UNIVERZITET UNION
RAČUNARSKI FAKULTET

DIPLOMSKI RAD

Indeksiranje i pretraživanje tekstualnih podataka

Srećko Toroman

Beograd, 2011

	UNIVERZITET „UNION“ **RAČUNARSKI FAKULTET** Knez Mihailova 6/VI 11000 BEOGRAD	Broj: Datum:

UNIVERZITET UNION

RAČUNARSKI FAKULTET

BEOGRAD

RAČUNARSKE NAUKE

DIPLOMSKI RAD

Kandidat: **Srećko Toroman**

Broj indeksa: 59/06

Tema rada: **Indeksiranje i pretraživanje tekstualnih podataka**

Mentor rada: **prof. dr Branko Perišić**

Beograd, 2011

Sadržaj

1 Uvod **1**

1.1 Struktuiran tip podatka....1

1.2 Nestruktuiran tip....1

1.3 Motivacija....2

2 Invertovani indeks **4**

2.1 Indeksiranje....4

2.2 Pretraživanje....5

2.3 Osobine rečnika....8

2.4 Pretraživanje intervala....9

3 Dinamičko indeksiranje **10**

3.1 Segmentacija indeksa....12

3.2 Organizacija i svrha segmenta....13

3.3 Performanse pretrage....13

3.4 Optimizacija segmentiranog indeksa....14

3.5 Logaritamska optimizaciona strategija....15

3.6 Algoritam za spajanje segmenata....19

4 Strukture podataka **21**

4.1 Statična rasuta datoteka....21

4.2 Dinamična rasuta datoteka....22

4.3 Kodiranje razlike....24

4.4 Kodiranje celih brojeva....24

4.5 LZW kompresija....25

4.6 Svrha kompresije....25

4.7 Strukture za brže izračunavanje preseka....26

4.8 Skok pokazivači....26

4.9 Alternative....28

4.10 Kodiranje skok pokazivača....29

5 Ocenjivanje i rangiranje **30**

5.1 Ponderisanje termina....30

5.2 Ocenjivanje reči unutar dokumenta....31

6 Distribuirano indeksiranje **32**

6.1 Vertikalna podela....32

6.2 Horizontalna podela....33

7 Zaključak **34**

1 Uvod

Pronalaženje informacija (engl. *information retrieval,* skr. *IR*) je deo informatičkih nauka koji se bavi organizacijom velikih količina nestruktuiranih podataka radi brze pretrage, odnosno pronalaženja informacija iz tih podataka[1]. Koristeći saznanja iz ove nauke, a za potrebe kompanije Wowd[2], razvijen je invertovani indeks koji je iskorišćen za skladištenje i pretraživanje lokalnih privatnih podataka korisnika. U ovom radu predstavljena je i analizirana teorijska podloga koju je autor koristio prilikom izrade Java biblioteke za potrebe Wowd aplikacije.

1.1 Struktuiran tip podatka

Relacione baze su već decenijama provereno rešenje za upravljanje struktuiranim podacima. To su problemi poput vođenja računa, evidencije zaposlenih itd. Podaci ovog tipa mogu se apstraktno modelovati, tzv. entitetima, sastavljenih od primitivnih obeležja, kao što su datum (npr. vreme isplate), kratak tekst (npr. broj računa), broj (npr. svota novca) itd. Pretraživanje po ovim obeležijma obavlja se uz pomoć sortiranih ili rasutih *indeksa* (B-stablo ili hash tabela). Pomoću njih izbegava se spora linearna pretraga i moguće je u *O(1)* ili *O(logn)* izvršiti selekciju po vrednosti. Uz pomoć B-stabla mogu se optimizovati i pretraživanja po prefiksu teksta.

1.2 Nestruktuiran tip

Ukoliko je potrebno pretraživati velike kolekcije teksta po unutrašnjem delu sadržaja, sa kompleksnim upitima (skup reči, fraze itd), indeksi kod relacionih baza nisu od pomoći. Iz tog razloga se takvi tekstualni podaci smatraju *nestruktuiranima* i nepogodnim za pretraživanje pomoću sistema relacionih baza.

U daljnjem toku rada više neće biti značajnijeg osvrtanja na klasične relacione sisteme,

već će se razmatrati samostalni IR sistem. Pre toga, bitno je napomenuti da se za jednostavnije pretraživačke potrebe može izbeći uvođenje zasebnog sistema za potpunu pretragu teksta. Većina kvalitetnih relacionih baza podataka podržava neki vid potpune pretrage (npr. PostgreSQL[3]), a drugi način jeste da DBA i programeri optimizuju pretragu upotrebom posebne tabele TIndeks (*Polje, Reč, Dokument_id, Ocena).* Iako ovaj pristup funkcioniše, nije ni blizu performansi i mogućnosti sistema posvećenom pretrazi teksta.

Kao osnovna jedinica predstave podatka u IR sistemu uzima se *dokument.* Osobine dokumenta su da ga je moguće rasčlaniti na elemente po kojima se kasnije može pretraživati – *reči* odn. *termine.* Taj proces naziva se *tokenizacija.* Kao što je već rečeno, iako se ova nauka bavi organizacijom i pretragom nestruktuiranih podataka, tekstom, često jedan dokument čine delom struktuirani podaci i jedan ili više tekstualnih podataka. Na primer, web stranica se, uprošteno, može posmatrati kao dokument sastavljen od *naslova*, zatim *sadržaja,* te *datuma objave.* Naslov i sadržaj dokumenta se smatraju nestruktuiranim, dok se datum objave smatra struktuiranim. Iz tog razloga većina IR sistema podržava mogućnost definisanja okvirne strukture dokumenta upotrebom imenovanih *polja.* Primera radi, web stranica može se definisati poljima *Naslov, Sadržaj, URL.* Sistem dopušta različitu tokenizaciju (pa i odsustvo iste) kod različitih polja, međutim, sve pojave polja jednog tipa koriste isti način tokenizacije.

1.3 Motivacija

Primeri sistema koji ubrzavaju pretragu velike količine podataka se mogu sresti i izvan računarskih nauka. Na primer, pre pojave i upotrebe računara, u bibliotekama su organizovani *indeksi* članova. To su bile (i još uvek jesu, negde) korpice sa članskim kartama, poredane leksikografski, koje su omogućavale da bibliotekar pronađe korpu u kojoj se nalaze podaci vezani za određenog člana bez ispitivanja svih korpi. Samo uz pomoć početnih slova imena člana, bibliotekar je u stanju da brzim pogledom identifikuje

odgovarajuću korpu ili ladicu i u njoj pronađe odgovarajuću fasciklu, koje su takođe poredane leksikografski, po prezimenu, imenu ili broju članske karte.

U slučaju kompanije Wowd, indeks sa potpunom pretragom teksta bio je potreban na dva mesta: upravljanje istorijom pregledanja web stranica i upravljanje Facebook podacima. Pre početka izrade sopstvenog sistema testirana su postojeća rešenja, ali su mahom odbačena uglavnom zbog prevelike upotrebe radne memorije. Zato je Wowd lokalni indeks, kodnog imena EDB, dizajniran s ciljem da minimizuje upotrebu radne memorije, a ispuni (i premaši) sve potrebe aplikacije.

Potrebe (i funkcionalnosti) aplikacije su:

1. skladištenje web stranica, Facebook prijatelja, objava i komentara
2. označavanje pojedinačnih zapisa sa "nalepnicama" (engl. *tag*)
3. pretraživanje po tekstu, frazama, datumu, opsegu datuma, oznakama, autorima, pomenutim osobama...
4. niska upotreba memorije, indeksiranje u realnom vremenu, vreme odziva koje ne iritira korisnika, optimizacija indeksa u pozadini bez ometanja ostatka aplikacije

U skladu sa zahtevima dizajniran je sistem koji podržava tražene osobine, a detalji implementacije su opisani u ostatku rada.

2 Invertovani indeks

Struktura koja omogućava brzu pretragu po rečima u velikim kolekcijama teksta naziva se invertovani (engl. inverted) indeks. U poređenju sa starijim bibliotečkim sistemima, to bi bilo ekvivalentno vođenju posebne fascikle za svaku moguću reč koja se pojavljuje negde u kolekciji. Reči bi se čuvale leksikografski uređene po korpama. To bi omogućilo bibliotekaru da pronađe fascikle sa spiskom knjiga koje sadrže reči *Pariz, devetnaesti, vek*. Zatim bi bilo potrebno ispitati te fascikle i pronaći spisak knjiga koje odgovaraju tim rečima – odn. upitu. Taj proces bi trajao različito dugo, u zavisnosti od veličine fascikli, međutim i ovaj proces se može ubrzati ukoliko je spisak knjiga u fasciklama uređen leksikografski ili po nekom drugom principu (npr. serijski broj knjige). U suštini to je ono što se i dešava na računaru.

2.1 Indeksiranje

U matematičkom smislu, invertovani indeks je funkcija koja mapira termin, reč, (engl. *term*, *keyword*) na *skup* dokumenata, obično predstavljenu kao rastući ili opadajući niz celobrojnih vrednosti koje predstavljaju serijske brojeve dokumenata. Takva funkcija se na računaru realizuje pomoću više datoteka.

Algoritam indeksiranja i nastanka eksternih struktura invertovanog indeksa u najkraćem obliku može se opisati sledećim koracima:

1. inicijalizacija

    1. rečnik – indeks sekvencijalna ili rasuta datoteka, skladišti meta podatke za sve termine, koristi se prilikom inicijalizacije pretrage

    2. invertovana datoteka – sekvencijalna datoteka, sadrži *liste dokument idenfitikatora* za termine iz rečnika, koristi se prilikom izvršavanja pretrage

3. skladište dokumenata – indeks-sekvencijalna datoteka koja se koristi prilikom učitavanja rezultata pretrage

2. obrada ulaznih podataka

    1. učitava se sledeći *dokument*

    2. dodeljuje se serijski broj dokumentu (1, 2, 3...)

    3. zapisuje se u skladište pod dodeljenim serijskim brojem

    4. vrši se *tokenizacija* teksta – za svaki termin vodi se lista dokument identifikatora (serijskih brojeva) u kojima je termin evidentiran ("foo": [1, 3], "bar": [2, 3])

    5. ponavlja se korak 2.1 dok se svi dokumenti ne obrade

3. zapisivanje invertovanog indeksa – za sve parove *termin, lista:*

    1. u invertovanu datoteku se zapisuje termin i njegova lista dokumenata ([foo, 1, 3], [bar, 2, 3])

    2. meta podaci o terminu skladište se u *rečniku* – njih čine sam termin, pomeraj u primarnoj datoteci, dokument frekvencija, tzv. term frekvencija i drugi

Postoje različiti načini da se proces implementira, ali to je okvirno šta se dešava. U daljem radu objašnjeni su najzahtevniji delovi procesa.

2.2 Pretraživanje

Zahtev IR sistemu za pretraživanjem indeksa definisan je *upitom.* Postoji više tipova upita, a u ovom radu je detaljno razmotren tzv. Bulov upit (engl. *Boolean query)*, nazvan

po istoimenoj Bulovoj logici na čijim principima se zasniva. Bulov upit definisan je logičkim izrazom koji se sastoji od *operatora* i *operanada*. Operand se posmatra kao primitivan upit (nedeljiv, tipa jedna reč) ili podupit (tipa drugi Bulov upit, između zagrada). Osnovni operatori su konjunkcija, disjunkcija i negacija.

Bulov upit moguće je izvršiti na par načina, konkretno sa ili bez normalizacije izraza. U pratećoj aplikaciji implementirana su oba pristupa, a ovde je objašnjen postupak bez normalizacije.

Bez upotrebe normalizacije, ugnježđeni upiti se prvi izvršavaju te se njihovi rezultati posmatraju kao rezultati primitivne pretrage i onda se spajaju sa ostatkom upita. Ova funkcionalnost trivijalno se može implementirati rekurzivnim algoritmom. Rezultati primitivnog upita je ništa više do lista dokument identifikatora, opcionalno sa pratećim meta podacima, npr. na koliko mesta u dokumentu se upit pronalazi. Ovi meta podaci koriste se za rangiranje i pretraživanje fraza o čemu je data reč u sledećim poglavljima. U ovom kontekstu može se smatrati da je za svaki *operand* upita dobavljena po jedna *lista* identifikatora, koja se u slučaju primitivnih operanada jednostavno učitava iz invertovane datoteke.

Nakon što se operandi obrade, izvršavaju se operatori između njih i nastaje nova lista dokumenata koja predstavlja *rezultat* upita. Prvo se svi operandi grupišu u tri grupe: **obavezni, poželjni i nedozvoljeni.** Ove grupe su optimalniji način predstavljanja upita od klasičnog Bulovog izraza. Na primer, umesto:

$$\text{foo} \wedge \text{bar} \wedge \neg \text{foobar} \wedge (\text{java} \vee \text{python})$$

Lakše je:

$$+\text{foo} +\text{bar} +(\text{java python}) -\text{foobar}$$

U drugom slučaju, svi operandi koji počinju sa znakom “plus” članovi su obavezne

grupe i njihove liste dokumenata se *presecaju.* Izračunavanjem preseka pojedinačnih listi pronalazi se lista dokumenata zajednička za sva tri operanda. Na primer, ako su dva operanda u grupi sa listama 1, 3, 5, 7 i 3, 4, 7, 9 onda je presek lista 3, 7.

Operandi koji ne počinju znakom minus ili plus su tipa "poželjni ali neobavezni", tako da se između njihovih listi pravi *unija.* Na primer, unija prethodnih listi je 1, 3, 4, 5, 7, 9. Glavna svrha poželjnih operanada jeste da se da nagoveštaj pretraživaču prilikom rangiranja, ali i kao obična disjunkcija (u smislu java ili python).

I poslednje, operandi koji počinju sa znakom minus su nedozvoljeni i između njih se takođe izvršava unija, čime se dobija lista nedozvoljenih (čvrsto nepoželjnih) dokumenata.

Ove tri grupe (od kojih neke mogu biti i prazne) se spajaju u konačan rezultat upita. Tu postoje posebni slučajevi. Kada su sve tri grupe neprazne, rezultat je:

$$R = O \cup (O \cap P) \setminus N$$

Formula izgleda suvišno zakomplikovana jer se čini da je $R = O \setminus N$. Ipak, prilikom izračunavanja preseka i unije ne proizvodi se samo lista dokument identifikatora, već se uz svaku *dokumentId* vrednost čuva i *ocena.* Ocena se pozitivno menja ukoliko se u dokumentu nalaze poželjne reči, na način objašnjen u poglavlju 5, Ocenjivanje i rangiranje.

Ukoliko ne postoje obavezni operandi, onda:

$$R = P \setminus N$$

Slučaj u kom je neprazan jedino skup nedozvoljenih operanada nije praktičan i obično se ne implementira. Ukoliko je ipak potreban, onda se omogućava dodavanjem posebnog termina u svaki dokument te izračunavanjem negacije u odnosu na listu svih dokumenata koja se dobija upitom u taj poseban termin. Time se dobija "skup svih dokumenata".

Optimalna implementacija ovog algoritma podrazumeva pametno izračunavanje

preseka, iterativno, te kvalitetno izračunavanje ocene u toku izvršavanja, o čemu će kasnije biti više reči.

Algoritam traženja unije ubrzava se upotrebom strukture *heap* (prev. gomila). Ukoliko se traži unija od m listi sa ukupno n elemenata, konstruiše se heap kapaciteta m. Pronalaženje sledećeg člana unije je prvi član heap-a, te je složenost te operacije konstantna. Međutim, pošto je ažuriranje heap-a $\log m$ operacija, onda je konačna složenost $O(n \log m)$.

Nakon što upit izračuna konačnu listu, potrebno je iz skladišta učitati odgovarajuće dokumente, te prezentovati ih korisniku, na način koji odgovara korisniku IR sistema.

2.3 Osobine rečnika

Rečnik (engl. *dictionary)* invertovanog indeksa često predstavlja usko grlo indeksa. On mora omogućiti brzo čitanje reči, a za to je poželjno da veći deo istog bude smešten u radnoj memoriji, što znači da rečnik mora biti dobro kompresovan. Razlikuju se rečnici koji su organizovani kao rasuta datoteka i rečnici sa indeks–sekvencijalnom datotekom, leksikografski uređeni. Dalje mogu se razlikovati po promenljivosti sadržaja – statični i dinamični.

Prvi je moguće implementirati pomoću statične heš mape, sa upotrebom otvorenog heširanja i dobrom heš funkcijom. Ulančavanje se ne preporučuje jer zahteva veću količinu radne memorije. Ukoliko je potrebno omogućiti centralni rečnik (zajednički za sve segmente, o čemu će kasnije biti reč) koji ne služi samo za čitanje već i upis novih vrednosti, onda se koristi dinamično heširanje, koje je korišćeno i u pratećoj aplikaciji.

Drugi tip rečnika, uređen tj. sortiran, implementira se kao stablo pretraživanja. Može biti binarno stablo ili *trie* ukoliko se nalazi u radnoj memoriji, odnosno **B-stablo** ukoliko je na spoljašnjem medijumu. Stabla su lošija u odnosu na heš mape u pogledu broja

neophodnih pristupa da bi se pronašla tražena vrednost. Složenost upisa i čitanja kod stabala su logaritamska, dok su kod heš mape konstantna. Ipak sortirani rečnici imaju velikih prednosti u pogledu kompresije jer je moguće kodirati razliku između susednih reči, umesto kodiranja cele reči. Ovaj metod je obavezno korišćen kod rečnika koji staju u radnu memoriju.

Rečnik pored same reči skladišti i određene metapodatke. Ti podaci potrebni za rangiranje detaljno su opisani u poglavlju 5, Ocenjivanje i rangiranje. Osim njih, rečnik može da skladišti i celobrojni identifikator reči. U tom slučaju svakoj reči pridružena je celobrojna vrednost označena kao *rečId*. Upotrebom ove vrednosti omogućava se bolja kompresija i brže spajanje segmenata, opisano u poglavlju 3, Dinamičko indeksiranje.

2.4 Pretraživanje intervala

Druga prednost sortiranih rečnika, a koju heš mape teško mogu da nadomeste, jeste pretraživanje opsega ili intervala (engl. *range search*). Primer upita koji može da se izvrši upotrebom opsega je:

1. "dati sve dokumente napisane između 1900. i 1950. godine" ili
2. "ime autora počinje sa Aleks"

Prvi deo upita može se implementirati i pomoću heš mape kao:

1900 ili 1901 ili 1902... ili 1950

Drugi upit je još više nepraktičan za izvršavanje na rasutim rečnicima, iako nije nemoguć.

Šta se dešava ukoliko indeks treba da izvrši upit "svi dokumenti iz perioda od 100. god. do 1900. god."? Očigledno, ekspanzija upita u Bulov upit sa 1800 operanada neće biti brza.

Rešenje kod rasutih datoteka jeste smanjiti količinu termina koje treba nabrojiti. To se postiže uvođenjem višestrukog indeksiranja datuma, po različitim rezolucijama. U tom slučaju, postojala bi određena redundansa, veća indeksna struktura (svaki dokument bi bio evidentiran pod svojom godinom, vekom i sl. po potrebi) ali bi gore navedeni upit izgledao kao "vek2 ili vek3 ili ... ili vek19 ili godina1900 ili godina1901... ili godina1950". Ovo rešenje je u upotrebi u pratećoj aplikaciji kod pretrage datuma, s tim što je granularizacija odrađena od nivoa dana, do nivoa godine.

Kod indeksa sa leksikografskim rečnikom pretraga je još elegantnija. Pošto su *dokId* liste kod ovih indeksa sortirane na isti način kao i rečnik, leksikografski, to znači da su nabrajanja za dokumente iz 1922. i 1923. susedna na disku. Dakle opseg iz upita odgovara fizičkom opsegu podataka na disku, te je dovoljno iz rečnika dobaviti samo jednu reč iz rečnika – onu prvu. Liste ostalih reči iz opsega nalaze se nakon liste prve reči te za ostatak termina nije potrebno vršiti upit u rečnik. Potreban je algoritam koji će upravljati čitanjem višestrukih susednih lista posmatrajući ih kao jednu, čime se dobija velika prednost u odnosu na rasute rečnike.

Uopšteno, da bi se iskoristila prednost sortiranih rečnika, neophodno je da leksikografski poredak odgovara logičkom poretku. Kod pretvaranja datuma u tekst, *godina-mesec-dan* je ispravan format, a *mesec-dan-godina* je neispravan.

3 Dinamičko indeksiranje

Dizajniranje i implementacija bilo kojeg informacionog sistema, algoritma ili strukture uveliko zavisi od načina upotrebe istog. Tako na primer, kod dizajniranja indeksa, pre donošenja odluka, korisno je poznavati osobine korpusa i korišćenja indeksa. Među svim parametrima koji određuju ovaj proces, ovde je razmotrena dinamičnost indeksa. Kod prateće aplikacije, bilo je neophodno omogućiti svakodnevno i instantno modifikovanje indeksa, dodavanje novih dokumenata, brisanje starih te konstantno paralelno izvršavanje

upita u bilo kom trenutku. Sama količina dokumenata nije velika, ali aplikacija je ograničena na malu upotrebu radne memorije, procesora i ulazno izlaznih operacija.

Brisanje dokumenata implementira se pomoću bitmaske koja staje u radnu memoriju ali ujedno se nalazi i na trajnoj spoljašnjoj memoriji. Ukoliko je potrebno obrisati dokumente starije od 3 meseca, prvo se izvrši upit koji dobavi dokumente iz tog vremenskog opsega, potom se u pomenutoj bitmaski označe bitovi na pozicijama *dokument identifikatora* vraćenih rezultata upita. Na kraju operacije te promene se sinhronizuju sa stanjem na disku. Međutim, tokom izvršavanja upita, ti indentifikatori su i dalje postojani u listama, iako su dokumenti obrisani. Zato je pre dodavanja pogotka u listu rezultata neophodno proveriti da li se dokId nalazi u bitmaski obrisanih dokumenata.

Promena dokumenta je moguća bez reindeksiranja jedino ukoliko se menjaju polja koja nisu indeksirana. Na primer, ukoliko je dokument definisan poljima *naslov, autor, prilog* i poslednje polje prilog predstavlja binarni objekat koji nije indeksiran i pogodan za pretragu, onda je moguće promeniti vrednost tog polja bez promena indeksne strukture.

Ukoliko je pak potrebno promeniti polje koje je predviđeno za pretragu, onda je neophodno reindeksiranje. Pošto su liste za pretragu gusto spakovane i kodirane u indeks datoteci, nije moguće modifikovati ih da predstavljaju novo stanje dokumenta. Zato je standardno rešenje u ovom slučaju reindeksiranje dokumenta. To podrazumeva označavanje aktuelne verzije dokumenta kao obrisane, zatim konstruisanje dokumenta sa osveženim vrednostima te ponovno indeksiranje, kada on dobija novi identifikator.

Ukoliko su ideksne strukture nepromenljive, postavlja se pitanje kako omogućiti dodavanje novih dokumenata u indeks. Jedno od rešenja jeste da se novi dokumenti čuvaju u nekom međuspremniku dok se on ne napuni do određene veličine. Potom se stari dokumenti i novi dokumenti indeksiraju u novi indeks i onda se taj indeks koristi umesto starog. Ako je dat indeks u kom se nalazi 100 dokumenata i međuspremnik koji prihvata

do 10 dokumenata, onda pri pristizanju 20 novih dokumenata u indeks, dešava se sledeće:

0. inicijalno stanje – indeks se sastoji od 100 dokumenata

1. dodaje se 10 dokumenata i indeks se ponovo pravi – reindeksira se svih 110

2. dodaje se još 10 dokumenata – reindeksira se svih 120

Dakle ukupno je reindeksirano 230 dokumenata, iako je indeksirano samo 20 novih. Količina reindeksiranja može se izračunati preko formule. Ako i predstavlja početnu veličinu indeksa, b je kapacitet međuspremnika, a n je broj novih dokumenata:

$$k = \lfloor \frac{n}{b} \rfloor$$
$$R(i, b, k) = (i + b) + (i + 2b) + \ldots + (i + kb)$$
$$R(i, b, k) = \frac{bk^2 + bk + 2ik}{2}$$

Ovaj pristup se koristi ukoliko nije potrebno indeksiranje u realnom vremenu, novi dokumenti retko kada dolaze ili je brzina pretrage bitnija od brzine indeksiranja i svežine dokumenata. U pratećoj aplikaciji iskorišćena je metoda logaritamskog indeksiranja koja pravi kompromis između brzine pretrage i složenosti reindeksiranja. Pre objašnjavanja ovog algoritma potrebno je definisati pojam *segmenta* ili *barela* u kontekstu indeksa.

3.1 Segmentacija indeksa

U ovom radu do sada je podrazumevano da se indeks sastoji od rečnika, datoteke sa dokumentima i datoteke sa indeksom. Sad se razmatra struktura segmenta, po kojem neke ili sve od pomenutih delova indeksa mogu da postoje u više instanci.

Segment se može posmatrati kao potpuno ili parcijalno samostalan i nezavistan deo indeksa. To je indeksna struktura definisana na podskupom dokumenata iz celog korpusa kojim indeks operiše. Do sada razmatrani indeks može se smatrati kao indeks sa samo

jednim segmentom.

3.2 Organizacija i svrha segmenta

Svi segmenti indeksa mogu da koriste neke zajedničke strukture, a i ne moraju, u zavisnosti od dizajna, sve ima svoje mane i prednosti. Prateća aplikacija nudi dve moguće organizacije. Kod *globalne* organizacije, svi segmenti dele zajednički rečnik i zajedničku datoteku dokumenata. Kod *lokalne* organizacije, deljen je samo rečnik. To znači da svaki segment poseduje svoju indeks sekvencijalnu datoteku za skladištenje dokumenata, te bitmasku obrisanih identifikatora. Svrha segmentacije je da se omogući dodavanje novih dokumenata u indeks bez potrebe za reindeksiranjem celog korpusa. Na primer, ukoliko se indeks sastoji iz dva segmenta, *seg1, seg2,* te međuspremnikom od 10 dokumenata, kada se on napuni, doći će do stvaranja trećeg segmenta, te će novo stanje indeksa biti sačinjeno od segmenata *seg1, seg2 i seg3.* Očigledno, složenost ovakvog dodavanja dokumenata je linearna u odnosu na broj dokumenata koje treba dodati. Veličina međuspremnika određuje broj novih segmenata i uopšteno, manji broj segmenata je poželjan zbog performansi pretrage, kao što je opisano u sledećem pasusu.

Centralni rečnik mapira reči u celobrojne identifikatore, a indeksna struktura segmenta mapira identifikator na poziciju na kojoj se u primarnoj datoteci segmenta nalazi dokument lista za tu reč. Liste dokumenata u segmentima sortirane su rastuće prema identifikatoru.

3.3 Performanse pretrage

Upit Q se izvršava tako što se nad svakim segmentom izvrši nezavisno od ostalih. Za svaku reč iz upita, potrebno je iz indeksne datoteke segmenta dobaviti listu dokumenata. To znači da je za indeks od S segmenata potrebno $Q * S$ pristupa disku za izračunavanje rezultata. Kao primer, ako je dat indeks sa međuspremnikom od 10 dokumenata i

dinamičnim korpusom (u smislu da dokumenti dolaze vremenom, tj. nisu svi odmah dostupni) od 120 dokumenata, ukupno će biti napravljeno 12 segmenata. Ako se nad tim indeksom izvrši upit $Q(q_1, q_2, q_3)$, biće izvršeno učitavanje 12 * 3 = **36** listi, što obično podrazumeva dva pristupa disku (jedan pristup mapi koja određuje poziciju liste i drugi pristup samom sadržaju liste). Pošto je vreme za pristupanje memoriji na slučajnoj lokaciji kod magnetnih diskova sporo i traje u proseku i po 10 milisekundi, samo za dobavljanje podataka biće potrebno 360 milisekundi, što je već značajno vreme u pogledu brzine indeksa sa korisničke strane.

3.4 Optimizacija segmentiranog indeksa

Pošto bi sa indeksiranjem desetina ili stotina hiljada dokumenata, čak i sa međuspremnikom od stotinu dokumenata, nastalo mnogo segmenata, očigledno je potrebno u nekom trenutku više segmenata spojiti u jedan. Ovaj proces naziva se *optimizacija* indeksa i neophodan je ukoliko se koristi segmentiran dizajn.

Najjednostavniji metod optimizacije jeste "optimizuj na svakih MaxSeg segmenata". To znači da se nakon stvaranja MaxSeg segmenata svi oni spoje u jedan – izvrši se potpuno reindeksiranje svih dokumenata. Ovo je kompromis između nesegmentiranog i trivijalno segmentiranog indeksa. Broj MaxSeg ne sme biti veći od 30 jer to znači sporo izvršavanje upita, a ne sme biti ni previše mali inače će reindeksiranje biti prečesto pozivano.

Analiza složenosti:

m – maksimalan broj segmenata

b – kapacitet međuspremnika

n – broj dokumenata u korpusu

Nakon prvih b dokumenata, nastaje prvi segment. Nakon još b dokumenata nastaje drugi itd. Nakon mb dokumenata, nastaće m segmenata i biće neophodno reindeksiranje

svih dokumenata i prethodnih m segmenata spojiće se u jedan. Zatim na svakih budućih $b(m–1)$ dokumenata sledi potpuno reindeksiranje korpusa. Formula za ukupan broj reindeksiranih dokumenata u opštem slučaju je:

$$R_{seg}(m, b, n) = R(b, b(m-1), k) + kb(m-1) + ostatak$$
$$n - b = kb(m-1) + ostatak$$

Primera radi, $R_{seg}(10, 10, 10000) = 570550$. Zbog toga se ova strategija može koristiti samo u slučaju kada se na međuspremnik može odvojiti veća količina memorije. Bez obzira da li se radi o radnoj ili spoljašnjoj memoriji, ova metoda nije bila prikladna za potrebe Wowd-a.

Iz ugla pretraživanja, performanse su solidne jer se broj segmenata kreće u rasponu od 1 do m, tako da je očekivan broj pristupa disku reda:

$$s = |Q| \frac{m}{2}$$

U narednom delu opisana je logaritamska optimizaciona strategija koja pravi idealan kompromis između količine reindeksiranja i broja segmenata, odnosno performansi pretraživanja indeksa.

3.5 Logaritamska optimizaciona strategija

Prethodno opisana strategija daje solidne performanse iz ugla pretraživanja. Šta se može učiniti da se broj segmenata zadrži na malom nivou, a količina reindeksiranja smanji? Rešenje je pronađeno u logaritamskoj optimizaciji. Za izvršavanje ove strategije, potrebno je voditi dodatnu informaciju, *stepen*, svakog segmenta. Osim toga, strategija je definisana i bazom logaritma u. Svakom novom segmentu koji je tek nastao pražnjenjem međuspremnika pridružuje se stepen 0 (nula). Optimizacija, tj. spajanje segmenata nastaje kada u skupu S postoji u segmenata istog stepena. Ti segmenti se spajaju u jedan segment

sledećeg stepena (s + 1). U pratećoj aplikaciji, korišćena je logaritamska optimizacija baze 2, bez međuspremnika. Drugim rečima, dokument se indeksira odmah po pristizanju. Time je omogućena maksimalna štednja radne memorije, a dokument se već nakon par desetina milisekundi nalazi spreman za pretraživanje. Naravno, implementirana je i mogućnost postojanja međuspremnika, te i veće baze. Ipak, sama baza ne može mnogo da varira. Već za bazu veličine 5 postoje problemi kod brzine izvršavanja upita. Po iskustvu autora, najbolje u je iz opsega od 2 do 4, u zavisnosti od hardverske podloge mašine. Računari sa sporijim slučajnim pristupom spoljašnjoj memoriji bolje rade sa manjim u, i obratno.

U sledećoj tabeli je dat primer funkcionisanja logaritamske strategije baze 2 za prvih 10 indeksiranih dokumenata. Kolona "ID" označava redni broj dokumenta koji se dodaje, a kolona "Segmenti" označava stanje indeksa – u kojoj svaki broj predstavlja jedan segment tog stepena.

ID	Segmenti	Objašnjenje
1	0	nastaje novi segment nultog stepena
2	1	novi segment se stapa sa prethodnim segmentom nultog stepena i nastaje segment prvog stepena
3	1 0	nastaje novi segment nultog stepena
4	2	algoritam utvrđuje da spajanjem dva nulta stepena nastaje segment prvog stepena i pošto takav već postoji, svi segmenti se spajaju u segment drugog stepena
5	2 0	nastaje novi segment nultog stepena
6	2 1	novi segment se stapa sa prethodnim segmentom nultog stepena i nastaje segment prvog stepena
7	2 1 0	nastaje novi segment nultog stepena
8	3	algoritam utvrđuje da je potrebno spajanje svih segmenata i nastaje segment trećeg stepena
9	3 0	nastaje novi segment nultog stepena
10	3 1	novi segment se stapa sa prethodnim segmentom nultog stepena i nastaje segment prvog stepena

Dakle, kada je baza *u=2,* u idealnom stanju indeksa, ne postoje dva segmenta istog stepena. Međutim, pošto je poželjno da se spajanje segmenata izvršava u paralelnoj niti, moguće su situacije da se, zbog prestanka rada aplikacije, segmenti ne spoje te da po sledećem pokretanju aplikacije postoje dva (ili čak i više ukoliko se ova situacija ponovi uzastopno više puta) segmenta istog stepena. Takođe, u toku spajanja segmenata, novi segmenti koji nastaju dodavanjem novih dokumenata u međuvremenu, biće optimizovani tek nakon završetka trenutne optimizacije. To ne predstavlja problem ukoliko je algoritam za spajanje napisan tako da može da prihvati proizvoljan broj segmenata, bez obzira na njihov stepen generacije. Bitno je još napomenuti da je redosled kod segmenata bitan. Najstariji segment se smatra prvim, a najnoviji segment poslednjim. Svi segmenti između takođe su uređeni po vremenu ili identifikatoru svog nastanka. Prilikom spajanja neophodno je da svi oni predstavljaju kontinualni podniz svih segmenata, inače se gubi na redosledu, što može biti nepoželjno jer unosi određene komplikacije.

Idealni slučaj je zgodan za posmatranje radi analize ponašanja ove strategije. Kao što se vidi u prethodnoj tabeli, do spajanja ne dolazi kod svakog neparnog dokumenta. Kod svakog parnog dokumenta dolazi do spajanja dva nulta segmenta, a možda i više, što se može lako izračunati ukoliko se primeti relacija između rednog broja dokumenta i stepena spajanja. Naime, postoji veza između rednog broja dokumenta, *rbr*, i njegove deljivosti sa brojevima stepena dvojke. Ukoliko je t najveći broj takav da je *rbr* deljivo sa 2^t, onda će indeksiranjem tog dokumenta nastati segment stepena t. Ovu formulu lako je proveriti sa gornjim primerom. Kao prvo, kod svih dokumenata neparnog indeksa, $t = 0$ jer je njihov najveći delilac 1 (odnosno 2 na nulti stepen) te njihovim umetanjem nastaje segment stepena nula. Kod dokumenata čiji je indeks deljiv sa četiri, odnosno dva na drugi stepen, ali ne sa osam, nastaje segment drugog stepena (u tabeli se to zaista dešava samo kod 4. dokumenta) itd.

Da bi se ustanovile performanse ove strategije, potrebno je izračunati očekivani broj

segmenata te količina (re)indeksiranih dokumenata. Neka je baza $u = 2$ i broj dokumenata n. Nakon indeksiranja svih n dokumenata, broj segmenata jednak je kardinalnosti binarne reprezentacije broja n. To je formula koja izračunava egzaktan broj segmenata. Za proizvoljnu bazu, tačan broj segmenata može se izračunati na sličan način. Najgori slučaj (stanje kada indeks ima najviše segmenata, iz svake generacije po u - 1 instanci) može se aproksimirati formulom:

$$s=(u-1)\log_u(n+1)=(u-1)\frac{\log(n+1)}{\log(u)}$$

Potpuno reindeksiranje svih dokumenata nastaje kada se n može predstaviti kao prirodni stepen broja u. Dakle za $u = 3$ postojaće potpuna optimizacija kod dodavanja dokumenata sa rednim brojem 9, 27, 81... itd. To je idealan slučaj sa stanovišta pretraživanja jer se pretražuje samo jedan segment. Sušta suprotnost je slučaj kada je n oblika u^t-1 kao u slučajevima 8, 26, 80... tada je broj segmenata maksimalan i odgovara gornjoj formuli. Na primer, za n=80 i u=3, broj segmenata je:

$$s=(u-1)\frac{\log(n+1)}{\log(u)}=(3-1)\frac{\log(81)}{\log(3)}=8$$

Ovo se može ilustrovati i ukoliko se broj 80 predstavi u ternarnom obliku:

$$(80)_{10}=2\cdot 27+2\cdot 9+2\cdot 3+2=(2222)_3$$

Analogno kardinalnosti binarnog broja, tačan broj segmenata za proizvoljnu bazu u jednak je sumi cifara predstave broja n u *u-arnom* obliku.

Količina reindeksiranja se takođe može predstaviti na jednostavan način. Kod binarnog modela, svaki segment stepena t u idealnom slučaju (bez brisanja dokumenata) sadrži 2^t dokumenata i on je nastao spajanjem segmenata 2^{t-1}, 2^{t-2}, ... , 2^0, $2^0 = 2^t$ tako da je ukupan broj reindeksiranja moguće predstaviti rekurentnom jednačinom:

$$R_{(u=2)}(1)=1$$
$$R_{(u=2)}(n)=R(n-1)+2^{t(n)}$$
$$t(n)=max\{i, 2^i|n\}$$

U sledećoj tabeli date su vrednosti R za različit broj dokumenata:

n=1	R=1
n=2	R=3
n=3	R=4
n=4	R=8
n=10	R=23

n=20	R=56
n=100	R=376
n=200	R=852
n=1000	R=5060
n=10000	R=71728

U poređenju sa trivijalnom optimizacijom i slučajem od 10000 dokumenata, količina reindeksiranja je približno šest puta manja, a očekivani broj segmenata je eksperimentalno utvrđen i iznosi 6, što je samo za jedan više od trivijalne optimizacije. Razlika je i u tipu optimizacije, naime, kod trivijalne strategije, optimizacija uvek podrazumeva spajanje svih segmenata, ali se ređe dešava (tek svakih $b * m$ dokumenata). Kod logaritamske strategije sa bazom dva, optimizacija se izvršava kod svakog drugog dokumenta, ali retko kad se dešava potpuno reindeksiranje. Konkretno, celo reindeksiranje se dešava samo log(n) puta. Prednost logaritamske strategije je dakle u tome što se optimizacija stalno izvršava i to *uglavnom po malo,* što mnogo bolje raspoređuje opterećenje računara nego trivijalna strategija.

3.6 Algoritam za spajanje segmenata

Spajanje segmenata podrazumeva pravljenje novog segmenta, bez tragova o obrisanim dokumentima. Osim što se time indeks čisti i smanjuje veličinu, smanjuje se i broj segmenata te poboljšavaju se performanse pretraživanja. U zavisnosti od organizacije segmenta, razlikuje se proces optimizacije, ali uglavnom se radi o spajanju nekoliko struktura u jednu celinu.

Spajanje indeksne datoteke može se izvesti na dva načina. Trivijalano rešenje jeste da se dokumenti iz oba segmenta ponovo indeksiraju u novu datoteku. Za ovu implementaciju nije potrebno pisati mnogo dodatnog koda. Mnogo brži i manje zahtevan način jeste da se napiše sekvencijalni čitač indeksne datoteke i algoritam za spajanje tih čitača, te pisač koji će zapisati spojeni segment. Preduslov za rad ovog algoritma jeste da su liste dokumenata, od kojih se indeksna datoteka sastoji, sortirane po terminu. Mogu biti uređene leksikografski (u zavisnosti od reči kojoj pripadaju) ili kao rastući prirodni brojevi (ukoliko se koristi centralno reč-rečId mapiranje).

Prilikom spajanja n segmenata, potrebno je pozicionirati sve čitače na istu reč, zatim spojiti njihove liste dokumenata. I u ovom delu korišćena je struktura za brzo pronalaženje najmanjeg člana, *heap,* tako da je uvek dostupna lista sa najmanjim *rečId*-om, a ukoliko više segmenata sadrži liste za tu reč, onda se one porede po starosti segmenta – u zavisnosti od dizajna indeksa, to može značiti da segmenti novije ili starije generacije imaju prednost. Kako svih k listi prolazi kroz *heap* kapaciteta n, složenost spajanja segmenata je reda $k*logn$.

4 Strukture podataka

Invertovani indeks, kao i baze podataka, koristi mnoge algoritme i strukture podataka, od kojih neki operišu sa spoljašnjom, a neki sa unutrašnjom memorijom. Po mišljenju autora ovo je bio jedan od najvažnijih i najtežih zadataka prilikom razvijanja prateće aplikacije, te je posvećeno posebno poglavlje u kome su opisani.

4.1 Statična rasuta datoteka

Pogodna je za mape sa statičnim sadržajem i mogućnošću brisanja elemenata. Zbog sporosti spoljašnjih medijuma, kolizije su izuzetno opasne po performanse, te je bitno izabrati dobru heš funkciju i odgovarajuću veličinu tabele. Ukoliko su zapisi koji se upisuju u mapu konstantne veličine (na primer, ključ k=4 bajta i vrednost v=20 bajtova), onda je za n ključeva potrebno rezervisati $rn(k+v)$ bajtova na disku. Faktor redundantnosti se utvrđuje eksperimentalno, a u pratećoj aplikaciji je korišćeno $r = 2$. Potrebno je voditi računa o slobodnim blokovima u datoteci, što se može učiniti dodavanjem kontrolnog bajta, uvođenjem posebnog ključa koji predstavlja praznu ćeliju (ukoliko je to moguće) ili uvođenjem bitovne maske. Kolizija se može rešiti zonom prekoračenja (zona nakon 2n(k+v) bajtova) u kojoj se čuvaju sva prekoračenja, ili otvorenim adresiranjem. Pošto su diskovi prilagođeni linearnom čitanju (spori skokovi, engl. *seek*) onda u obzir dolazi samo *linearno* otvoreno adresiranje. To znači da ukoliko se K mapira u ćeliju j koja je zauzeta, onda se isprobava ćelija $j+1 \pmod m$ itd.

Korisna tehnika kod dizajniranja i statičnih i dinamičnih struktuiranih datoteka jeste blokiranje zapisa, u smislu da se datoteka izdeli u blokove od po 4KiB, 32KiB, ili sl. U tom slučaju heš funkcija mapira ključ na indeks bloka u kom se vrednost za taj ključ nalazi (pravilniji izraz za blok u ovom slučaju je *korpa* od engl. *bucket)*. Ukoliko se ključ-vrednost par ne nalazi u bloku, znači da ne postoji vrednost za dati ključ. Ukoliko se otvoreno

adresiranje koristi sa korpama, onda je potrebno označiti korpe koje su prepunjene, da bi se u prethodnom opisanom slučaju nastavilo sa pretraživanjem sledeće korpe.

Blokiranje se može koristiti i u sprezi sa zonom prekoračenja, a neophodno je u slučaju kada su ključevi i (ili) vrednosti varijabilne dužine, doduše ne previše varijabilne (u tom slučaju podaci nisu pogodni za čuvanje u ovakvoj strukturi). U tom slučaju jedan blok prima ključeve i vrednosti dok u njemu ima mesta, a dužine ključeva i vrednosti se zapisuju unutar bloka jer su neophodne prilikom čitanja.

4.2 Dinamična rasuta datoteka

Odlikuje se mogućnošću progresivne ekspanzije. Nije neophodno rezervisati ogroman prostor na memorijskom uređaju da bi se izbegle kolizije, tj. nije potrebno unapred znati broj elemenata koji će biti smešten u mapi – dinamična rasuta datoteka sama usklađuje svoju veličinu po potrebi. Kao i statična, i ova struktura zadržava konstantnu složenost upisivanja i čitanja. Od više načina da se razvije ova struktura, u pratećoj aplikaciji koristi se varijanta sa *katalogom*[4]. Katalog je niz celih brojeva koji se, poželjno, čitav čuva u radnoj memoriji i u sekundarnoj datoteci radi trajnog skladištenja. Primarna datoteka je blokirana i to tako da količina elemenata koja može da stane u bloku i maksimalna veličina kataloga koja sme da se nalazi u radnoj memoriji budu u skladu sa očekivanim opterećenjem mape. Razlog je taj što na svaku korpu (blok) ide barem jedan član u katalogu (teži ka jedan, ali u praksi varira između 1 i 2).

Katalog se koristi kao dodatna heš funkcija za pronalaženje korpe koja je zadužena za traženi ključ. Vrednosti u katalogu predstavljaju indekse korpi, a pošto su korpe statičnog kapaciteta, pomoću indeksa je moguće odrediti i poziciju korpe u memoriji, te je učitati i pronaći vrednost. Katalog je uvek dimenzija 2^d gde $d = 0$ označava DRD sa jednom korpom. Kada se zatraži vrednost pod određenim ključem, prvo se izvrši *transformacina funkcija* pomoću koje se iz kataloga pronađe indeks korpe. Transformaciona funkcija uzima

ključ, izvršava pomoćnu heš funkciju nad njim te na osnovu prvih d bitova (koje pretvori u celi broj) određuje poziciju u katalogu iz koje će pročitati indeks korpe. Dakle, zajedničko za sve ključ-vrednosti iz neke korpe jeste da im je prvih d' bitova ove heš funkcije jednako. Dužina zajedničkog prefiksa čuva se u svakoj korpi, označava se sa d' i pripada skupu {0... d}. Sa dobrom heš funkcijom, d' je ili jednako d ili $d - 1$.

Lokalna d' vrednost raste prilikom *cepanja* korpe. Cepanje nastupa kada je potrebno dodati novi ključ-vrednost par u korpu, a korpa nema dovoljno slobodnog mesta za tu operaciju (ili ako prilikom osvežavanja vrednosti nova vrednost zahteva više memorije nego što je dostupno). U tom slučaju, ukoliko je d' te korpe jednako d, na kraju primarne datoteke alocira se nova korpa. U suprotnom slučaju nastaje ekspanzija kataloga, što će biti opisano kasnije, nakon čega sledi isti postupak cepanja. Ukoliko je prefiks transformacione vrednosti pv, za koji je prepunjena korpa zadužena, onda će nova korpa biti zadužena za prefiks $2pv + 1$, a stara korpa za prefiks $2pv$. Ujedno time obe korpe sada odgovaraju za prefiks od jednog bita duži, te novo d' postaje $d' + 1$. Za sve k-v parove iz stare korpe izvršava se transformaciona funkcija i ukoliko je bit na d' + 1 mestu jednak nuli, onda k-v par ostaje u staroj korpi. U suprotnom, taj par prelazi u novu korpu na kraju primarne datoteke.

Nakon podele korpe na dva dela, potrebno je osvežiti katalog. Činjenica da svaki član kataloga ukazuje na korpu, a da broj korpi ne može biti veći od veličine kataloga, a obratno može, ukazuje da se jedna korpa (odn. njen indeks) vodi pod više mesta u katalogu. Ta osobina nije slučajna i usko je vezana uz odnos d' i d. Ako je pv prefiks neke korpe, onda svi članovi kataloga ukazuju na tu korpu, počevši od indeksa $pv * 2^{d-d'}$ pa do $(pv + 1) * 2^{d-d'}$, isključujući poslednji član. U tom intervalu, koji je dužine $2^{d-d'}$, potrebno je osvežiti drugu polovinu da pokazuje na novu korpu.

Kao što je već nagoviješteno, ukoliko je d = d', onda cepanje nije moguće dok se ne izvrši ekspanzija kataloga, što podrazumeva dupliranje svakog člana tog niza, tako da

katalog sa vrednostima c_1, c_2, c_3, c_4 postaje c_1, c_1, c_2, c_2, c_3, c_3, c_4, c_4. Sad je moguće izvršiti cepanje i na jednom mestu u katalogu će se zapisati indeks novog kataloga ("druga polovina" iz prethodnog paragrafa u ovom slučaju označava neko c_{2x+1}).

Čak i nakon cepanja korpe, moguć je (iako vrlo neverovatan i nepoželjan) slučaj da su sve k-v vrednosti imale isti ključni bit kao i nova vrednost koja se dodaje (ili menja) tako da na opet svi završavaju u istoj korpi, dok druga korpa ostaje prazna. Dakle teoretski moguće je uzastopno izvršiti dve ili više ekspanzija kataloga.

4.3 Kodiranje razlike

U ovom kontekstu, ovaj metod kodiranja se koristi za kompresovanje sortiranih nizova prirodnih brojeva. U izradi indeksa ovi nizovi se pojavljuju kod evidentiranja dokumenata koji sadrže neku reč, ili kod pozicija unutar dokumenta na kojima se ta reč nalazi. Pošto su podaci već sortirani radi bržeg pronalaženja preseka, uvek se prvo primenjuje kodiranje razmaka. Ukoliko je dat neopadajući niz $a_1, a_2, a_3 \ldots a_n$, nakon kodiranja razmaka dobija se niz $a_1, (a_2 - a_1), (a_3 - a_2) \ldots (a_n - a_{n-1})$. Dok sam niz raste, kodirani razmaci variraju u užem opsegu. Takav niz je pogodno dalje kompresovati i tu je u izradi pratećeg rada korišćen univerzalni kod za zapis celih brojeva, na nivou bajta.

4.4 Kodiranje celih brojeva

Skladištenje celog broja u radnoj memoriji zauzima od 1 do 8 bajtova, u zavisnosti od potrebe i kako je programer definisao varijablu. Tako na primer, brojevi veličine do 2 milijarde staju u četiri bajta. Međutim, broj koji je potrebno zapisati u datototeku, predviđenu za sekvencijalno čitanje, može se kodirati na način da ne zauzima fiksan prostor, nego promenljiv, u zavisnosti od njegove veličine. Na taj način, ukoliko je većina brojeva mala, moguće je uštedeti mnogo memorije. U izradi prateće aplikacije na više mesta korišćen je tzv. univerzalni kod za varijabilno bajtovno kodiranje prirodnih brojeva[5].

Univerzalan je u smislu da algoritam kodiranja nije određen parametrima, već je uvek isti, bez obzira na raspodelu brojeva koji se kodiraju. Kodiranje je na nivou bajta, od 8 bitova bajta, prvih 7 je korišćeno kao memorija, a 8. bit je korišćen kao kontrolni (moguće je koristiti i prvi bit kao kontrolni). On ukazuje na potrebu čitanja sledećeg bajta (koji je organizovan na isti način). Po tome, brojevi koji se mogu predstaviti pomoću 7 bitova (0... 127) staju u jedan bajt i njihova reprezentacija u serijskoj datoteci je identična reprezentaciji bajta u radnoj memoriji. Brojevi koji se u binarnom zapisu predstavljaju sa od 8 do 14 bitova (128... 16383) staju u dva bajta itd. Sam proces kodiranja i dekodiranja je jednostavan, brz i pogodan u smislu da male vrednosti zauzimaju jedan ili dva bajta, dok velike vrednosti ne zauzimaju mnogo više od teorijskog minimuma. U sprezi sa kodiranjem razlike moguće je kompresovati dugačke sortirane nizove u idealan *jedan bajt po elementu* slučaj.

4.5 LZW kompresija

Kod zapisa dokumenata i njihovih polja, u pratećoj aplikaciji, koristi se LZW (Lempel-Ziv-Welch[6]) kompresija, ukoliko za to ima potrebe. Da li vredi kompresovati neki niz bajtova heuristički se može utvrditi analizom entropije u linearnom vremenu, proverom veličine polja (male vrednosti ne vredi kompresovati) ili statički. U pratećoj biblioteci za indeksiranje, korisniku je omogućeno da, prilikom dodavanja polja u dokument, navede da li želi da se određeno polje dokumenta kompresuje ili ne tokom zapisivanja.

4.6 Svrha kompresije

U ovom radu pomenute su samo neke od osnovnih kompresija koje se mogu upotrebiti u izradi sistema za pretraživanje. Postoje tri glavne prednosti upotrebe iste. Prva korist je ta što je kompresovani zapis manje veličine te se na određenoj (spoljašnjoj) memoriji može skladištiti veća količina podataka. Druga korist je ta što čitanje podataka kraće traje, jer vreme potrebno za komuniciranje sa spoljašnjim uređajem prevazilazi vreme potrebno da

se podaci dekompresuju, ukoliko je metod dekompresije brz. Treća pogodnost se ogleda u mogućnosti privremenog skladištenja (keširanja) veće količine podataka u radnoj memoriji. Dakle, ukoliko indeks podržava čuvanje najčešće korišćenih podataka u radnoj memoriji, a vreme potrebno za dekompresiju podataka je zanemarljivo, onda je praktično keširati podatke u kompresovanom obliku i dekompresovati po potrebi.

4.7 Strukture za brže izračunavanje preseka

Složenost izračunavanja preseka dvaju listi je linearna, ukoliko se koristi klasičan *merge* algoritam. Ukoliko su liste dugačke ili je Bulov upit kompleksan (sastoji se iz više reči), ovakvo traženje preseka može potrajati duže nego što je neophodno. Mogućnost izvođenja preseka može se unaprediti nadograđivanjem strukture liste sa dodatnim informacijama, tzv. preskocima iliti skokovima (engl. skip). Informacije o mogućim skokovima omogućavaju zaobilaženje isčitavanje celog sadržaja liste, nauštrb malog povećanja veličine indeksa. Pokazivači nisu od koristi pri izračunavanju unije.

4.8 Skok pokazivači

Ovaj metod radi na principu dodavanja skok-pokazivača unutar liste dokument identifikatora. Pokazivač se nalazi na početku liste, te nakon svakih m elemenata liste, koji se smatraju jednim nedeljivim blokom. Drugi način da se ovo predstavi jeste da je lista podeljena na manje podliste između kojih se nalazi po jedan pokazivač koji omogućava preskakanje sledećeg bloka. Svi blokovi sadrže po m zapisa, osim poslednjeg, koji može sadržati manje ukoliko broj zapisa cele liste nije deljiv sa m. Sledi da je, ukoliko je lista od n elemenata, broj blokova i preskočnih pokazivača jednak:

$$pk(n, m) = \lceil \frac{n}{m} \rceil$$

Pokazivač je definisan parom celih brojeva *(skokId, skokPomeraj)*. Broj *skokId* je

identifikator dokumenta koji se nalazi na početku nakon-sledećeg bloka. Pomeraj služi za određivanje lokacije tog bloka i to je u suštini dužina sledećeg bloka u bajtovima. Prilikom izračunavanja preseka, pomoću ove informacije moguće je preskočiti ostatak trenutnog bloka jer je poznat početak sledećeg bloka.

Kod odabira dužine bloka m, manja vrednost broja m omogućava češće korišćenje pokazivača, ali smanjuje samu korist od preskakanja - i važi obratno - što je m veće, verovatnoća korišćenja pokazivača je manja, ali je stvarna dobit od istog mnogo veća. Pravilo koje se pokazalo kao dobro u praksi je da se m izračunava u zavisnosti od dužine liste, i to prema formuli:

$$m = \sqrt{n}$$

Dakle lista od milion elemenata bila bi podeljena u hiljadu blokova od po hiljadu elemenata.

Prilikom izvođenja preseka dve ili više listi, ukoliko je trenutni maksimum *idMax* (ili minimum, ukoliko su dokument-identifikatori uređeni opadajuće), onda se u svim listama koriste svi pokazivači kod kojih je *skokId* manji ili jednak *idMax*. Praktična upotrebljivost ovog mehanizma demonstrirana je na sledećem primeru:

```
foo: (5, p0) 1,2,3,4 (9, p1) 5,6,7,8 (13, p2) 9,10,11,12 (17, kraj) 13,14,15,16
bar: 1, 10, 12, 20
```

Pokazivači su dati u zagradama kod termina *foo*. Bez upotrebe skokova, potrebno je ispitati svih 16 + 3 = 19 identifikatora iz obe liste i pronaći preseke (1, 10, 12). Međutim, upotrebom skokova moguće je preskočiti deo prvog, ceo drugi i četvrti blok. U gornjem primeru, korišćena je specijalna vrednost kraj koja označava kraj liste, odnosno, ukoliko se taj pokazivač iskoristi, lista je pročitana i presek je izračunat. Ova vrednost se može definisati kao 0, pri čemu se sve ostale vrednosti *k* kodiraju kao *k + 1*.

	Stanje: “foo”	“bar”	maxId	Posledica
1.	`id=1 skok=(5, p0)`	`id=1`	`1`	Dodaje se id=1 u listu preseka, čitači se pomeraju na sledeći indentifikator
2.	`id=2 skok=(5, p0)`	`id=10`	`10`	skokId(foo) < maxId => iskorištava se pokazivač na sledeći blok za foo
3.	`id=5 skok=(9, p1)`	`id=10`	`10`	pokazivač se opet iskorištava
4.	`id=9 skok=(13, p2)`	`id=10`	`10`	skok se ne može iskoristiti jer 13 > 10, napreduje se linearno kroz listu foo
5.	`id=10 skok=(13, p2)`	`id=10`	`10`	Dodaje se id=10 u listu preseka, čitači se pomeraju na sledeći identifikator
6.	`id=11 skok=(13, p2)`	`id=12`	`12`	Analogno... foo čita dalje
7.	`id=12 skok=(13, p2)`	`id=12`	`12`	Dodaje se id=12
8.	`id=13 skok=(17,kraj)`	`id=20`	`20`	koristi se pokazivač
9.	`kraj`		`max`	kraj izvršavanja

4.9 Alternative

Oznaku za kraj liste moguće je kodirati i na drugi način. Ukoliko je broj blokova poznat u svakom trenutku (prilikom izvršavanja pretrage i optimizacije indeksa), kod poslednjeg pokazivača je dovoljno znati samo skokId te moguće je usaglasiti koder i dekoder liste kod kojih je poslednji skok definisan jednim brojem. Ipak, tokom izrade prateće aplikacije, autor je primetio da ovakve optimizacije nisu uvek najsrećnije (ušteda je veoma mala, a moguće su komplikacije).

Naprednija tehnika od skok pokazivača jeste upotreba skok liste (engl. *skip list)* koje

omogućavaju logaritamsku složenost pronalaženja bilo kog dokument identifikatora, nasuprot korenske složenosti. Ovaj metod koristi se u indekseru *Lucene,* koji je otvorenog koda. U eksperimentalnom indeksu, sa 1.2 GiB wikipedia stranica, ovaj metod ostvaruje 42% manje ulazno-izlaznih operacija, 16% smanjeno vreme izvršavanja prosečnog konjunktivnog upita od tri termina, uz neznatno povećanje veličine indeksne strukture (od 1 do 2%[7]).

4.10 Kodiranje skok pokazivača

Vrednost *skokId* može se kodirati pomoću bajtovnog varijabilnog zapisa brojeva i uz to moguće je koristiti kodiranje razlike između susednih *skokId* vrednosti. Pošto su svi blokovi sastavljeni od jednakog broja elemenata, njihova dužina u zapisu na medijumu se grupiše oko neke srednje vrednosti, te se nameće očigledno rešenje da se i ova vrednost takođe kodira varijabilnim kodom i to kao razlika trenutnog pomeraja od prethodnog pomeraja. Pri tom, pošto pomeraj može biti i pozitivan i negativan, potrebno je mapirati skup celih na skup prirodnih brojeva i obratno, što nije teško.

5 Ocenjivanje i rangiranje

Bulov upit pronalazi sve dokumente koji odgovaraju upitu. Ukoliko je rezultate potrebno predstaviti čoveku na razmatranje, kao što je to u slučaju web pretraživača, nerealno je očekivati da će pregledati hiljade dokumenata u potrazi za relevantnim informacijama. U tu svrhu postoje matematičke metode (kada se implementiraju postaju mašinske) ocenjivanja relevantnosti dokumenta u odnosu na postavljeni upit. Time je moguće urediti rezultate u niz tako da je veća verovatnoća da će korisnik među prvim rezultatima pronaći relevantan *pogodak*. Osim pogodnosti iz korisničkog ugla posmatranja, ova mogućnost korisna je i iz ugla računarskog sistema, jer se može iskoristiti kako bi se povećale performanse. Korisnik će uz ispitivanje manjeg broja dokumenata brže pronaći svoj predmet istraživanja, što za sistem znači manju komunikaciju sa skladištem podataka. Ovo se postiže time što se rezultat upita, niz *dokumentId* vrednosti, čuva u korisnikovoj sesiji, a dokumenti se straniče i učitavaju po potrebi, tako da je za prikaz prve strane rezultata potrebno 10 ili 20 pristupa datoteci dokumenata. Ukoliko korisnik zatraži još dokumenata, oni se serviraju pomoću zapamćenog niza identifikatora. U praksi kod (svih?) web pretraživača se koristi skraćivanje liste – jer korisnici retko kad idu dalje od desete stranice, tako da dovoljno je čuvati prvih 1000 identifikatora, osim u slučajevima kada su svi rezultati neophodni.

Konačno, za implementaciju sledećih algoritama neophodna je dodatna informacija u dokument listama – a to je dokument frekvencija. To znači da posle svakog dokument identifikatora sledi još jedan celi broj koji predstavlja frekvenciju reči u dokumentu.

5.1 Ponderisanje termina

U rečniku indeksa uz svaku reč se čuva broj dokumenata koji sadrži tu reč (učestanost, frekvencija), te se na osnovu te vrednosti izračunava njena bitnost (težina) – što se reč

ređe pojavljuje, ona se smatra bitnijom i obratno. Ova vrednost se označava kao ***df(t).*** Inverzna dokument frekvencija, ***idf(t)***, je vrednost koja se dobija skaliranjem dokument frekvencije u odnosu na ukupan broj dokumenata N, po formuli:

$$idf(t) = \log \frac{N}{df(t)}$$

Na primer, ako je *df(x)* = 10, to ne govori ništa o važnosti te reči, jer nedostaje informacija o veličini korpusa. Ali ako se zna ukupan broj dokumenata, npr $N = 500$, onda:

$$idf(x) = \log\left(\frac{500}{10}\right) = \log(50) \approx 3.912$$

Ova vrednost je univerzalna (nebitno je da li se u sistemu nalazi pet stotina ili pola milijarde dokumenata).

5.2 Ocenjivanje reči unutar dokumenta

Vrednovanje dokumenta *d* u odnosu na upit $q(t_1, t_2 \ldots t_n)$ zavisi od *idf(t),* za svaku reč ponaosob, te od učestanosti reči *t* unutar dokumenta *d,* označeno kao ***tf(t,d)***. Ova vrednost označava se kao ***tf-idf(t,d)*** i jednaka je *tf(t,d) * idf(f).* Jednostavan metod za izračunavanje težine celog dokumenta jeste da se sumiraju *tf-idf* vrednosti za sve reči iz upita, međutim postoje i bolje formule. Osim toga, pošto duplo veći broj iste reči ne predstavlja nužno duplo veći kvalitet, *tf(t,d)* se skalira, kao na primer:

$$wt(t, d) = 1 + \log tf(t, d), t \in d$$
$$wt(t, d) = 0, t \notin d$$

U tom slučaju vrednovanje termina je:

$$wt - idf(t, d) = wt(t, d) \times idf(t)$$

Konačna ocena dokumenta u odnosu na upit dobija se sumiranjem ovih vrednosti za sve termine iz upita:

$$ocena(q, d) = \sum_{i=1}^{n} wt - idf(t_i, d)$$

Ova formula je jedna od najosnovnijih metoda rangiranja i u pratećoj aplikaciji korišćene su softificiranije metode. Kod pretraživanja web stranica, hronološko uređivanje rezultata pokazalo se kao najpraktičnije. Kod pretraživanja podataka iz društvenih mreža korišćena je formula koja daje prednost svežim i kvalitetnim objavama. Kvalitet nekog objekta zavisi od afiniteta korisnika prema autoru, te drugim korisnicima koji su diskutovali ili označili objavu. Pokazalo se da ovaj metod personalizovanog polustatičkog rangiranja odlično funkcioniše u kombinaciji sa Bulovim upitima.

6 Distribuirano indeksiranje

Kada jedan računar ne može sam da opsluži sve klijente ili je korpus prevelik za njegov kapacitet, onda se koristi distribuiran indeks. Uz pomoć opisanih metoda za stvaranje lokalnog indeksa, moguće je napraviti i distribuirani indeks. U sledeća dva paragrafa ukratko su predstavljena dva osnovna principa, po načinu deljenja dokumenata.

6.1 Vertikalna podela

Pod vertikalnim deljenjem podrazumeva se da je rečnik distribuiran, odnosno da jedna mašina vodi računa samo o užem podskupu termina i da su svi indeksni podaci vezani za jednu reč samo na jednom računaru. U ovom slučaju prilikom indeksiranja dokumenta neophodno je izvršiti njegovu tokenizaciju i potom poslati dokument identifikator mašinama koje rukovode terminima iz dokumenta. Uz termin i identifikator dokumenta, šalju se i ostali podaci potrebni za pretraživanje, poput frekvencije reči u dokumentu, ili pozicije u kojima se pojavljuje, radi pretraživanja fraza.

Prilikom pretraživanja, potrebno je utvrditi koji računari su zaduženi za termine iz upita, te sa njih dovući dokument liste i izvršiti presecanje, a moguće su i drugačije organizacije. Optimizacije u ovakvom sistemu su moguće upotrebom probabilističkih struktura *(bloom filter).* Skladištenje dokumenata i pridruživanje identifikatora moraju se posebno rešiti.

6.2 Horizontalna podela

Kod ovog pristupa, svaki računar odgovoran je za sve reči, ali ne i za sve dokumente, već samo za deo njih (što više računara, to manji deo, što pruža odličnu skalabilnost po pitanju skladištenog kapaciteta). Po pristizanju dokumenta bira se računar, koji je najmanje opterećen ili slučajnom metodom, i njemu se prosleđuje na indeksiranje. Ovakav sistem može se najlakše oponašati upotrebom već postojećeg lokalnog indeksa. Skladištenje i dodeljivanje identifikatora obavlja se lokalno. U odnosu na vertikalno deljenje, ovaj metod je mnogo jednostavniji za indeksiranje.

Kod pretraživanja horizontalno podelenog indeksa kontaktiraju se svi računari, upit se izvršava na svakom podjednako i izvršava se spajanje rezultata pojedinačnih mašina. Pošto je neophodno kontaktirati sve računare, broj mogućih upita u sekundi lošije se skalira od indeksiranja.

7 Zaključak

Prva verzija Wowd indeksera napravljena je za samo par meseci. Nakon višenedeljnog, temeljnog testiranja i ispravljanja grešaka unutar firme, EDB je integrisan u glavni paket aplikacije. U toku prve godine korišćenja, na svoje kućne računare instaliralo ga je je i koristilo preko pola miliona korisnika širom sveta.

Autor je stekao mišljenje da je korist od razvijanja privatnog sistema, nasuprot korišćenju javno dostupnog, bila dvostruka. Prvo, sistem je ispunio tražena očekivanja. Drugo, autor je u toku rada stekao iskustvo koje se teško može naučiti korišćenjem već postojećeg softvera, pogotovo ukoliko je reč o biblioteci ili sistemu zatvorenog koda.

[1] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze, *Introduction to Information Retrieval*, 1st edn (Cambridge University Press, 2008).

[2] 'Wowd' <http://www.wowd.com/> [pristupljeno 21 januara 2011].

[3] 'PostgreSQL: Documentation: Manuals: PostgreSQL 8.3: Full Text Search' <http://www.postgresql.org/docs/8.3/static/textsearch.html> [pristupljeno 21 januara 2011].

[4] Ronald Fagin and others, 'Extendible hashing\—a fast access method for dynamic files',*ACM Transactions on Database Systems (TODS)*, 4 (1979), 315–344 <doi:10.1145/320083.320092>.

[5] 'Variable-length quantity - Wikipedia, the free encyclopedia' <http://en.wikipedia.org/wiki/Variable-length_quantity> [pristupljeno 24 januara 2011].

[6] 'Lempel–Ziv–Welch - Wikipedia, the free encyclopedia' <http://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch> [pristupljeno 25 januara 2011].

[7] '[#LUCENE-866] Multi-level skipping on posting lists - ASF JIRA' <https://issues.apache.org/jira/browse/LUCENE-866> [pristupljeno 25 januara 2011].