UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DOMENIUL: Calculatoare și Tehnologia Informației
SPECIALIZAREA: Tehnologia Informației

Indexare și căutare

--REGĂSIREA INFORMAȚIILOR PE WEB--

Profesor îndrumător
Ș.l. dr. ing. Alexandru Archip

Student
Temătoru Daniel-grupa 1410B

Capitolul 1. Problema propusă spre rezolvare

Prima componentă de proiect a disciplinei „Regăsirea Informațiilor pe WEB” vă propune să realizați un set de modificări asupra aplicațiilor dezvoltate pe parcursul laboratoarelor 1 – 4, astfel încât:

1. să obțineți o procesare adecvată a cuvintelor determinate în cadrul unui text;
2. să studiați și alte tipuri de baze de date și mecanisme de stocare de date (precum MongoDB);
3. să studiați alte metode de realizare a operațiilor de căutare.

Astfel, pentru prima cerință de mai sus, metodele de construire a indecșilor direcți și inverși trebuie să implementeze mecanisme prin intermediul cărora un cuvânt de dicționar să fie adus la așa numita „**formă canonică**”. Indecșii astfel determinați, vor fi apoi stocați prin intermediul unor baze de date ne-relaționale, precum MongoDB.

După cum a fost amintit în cadrul laboratorului nr. 4, căutarea booleană nu include mecanisme de determinare a unui eventual scor de relevanță pentru rezultatele identificate. A treia cerință vă propune să studiați impactul distanței de tip cosinus asupra rezultatelor obținute de un motor de căutare.

Capitolul 2. Prezentarea aplicației propuse ca soluție

Aplicația propusă a fost scrisă în limbajul Java, utilizând mediul de dezvoltare Eclipse IDE. De asemenea, au fost utilizate 2 librării externe, și anume **JSoup** (pentru procesarea fișierelor HTML) și **Gson** (pentru procesarea fișierelor JSON).

Interfața cu utilizatorul este reprezentată de linia de comandă, meniul și interacțiunea fiind în mod text, simplu.

Exemplu de execuție a aplicației

1. ***căutare booleană***

```
------------------------------------------------------------------------
Aplicare cautare booleana
Introduceti ce vreti sa cautati:
technical work
------------------------------------------------------------------------
Incepe cautarea.
Rezultatele gasite din cautarea booleana  sunt:

        E:\Eclipse\New folder\Proiect_01_RIW\stackoverflow\about\groups\iesg\appeals\index.html
        E:\Eclipse\New folder\Proiect_01_RIW\stackoverflow\about\index.html
        E:\Eclipse\New folder\Proiect_01_RIW\stackoverflow\about\groups\iab\index.html
        E:\Eclipse\New folder\Proiect_01_RIW\stackoverflow\about\groups\iaoc\index.html
        E:\Eclipse\New folder\Proiect_01_RIW\stackoverflow\about\groups\directorates\index.html
        E:\Eclipse\New folder\Proiect_01_RIW\stackoverflow\about\groups\iesg\index.html
        E:\Eclipse\New folder\Proiect_01_RIW\stackoverflow\about\groups\index.html
------------------------------------------------------------------------
```

2. ***căutare vectorială***

```
------------------------------------------------------------------------
Aplicare cautare vectoriala
Introduceti ce vreti sa cautati:
technical work
------------------------------------------------------------------------
Incepe cautarea.
Rezultatele gasite din cautarea vectoriala sunt:

        E:\Eclipse\New folder\Proiect_01_RIW\stackoverflow\about\index.html=1.0
        E:\Eclipse\New folder\Proiect_01_RIW\stackoverflow\about\groups\iab\index.html=1.0
        E:\Eclipse\New folder\Proiect_01_RIW\stackoverflow\about\groups\directorates\index.html=1.0
        E:\Eclipse\New folder\Proiect_01_RIW\stackoverflow\about\groups\iesg\index.html=1.0
        E:\Eclipse\New folder\Proiect_01_RIW\stackoverflow\about\groups\index.html=1.0
```

Capitolul 3. Explicarea modulelor aplicației

1. Creare index direct + indice „tf”

Acest modul creează index-ul direct al tuturor documentelor HTML găsite în directoarele și subdirectoarele website-ului sursă. Pentru parcurgerea directoarelor, este folosită o coadă, deoarece se dorește o parcurgere secvențială și nu recursivă.

Fișierele țintă sunt cele cu extensia „**.html**”, însă aplicația verifică și conținutul fișierelor, nebazându-se doar pe extensie. Există fișiere fără extensie ce au conținut HTML, ce sunt luate totuși în considerare.

Din fiecare fișier HTML, este preluat textul folosind parser-ul intern al librăriei **JSoup**, acesta fiind stocat într-un fișier separat. Textul rezultat este procesat caracter cu caracter, din aceleași considerente în ceea ce privește viteza de lucru. Se extrag astfel toate cuvintele din textul obținut, iar acestea sunt trecute prin 3 filtre:

1. sunt testate contra unei liste de **excepții** = cuvinte ce prezintă interes, dar nu se găsesc în dicționar, așa încât trebuie tratate ca atare, în forma lor curentă
2. se verifică dacă nu sunt de tip **stopwords** = cuvinte ce nu prezintă interes pentru relevanța documentelor rezultate din căutare; sunt ignorate, pur și simplu
3. **cuvintele de dicționar** sunt trecute printr-un proces de **stemming**, utilizând ***algoritmul lui Porter***; se aduc astfel la o formă de bază, eliminându-se terminațiile ce determină forme diverse ale aceleiași noțiuni.

Se creează, de asemenea, un fișier de mapare ce indică, pentru fiecare document în parte, locația fișierului de index direct, ce conține perechi de forma <cuvânt, număr de apariții>.

Indicele ***„tf” (frecventa de aparitie a cuvantului k in cadrul documentului d)*** este creat tot în această etapă, datorită ușurinței cu care se poate accesa fiecare document în parte, local

2. Creare index indirect + indice „idf”

În această etapă, este creat index-ul indirect, ce se bazează, în mare, pe logica de funcționare a etapei precedente. Se preia fiecare fișier de index direct creat anterior pentru a obține lista de cuvinte din acel document. Fiecare cuvânt în parte este luat în considerare în crearea structurii finale de date, ce conține perechi de forma <cuvânt, <document, număr de apariții>>. Astfel, sunt „unificate” toate documentele într-un index indirect global, menținut într-un fișier JSON.

Indicele ***„idf”(frecventa inversa de aparitie a cuvantului k in cadrul multimii de documente D)*** este creat în această etapă, deoarece avem acces global la toate documentele, iar formula acestui indice utilizează numărul total de documente și numărul de documente în care apare un cuvânt. Este cel mai eficient în a accesa aceste informații chiar atunci când este creat index-ul indirect.

La fel ca la etapa anterioară, se obține și un fișier de mapare ce va conține locația index-ului indirect pentru fiecare fișier în parte, deși se creează și unul global, cu toate cuvintele din sursele extrase.

3. Încărcare index indirect în memorie (pentru căutarea booleană)

Atunci când se dorește o căutare booleană, trebuie să avem index-ul indirect încărcat în memorie, într-o structură de date ce ne permite căutarea pe bază de chei (cuvintele introduse de utilizator, legate de operatori). În această aplicație, am folosit **TreeMap<String, HashMap<String, Integer>>**. Practic, se parsează fișierul JSON cu index-ul indirect și se încarcă, intrare cu intrare, în structura de date menționată, perechile cheie : valoare.

4. Căutare booleană

Este cel mai simplu model de căutare, ce ne arată dacă un document conține sau nu o anumită cheie dată de utilizator în interogare. Nu are posibilitatea de a oferi informații legate de relevanță, sub o anumită ordonare.

Aplicația folosește, pentru căutarea booleană, index-ul indirect încărcat în memorie, respectiv interogarea dată de utilizator, sub forma: CHEIE ***OPERATOR*** CHEIE ***OPERATOR*** CHEIE ..., unde ***OPERATOR*** = un element din mulțimea {AND, OR, NOT}.

Pentru parsarea interogării, se folosesc 2 stive: una de operanzi și una de operatori. Ordinea de parsare este de la dreapta la stânga. Toate cuvintele din interogare sunt trecute prin aceleași 3 procese de filtrare menționate anterior.

Principiul de căutare este simplu: pentru o anumită cheie găsită, se returnează setul de documente care conține acea cheie, folosind index-ul indirect. Apoi, pe acel set se aplică operatorul din stânga cheii de căutare curente.

Pentru operatorul **AND**, se parcurge mulțimea cu cardinalul mai mic și se adaugă la rezultat documentul curent, doar dacă acesta există în cealaltă mulțime. La operatorul **OR**, se aplică principiul invers: se parcurge mulțimea cu cardinalul mai mare. Aceste „optimizări" rezultă din dorința unei viteze cât de cât mai bune față de cazul obișnuit.

Operatorul **NOT** este cel mai simplu, fiind necomutativ: se iterează prima mulțime și se verifică dacă documentul curent nu există în cea de-a doua, caz în care se adaugă la rezultat.

5. Creare vectori asociați documentelor HTML

Conform algoritmului de căutare vectorială, documentele HTML trebuie reprezentate sub formă de vectori, ce conțin elemente de tipul <cuvânt, *indice* ***tf*** x *indice* ***idf***>.

În primul rând, se folosește fișierul de mapare al index-ului indirect pentru preluarea listei de documente (mai simplu ca viteză decât să parcurgem din nou lista de directoare).

Pentru fiecare document în parte, se preiau toate cuvintele ce există în acel document, folosind index-ul direct local, creat anterior (se află într-un fișier JSON pe disc).

Pentru fiecare cuvânt din acel document, se calculează indicii ***tf*** și ***idf***, și se adaugă intrarea <cuvânt, *tf* x *idf*> în structura de date finală.

Deoarece acest proces este de durată pentru seturi mari de date, rezultatul este stocat într-un fișier JSON și este încărcat la cerere pentru efectuarea de căutări vectoriale. Procesul de încărcare are un timp infim față de procesarea propriu-zisă

6. *Încărcare vectori asociați în memorie (pentru căutarea vectorială)*

La fel ca la căutarea booleană, este necesar ca structura de date cu ajutorul căreia căutăm să se afle în memorie, deci înainte de a folosi căutarea vectorială pentru prima dată, se încarcă în memorie, într-o structură de tip **HashMap<String, TreeMap<String, Double>>** vectorii asociați documentelor HTML, pregătind astfel calculul de similarități ce urmează în algoritm.

7. *Căutare vectorială*

Principiul căutării, în aplicația propusă, este similar cu cel de la căutarea booleană. Se parsează interogarea utilizatorului și se împarte în cuvinte, utilizând aceleași 3 filtre pentru cuvintele rezultate.

Lista de cuvinte cheie este transformată, apoi, într-un vector de aceeași formă ca și cei creați în etapa anterioară: <cuvânt, *tf* x *idf*>, cu observația că indicele ***tf*** este ***local interogării***! Cu alte cuvinte, interogarea este tratată ca un document în sine, așa încât *tf*-ul este calculat folosind interogarea ca și parametru pentru document.

Se sortează descrescător structura de documente obținute, folosind criteriul de relevanță ca și element de comparație, ignorându-se documentele cu relevanță 0. Utilizatorului îi este prezentată lista finală, ordonată de documente, preluate dintr-o structură **SortedSet<HashMap.Entry<String, Double>>**.

```
static <k1,k2 extends Comparable<? super k2>>SortedSet<Map.Entry<k1,k2>>Sorteaza_Scor(Map<k1,k2>map){

    SortedSet<Map.Entry<k1,k2>>Input=new TreeSet<Map.Entry<k1, k2>>(
            new Comparator<Map.Entry<k1, k2>>() {
                @Override public int compare(Map.Entry<k1, k2>set1,Map.Entry<k1,k2>set2) {
                    int rez=set2.getValue().compareTo(set1.getValue());
                    if(rez!=0)
                    {
                        return rez;
                    }
                    else
                    {
                        return 1;
                    }
                }

            }
            );
    Input.addAll(map.entrySet());
    return Input;

}
```