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

Proiect

Regăsirea Informațiilor pe WEB

Student
Dobrincu Octavian
Grupa 1408A

Iași, 2020

Cuprins

Etapa 1. Indexare și căutare

1.1. Contextul primei etape

Prima etapă a proiectului la disciplina *Regăsirea Informațiilor pe WEB* presupune rezolvarea următoarelor probleme:

1. Obținerea unei procesări adecvate a cuvintelor determinate în cadrul unui text;
2. Studierea unor mecanisme de stocare de date (precum MongoDB);
3. Studierea unor metode de realizare a operațiilor de căutare, altele decât cele studiate la laborator.

În acest context, realizarea indecșilor direcți și inverși trebuie realizată pentru obținerea unei *forme canonice* ce urmează a stoca aceasta într-o bază de date ne-relațională, în cazul nostru MongoDB. De asemenea, ca mecanism de căutare se va implementa căutarea booleană studiată la laborator, la care se adaugă implementarea unei soluții pentru calcularea distanței de tip cosinus pentru realizarea unui *criteriu de relevanță* asupra rezultatelor căutării.

1.2. Procesarea cuvintelor

Prima parte a acestei etape constă în procesarea cuvintelor din cadrul unui text. În acest context, s-a creat în primă fază, indexul direct al fiecărui fisier procesat. În acest scop, documentele din colecția „index_direct” din baza de date au o structură ca cea din Figura 1.1:


```
{
    "_id" : ObjectId("5e8c864c4d82134d3add1584"),
    "document" : "..\\input\\3.html",
    "words" : {
        "Chris" : 56,
        "evan" : 86,
        "actor" : 40,
        "-" : 1,
        "wikipedia" : 21,
        "document" : 1,
        "documentel" : 1,
        "classnam" : 1,
        "client-j" : 1,
        "rlconf" : 1,
        "wgbreakfram" : 1,
        "1" : 27,
        "wgseparatortransformt" : 1,
        "wgdigittransformt" : 1,
```


Figura 1.1: Structura indexului direct

În ceea ce privește indexul invers, acesta a fost realizat după crearea indexului direct. Astfel, s-a obținut structura din Figura 1.2:


```
"_id" : ObjectId("5e8c86724d82134d3addcb9b"),
"word" : "Avengers",
"documents" : [
    [
        "..\\input\\1.html",
        254
    ],
    [
        "..\\input\\2.html",
        206
    ],
    [
        "..\\input\\3.html",
        14
    ],
    [
        "..\\input\\5.html",
        1
    ]
]
```


Figura 1.2: Structura indexului invers

1.2.1. Extragerea rădăcinii cuvântului (stemming)

În procesarea cuvintelor, obținerea rădăcinii din care provine un cuvânt reprezintă o parte importantă datorită utilității ce se obține atât din punct de vedere al memoriei folosite cât și a rezultatelor căutărilor ulterioare ce se realizează pe baza acestora. Astfel, în baza de date nu regăsim cuvintele în forma în care au fost scrise ci sub forma acestei rădăcini.

Algoritmii de *stemming* sugerați au fost algoritmul lui Porter și algoritmul lui Lovins, primul dintre acestea fiind cel folosit.

Algoritmul lui Porter constă în 5 faze, dintre care, prima și ultima sunt la rândul lor împărțite, ajungându-se la un total de 8 faze. În acestea, se aplică diferite operațiuni în care sufixul unui cuvânt este transformat în alt sufix pe baza unei condiții.

Acest algoritm consideră că orice cuvânt are următoarea structură: $[C](VC)^m[V]$.

În această structură C reprezintă o consoană sau un grup de consoane, V reprezintă o vocală sau un grup de vocale, iar m este numărul de apariții a grupului VC. Astfel, un cuvânt poate avea un grup opțional de consoane la început și unul opțional de vocale la sfârșit.

Asupra acestor cuvinte, se aplică operații de tipul:

$$(\text{condiție})\ S_1 \rightarrow S_2\ ,$$

ce au următoarea semnificație: dacă un cuvânt are sufixul S_1 și îndeplinește condiția, sufixul S_1 se transformă în sufixul S_2. De exemplu: (m>0) EED → EE pentru agreed → agree [1].

Comparativ, algoritmul lui Porter este mai lent decât algoritmul lui Lovins, acesta din urmă având doar 2 faze prin care trece un cuvânt, însă consumul de memorie este mai mic, din moment ce algoritmul lui Lovins reține 294 de sufixe, 29 de condiții și 35 de reguli de transformare[2]. De asemenea, în algoritmul lui Lovins există anumite probleme în ceea ce privește anumite sufixe și este mai instabil în ceea ce privește cuvintele de mici dimensiuni.

1.3. Stocarea structurilor folosite într-o bază de date ne-relațională (MongoDB)

În baza de date de tip MongoDB s-au stocat structurile folosite pentru indexul direct și indexul invers, observate în Figura 1.1 și Figura 1.2.

În indexul direct, s-au stocat pentru fiecare fișier analizat, un subdocument al căror structuri cheie: valoare sunt reprezentate de cuvinte și numărul de apariții în fișierul analizat, cât și o normă corespunzătoare fișierului respectiv, folosită ulterior pentru calcularea cosinusului din *criteriul de relevanță*.

Indexul invers este o rearanjare a datelor stocate în indexul direct, de această dată, colecția având pentru fiecare document, un cuvânt și o listă de perechi de tip (fișier, număr de apariții).

1.4. Operația de căutare

Pentru efectuarea căutării unor cuvinte în fișierele preprocesate anterior s-au realizat mai multe funcții ce calculează componentele specifice.

Funcția care manageriază întreaga căutare este prezentată mai jos:

```
def input_query(self):
    query = input("Cautare: ")
    word = []
    word_to_lower = []
    word_queue = []
    operators_queue = []
    for letter in query:
        if Mapper.is_operation(letter):
            if len(word) == 0:
                continue
            word = "".join(word)
```

```
            word_to_lower = "".join(word)
            if self.is_exception(word):
                word_queue.append(word)
                operators_queue.append(letter)
            else:
                if not self.is_stop_word(word_to_lower):
                    word_queue.append(word_to_lower)
                    operators_queue.append(letter)
            word = []
            word_to_lower = []
        else:
            word.append(letter)
            word_to_lower.append(letter.lower())
    if len(word) != 0:
        word = "".join(word)
        word_to_lower = "".join(word)
        if self.is_exception(word):
            word_queue.append(word)
        else:
            if not self.is_stop_word(word_to_lower):
                word_queue.append(word_to_lower)
    previous = word_queue.pop(0)
    query_count = dict()
    query_count[previous] = 1
    for word in word_queue:
        if word in list(query_count.keys()):
            query_count[word] += 1
        else:
            query_count[word] = 1
        operator = operators_queue.pop()
        previous = self.execute_operation(previous, word, operator)
    raw_search = previous
    if isinstance(raw_search, str):
        raw_search = self.execute_operation(previous, "", " ")
    cosine = dict()
    word_count = len(word_queue)
    for document in list(raw_search.keys()):
        words = raw_search[document]
        c = self.calculate_cosine(document, words, query_count, word_count)
        cosine[c] = document
    print("Documente pentru <"+query+"> :")
    for key in sorted(list(cosine.keys()), reverse=True):
        print(cosine[key])
```

Prima parte din aceasta acceptă ca input de la tastatură o serie de cuvinte pe baza cărora se va face o căutare booleană. În acest sens, cuvintele din interogarea introdusă sunt analizate în aceeași manieră ca cele din faza de creare a indexului direct. După obținerea termenilor analizați, se realizează căutarea booleană pe baza interogării introduse. Pentru realizarea acesteia, se iau în considerare caracterele predefinite ce substituie efectuarea unei anumite operații între termeni, aceste operații fiind „și”, „sau” și ”not”. Rezultatul căutării booleene este un dicționar al căror perechi cheie: valoare sunt reprezentate de documente și o serie de dicționare ce conțin cuvintel și numărul lor de apariții. Pe baza acestora se calculează cosinusul fiecărui document, folosind de asemenea și două metode ce calculează norma unui fișier, respectiv a interogării introduse.

Rezultatele calculului cosinusurilor sunt ulterior grupate într-un dicționar ce leagă numele fișierelor de aceste valori. Cosinusurile sunt sortate descrescător, obținându-se astfel lista de fișiere ordonată după relevanță.

1.5. Îmbunătățirea performanțelor

Prima etapă a proiectului reprezintă un experiment într-o fază incipientă a ceea ce reprezintă motoarele de căutare. Mecanismul ce stă la baza acestora a fost implementat sub forma unui program cu un set de date relativ restrâns, ce nu poate fi mapat în totaliate pe un model mai mare fără îmbunătățirea performanțelor. Paralelizarea operațiilor, realizată în acest caz prin folosirea unui ThreadPoolExecutor, poate reprezenta un mod de optimizare, însă simplificarea operațiilor atomice ar fi calea optimă.

1.6. Concluziile etapei

În prima etapă a proiectului, s-a reușit realizarea unui model similar în funcționalitate cu mecanismele motoarelor de căutare din ziua de astăzi, prin utilizarea unor metode de stocare cu un mai bun randament dar și prin analizarea performanțelor căutării pe baza unui criteriu de relevanță.

Etapa 2. Colector de date web – Web Crawler

2.1. Contextul celei de-a doua etape

A doua etapă a proiectului presupune crearea unui colector de date web (web crawler) care să parcurgă și să colecteze date de pe internet, începând cu un link sursă. Acest crawler va colecta de pe paginile accesate, toate legăturile permise pe care le va accesa ulterior în aceeași manieră. În contextul creării acestui mecanism, s-au propus următoarele elemente componente:

1. Componenta DNS
2. Componenta HTML
3. Colectorul propriu zis

2.2. Componenta DNS

Pentru a putea realiza conectarea crawlerului la diferite pagini web, a fost necesară crearea unei componente cu rol de client DNS (Domain Name System), ce trimite mesaje cu privire la domeniile ce se vor a fi accesate și primește înapoi o serie de informații, printre care se regăsesc și adresele IP la care se pot găsi respectivele resurse.

Mesajele trimise și primite către și de la serverul DNS urmăresc structura din Figura 2.1:

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Identification																QR	Opcode				AA	TC	RD	RA	Z	AD	CD	Rcode			
Total Questions																Total Answer RRs															
Total Authority RRs																Total Additional RRs															
Questions [] :::																															
Answer RRs [] :::																															
Authority RRs [] :::																															
Additional RRs [] :::																															

Figura 2.1: Structura unei cereri DNS - preluat [3]

Rezultatele obținute cu ajutorul clientului DNS au fost apoi stocate într-o structură proprie cu rolul de cache, din moment ce răspunsul provenit de la serverul DNS are o durată de viață specifică. Prin folosirea acestui tip de structură se evită operațiile inutile de comunicare, obținându-se prin aceasta o optimizare a timpului de acces la adresele IP pentru paginile accesate, cu prețul unei părți adiționale de memorie locală.

2.3. Componenta HTML

Odată obținută adresa IP pentru o anumită resursă, scopul este de a accesa resursa prin trimiterea unei cereri HTTP către serverul care o deservește. În acest scop, colectorul de date trimite cu ajutorul unei structuri predefinite de tip socket, un mesaj către portul 80 în cazul paginilor nesecurizate, sau 443 în cazul paginilor securizate (încapsulând comunicația în acest caz într-un flux SSL/TSL).

2.3.1. Permisiunea accesului la resurse. Fișierul „robots.txt”

Scopul acestei componente este de obține pagina HTML aferentă resursei cerute, într-un mod cât mai sigur, fără a încălca vreo regulă impusă de deținătorii respectivelor date. Acestea fiind spuse, crawlerul web trebuie să se supună unor reguli impuse cu ajutorul unor convenții încapsulate în fișierele *„robots.txt”*. Prin aceste fișiere disponibile, crawlerului i se comunică resursele care nu îi sunt permise a fi accesate, cât și o serie de alte date.

Axându-ne astfel pe caracteristica de interzicere a accesării anumitor resurse web, crawlerul deține o componentă secundară care accesează în primul rând acest fișier, înainte de a face cereri către resursele disponibile pe un anumit domeniu. Odată confirmării permisiunii de

acces la o anumită resursă, crawlerul web poate să salveze pagina respectivă.

2.3.2. Gestionarea răspunsurilor resurselor accesate

Accesarea resurselor poate fi uneori dificilă datorită răspunsurilor oferite de gazda acestora. Astfel, componenta HTML obține în primă fază tipul de răspuns al gazdei, urmând ca în funcție de acesta, resursa să fie salvată, reaccesată sau refuzată. O resursa HTML accesată este așadar salvată dacă răspunsul primit face parte din categoria 200 (Succes) dar nu înainte de a se face o verificare a tag-ului meta, adresat specific crawlerului, care să confirme permisiunile acordate în fișierul „*robots.txt*”. Dacă, însă, răspunsul primit face parte din categoriile 400 (Eroare client) sau 500 (Eroare server), accesarea resursei nu mai are sens. De asemenea problematice sunt și răspunsurile din categoria 300 (Redirecționare) care anunță o redirecționare către o altă locație web, motiv pentru care componenta HTML va face alte cereri, dăundând astfel performanței timp.

```
def redirect(self, location, ip, protocol, useragent):
    url = urlparse(location)
    host = url.netloc
    if url.scheme.lower() == "https":
        port = 443
    else:
        port = 80
    message = RequestCreator.new_get_request("/robots.txt", protocol)
    message = RequestCreator.add_header("Host", host, message)
    message = RequestCreator.add_header("User-Agent", useragent, message)
    message = RequestCreator.end_message(message)
    response_headers = ""
    file_data = ""
    write_flag = False
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        if port == 443:
            context = ssl.create_default_context()
            s = context.wrap_socket(s, server_hostname=host)
        s.connect((ip, port))
        s.send(message.encode())
        former_first = ""
        data = bytes(0)
        while True:
            try:
                try:
                    data += s.recv(1024)
                    data = data.decode()
                except UnicodeDecodeError:
                    continue
                try:
                    firstline = data.splitlines()[0]
                except:
                    if data == "":
                        break
                    raise Exception
                if firstline == former_first:
                    break
                former_first = firstline
            except socket.timeout:
                break
            if write_flag is False:
                try:
                    idx = data.index("User-Agent:")
                    write_flag = True
```

```
                except ValueError:
                    try:
                        idx = data.index("User-agent:")
                        write_flag = True
                    except ValueError:
                        idx = None
                response_headers += data[0:idx]
                if idx is not None:
                    file_data += data[idx:]
            else:
                file_data += data
            data = bytes(0)
    return file_data, response_headers
```

2.4. Colectorul

Pentru a putea face uz de informațiile accesate de către componenta HTML, colectorul folosește o coadă de adrese web, ce reprezintă fluxul de date de intrare. Crawlerul primește așadar o serie de adrese ce reprezintă resursele accesate inițial.

2.4.1. Accesarea de noi resurse

Una din caracteristicile principale ale colectorului de date web este faptul că trebuie să funcționeze în continuu până la oprirea la cerere sau atingerea unei limite prestabilite. În acest context, obținerea de noi date cu ajutorul datelor deja existente este una prioritară.

Existența acestor date depinde, desigur de permisiunile acordate de resursele accesate și salvate. Pentru a nu încălca nicio regulă, crawlerul are obligația să verifice informațiile oferite de tag-ul meta al paginilor accesate, care îi sunt adresate direct. Odată permis accesul, crawlerul poate considera apoi legăturile din paginile indexate ca noi resurse de accesat.

2.4.2. Salvarea datelor

Paginile HTML ce rezultă în urma colectării, sunt stocate local într-o structură de fișiere vizibilă în Figura 2.2:

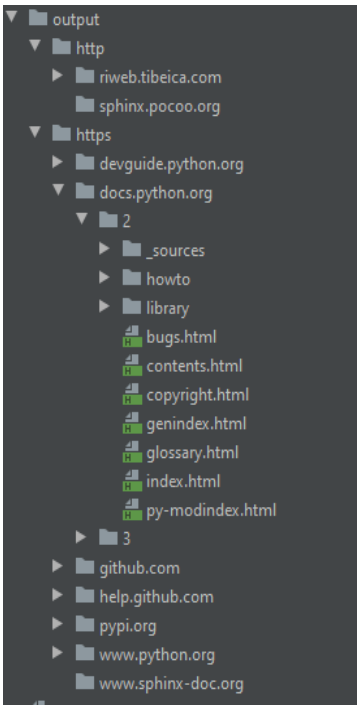

Figura 2.2: Structura de fișiere obținută

Deci în directorul „output”, se creează directoare pentru schemă, domeniu și fiecare secțiune delimitată de caracterul „/” din cadrul căii resursei, exceptând ultima, care reprezintă numele fișierului html. Singura abatere de la această regulă este atunci când după numele domeniului, nu mai există o cale specifică, motiv pentru care conținutul resursei se salvează sub numele „index.html”.

Concluzii

În acest proiect s-a reușit crearea unor componente din structura unui motor de căutare dar și un model de colector de date web, lucruri esențiale în existența internetului ca un tot unitar. Dificultatea optimizării acestora arată totuși că, deși simple de interpretat, modelele urmărite sunt greu de stăpânit în totalitatea lor.

Bibliografie

[1] The Porter stemming algorithm, accesat on-line (aprilie 2020)
[2] The Lovins stemming algorithm, accesat on-line (aprilie 2020)
[3] DNS, Domain Name Systerm, accesat on-line (mai 2020)