Indexation de grandes quantités de données textuelles

Groupe : Paul Dumaitre, Lu Zhengrong, Zineb Lahjouji, Arthir Pichot Utrera, PITOLIN Kévin

Fonctionnalités du programme

Preprocessing
La tokenisation a été implémentée avec simple split et match d'une regex mais aussi en utilisant nltk. Dans la solution finale nous avons utilisé un split (plus rapide).
Il y a possibilité de filtrer le texte avec quelques options (qui peuvent être cumulées) : avec tags XML ou sans, case sensitive ou pas ,stemming, stop-word removal.

Indexing
Nous avons commencé par construire l'index en mémoire, ce qui donnait des bonnes performances en terme de temps mais était gourmand en mémoire.
Par la suite nous avons implémenté l'approche merged-based pour construire l'index sur disque. Cette méthode donne de moins bonnes performances mais elle permet de traiter des quantités de données plus grandes sur une mémoire vive limitée. Il est possible de moduler l'impact en mémoire de l'indexation en modifiant le paramètre memory_limit exposé sous la forme de l'option *index -m.*
Nous n'avons par contre pas implémenté la compression qui permettrait de réduire l'espace disque pris par l'index final.

Querying
Il est possible de réaliser des requêtes conjonctives ou disjonctives sur l'index.
Nous avons implémenté l'algorithme de Fagin (Threshold Algorithm) qui peuvent traiter des requête disjonctives et conjonctives.

Architecture du programme

Au lancement du programme, on exécute la méthode main du fichier querying.py. Ses options sont détaillés dans le Readme.

querying.py
Contient la méthode main, dans laquelle on commence par construire l'index en appelant la méthode indexing du fichier indexing.py.
Puis, on demande à l'utilisateur les termes de sa requête, et on lui renvoie les résultats pour les quatre options (naïf/Fagin, Conjonctif/Disjonctif).

Le requêtage naïf se fait en récupérant les résultats par les méthodes query_with_naive_disj_algo et query_with_naive_conj_algo, sous forme d'un dictionnaire dont les clefs sont des noms de documents et les valeurs leurs scores.
Ces deux méthodes appellent respectivement naive_disj_algo et naive_conjj_algo.
Par la suite, nous trions et affichons ces résultats avec la méthode sort_and_print_dict.

Le requêtage avec l'algorithme de Fagin s'opère avec la méthode query_with_threshold_algo, qui extrait la partie de l'inverted index qui concerne les termes de la requête (triés par nom de document) et reproduit cet extrait, cette fois trié par ordre de score décroissant. On appelle ensuite la méthode threshold_algo, qui implémente l'algorithme du même nom. Les paramètres de cette méthode comprennent le nombre de résultats désirés et un flag indicant si la requête est conjonctive ou disjonctive.

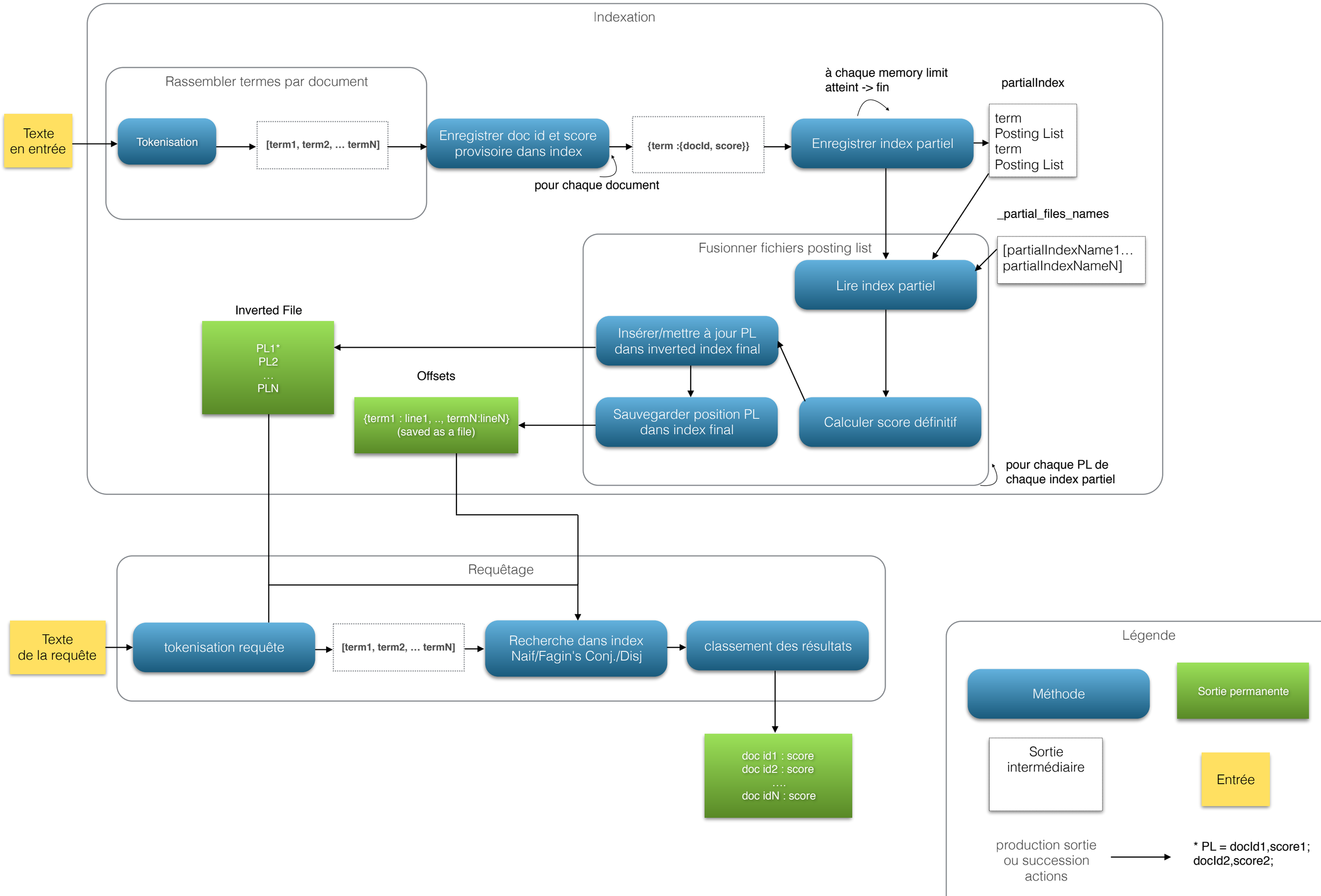
Indexing.py

Le schéma de la page suivante donne la structure de notre algorithme en utilisant l'indexation merged-based. Pour l'indexation en mémoire, on n'enregistre pas de couple docid-score provisoires dans un fichier mais on crée directement l'index, une structure suivant cette forme : {« term1 » :{docId1:score, docId2:score}, « term2 »:{docId1:score}}.

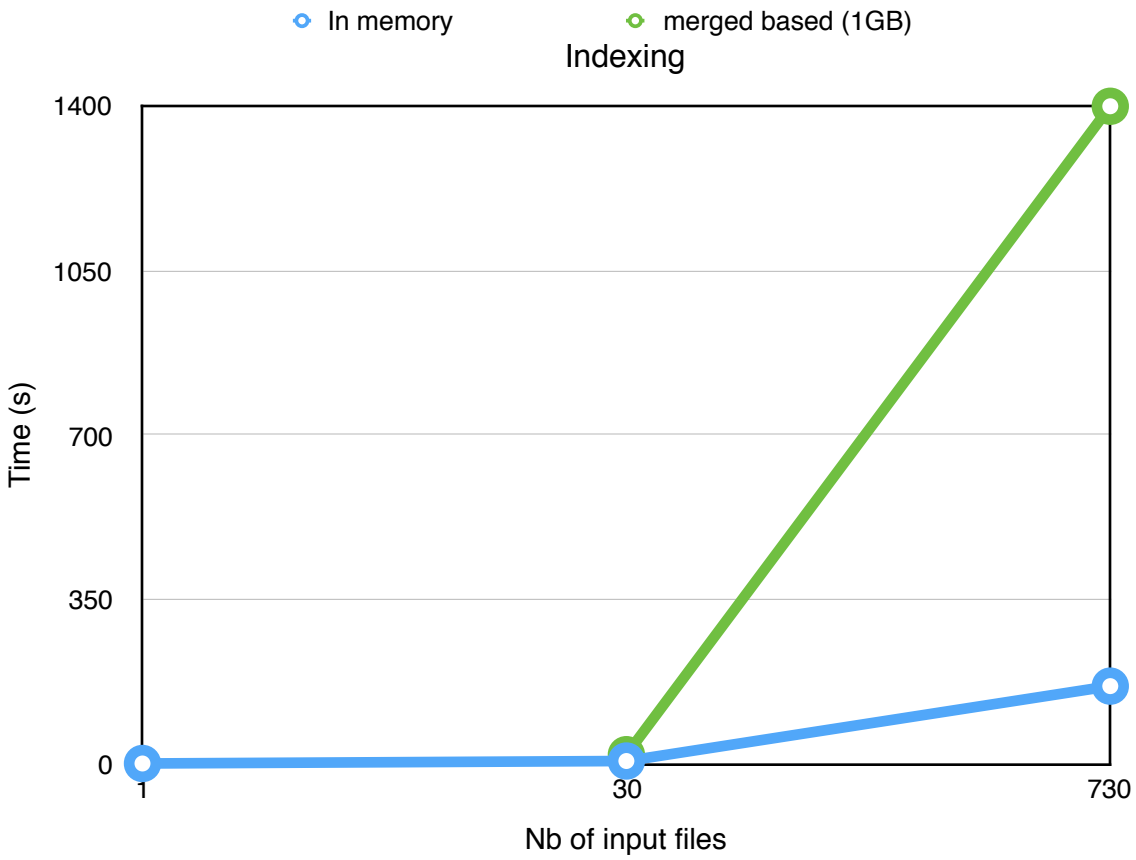
Dans cette structure de données, les scores sont provisoires dans un premier temps (simple fréquence des termes) puis ils sont mis à jour (multipliés par l'idf)

Le tableau ci-dessous fait le lien entre le schéma et les méthodes utilisées :

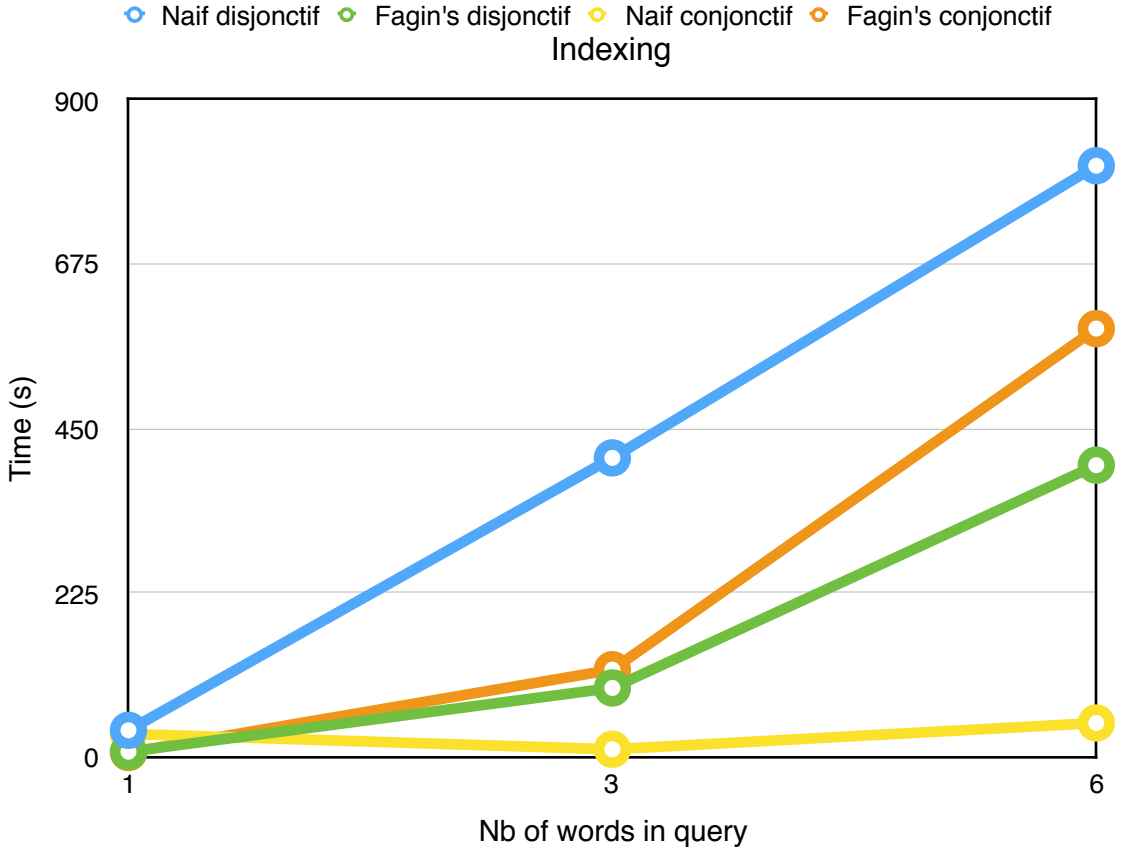
Elément du graphe	Méthode(s) principales
tokenisation	tokenize_string_split (ou tokenize_string_by_doc_nltk)
indexation	index_files ou index_text
rassembler termes par doc id	index_documents
enregistrer doc id et score dans index partiel	write_partial_index
fusionner fichiers posting lists	merge_partial_indexes
lire index partiel	read_terms_from_partial_file
calculer scores définitifs	update_scores_with_idf (memory), calculate_all_term_pl_scores (merged-based)
tokenisation requête	get_terms
recherche dans index	query_with_threshold_algo, query_with_naive_disj_algo, query_with_naive_conj_algo
classement des résultats	sort_and_print_dict


Indexation
Rassembler termes par document
Texte
en entrée
Tokenisation
[term1, term2, … termN]
Enregistrer doc id et score
provisoire dans index
{term :{docId, score}}
pour chaque document
à chaque memory limit
atteint -> fin
Enregistrer index partiel
partialIndex
term
Posting List
term
Posting List
_partial_files_names
[partialIndexName1…
partialIndexNameN]
Fusionner fichiers posting list
Lire index partiel
Insérer/mettre à jour PL
dans inverted index final
Sauvegarder position PL
dans index final
Calculer score définitif
pour chaque PL de
chaque index partiel
Inverted File
PL1*
PL2
…
PLN
Offsets
{term1 : line1, .., termN:lineN}
(saved as a file)
Requêtage
Texte
de la requête
tokenisation requête
[term1, term2, … termN]
Recherche dans index
Naif/Fagin's Conj./Disj
classement des résultats
doc id1 : score
doc id2 : score
….
doc idN : score
Légende
Méthode
Sortie permanente
Sortie
intermédiaire
Entrée
production sortie
ou succession
actions
* PL = docId1,score1;
docId2,score2;

Performances du programme



On voit ici que l'impact en mémoire réduit de la méthode merged-based est compensée par son impact en temps (à cause des opération I/O). Pour une indexation rapide il parait judicieux d'utiliser l'index en mémoire dès que c'est possible.



Il n'y a ici quasiment pas de différence de tendance entre fagin's conjonctif et disjonctif. Cependant on remarque que le requêtage naif disjontif est le plus couteux en temps (sa progression semble linéaire). Les chiffres du naif conjonctif semblent curieusement bas.