Project DAAR
Mark-Compact GC for MINIZAM in C

Submitted by

Hejun Cao
Ruolin Zhou
Weida Liu

M1 Science et Technologie du Logiciel 2023-2024
Sorbonne Universite (SU UMPC)

Under the guidance of

Bùi Xuân Bình Minh
Sorbonne Universite (SU UPMC)

20/01/2025

Contents

1 Introduction

Dans le cadre de notre projet DAAR, nous avons choisi de développer une application web/mobile de moteur de recherche pour une bibliothèque numérique, conformément aux exigences du "CHOIX A". Ce projet s'inscrit dans un contexte où les bases de données textuelles de grande envergure, comme la bibliothèque de Gutenberg, rendent la recherche manuelle inefficace. L'objectif principal est donc de concevoir une solution performante et intuitive permettant aux utilisateurs de localiser efficacement des documents textuels au sein d'une bibliothèque comprenant au minimum 1664 livres, chaque livre contenant au moins 10 000 mots.

Le projet s'articule autour de plusieurs objectifs clés :

- Un système de gestion des utilisateurs.
  Pour gérer les recommandations personnalisées uniques de chaque utilisateur et le tri des résultats de recherche, ainsi que la fonction de favoris correspondante.

- KMP Search.
  La recherche de contenu par expressions régulières à l'aide de KMP Search appris dans les cours de DAAR permet de trouver tous les contenus textuels de tous les livres et les résultats sont triés en fonction du nombre d'occurrences.

- Recommandation de contenu.
  Nous utilisons l'algorithme PageRank pour calculer le score PageRank de chaque nœud (livre) dans le graphe de similarité des livres, les livres ayant un score élevé étant considérés comme les plus appréciés par l'utilisateur, tout en filtrant les résultats en fonction de l'auteur et de la catégorie afin de nous assurer que les livres recommandés correspondent aux intérêts de l'utilisateur.

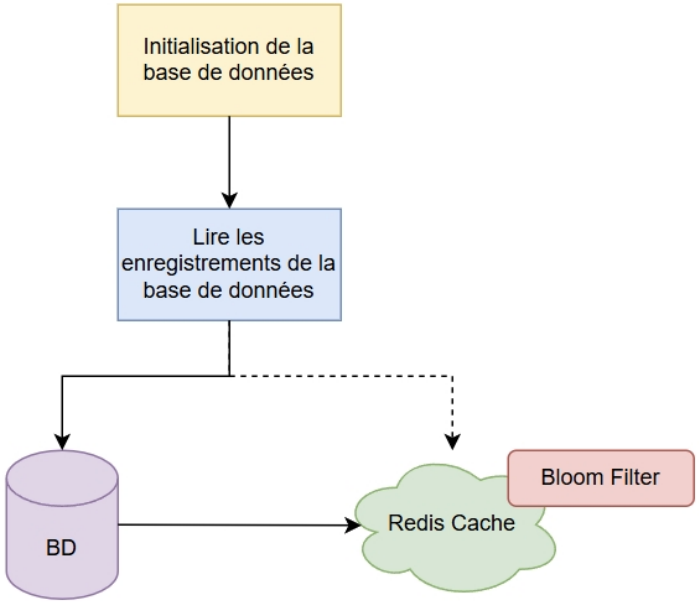
En plus des défis techniques liés à l'implémentation des algorithmes de recherche et de classement, une attention particulière sera portée à la performance de l'application, mesurée à travers des tests sur des volumes de données conséquents.

2 Implémentation détaillée

2.1 Gestion des utilisateurs

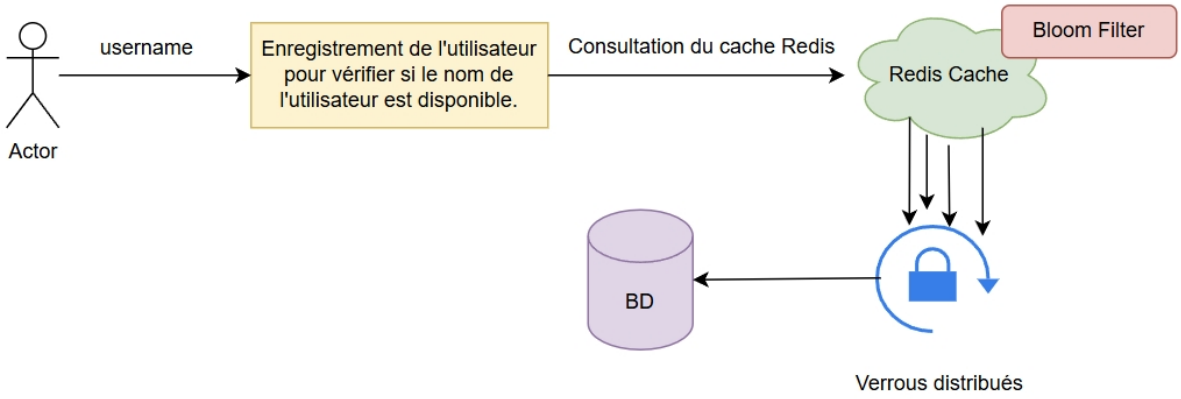
Notre module utilisateur met en œuvre les fonctions d'enregistrement, de connexion, de modification et de déconnexion des utilisateurs, ainsi que les fonctions de collecte de livres.

Nous avons mis en œuvre des modules utilisateurs pour la protection des bases de données à l'aide de filtres de Bloom, de verrous distribués, de buckets limitant les flux et de la mise en cache Redis.



À chaque fois que le projet s'exécute, nous lisons d'abord les données de la base de données, puis nous les synchronisons avec le cache Redis pour nous assurer qu'il n'y a pas de doublons.

Entre-temps, nous avons conçu les solutions suivantes pour faire face à d'éventuelles attaques de bases de données : S'il y a un grand nombre d'appels à l'interface d'enregistrement/de connexion avec le même nom d'utilisateur, nous utiliserons d'abord des filtres Bloom pour le filtrage, puis des verrous distribués pour verrouiller les noms d'utilisateur afin de garantir qu'une seule demande peut interroger la base de données en même temps.



En outre, nous avons mis en place une fonction de collecte facile. L'utilisateur crée un groupe et y place le livre correspondant. Les données du livre dans la collection affectent le poids de l'algorithme PageRank.

2.2 KMP Search

2.2.1 Analyse

L'algorithme de Knuth-Morris-Pratt, également connu sous le nom d'algorithme KMP, est un algorithme de correspondance de chaînes de caractères. Son idée de base est que lorsqu'il y a une non-concordance de chaîne, vous pouvez savoir quelle partie du texte doit correspondre.

L'idée centrale de l'algorithme KMP est d'utiliser des informations connues pour réduire autant que possible le nombre de correspondances. Plus précisément, il peut être divisé en deux étapes :

Compute le tableau Carryover et puis utilise KMP Search.

2.2.2 Structure de donnees

Comme pour le Projet précédent, nous lisons toujours le texte entier, ligne par ligne, et nous le recherchons. Par conséquent, sa structure de données est une simple chaîne de caractères.

2.2.3 Implémentation

L'algorithme KMP peut générer un tableau LST à partir d'une chaîne de motifs prétraitée, puis convertir le tableau LST en un tableau de report, où report[i] indique la position à laquelle la chaîne de motifs doit sauter pour poursuivre la comparaison si le i-ème caractère de la chaîne de motifs ne correspond pas à un caractère de la chaîne de texte.

Algorithm 1: ComputeCarryover

```
1  Inputs character array
2  Outputs The final LST array
3  Calculations
4  for i ← 0 to len − 1 by 1 do
5  |  if j == −1 OR pattern[i] == pattern[j] then
6  |  |  i ← i + 1
7  |  |  j ← j + 1
8  |  |  LST[i] ← j
9  |  end
10 |  else
11 |  |  j ← next[j]
12 |  end
13 end
14 for i ← 1 to len by 1 do
15 |  if LST[i] ≥ 0 and i < len and pattern[i] == pattern[LST[i]] then
16 |  |  LST[i] ← LST[LST[i]]
17 |  end
18 end
```

L'algorithme utilise une table de sauts précalculée (carryover) pour gérer efficacement le processus de correspondance. Chaque fois qu'une correspondance échoue, l'algorithme ajuste rapidement le pointeur de motif à travers la table de saut pour continuer à rechercher une correspondance. Si une correspondance est réussie, elle est enregistrée et le pointeur est réinitialisé pour poursuivre la recherche de l'occurrence suivante. Enfin, le contenu correspondant est renvoyé.

Algorithm 2: KPM (Knuth-Morris-Pratt)

Inputs

- $factor$ = a string (the pattern to search for)
- $file$ = a string (the file path)

Initializations

- $factorChar$ = Convert $factor$ to a character array
- $carryover$ = Result of calling $computeCarryover(factorChar)$
- $lineNumber = 0$

Calculations

while *there is a new line in the file* **do**
  $lineNumber \leftarrow lineNumber + 1$
  $line$ = read the next line from the file
  $i \leftarrow 0$, $j \leftarrow 0$
  $textLen$ = length of $line$
  $factorLen$ = length of $factorChar$
  while $i < textLen$ **do**
    if $j == -1$ *OR* $line[i] == factorChar[j]$ **then**
      $i \leftarrow i + 1$
      $j \leftarrow j + 1$ **if** $j == factorLen$ **then**
        $j \leftarrow carryover[j]$
      end
    end
    else
      $j \leftarrow carryover[j]$
    end
  end
end

Outputs

- The matches found in the file (if any) with the corresponding line number

2.3 PageRank

2.3.1 Analyse

L'algorithme PageRank a été proposé à l'origine par Larry Page et Sergey Brin pour mesurer l'importance des pages web. L'idée de base est que s'il y a beaucoup de pages de haute qualité (PageRank élevé) qui pointent vers une page, alors cette page a plus de chances d'être de haute qualité. Mathématiquement, le PageRank peut être considéré comme une sorte de « distribution à l'état stable » sur un graphique, modélisant la distribution de probabilité d'un spectateur aléatoire qui continue à cliquer sur des hyperliens ou à passer aléatoirement d'un nœud à l'autre pour atteindre d'autres nœuds.

Pour un graphe orienté contenant N nœuds, la valeur de PageRank de chaque nœud i est désignée par PR(i) et la formule itérative de base pour PageRank peut être écrite :

$$PR(i) = \frac{1-d}{N} + d \sum_{j \in \mathrm{In}(i)} \frac{PR(j)}{\mathrm{OutDegree}(j)}$$

Parmi eux:

- d est le facteur d'amortissement, généralement 0,85, ce qui signifie que dans le modèle de marche aléatoire, la probabilité d'utiliser un lien hypertexte pour continuer à sauter est d, tandis que la probabilité de sauter vers n'importe quel nœud directement et aléatoirement sans utiliser de lien hypertexte est de 1 - d.
- N est le nombre total de nœuds.
- In(i) désigne l'ensemble des nœuds dont l'arête pointe vers i (c'est-à-dire les voisins entrants de i).
- OutDegree(j) indique le degré de sortie du nœud j (c'est-à-dire le nombre d'arêtes partant du nœud j).

Dans l'algorithme PageRank, si un nœud n'a pas d'arêtes sortantes (outdegree 0), toute sa valeur PageRank ne peut pas être transmise à d'autres nœuds par le biais de « liens ».

PageRank est un algorithme itératif :

- Commencez par donner à tous les nœuds une valeur initiale.(Par example 1/N)
- Il est ensuite mis à jour de manière itérative selon la formule. Jusqu'à ce que la différence entre les deux itérations précédentes et suivantes soit inférieure à un certain seuil ou que le nombre maximal d'itérations soit atteint, la convergence est considérée comme atteinte et l'itération est arrêtée.

2.3.2 Structure de donnees

Dans la mise en œuvre du PageRank de ce projet, les quatre types suivants de structures de données centrales sont utilisés pour construire et traiter efficacement des structures de graphe à grande échelle :

- Liste d'adjacence positive
  Définition de la structure : Map<Long, List<Long>> graph
  Signification : Key représente l'identifiant du noeud (dans ce projet, il correspond à l'identifiant du livre), Value est une liste contenant les identifiants de tous les noeuds vers lesquels ce noeud pointe (ou auxquels il est similaire).

- Liste d'adjacence inversée
  Définition de la structure : Map<Long, List<Long>> reverseGraph
  Signification : la clé représente l'identifiant du noeud, la valeur est l'ensemble des « autres identifiants de noeuds pointant vers ce noeud ».

- Carte des degrés de sortie (Out-degree Map)
  Définition de la structure : Map<Long, Integer> outDegreeMap
  Signification : Enregistrer la relation de correspondance entre l'identifiant du nœud et son degré de sortie (nombre d'arêtes sortantes).

- Stockage du PageRank (PageRank Map)
  Définition de la structure : Map<Long, Double> pageRank
  Signification : Enregistre le score PageRank actuel de chaque noeud.

2.3.3 Implémentation

Nous devons d'abord construire la table de voisinage inverse et la table des degrés sortants.

Algorithm 3: Build Reverse Graph

Input: *graph*: A map where each key is a node, and the value is a list of its outgoing neighbors.
Output: *reverseGraph*: A map where each key is a node, and the value is a list of its incoming neighbors.

```
1  reverseGraph ← empty map with lists as values;
2  foreach node ∈ graph.keys() do
3  |  reverseGraph.put(node, empty list);
4  end
5  foreach node ∈ graph.keys() do
6  |  outNodes ← graph.get(node);
7  |  if outNodes is not null then
8  |  |  foreach outNode ∈ outNodes do
9  |  |  |  reverseGraph.get(outNode).add(node);
10 |  |  end
11 |  end
12 end
13 return reverseGraph;
```

Algorithm 4: Build Out-Degree Map

Input: *graph*: A map where each key is a node, and the value is a list of its outgoing neighbors.

Output: *outDegreeMap*: A map where each key is a node, and the value is the count of its outgoing neighbors.

```
outDegreeMap ← empty map;
foreach node ∈ graph.keys() do
    outList ← graph.get(node);
    outDegree ← (outList is null) ? 0 : outList.size();
    outDegreeMap.put(node, outDegree);
end
return outDegreeMap;
```

L'idée spécifique de la mise en œuvre de notre algorithme est la suivante :

Nous commençons par déterminer si le graphe d'entrée est vide. Si le graphe est vide ou n'a pas de nœuds, il renvoie directement le résultat null. Cela permet d'éviter les exceptions de pointeur nul dans les calculs ultérieurs.

Les structures de données auxiliaires (graphe inversé et carte des degrés de sortie) sont ensuite construites.

Ensuite, nous attribuons une valeur initiale de PageRank à chaque nœud PR(node) = $\frac{1}{N}$ dont N est le nombre total de nœuds, ce qui indique que tous les nœuds ont la même importance initiale.

Nous calculons ensuite la contribution des nœuds en suspens et répartissons la valeur du PageRank de tous les nœuds en suspens de manière égale entre tous les nœuds du graphe selon la formule suivante.

$$\text{danglingSum} = \sum_{\text{node}\in\text{graph.keys()},\text{outDegree[node]}=0} PR(\text{node})$$

La contribution de chaque nœud au nœud suspendu est alors $\frac{\text{danglingSum}}{N}$.

Immédiatement après, nous mettons à jour le PageRank nœud par nœud et, pour chaque nœud, nous calculons la nouvelle valeur du PageRank à l'aide de la formule :

$$PR(\text{node}) = \frac{1-d}{N} + d \cdot \frac{\text{danglingSum}}{N} + d \cdot \sum_{\text{inNode}\in\text{reverseGraph[node]}} \frac{PR(\text{inNode})}{\text{outDegree[inNode]}}$$

Enfin, nous calculons le changement total entre cette itération et l'itération précédente :

$$\text{diff} = \sum_{\text{node}\in\text{graph.keys()}} |PR_{\text{new}}(\text{node}) - PR_{\text{old}}(\text{node})|$$

Si diff¡ϵ, les valeurs de PageRank sont considérées comme convergentes et l'itération peut être arrêtée prématurément pour renvoyer les valeurs de PageRank de tous les nœuds.

Algorithm 5: Compute PageRank

Input: *graph*: A map where keys are book IDs, values are lists of directly connected book IDs.
dampingFactor: Damping coefficient, typically set to 0.85.
maxIterations: Maximum number of iterations.
epsilon: Convergence threshold, used to terminate early if the difference between two iterations is small.
Output: *pageRank*: A map where keys are book IDs and values are their corresponding PageRank scores.

if *graph is null or empty* **then**
  return *empty map*;
end
$reverseGraph \leftarrow$ **BuildReverseGraph**$(graph)$;
$outDegreeMap \leftarrow$ **BuildOutDegreeMap**$(graph)$;
$nodeCount \leftarrow$ size of $graph$;
$pageRank \leftarrow$ map initialized to $1.0/nodeCount$ for each node;
for $iter \leftarrow 0$ ***to*** $maxIterations - 1$ **do**
  $danglingSum \leftarrow 0.0$;
  foreach $node \in graph.keys()$ **do**
    if $outDegreeMap[node] = 0$ **then**
      $danglingSum \leftarrow danglingSum + pageRank[node]$;
    end
  end
  $newPageRank \leftarrow$ empty map;
  foreach $node \in graph.keys()$ **do**
    $rank \leftarrow \frac{1.0 - dampingFactor}{nodeCount} + dampingFactor \cdot \frac{danglingSum}{nodeCount}$;
    $inNeighbors \leftarrow reverseGraph[node]$;
    if $inNeighbors$ *is not empty* **then**
      foreach $inNode \in inNeighbors$ **do**
        $outDeg \leftarrow outDegreeMap[inNode]$;
        if $outDeg > 0$ **then**
          $rank \leftarrow rank + dampingFactor \cdot \frac{pageRank[inNode]}{outDeg}$;
        end
      end
    end
    $newPageRank[node] \leftarrow rank$;
  end
  $diff \leftarrow$ **CalculateDifference**$(pageRank, newPageRank)$;
  $pageRank \leftarrow newPageRank$;
  if $diff < epsilon$ **then**
    break;
  end
end
return *pageRank*;