length; i++) matrix[i] = [i];
  for (let j = 0; j <= a.length; j++) matrix[0][j] = j;

  // Remplissage
  for (let i = 1; i <= b.length; i++) {
    for (let j = 1; j <= a.length; j++) {
      if (b[i-1] === a[j-1]) {
        matrix[i][j] = matrix[i-1][j-1];
      } else {
        matrix[i][j] = 1 + Math.min(
          matrix[i-1][j-1],  // substitution
          matrix[i][j-1],    // insertion
          matrix[i-1][j]     // suppression
        );
      }
    }
  }
  return matrix[b.length][a.length];
}
```

7.1.5 Complexité

— **Temps** : $O(m \times n)$
— **Espace** : $O(m \times n)$, réductible à $O(\min(m, n))$ avec optimisation

7.1.6 Expansion de Requête

Lors d'une recherche floue, chaque terme de la requête est étendu avec les termes de l'index ayant une distance inférieure ou égale au seuil (par défaut 2) :

```
entrée: terme de requête, distance maximale
sortie: ensemble de termes correspondants

pour chaque terme t dans l'index:
    si levenshtein(requête, t) <= distance_max:
        ajouter t aux résultats
trier par distance croissante
```

7.2 Highlighting

7.2.1 Objectif

Le highlighting consiste à générer des extraits de texte (snippets) mettant en évidence les termes recherchés. Cela permet à l'utilisateur d'évaluer rapidement la pertinence d'un résultat.

7.2.2 Génération de Snippets

L'algorithme utilise les positions stockées dans l'index inversé :

1. Récupérer les positions de tous les termes de la requête
2. Trier les positions par ordre croissant
3. Pour chaque position, extraire un contexte (caractères avant/après)
4. Fusionner les contextes adjacents si ils se chevauchent
5. Appliquer les balises `<mark>` autour des termes

7.2.3 Multi-Pattern Matching avec Aho-Corasick

Pour le highlighting de plusieurs termes simultanément, nous utilisons l'algorithme Aho-Corasick (décrit dans le rapport du projet 1). Cet algorithme permet de rechercher tous les termes en un seul parcours du texte.

7.2.4 Paramètres

Paramètre	Valeur	Description
`snippetCount`	3	Nombre de snippets par résultat
`snippetLength`	150	Longueur maximale d'un snippet
`contextBefore`	100	Caractères de contexte avant
`contextAfter`	100	Caractères de contexte après

7.2.5 Exemple de Sortie

Pour la requête "treasure island" dans un document :

```
...the young Jim Hawkins discovers a
<mark>treasure</mark> map leading to a distant
<mark>island</mark>...
```

8 Tests et Performances

8.1 Environnement de Test

8.1.1 Corpus

Le corpus de test est constitué de 4000 premier livres du projet Gutenberg :

8.1.2 Configuration Matérielle

Les tests ont été réalisés sur une machine avec : - Processeur : Intel Core i7 - 6 coeurs @ 3.8 GHz - Mémoire : 24 Go RAM

8.2 Performances d'Indexation

8.2.1 Temps d'Indexation

À partir du graphique nous pouvons remarquer : - La tendance quadratique de la construction du graphe de jaccard est bien visible. - PageRank est extremement rapide, son temps d'execution augmente à peine avec le

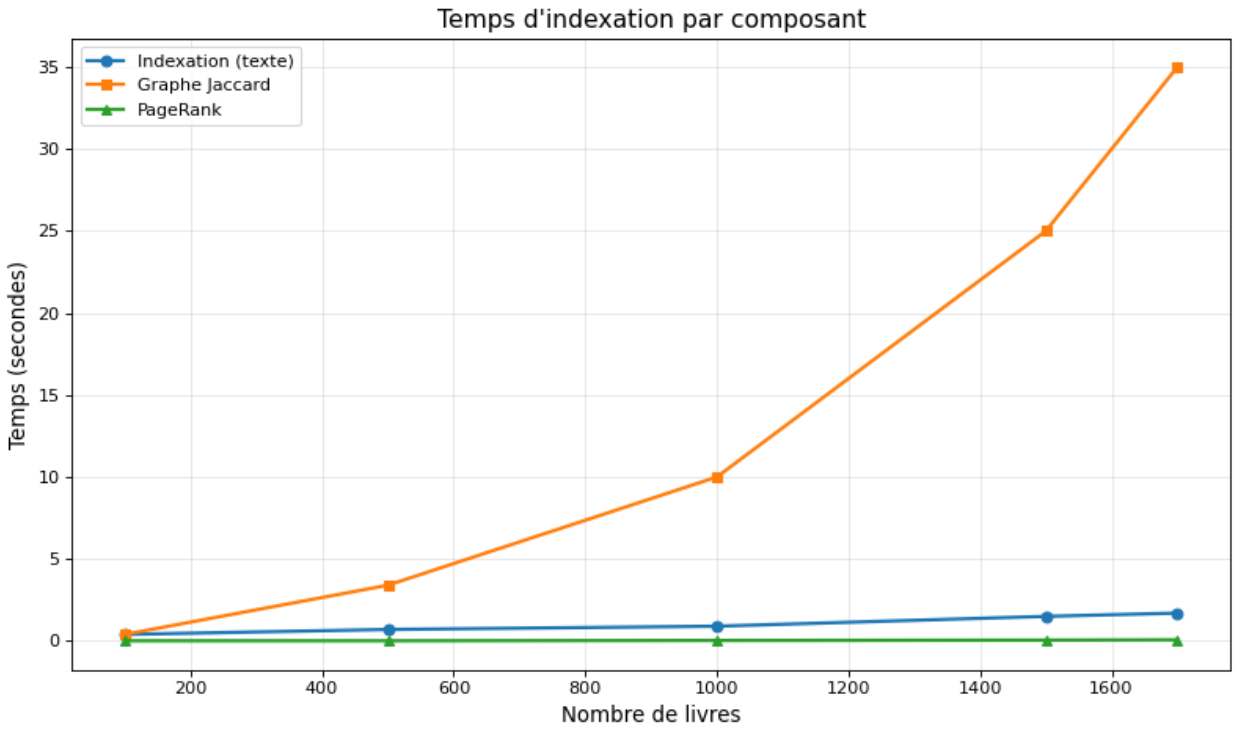

FIGURE 3 – Temps d'indexation en fonction du nombre de livres

nombre de livres. - L'indexation augmente légèrement avec le nombre de livres, probablement du a un facteur dans sqlite. En théorie, l'indexation est constante et devrait plutôt dépendre de la taille du livre à indexer.

8.3 Performances de Recherche

8.3.1 Temps de Réponse

La réponse est extrêment rapide, avec une médiane de 15ms et un P95 de 50ms.

9 Conclusion

9.1 Bilan

L'implémentation d'un moteur de recherche complet pour une bibliothèque numérique nous a permis d'explorer différentes approches algorithmiques pour la recherche d'information. Au-delà de la construction classique d'un index inversé avec positions, nous avons développé un graphe de similarité de Jaccard pondéré par IDF pour capturer les relations thématiques entre documents, et intégré l'algorithme PageRank pour identifier les ouvrages centraux du corpus.

9.2 Perspectives

Plusieurs limitations ont été identifiées :

— La construction du graphe de Jaccard reste quadratique, limitant le passage à l'échelle
— La recherche regex peut être lente sur des patterns complexes, nécessitant un passage de l'automate sur tout les termes de l'index.

Plusieurs améliorations pourraient être envisagées :

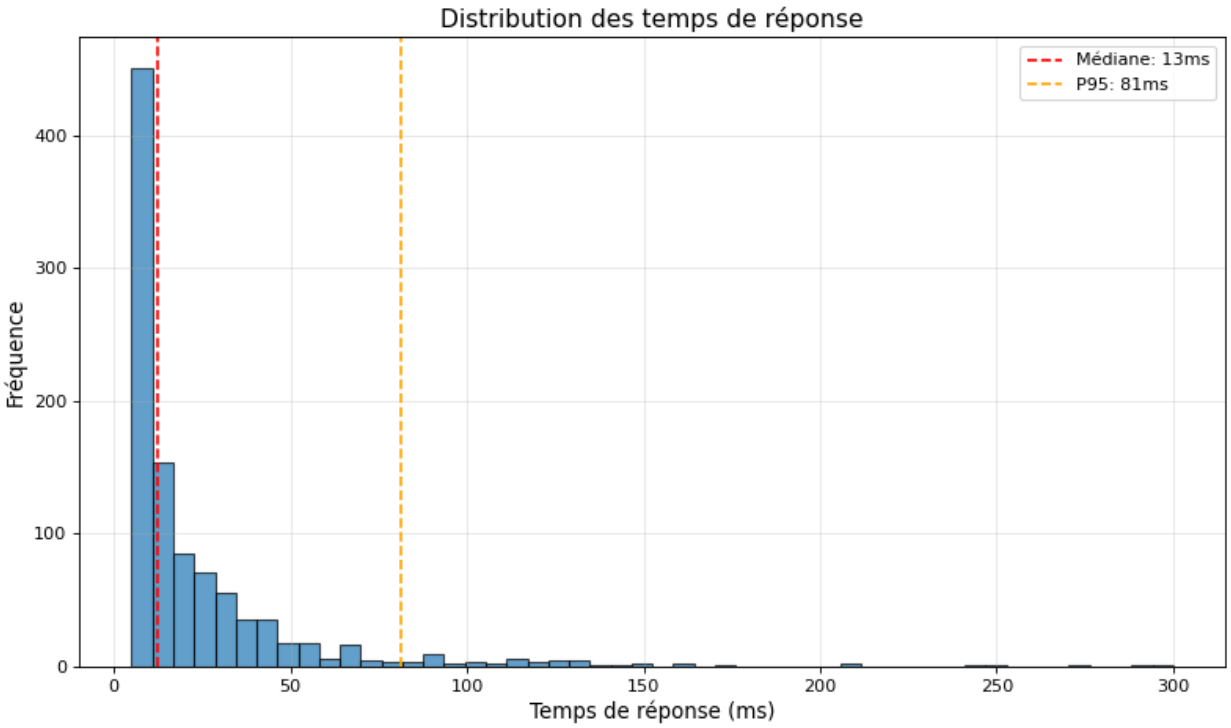

FIGURE 4 – Distribution des temps de réponse pour 1000 requêtes

— Utiliser des techniques de hachage (MinHash, LSH) pour approximer la similarité Jaccard en temps sous-quadratique.
— Intégration d'embeddings de mots (Word2Vec, BERT) pour capturer les relations sémantiques.