

Search Engine DAAR

Lounes & Oussama

Février 2022

Table des matières

1	Introduction	3
2	Architecture du projet	3
2.1	Base de données	3
2.2	Table d'indexage	4
2.3	Recherche par chaîne	4
2.4	Recherche par regex	4
2.5	Classement	4
2.6	Suggestions	5
3	Implémentation	5
3.1	Modèles	5
3.2	Construction de la base de données	6
3.3	Table d'indexage	6
3.4	Recherche Simple et Avancée	7
3.5	Suggestions	8
3.5.1	Distance de Jaccard	9
3.6	Classement	9
4	Frontend	9
4.1	Recherche	10
4.2	Suggestion	10
4.3	Classement	10
5	Expérimentation et Tests	11
5.1	Test de performance	11
5.2	Tests unitaires	12
6	Conclusion	13

1 Introduction

Un moteur de recherche est une application permettant d'effectuer une recherche sur une base de données locale ou en ligne, le résultat qui n'a pas de contrainte de type est une réponse à une requête qui peut être simple tel une chaîne de caractère ou complexe tel une expression régulière.

Un moteur de recherche de grande envergure peut être relativement complexe et être composé de plusieurs algorithmes, car il devra répondre à plusieurs problématiques, que ça soit dans la partie recherchée ou dans la partie résultat. Parmi ces contraintes, la rapidité du processus, la pertinence des résultats vis-à-vis de ceux souhaités par l'utilisateur.

L'objectif de notre projet est de développer une application web/mobile représentant un moteur de recherche sur une bibliothèque numérique contenant des livres textuels.

2 Architecture du projet

L'application est développée en se basant sur deux technologies : Angular pour faire le frontend et Django pour le backend. Pour le stockage des données on a utilisé SQLite

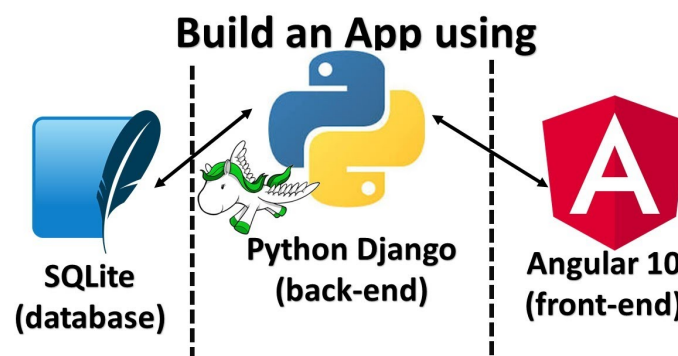


FIGURE 1 – Modèle D'un livre

2.1 Base de données

La base de données de l'application est construite instantanément après l'installation de cette dernière, elle sera composée de 1664 livres issus de la base Gutenberg, la taille minimum d'un

livre chargé est de dix mille mots.

2.2 Table d'indexage

Afin de réduire la complexité de la recherche sur la base de données, une première étape est effectuée, cette dernière est la conversion des données vers une nouvelle structure de représentation sous forme d'une table d'indexage.

Un index est une structure de données utilisée et entretenue par le système de gestion de base de données pour lui permettre de retrouver rapidement les données. L'utilisation d'un index simplifie et accélère les opérations de recherche.

2.3 Recherche par chaîne

La requête est sous la forme d'un mot-clef saisie par l'utilisateur, interprétée telle une simple chaîne de caractère. L'algorithme scan la table d'indexage pour générer l'ensemble des livres numériques qui vérifient le lien entre le mot-clef recherché et les index.

2.4 Recherche par regex

Cette recherche a la particularité que la requête est sous la forme d'une expression régulière, cette dernière est restreintes aux éléments suivants : "les parenthèses", "l'alternative", "la concaténation", "l'opération étoile", "le point" et "les lettres ASCII".

L'algorithme parcourt la table d'indexage pour générer l'ensemble des livres qui contiennent au moins une chaîne de caractères qui vérifie l'expression régulière.

2.5 Classement

Les résultats d'une recherche peuvent être nombreux, il est important que l'utilisateur puisse trier ces résultats selon certains critères par exemple, par leurs titres ou par leur pertinences, pour réaliser cette dernière nous nous sommes basés sur l'algorithme closeness [2].

2.6 Suggestions

Suite à une recherche par chaîne ou par regex, le moteur renvoie deux types de résultats : une liste de livres et une liste de suggestions basées sur les dernières recherches, l'algorithme se base sur un graphe appelé graphe de jaccard dont chaque noeud représente un livre déjà recherché.

3 Implémentation

3.1 Modèles

Un livre est représenté dans notre base de données par un identifiant, un titre, ses auteurs, la langue, un lien vers son texte (cela permet d'avoir un gain en mémoire), potentiellement une couverture et un range de classement.

```
▼ {  
  "id": 11,  
  "title": "Alice's Adventures in Wonderland",  
  "author": "Carroll, Lewis",  
  "lang": "en",  
  "body": "https://www.gutenberg.org/files/11/11-0.txt",  
  "cover": "https://www.gutenberg.org/cache/epub/11/pg11.cover.medium.jpg",  
  "rank": 1.2933  
},
```

FIGURE 2 – Modèle D'un livre

Un graphe de jaccard est représenté dans notre base de données par un identifiant propre à lui, un autre qui est celui du livre, les voisins dans le graphe, et la distance totale.

```
▼ {  
  "id": 63,  
  "bookId": 1680,  
  "neighbors": [  
    7065,  
    55860  
  ],  
  "totalDistance": 5.034571935366012,  
  "centrality": 1.1474487746082764  
},
```

FIGURE 3 – Modèle D'un graphe de Jaccard

3.2 Construction de la base de données

Une commande à été implémenté, cette dernière régit à un algorithme pour charger la base de données automatiquement en prenant soin de respecter les contraintes imposées (minimum dix mille mots et 1664 livres et 1000 motes dans chaque livres).

On utilise l'api Gutendex qui elle à son tour communique avec la base Gutenberg. On a décidé de procéder ainsi car Gutendex génère des JSON très riches et qui facilitent l'accès et la manipulation des données de la base de données principale.

L'algorithme parcourt les livres de la base de données, et pour chaque entité, il charge son texte depuis Gutenberg sous format utf-8, il teste si ce dernier est bien composé de minimum dix mille mots, et cela en faisant un split de la String en une liste, le motif du split est l'espace " ", ainsi la taille de la liste générée indique le nombre de mots, une fois la condition vérifiée le livre est chargé dans la base de données interne sous la forme du modèle "BookM" ainsi sa correspondance dans la table d'indexage "BookMIndex".

3.3 Table d'indexage

L'indexation des livres est une étape dans la construction de la base de données citée précédemment, ainsi à chaque fois qu'un livre est valide et qu'il est ajouté à notre base, une entrée correspondant à ce livre est ajoutée dans la table d'indexage. Le titre et le sujet sont transformés en une liste de mots-clefs en supprimant les "stopword". La liste des mots résultants est ajoutée comme attribut d'un livre dans la table d'indexage "BookMIndex".

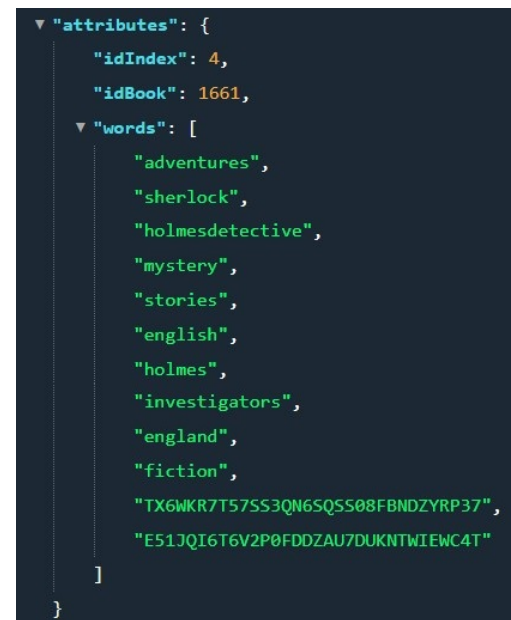


```

{
  "attributes": {
    "idIndex": 12,
    "idBook": 844,
    "words": [
      "importance",
      "earnest",
      "trivial",
      "comedy",
      "serious",
      "peoplecomedies",
      "england",
      "drama",
      "identity",
      "psychology",
      "drama"
    ]
  }
}

```

FIGURE 4 – Table d’indexage avant recherche par regex



```

{
  "attributes": {
    "idIndex": 4,
    "idBook": 1661,
    "words": [
      "adventures",
      "sherlock",
      "holmesdetective",
      "mystery",
      "stories",
      "english",
      "holmes",
      "investigators",
      "england",
      "fiction",
      "TX6WKR7T57SS3QN6SQ5S08FBNDZYRP37",
      "E51JQI6T6V2P0FDDZAU7DUKNTWIEWC4T"
    ]
  }
}

```

FIGURE 5 – Table d’indexage après recherche par regex

3.4 Recherche Simple et Avancée

L’algorithme commence par vérifier si la chaîne de caractères saisie par l’utilisateur est une expression régulière.

Si la requête est réduite à un simple mot-clef, il parcourt la table d’indexage, et filtre l’ensemble des livres pour en garder que ceux d’en la liste de mots contient au moins un qui est équivalent au mot-clef recherché.

Pour diminuer le temps d’exécution de la recherche, seuls les dix livres les plus pertinents au maximum sont conservés après chaque recherche.

On sélectionne Ensuite les 3 qui ont le plus d ’occurrences du mot recherché c’est les plus (vrais) pertinents.

Si la chaîne de caractères recherchée est une expression régulière, on utilise un projet qu’on a développé pour l’ue DAAR de Sorbonne Université, qui est un clone de la commande egrep de unix, le projet a été développé en JAVA, ce dernier a été légèrement modifier pour ne rien renvoyer si aucune chaîne de caractère cible ne satisfait la regEx recherchée. La liste de mots indexés concaténée ainsi que la regEx sont envoyé à l’application Java qu’on a cité à travers une commande python qui nous permet de communiqué (exécuter et récupéré le résultat)

avec un projet JAR.

Une expression régulière pouvant correspondre à plusieurs chaînes de caractères différentes, les mots-clefs qui la satisfont dans deux livres différents peuvent être différents, et celle recherché en tant que String n'est équivalente à aucune d'elles. Pour résoudre ce problème un hash Code de 32 bits est généré aléatoirement, ce dernier fera donc référence à la regEx recherchée, le hash Code est injecté dans la table d'indexage des livres qui répondent à la contrainte et le mot rechercher devient ce hashCode tel qu'on peut le voir sur la figure suivante. Enfin, l'algorithme sélectionne les 3 livres qui ont le plus d'occurrences du hash Code recherché, c'est les plus pertinents.

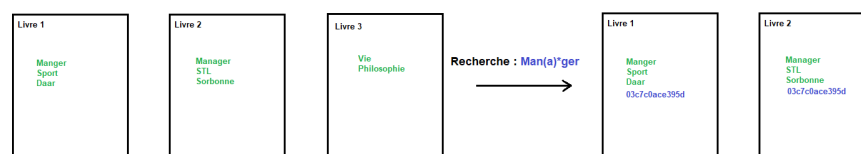


FIGURE 6 – Exemple de recherche avancée

3.5 Suggestions

La Liste des suggestions est générée en se basant sur un graphe de jaccard. pour cela et suite à une recherche par l'utilisateur, l'algorithme stocke les 3 livres pertinents dans le graphe, pour chaque livre il calcule la distance de jaccard (c.f Figure 7) entre ce dernier et tous ses livres voisins et conserve que ceux qui respectent cette distance, l'algorithme ensuite rajoute une arrêt entre ce noeud et chaque voisin de cette nouvelle liste des voisins. Une fois l'utilisateur fait une nouvelle recherche, l'algorithme renvoie les livres correspondants ainsi que la liste des voisins enregistrées dans le graphe comme suggestions.

La construction du graphe est fait dynamiquement, c'est à dire après chaque recherche, le graphe est manipulé par l'algorithme en mettant à jour la distance totale de chaque noeud et tous les autres noeuds du graphe, ainsi la centralité de chaque noeud.

En raison que le graphe est devient plus en plus volumineux après chaque recherche, ce qui augmente la complexité de la recherche, une solution assez simple est développée. Le graphe ne doit pas contenir plus que 25 noeuds, une fois cette limite est dépassée, l'algorithme conserve que les livres/noeuds plus centralisés (en se basant sur l'attribut "centrality") et supprime ceux qui sont au borne du graphe.

3.5.1 Distance de Jaccard

La distance de jaccard entre deux noeuds/livres est calculée en se basant sur la formule :

$$d(D_1, D_2) = \frac{\sum_{(w,k_1) \in D_1 \wedge (w,k_2) \in D_2} \max(k_1, k_2) - \min(k_1, k_2)}{\sum_{(w,k_1) \in D_1 \wedge (w,k_2) \in D_2} \max(k_1, k_2)}$$

FIGURE 7 – Distance de Jaccard

Cette distance permet de paramétrer notre graphe en ajoutant une arrêt entre un noeud/livre l et chaque noeud voisin v avec $d(l,v) < m$, à noter que m est une probabilité choisit par le moteur de recherche [4] [3].

3.6 Classement

L'utilisateur peut classer les résultats par titre ou par la pertinence, le classement ce fait en se basant sur l'algorithme de closeness [1].

Pour chaque nœud, l'algorithme closeness calcule la somme de ses distances (c.f distance de jaccard) à tous les autres nœuds. La somme résultante est ensuite inversée pour déterminer le score de centralité de proximité pour ce nœud. Il est plus courant de normaliser ce score afin qu'il représente la longueur moyenne des chemins les plus courts plutôt que leur somme. Cet ajustement permet de comparer la centralité de proximité des nœuds de graphes de tailles différentes.

L'algorithme utilise la formule ci-dessous :

$$\text{centrality} = (\text{number of nodes} - 1) / \text{sum}(\text{distance from node to all other nodes})$$

FIGURE 8 – Centralité dans un graphe

4 Frontend

Pour communiquer avec le backend et visualiser les résultats nous avons implémenter un simple projet développé avec le framework angular.

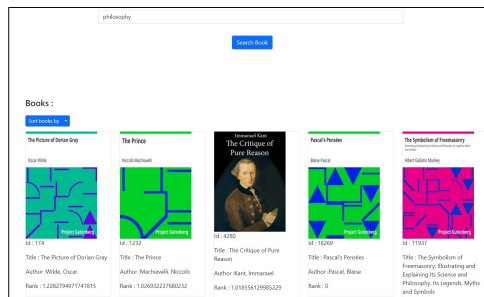


FIGURE 9 – Recherche par chaîne

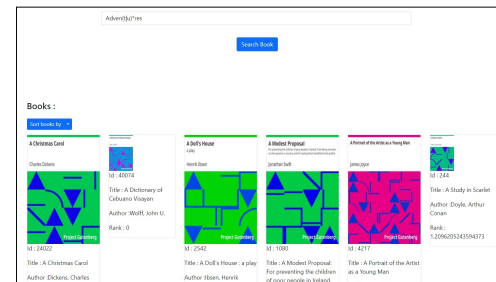


FIGURE 10 – Recherche par regex

4.1 Recherche

L'utilisateur peut faire la recherche par chaîne simple ou par regex en utilisant une barre de recherche, le résultat est généré sur la même page. Les deux figures ci-dessus illustrent le résultat de la recherche par chaîne simple et par regex

4.2 Suggestion

Basé sur des recherches antérieures, le moteur de recherche affiche également des suggestions de livres que l'on affiche dans la même page que le résultat initial.



FIGURE 11 – Suggestion de livres

4.3 Classement

L'utilisateur peut également faire le classement des résultats en se basant sur deux options : titre et pertinence.

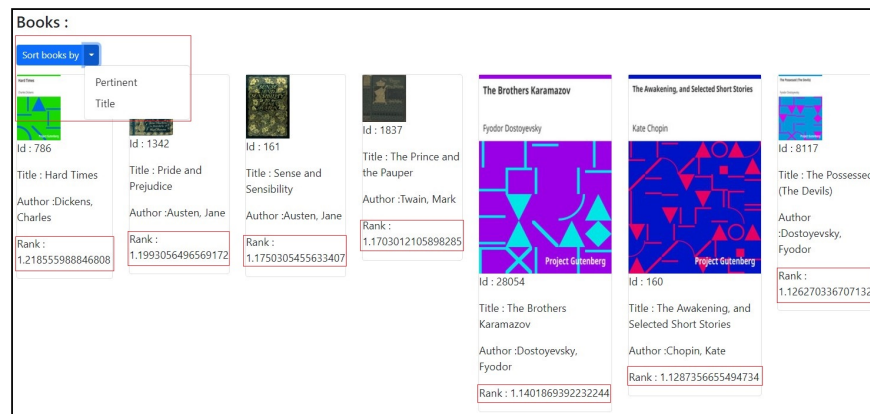


FIGURE 12 – Classement par titre / pertinence

5 Expérimentation et Tests

5.1 Test de performance

Nous avons opté pour utiliser les mots du titre et du sujet pour constituer la table d'indexage car c'est cet ensemble de mots qui définit le caractère du livre de la façon la plus concise.

La recherche simple est presque instantanée, celle avancée est plus lente car elle procède en plusieurs étapes et les possibilités se multiplient et deviennent rapidement exponentielles. Néanmoins, cette dernière s'exécute en un temps très raisonnable.

Si la table d'indexage était composée des mots du texte, il faudrait probablement repenser le processus de recherche, et cela par exemple en stockant les mots sous une structure d'arbre digital, car le nombre de mots indexés par pour un livre influe sur la complexité.

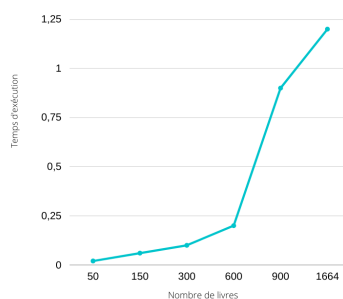


FIGURE 13 – Temps d'exécution pour la recherche simple

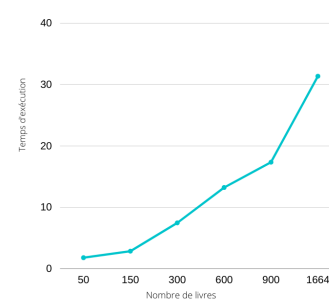


FIGURE 14 – Temps d'exécution pour la recherche avancée

5.2 Tests unitaires

Des tests unitaires ont été développés pour vérifier l'intégrité de la structure et le bon fonctionnement de l'application.

Dans la figure suivante on teste si un livre a bien été sauvegardé. Dans la figure suivante on

```
b1 = {
    "id": "30",
    "title": "daar final project one",
    "author": "oussama",
    "lang": "english",
    "cover": "sorry, cover not available",
    "body": "daar final project is a search engine...etc"
}

def test_bookTest(self):
    serializer = BookMSerializer(data=self.b1)
    if (serializer.is_valid()):
        serializer.save()
        book = BookM.objects.filter(id="30")
        self.assertEqual(len(book), 1)
```

FIGURE 15 – Test unitaire 1

teste le calcul de la distance de jaccard, pour cela on sauvegarde deux livres, on calcule la distance à partir du premier, on fait de même à partir du deuxième, suite à cela les deux valeurs résultantes doivent être égales.

```
def test_graphTest(self):
    for i in range(2):

        serializerb1 = BookMSerializer(data=self.b1)
        serializerb2 = BookMSerializer(data=self.b2)
        if(serializerb1.is_valid()):
            serializerb1.save()
        if(serializerb2.is_valid()):
            serializerb2.save()

        wordsb1 = getWordList(self.b1['title'], self.b1['lang'])
        wordsb2 = getWordList(self.b2['title'], self.b2['lang'])

        d1 = calculJaccardDistance(Counter(wordsb1), Counter(wordsb2))
        d2 = calculJaccardDistance(Counter(wordsb2), Counter(wordsb1))

        serializerGraph = JaccardGraphSerializer( data = {
            "bookId" : self.b1['id'],
            "neighbors" : [self.b2['id']],
            "totalDistance" : d1
        })
        if(serializerGraph.is_valid()):
            serializerGraph.save()

        self.assertEqual(d1,d2)
```

FIGURE 16 – Test unitaire 2

6 Conclusion

Le projet fut très constructif, car il nous a permis de mettre en pratique des notions de théories qui ont été abordés plusieurs fois pendant notre cursus (l'élaboration d'un moteur de recherche performant sur une base der donnée conséquente, une application capable de s'exécuter sur différentes machines et de communiqué).

Le projet nous a permis de comprendre la difficulté et les différentes solutions pour chercher une donnée voulue dans une grande base de données, mais aussi de se familiariser avec Django et de gagner en compétence de la manipulation des APi.

Références

- [1] Closeness Centrality - Neo4j Graph Data Science.
- [2] An Estimated Closeness Centrality Ranking Algorithm and Its Performance Analysis in Large-Scale Workflow-supported Social Networks. *KSII Transactions on Internet and Information Systems*, 10(3), 2016.
- [3] Brad Rees. Similarity in graphs : Jaccard versus the Overlap Coefficient, 12 2021.
- [4] Nagiza F. Samatova, William Hendrix, and John Jenkins. Graph-based Proximity Measures. Technical report, 2014.