SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

BIOINFORMATIKA

PROJEKT

BLOOM FILTERI

Autori:
Hrvoje Bagarić
Ivan Herak
Ivan Kraljević

Zagreb, 2014.

Sadržaj

Opis problema

1.1 Bloom Filter

Bloom filter je memorijski efikasna vjerojatnosna struktura podataka (engl. *space-efficient probabilistic data structure*) koju je 1970. godine osmislio Burton Howard Bloom. Bloom filteri se koriste za provjeru pripadnosti elementa skupu. Prilikom provjere pripadnosti mogući su lažno pozitivni odgovori, ali lažno negativna nisu.

U svojoj osnovi Bloom filter se sastoji od niza bitova duljine m i k funkcija sažimanja (engl. *hash functions*). Jedine dvije funkcije koje podržava su za dodavanje elementa u skup i provjeru pripadnosti elementa skupu. Uklanjanje elemenata iz skupa nije dozvoljeno jer bi ono omogućavalo pojavljivanje lažno negativnih odgovora, no postoji posebne varijante Bloom filtera kod kojih je ono dopušteno (*Counting filters*).

Prilikom dodavanja elementa u skup, nad elementom se izračunava k sažetaka koje određuju k pozicija u m bitovnom nizu. Nakon toga se bitovi na pripadnim pozicijama postavljaju u 1 i time je funkcija dodavanja elementa gotova.

Prilikom provjeravanja pripadnosti elementa skuput, nad elementom se izračunava k sažetaka koje određuju k pozicija u m bitovnom nizu (isti postupak kao i kod dodavanja elementa). Nakon toga se provjeravaju bitovi na pripadnim pozicijama te ako barem jedan nije postavljen u 1 onda se element sigurno ne nalazi u skupu, a ako su bitovi na svim pozicijama postavljene u 1 onda je element vrlo vjerojatno u skupu.

Pošto je duljina binarnog niza ograničene duljine, prekomjernim dodavanjem novih elemenata u skup povećava se vjerojatnost pojave lažno pozitivnih rezultata prilikom upita. Prema Almeida et al. (2007) funkcije pomoću kojih se može odrediti prikladna veličina binarnog polja i broj potrebnih funkcija sažimanja su:

$$k = \log_2 \frac{1}{P}$$

$$m = n \frac{|lnP|}{(ln2)^2}$$

gdje je k broj funkcija sažimanja, P prihvatljiva vjerojatnost pojave lažno pozitivnih, m duljina binarnog niza, n očekivani broj elemenata koji će se dodati u skup.

1.1.1 Partitioned Bloom Filter

Partitioned Bloom Filter je posebna varijanta Bloom filtera kod kojega je binarni niz duljine m podijeljen na k jednakih (ili podjednakih ukoliko je ostatak dijljena m sa k različit od 0) dijelova. Pri tome svaka od k funkcija sažimanja može postavljati bitove samo na svome dijelu niza tj. u svojoj particiji.
Ovakvom podjelom binarnog niza ostvaruje se veća robusnost Bloom filtera i veća otpornost na pojavu lažno pozitivnih odgovora pošto nijedna funkcija sažimanja neće generirati isti indeks kao i neka druga funkcija.

1.1.2 Scalable Bloom Filter

Scalable Bloom Filter koji je predložen u Almeida et al. (2007) je posebna varijanta Bloom filtera čija je glavna svrha rješavnje problema apriornog zadavanja očekivanog broja elemenata.
Scalable Bloom Filter dotični problem rješava tako da interno sadrži listu ("običnih") Bloom filtera koja dinamički raste kada se posljednji filter napuni. Ispitivanje pripadnosti elementa filteru se radi tako da se provjerava da li element pripada nekom filteru iz liste. Ukoliko element pripada i-tom filteru liste onda element sigurno pripada Scalable Bloom Filteru.

Uz to, svaki novi Bloom filter koji se dodaje u dinamičku listu se gradi tako da ima manju vjerojatnost lažno pozitivnih odgovora od svog prethodnika. Za tu potrebu se uvodi poseban parametar r (*tightening ratio*). Parametar se zadaje u rasponu od 0 do 1.
Vjerojatnost pojave lažno pozitivnih odgovora i-tog filtera je:

$$P_i = r \cdot P_{i-1}$$

Posljedica uvođenja dotičnog parametra je rast broja funkcija sažimanja i rast duljine binarnog niza.

1.2 Funckije sažimanja (Fowler–Noll–Vo i Murmur-Hash)

U svojoj teorteskoj osnovi za izgradnju Bloom filtera potreban je veći broj funkcija sažimanja. Prema Kirsch and Mitzenmacher (2007) proizvoljan broj hash

funkcija može se simulirati korištenjem samo dvije različite funkcije sažimanja. Simulacija i-te funkcije sažimanja određena je izrazom:

$$h_i(x) = h_1(x) + i \cdot h_2(x)$$

U gornjem izrazu x predstavlja ključ koji se treba sažeti a h_1 i h_2 predstavljaju dvije različite funkcije sažimanja.Ovakav način izračuna sažetka olakšava izradu bloom filtera te iz razloga što ne koristimo *prave* hash funkcije ukupan broj potrebnih računskih operacija je drastično manji.
U sklopu projekta za implementaciju bloom filtera korištene su funkcije sažimanja Fowler–Noll–Vo (FNV) i MurmurHash. Obe funkcije obilježava to što su nisu kriptografske stoga je izračun sažetka puno brži te pogodne su upravo za raspršivanje ključeva.
FNV funkcija sažimanja se temelji na usporedbi bitova i množenjem sa pripadnim prostim brojem. Pesudokod FNV funkcije sažimanja (izvor: Noll (2014)):

Algorithm 1 FNV-1 algoritam

```
hash = FNV_offset_basis
for each octet_of_data to be hashed do
    hash = hash x FNV_prime
    hash = hash xor octet_of_data
end for
return hash
```

Varijable FNV_offset_basis i FNV_prime odabiru se prema broju bitova koje će rezultat imati. Za 32 bitnu implementaciju FNV_offset_basis iznosi 2166136261, a FNV_prime 16777619. Prikazani pseudokod je inačica FNV-1, ali postoji još i inačica FNV-1a i jedina razlika između dviju funkcija je poredak množenja i XOR funkcije. U inačici FNV-1a prvo se obavlja operacija XOR pa tek onda množenje. Inačica FNV-1a je bolja u raspršivanju malih ključeva i radi toga je impelemntirana ta inačica za potrebe bloom filtera.

Murmur funkcija sažimanja temelji se na množenju i rotaciji bitova. Pseudokod Murmur funkcije sažimanja (izvor: Wikipedia (2014)):

Algorithm 2 MurmurHash algoritam

hash = seed
for each 4 bytes k in data **do**
  k = k $*c_1$
  k = rotl(k, r_1)
  k = k $*c_2$
  hash = hash **xor** k
  hash = rotl(hash, r_1)
  hash = hash$*m_1 + n_1$
end for
hash = hash **xor** $(\text{hash} >> 16)$
hash = hash $*$ 0x85ebca6b
hash = hash **xor** $(\text{hash} >> 13)$
hash = hash $*$ 0x85ebca6b
hash = hash **xor** $(\text{hash} >> 16)$
hash
hash
return hash

Posebnost murmur funkcije sažimanja je u tome što obrađuje po četiri bajta podataka odjednom. Kao i kod FNV funkcije sažimanja postoje 32 bitna i 64 bitna verzija algoritma. One se razlikuju samo po brojevima završnih dvaju množenja. Varijable c_1,r_1,m_1 i n_1 su konstante i podložne su promjeni kako od tvorca algoritma tako i od pojedinih implementacija.

Jednostavni primjeri

Bloom filteri često se upotrebljavaju kada je potrebno samo provjeriti ako nešto ne postoji, obzirom da je kod njih moguća pojava lažnih pozitivnih rezultata. Na taj način se smanjuje broj pretraga za nepostojećim ključevima.

U osnovi, bloom filteri sadrže dvije osnovne funkcije: dodavanje elemenata i postavljanje upita. Prilikom dodavanja elementa računa se k hash funkcija te se u bloom filteru postavi k bitova na mjesta dobivena hash funkcijama. Na primjer, ukoliko imamo 2 hash funkcije i bloom filter od 15 bitova, za svaki element koji se dodaje računamo dvije vrijednosti (npr. računamo dvije različite hash funkcije, poput FNV i MurmurHash).
Recimo da želimo dodati ključ "fakultet". Računamo FNV hash vrijednost riječi "fakultet" (unutar opsega filtera) i tako dobijemo vrijednost "0". Nakon toga računamo murmur hash vrijednost iste riječi i na primjer dobijemo vrijednost "6". Završni dio dodavanja ključa je postavljanje bitova u bloom filteru na dobivnim mjestima:

1						1								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Kod postavljanja upita opet se računaju hash vrijednosti traženog ključa i zatim se testiraju dobiveni bitovi u filteru. Na primjer, ukoliko bi postavili opet upit s riječi "fakultet", FNV i Murmur hash funkcije opet bi dale isti rezultat (0 i 6). Provjerom bitova 0 i 6 u filteru ustanovilo bi se da su oni uistinu postavljeni, te da ključ "fakultet" (vjerojatno) postoji. S druge strane, ukoliko bi postavili upit s nekom drugom riječi, npr. "fer", FNV i Murmur funkcije bi nam dale drugacije vrijednosti (npr. 4 i 7). Tada bi provjerom u filteru vidjeli da bitovi 4 i 7 nisu postavljeni te bi tako zaključili da ključ ne postoji.
Naravno, moguća je i pojava lažnih pozitivnih rezultata. Zamislimo sljedeću situaciju: nakon ključa "fakultet" u filter dodajemo i ključ "fer". Filter nam tada izgleda ovako:

1				1		1	1							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Postoji mogućnost da će neki upit, npr. "racunarstvo" generirati FNV i Murmur hasheve redom 0 i 7. Provjerom u filteru vidimo da su ti bitovi postavljeni, iako ključ "racunarstvo" nije dodano u filter.

Implementacija i testiranje

3.1 Testiranje

Testiranje je izvršeno na C++, Python i Java implementacijama.
U sklopu testiranja koristila se datoteka koja sadrži milijun različitih elemenata duljine od 3 do 15 znakova.
U Java implementaciji se koristio Scalable Bloom Filter sa parametrima $r = 0.9$ i $fillRatioLimit = 50000$, dok se u ostalim implementacijama koristila osnovna inačica Bloom filtera. Rezultati testiranja su sljedeći:

Programski jezik	$t_{element}$	t_{ukupno}	t_{query}	Memorija
C++	1s	3ms	2ms	18MB
Python	25s	11ms	3ms	223MB
Java	0.8s	4ms	4ms	122MB

Tablica 3.1: Vrijednosti testiranja implementacija za $P = 0.0001$

Programski jezik	$t_{element}$	t_{ukupno}	t_{query}	Memorija
C++	0.7s	3ms	2ms	9MB
Python	20s	10ms	3ms	113MB
Java	0.7s	4ms	3ms	106MB

Tablica 3.2: Vrijednosti testiranja implementacija za $P = 0.01$

Iz gornjih tablica je vidljivo da je C++ implementacija po korištenju memorijskog prostora daleko bolja od Java i Python implementacija. Također, može se primjetiti kako su kod dodavanja novih elemenata Java i C++ implementacije daleko efikasnije, a za provjeru pripadnosti elementa sve 3 implementacije daju pojednake rezultate.

3.2 Upute za korištenje

3.2.1 Python implementacija

Kako bi se pokrenula Python implementacija potrebno je pokrenuti skriptu `Start.py`. Ostale skripte `BloomFilter.py`, `FNVHasher.py` i `MurmurHasher`

moraju biti u istoj datoteci. Sve skripte su pisane za 2.7 verziju Pythona. Nikakve vanjske biblioteke nisu potrebne za njihovo izvođenje.

Skripta `Start.py` zahtjeva tri argumenta prilikom pokretanja. Prvi argument mora biti put do tekstualne datoteke koja u svakoj zasebnoj liniji sadrži ključ koji će biti dodan u bloom filter. Drugi argument mora biti put do datoteke u kojoj će se zapisivati rezultati upita u bloom filter. Treći argument mora biti dozvoljeni postotak greške.

Nakon pokretanja skripta će učitati ključeve, ispisati parametre s kojim je bloom filter inicijaliziran te ponuditi korisniku da upiše ključ te provjeri da li se rezultat nalazi u bloom filteru. Rezultat će biti ispisan na ekranu te zapisan u izlaznu datoteku.

3.2.2 C++ implementacija

Za prevođenje izvornog koda potrebno je pokrenuti naredbu:

```
g++ -Wall -g Filter.cpp BioTest.cpp -lm -o biobloom
```

Ukoliko već postoji datoteka `biobloom`, nije potrebno izvršiti prethodnu naredbu. Implementacija se pokreće naredbom:

```
./biobloom ["-f"] "textfile.txt" ["queryFile.txt"]
```

Dakle, prilikom pokretanja je nužno navesti barem jedan argument ("textfile.txt"). Taj argument mora biti put do tekstualne datoteke koja u svakoj zasebnoj liniji sadrži ključ koji će biti dodan u bloom filter. Pritom je u prvoj liniji moguće navesti i željeni postotak pogreške u rasponu [1-99]%. Redak u tom slučaju mora započinjati s '?'.
Npr:

```
?1
wlhhxvd0
...
```

Datoteka mora biti formatirana za uporabu u unix okruženju (lf umjesto crlf).

Ukoliko se implementacija pokrene u ovom modu rada, tada se nakon učitavanja svih ključeva korisniku omogući postavljanje upita putem konzole. Iz tog modula rada se izlazi postavljanjem upit ":Q".

Drugi način rada je da se osim ulazne datoteke s ključevima definira i datoteka u kojoj je zapisan neki upit ("queryfile.txt"). U tom se slučaju nakon učitavanja

svih ključeva otvara i ta datoteka, te se vrši provjera nalazi se ključ koji je tamo zapisan u bloom filteru, te se rezultat ispisuje na ekran.

Prilikom pokretanja je moguće i postaviti zastavicu "-f". Ukoliko se postavi, implementacija će pretpostavljati da su svi zapisi u datotekama u FASTA formatu.

Uz priložene testne datoteke, implementacija se pokreće naredbom:
`./biobloom keyList.txt`

Nakon toga je moguće upisati elemente koji postoje (npr. wlhhxvd0, e74...), ili neke koji ne postoje u bloom filteru (npr. fer, fakultet, otorinolaringologija...). Rezultat se ispisuje na ekran.

3.2.3 Java implementacija

Za prevođenje Java implementacije potrebno je pozicionirati se u direktorij implementacije te unijeti:
`find . -name "*.java" | xargs javac`

Program se pokreće sljedećom naredbom:
`java -cp . hr.fer.bioinformatika.projekt.Main <inputFilePath> <fasta>`

Argument `<inputFilePath>` predstavlja putanju do datoteke sa riječima koje će se unijeti u bloom filter, dok drugi argument predstavlja prihvatljivu vjerojatnost pojavljivanja lažno pozitivnih upita.
Ako je argument `<fasta>` 1 pretpostavlja se da je datoteka zadana u FASTA formatu, inače se pretpostavlja da se učitava tekstualna datoteka gdje je svaka stavka zapisana u zasebnom retku.
Ukoliko se prvi argument ne navede pretpostavlja se da je rječnik pohranjen u datoteci `keyList.txt` koja se nalazi u istom direktoriju kao i program.

Zaključak

Zbog efikasne uštede u potrebnom prostoru i brzog provjeravanja pripadnosti elementa skupu Bloom filteri su danas sve više korišteni. Neki poznati primjeri korištenja Bloom filtera uključuju identificiranje malicioznih hiperlinkova (Google Chrome), reduciranje broja pristupa disku i mreži te brojni drugi.

Tokom posljednih nekoliko godina pojavio se nemali broj varijanti originalnog Bloom filtera od kojih neki pokušavaju riješiti postojeće nedostatke Bloom filtera, dok drugi proširuju mogućnosti Bloom filtera. Ove izjave dovoljno govore o popularnosti Bloom filtera te o tome kako postoji još dosta prostora za daljnji napredak.

U poglavlju o testiranju implementacije empirijski je utvrđeno kako Bloom filter efikasno koristi raspoloživi prostor za pohranu pripadnosti elementa skupu. Također, prikazano je kako u jako kratkom vremenu može dodati novi element u skup i također kako u kratkom vremenu odgovara na upite o pripadnosti elementa skupu.

Literatura

P. S. Almeida, C. Baquero, N. Preguica, and D. Hutchison. Scalable bloom filters, 2007.

A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better bloom filter, 2007.

L. C. Noll. Fnv hash. http://www.isthe.com/chongo/tech/comp/fnv/, 2014. [Online; accessed 13-January-2014].

Wikipedia. Murmurhash. http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/, 2014. [Online; accessed13-January-2004].