Strukture podataka

Seminarski rad - Bloomovi filteri

Izradili:
Ivan Lukšić
Bruno Grbavac
Matej Dražić Balov

Strukture podataka 1

Uvod 3

Kako radi Bloomov filter? 4
- Kirsch-Mitzenmacher optimizacija 6

False positive - matematika iza Bloomovog filtera 9
- False positive i false negative. 9
- Vjerojatnost false positive-a i matematika 9
- False negative? 10
- Formula false positive-a 11
- Optimalan broj hash funkcija 13

Hash funkcije 16

Svojstva hash funkcije 16
- UNIFORMNOST 16
- JEDNOZNAČNOST 17
- BRZINA 18

Kriptografske hash funkcije 19

Salting 20

Primjena 21

Korištena literatura: 24

Uvod

Zašto Bloom filter kao tema? Za izbor teme smo se odlučili jer nam širina i primjena teme daje dosta velik prostor za istraživanje i odabir načina za vlastitu primjenu za neki veći i dosta realan problem koji se javlja kod skaliranja sustava, ali i optimizacije i manjih i većih sustava. Bloom filteri se koriste kada želimo provjerit postoji li neki element u našoj bazi podataka, listi itd. Prednost korištenja njih što su zbog načina rada brži od drugih metoda te se ovisno o izboru parametara, o čemu ćemo među ostalim pisati u seminarskom radu, uštedi 10 pa i više puta memorije od nekih drugih konvencionalnih metoda, što se uvelike isplati kada trebamo pretraživati velike sustave. Bloom filteri se zasnivaju na vjerojatnosti postoji li neki element u nekom datasetu, koristimo ga kad nam je vjerojatnost false positiva prihvatljiva. Iza računanja te vjerojatnosti postoji matematički model kojeg ćemo također obraditi. No da bi razumjeli kako taj model funkcionira prvo ćemo morati objasniti neki osnovni princip rada tih filtera. Princip rada ima bazu u korištenju hash funkcija, u praksi više njih u jednom modelu, tako da je odabir hash funkcija, te njihova implementacija poprilično važna u razvoju Bloomovog filtera pa ćemo u seminaru i hash funkcijama dati vlastito poglavlje. Također, zbog raznolike mogućnosti primjene ovih filtera istraživati ćemo gdje se oni sve primjenjuju i kako, tko sve koristi i pokušati naći neke ne toliko rasprostranjene metode korištenja Bloomovih filtera

Kako radi Bloomov filter?

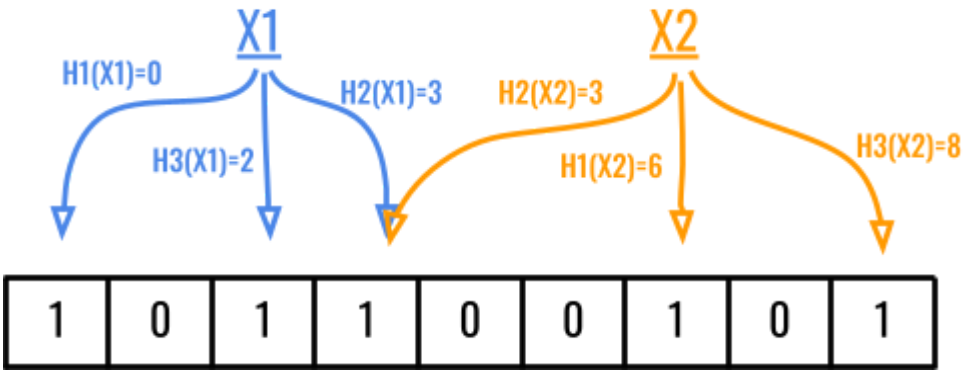
Bloomov filter je probabilistička struktura koja služi za brzu provjeru postojanja određenog elementa unutar nekog seta[1] elemenata.

Filter se sastoji od *m bitova* kojima je u početku dodijeljena vrijednost 0.

0	0	0	0	0	0	0	0	0

1.1. primjer Bloomova filtera s 9 bitova

Konstrukcija filtera se najčešće odvija pri pohrani podataka[2]. Svaki element kojeg pohranjujemo u set "uvrštavamo" u ***k hash funkcija***. Za svaki od ***n elemenata*** koje pohranjujemo, stoga dobivamo po ***k*** indeksa[3]. Bitove filtera koje predstavljaju dobiveni indeksi, ako već nisu, postavljamo na vrijednost 1.

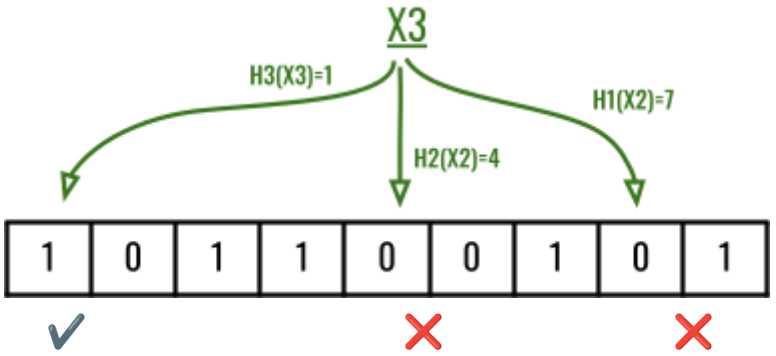


1.2. unos elemenata X1 i X2 u Bloomov filter s 9 bitova i 3 hash funkcije H1, H2, H3

[1] struktura za pohranu podataka - hash tablica, vezana lista, niz, itd.
[2] sam Bloomov filter se ne koristi za pohranu podataka
[3] hash funkcija vraća indeks elementa filtera tj. cijeli broj između od 0 do m

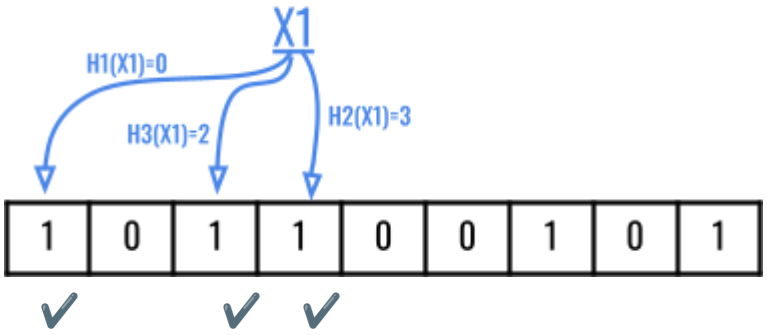
Provjera postojanja određenog elementa u nekom setu upravo je funkcija Bloomova filtera. Element za kojeg se vrši provjera uvrštava se u hash funkcije i dobiva se k indeksa. Zatim se očitava vrijednost bitova Bloomova filtera koje predstavljaju dobiveni indeksi.

Ako je barem jedan od tih bitova filtera jednak 0, možemo zaključiti da element nije dio spomenutog seta. Takav zaključak donosimo jer kada bi element za kojeg vršimo provjeru bio dio tog seta, tada bi hash funkcije vratile iste indekse, pa bi bitovi na tim indeksima imali vrijednost 1.



1.3. filterom sa prijašnjih primjera provjeravamo postojanje X3 u setu {X1,X2}

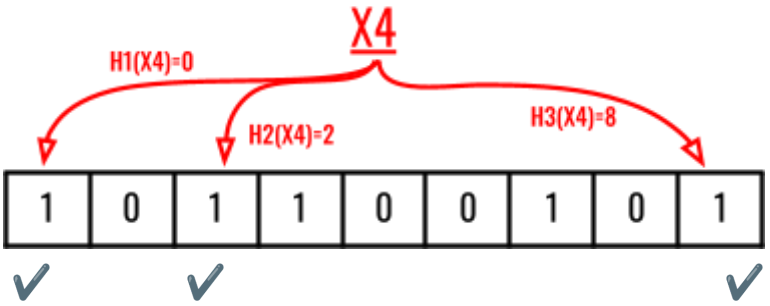
U protivnom, tj. ako su svi indeksima pridruženi bitovi jednaki 1, zbog postojanja mogućeg *false positive* ishoda, možemo reći da je traženi element "vjerojatno/možda [4]" element spomenutog seta.



1.4. filterom sa prijašnjih primjera provjeravamo postojanje X1 u setu {X1,X2}

[4] nije moguće sa potpunom sigurnošću reći da jest element

False positive je općenito ishod u kojem neka booleova funkcija vraća istinu tamo gdje bi trebala vratiti laž. Kod Bloomova filtera mogućnost false positive ishoda proizlazi iz ***preklapanja rezultata hash funkcija***, tj. za različite element, ista ili različite hash funkcije mogu dati iste indekse.
Ako su bitovi Bloomova filtera koje provjeravamo za utvrđenje postojanja određenog elementa u setu, postavljeni na vrijednost 1 pri pohrani nekih drugih elemenata, a sam element nije sadržan u setu, Bloomov filter će svejedno "reći" da set sadrži taj element iako to nije istina.



1.5. filterom sa prijašnjih primjera provjeravamo postojanje X4 u setu {X1,X2}

Kirsch-Mitzenmacher optimizacija

Za realizaciju Bloomova filtera nije nam potrebno k različitih međusobno neovisnih funkcija, naime dovoljne su samo dvije. Odabranim funkcijama se zatim na jedan od dva preporučena načina simuliranje k međusobno neovisnih hash funkcija, bez promjene vjerojatnosti false positive-a.

1) Particijski Bloomov filter - "partition scheme"

Bloom filter od m bitova se podijeli na k područja od p bitova. Željeni broj tj. k hash funkcija realizira se tako da svako područje hashira zasebna hash funkcija g_i koja se kreira pomoću dvije hash funkcije h_1 i h_2 tako da:

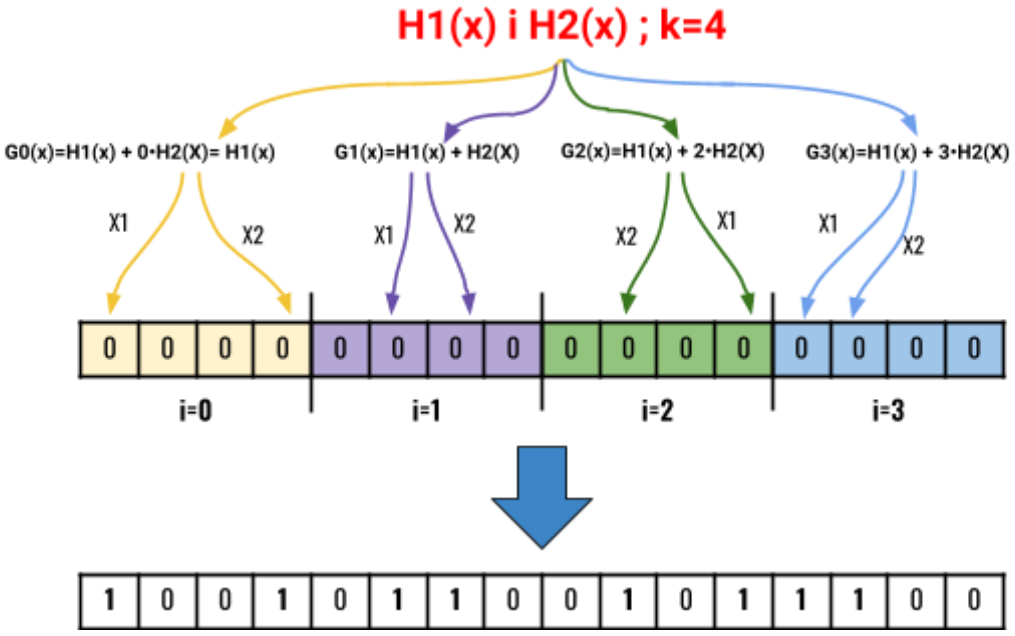
$$g_i(x) = (h_1(x) + i \cdot h_2(x))(mod\ p);\ x \in U,\ i \in \{0, 1, 2,..., k-1\}$$

2) Dvostruko hashiranje - "enhanced double hashing scheme"

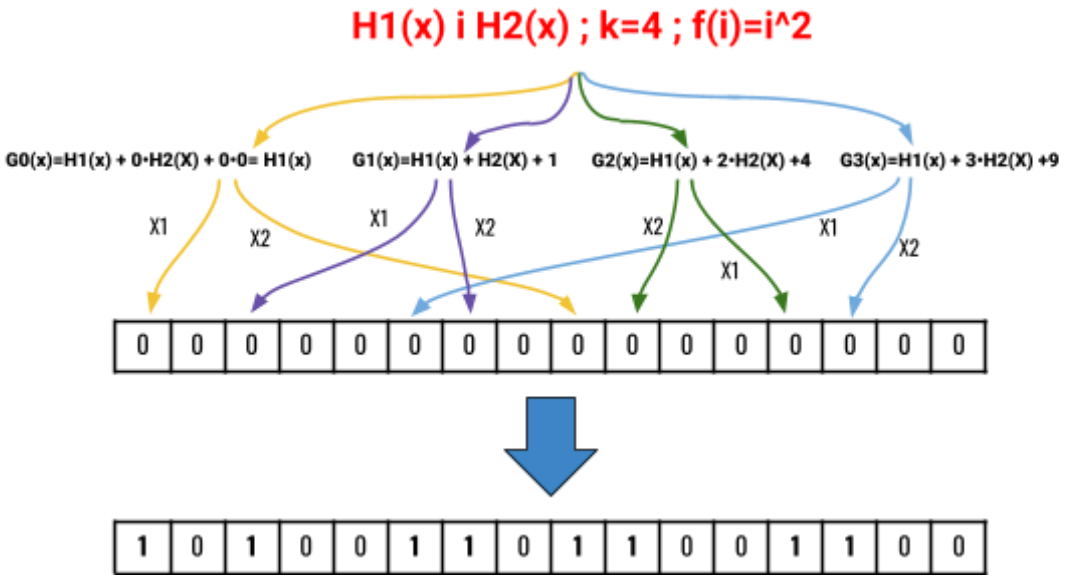
Željenih k hash funkcija generiraju se pomoću dvije nezavisne hash funkcije po navedenoj formuli gdje je f(i) funkcija jedne varijable tj rednog broja simulirane hash funkcije.

$$g_i(x) = (h_1(x) + i \cdot h_2(x) + f(i))(mod\ m);\ x \in U,\ i \in \{0, 1, 2,..., k-1\}$$

Eksperimentom u radu *"Less Hashing, Same Performance: Building a Better Bloom Filter - Adam Kirsch, Michael Mitzenmacher"* je dokazano da double hashing postupak rezultira zanemarivom, a particijski postupak gotovo nikakvom devijacijom u generiranju false positive u odnosu na standardni model Bloomova filtera.



1.6. u particijski Bloomov filter s 16 bitova unose se 4 elementa koristeći 4 simulirane hash funkcije



1.7. u Bloomov filter s 16 bitova unose se 4 elementa koristeći 4, dvostrukim hashiranjem simulirane, hash funkcije u kojima je $f(i) = i^2$

False positive - matematika iza Bloomovog filtera

Nalazi li se naš element u skupu?

Nakon nekog vremena *spremanja podataka* u bloom filtere, to jest ‘gađanja’ bitova koje nam pridjele hash funkcije, želimo provjeriti nalazi se neki element x u skupu S. Na primjer, nalazi li se slovo **‘a’ u skupu abecede**. Kada smo *spremali*[5] slova iz abecede u bloom filter, svakom je slovu pridjeljeno onoliko bitova koliko ima hash funkcija. Ako želimo provjeriti nalazi se ‘a’ u skupu abecede to radimo gledajući jesu li bitovi pridjeljeni tom podatku, koje je hash funkcija pogodila kod spremanja, vrijednosti[6] **1**.
Ako jesu, ona se zasigurno nalazi tu. Ili ipak ne?

False positive i false negative

Općenito znamo da se slovo ‘a’ nalazi u skupu slova abecede pa se ne bi trebali čuditi ako vidimo sve potrebne bitove vrijednosti 1, no što ako želimo provjeriti nalazi se u tom skupu slovo ‘y’? Sigurno očekujemo kako ćemo naići na neku ili neke bitove vrijednosti 0. To se čini logičnim, no nije nemoguće da se dogodi situacija kada će se nekom slovu iz abecede (npr. ‘m’) iz skupa prethodno dodanog, pridijeliti isti oni bitovi koji će pridijeliti i slovu ‘y’. Tada će nam javiti da postoji slovo ‘y’ iako svi znamo da se ono ne nalazi u Hrvatskoj abecedi.
Kako je došlo do toga?

[5] Zapravo ne spremamo, već mijenjamo vrijednost pojedinih bitova u Bloom filteru.
[6] Ona mijenja vrijednost bita iz 0 u 1 kad god se redni broj tog bit adresira u hash funkciji.

Vjerojatnost false positive-a i matematika

Količina bitova koje pogađaju[7] hash funkcije je zasigurno ograničena. Dakle, nije naodmet zaključiti kako će se zasigurno dogoditi slučaj kada će nekoliko istih bitova biti pridijeljeno različitim podacima.
Nameće se pitanje, koliko često će se to događati i o čemu će to ovisiti?
Ako malo razmislimo, to će sigurno ovisiti o veličini niza bitova gdje spremamo vrijednosti jer ako je taj niz malen i mi spremamo podatke brzo će se ispuniti sve vrijednosti to jest, svi bitovi će brzo prijeći iz 0 u 1 i onda će za svaki provjereni podatak vraćati kako se nalazi tu.

ZAKLJUČAK**:** Sigurno ovisi o veličini seta **m**, tj. veličini Bloom filtera.

Spomenuli smo već podatke. Nije isto spremamo li malen broj podataka ili veliki broj istih. Logično je pretpostaviti da će nam vjerojatnije vratiti *false positive* prijašnjeg spremanja ako smo prije toga spremili veliku količinu podataka.

ZAKLJUČAK: Ovisi o broju elemenata koje *spremamo* **n**.

Vratimo li se malo na to kako se uopće bitovi '*gađaju*' ili kako se uopće nekoj informaciji pridjeljuje redni broj bita čiju vrijednost mijenja, dolazimo do hash funkcija.
Broj hash funkcija je uvelike bitan i broj tih funkcija označavamo s **k**.

ZAKLJUČAK: Ako nekoj informaciji pridjeljujemo više od jednog bita, to jest koristimo više od jedne hash funkcije (**k > 1**) to matematički sigurno utječe na zauzetost bloom filtera.

[7] Ili veličina bloom filtera.

False negative?

No prije toga, zapitat ćemo se prije ili kasnije, postoji li mogućnost da nam vrati negativnu vrijednost iako smo element prethodno dodali u filter. Zapravo ne. Ako su hash funkcije promijenili vrijednost odabranim bitovima oni zastalno ostaju u **TRUE** stanju tj. logičkoj jedinici.

Formula vjerojatnosti false positive-a

Sada malo matematike. Kako zapravo dođemo do formule koja nam govori da se neki element nalazi u setu iako on nije prethodno uvršten? Uzmimo za primjer jedan set od **m** bitova koji će nam poslužiti kao bloom filter. Označimo njegove bitove rednim brojevima od **0 do m-1**.

Pretpostavimo li da je korištena hash funkcija uniformna, spremanjem podataka u naš set bitova vjerojatnost događaja da je nasumični **bit indeksa j** "pogođen" hash funkcijom[8] je:

$$p = \frac{1}{m}$$

Ovaj izraz proizlazi iz formule za vjerojatnosti događaja:

$$p = \frac{broj\ povoljnih\ događaja}{broj\ mogućih\ događaja}$$

U ovom slučaju povoljan je događaj da, pri spremanju nekog elementa u naš set, hash funkcija Bloomova filtera vrati **indeks j**. Hash funkcija vraća neki od m indeksa, pa upravo tih **m ishoda** čini skup mogućih događaja.
Nas zanima vjerojatnost da bit nije pogođen to jest :

$$p = 1 - \frac{1}{m}$$

Ovaj izraz vjerojatnost je aktivacije bita j pri hashiranju jednog elementa jednom hash funkcijom, u Bloomovom filetru imamo k hash funkcija tj. hashiramo k puta. Tada se vjerojatnosti pojedinih hashiranja tog elementa različitim hash funkcijama množe[9] :

---

[8] tj. vrijednost tog bita postavljena je u 1
[9] ovo proizlazi iz osnovnih formula vjerojatnosti - pravilo produkta

$$p'^{k} = p' \cdot p' \cdot \ldots \cdot p'$$

Ako u naš set unosimo n elemenata, proces **hashiranja se vrši** $n \cdot k$ puta, pa je naša vjerojatnost da je bit **j aktiviran**, nakon unošenja n elemenata u set:

$$p = (1 - \frac{1}{m})^{kn}.$$

Ova formula da se pojednostaviti koristeći činjenicu da je [10]

$$e^{-1} = \lim_{x \to \infty} (1 - \frac{1}{x})^{x}$$

Za dovoljno velik $n \cdot k$ imamo:

$$p = ((1 - \frac{1}{m})^{m})^{\frac{kn}{m}} = (e^{-1})^{\frac{kn}{m}} = e^{-\frac{kn}{m}}$$

Vjerojatnost pojave false positive-a je vjerojatnost da je svih k bitova koje dobivamo hashiranjem elementa koji nije u setu već aktivirano tj. postavljeno u vrijednost 1. Vjerojatnost da je neki bit aktiviran:

$$p' = 1 - e^{-\frac{kn}{m}} \Rightarrow vjerojatnost\ da\ je\ svih\ k\ bitova\ aktivirano \Rightarrow p = (p')^{k}$$

Tj. vjerojatnost pojave false positive-a u Bloomovu fileru od **m bita**, s **k hash funkcija** kojim se provjerava postojanje elementa u setu od **n elemenata** je:

$$p = (1 - e^{-\frac{kn}{m}})^{k}$$

Pogledajmo sada našu formulu i zaključimo kako promjena vrijednosti pojedine varijable[11] utječe na vjerojatnost za 'false positive':

1. N (broj elemenata koji set sadrži):
   Povećanjem broja elemenata seta kojeg predstavlja filter, **n** , raste razlomak **-kn/m**, time se **smanjuje izraz e^(-kn/m)**, a raste izraz u zagrada te cjelokupni izraz, tj. **raste vjerojatnost pojave false positive-a.** Ta pojava u skladu je sa stvarnim zbivanjima jer je pri većem broju pohranjenih elemenata Bloomov

[10] e je baza prirodnog logaritma e=2.7182818
[11] veličine koje opisuju Bloom filter

filter sve “zasićeniji jedinicama”, pa je veća mogućnost false positive-a te će kada se cijeli filter popuni ona porasti na 100%.

2. ### M (veličina bloom filtera):
   Rastom varijable m - veličine filtera, **raste izraz e^(-kn/m),** a time se **smanjuje cjelokupni izraz, tj. vjerojatnost za false positive** ishod. Opravdanje za takvo ponašanje u stvarnosti je činjenica da povećanje filtera nudi veći raspon za hashiranje čime se intuitivno smanjuje vjerojatnost hash sudara.

3. ### K (broj hash funkcija):
   Kako je broj u zagradi uvijek manji od 1, vanjski eksponent vrijednosti **k** bi u pravilu morao **smanjivati** vjerojatnost ‘false positive’-a pošto je **k** cijeli broj veći od 0. Ipak, unutar zagrade on se nalazi u brojniku pa istim pravilom kao i kod **n** zaključujemo da se njegovim povećanjem **povećava** mogućnost ‘false positive’-a.

Na sreću, mi smo ti koji odabiremo koliko ćemo hash funkcija koristiti dok će količina podataka i veličina bloom filtera uglavnom biti unaprijed zadana. Stoga nam je sljedeći cilj odabrati optimalan broj hash funkcija.

Optimalan broj hash funkcija

Označimo **vjerojatnost false positive-a sa F** i zapišimo taj izraz kao eksponencijalnu funkciju koja se međusobno poništava, čisto da bismo mogli dobiti funkciju s kojom ćemo lakše odrediti optimalni **k**.

$$F = exp(k(ln(1 - e^{-\frac{kn}{m}})))$$

funkciju u zagradama označimo s G

$$G = k(ln(1 - e^{-\frac{kn}{m}}))$$

Pronalaženjem minimalnog **G** pronašli smo i minimalni F jer je *exp* stalno rastuća funkcija i jer je $F = exp(G)$,dakle funkcija varijable G.

Derivacijom G po k (tražimo optimalni minimum 'false positive'-a) dobivamo sljedeći izraz:

$$\frac{\sigma G}{\sigma k} = ln(1 - e^{\frac{-kn}{m}}) + \frac{k}{1-e^{\frac{-kn}{m}}} \frac{n}{m} e^{\frac{-kn}{m}}$$

Iz izraza, ne baš očigledno, vidimo da je optimalno za izraz uvrstiti

$k = \frac{m}{n} ln(2)$ jer je tada ova derivacija jednaka 0, to jest, naišli smo na minimum.

Primjetimo da ovaj optimalni broj hash funkcija ovisi o omjeru broja bitova 'rezerviranih' za pojedine elemente. $\frac{sveukupna\ veličina\ bloom\ filtera}{količina\ podataka\ koje\ spremamo} = \frac{m}{n}$

Svojstvo simetrije

Optimalni broj hash funkcija možemo dobiti na jednostavniji način poznavajući *svojstvo simetrije*. Iz relacije $p = e^{\frac{-kn}{m}}$ možemo izvući ***k*** i dobiti $k = \frac{m}{n} ln2.$

$\Rightarrow$ koristimo "trik" iz prethodnog izvoda, uvodimo G $\Rightarrow$

$$G = k(ln(1 - e^{-\frac{kn}{m}}))$$

$\Rightarrow$ uvrštavamo dobiveni izraz za k $\Rightarrow G = -\frac{m}{n} ln(p) ln(1 - p)$

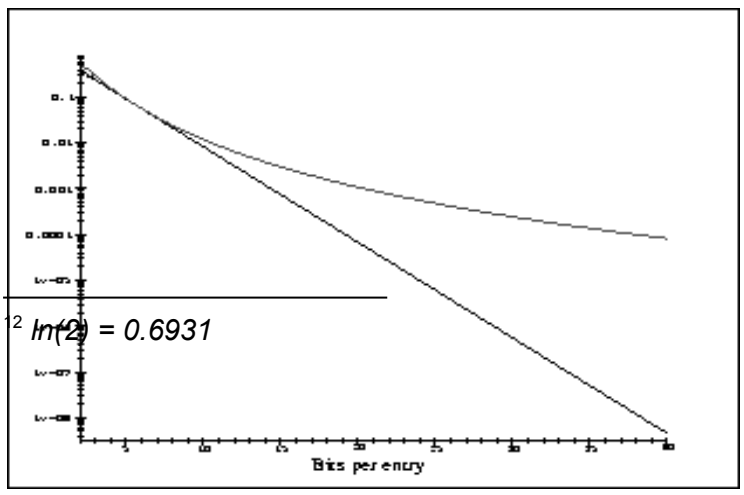
Svojstvo simetrije nam nalaže da je vrijednost G najmanja kada je $p = ½$. Iz toga, ponovo zaključujemo, optimalan $k = \frac{m}{n} ln2.$ Koristeći izraz za optimalan k, false positive poprima vrijednost:

$$F = (1 - p)^{\frac{m}{n}ln2} = (0.5)^{\frac{m}{n}ln2} \approx (0.6185)^{\frac{m}{n}}$$ [12]

- k bi naravno u praksi trebao biti cijeli pozitivan broj.
- Odabiremo najmanje cijelo vrijednosti k, da spriječimo preopterećenje hash-a.
- Očigledno k_{opt} ovisi o omjeru veličine bloom filtera i količine podataka.

Promotrimo ponašanje 'false positive'-a ovisno o tom omjeru:

$\frac{m}{n}$**(bit po elementu)**	***hash funkcije (k)***	***false positive (%)***
6	4	0.0561
8	6	0.0215
12	8	0.00314
16	11	0.000458



•**X os** - $\frac{m}{n}$- bitovi filtera po elementu seta
•**Y os -** vjerojatnost false positive-a

[12] *ln(2) = 0.6931*

•**ravna crta** - optimalan broj hash funkcija
•**k=4** koriste se 4 hash funkcije

Hash funkcije

Hash funkcija je bilo koja funkcija/preslikavanje koja ulaznoj vrijednosti proizvoljne konačne veličine pridružuje vrijednost iz fiksnog intervala veličina. Izlazna vrijednost hash funkcije naziva se hash vrijednost, hash kod ili samo hash.

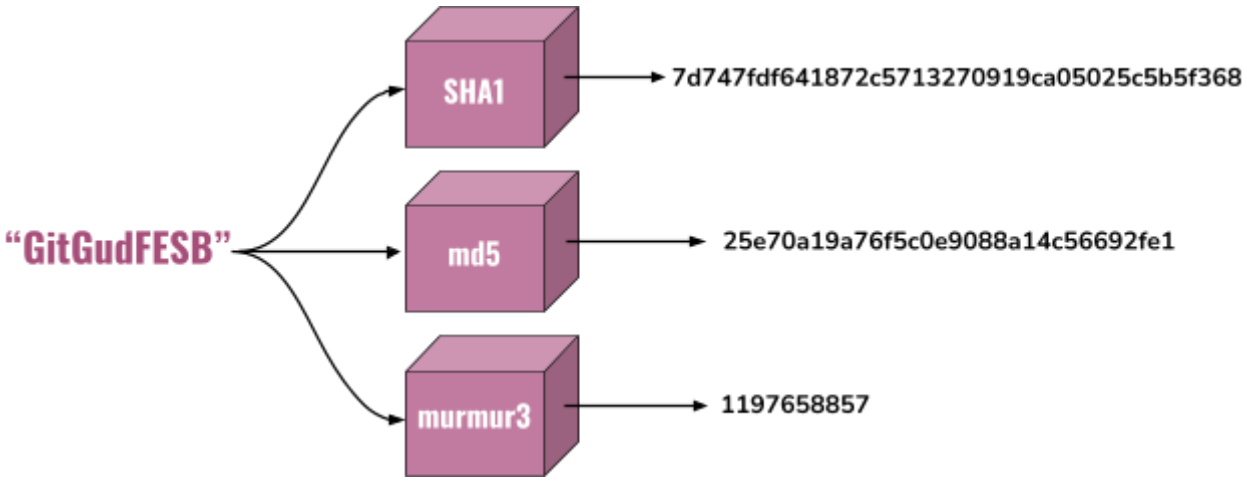
Hash vrijednost koristi najčešće za indeksiranje hash tablica u koje se mapiraju[13] ulazne vrijednosti hash funkcije, tj. podaci proizvoljnih konačnih veličina.

$$h(x)\colon X \to I$$

X - skup ulaznih vrijednosti proizvoljne konačne veličine[14]

I - skup mjesta hash tablice tj. $I = \{ x \in Z\colon 0 \geq x \geq m \}$, gdje je m veličina hash tablice

[13] takozvano hashiranje ili raspršeno pohranjivanje
[14] tj. beskonačan skup elemenata konačne veličine



2.1. Hashiranje stringa "GitGudFESB" pomoću nekih od najpoznatijih hash funkcija

Svojstva hash funkcije

UNIFORMNOST

Hash funkcija je uniformna ako unesene podatke jednoliko mapira preko intervala izlaznih vrijednosti, tj. ako je svaka izlazna hash vrijednost ispoljavana sa približno jednakom vjerojatnošću[15].

Uniformnost hash funkcije provjerava se *chi-squared testom* u kojem se stvarna raspodjela ulaznih vrijednosti po izlaznim vrijednostima uspoređuje s željenom/očekivanom tj. uniformnom.

Pri uvrštavanju n ulaznih vrijednosti u hash funkciju koja mapira hash tablicu s m hash vrijednosti , idealna uniformnost bi bila postignuta ako se u svaki "bucket[16]" hash tablice preslika točno $E_i = \frac{n}{m}$[17] ulaznih vrijednosti.

[15] npr. kod pohrane u Hash tablicu ne želimo da sve funkcije budu spremljene pod isti indeks - tada zapravo imamo pohranu u vezanu listu
[16] u literaturi čest naziv za mjesto - poziciju unutar hash tablice
[17] n - broj unesenih elemenata, m- broj mogućih izlaznih vrijednosti

$$X = \frac{\sum_{j=0}^{m-1} b_j \cdot (b_j + 1)}{(n/m)(n + 2m - 1)}$$

- prilagođena i lako primjenjiva “verzija chi-square” testa, za uniformnu hash funkciju
  $1.05 \geq X \geq 0.95$

Iz uniformnosti proizlazi i manja vjerojatnost “sudara” čime se smanjuje vjerojatnost false positive-a u primjeni hash funkcije u Bloomovu filteru.

Jedno od svojstava hash funkcija koje implicira da je funkcija uniformna je “kriterij lavine”, tj. strict avalanche criterion. Kriterij lavine zahtjeva da se sa promjenom vrijednosti bilo kojeg bita ulazne kodne riječi svaki bit izlazne kodne riječi promijeni sa 50%-om vjerojatnošću. Ovo svojstvo izrazito je bitno ako radimo sa ulazima s malom varijacijom vrijednosti, a želimo uniformno hashiranje tj. da se izlazi razlikuju.

JEDNOZNAČNOST

Jednoznačnost/determiniranost hash funkcije svojstvo je na kojem se temelji rad Bloomovog filtera.
Hash funkcija je jednoznačna akko za određenu ulaznu vrijednost uvijek daje istu izlaznu vrijednost.

Stoga su za realizaciju Bloomovog filtera beskorisne funkcije koje koriste promjenjive varijable poput “Time Of Day” varijabli ili pseudo-random varijabli.

Neke funkcije kao ulaz ne primaju isključivo vrijednost koju hashiramo, već izlaz ovisi i o dodatnim ulaznim vrijednostima - seedovima. Jednu od takvih funkcija Murmur3 obradit ćemo u kasnijem dijelu teksta. Takve funkcije mogu biti jednoznačne pod uvjetom da su seedovi zadržavaju konstante vrijednosti pri svakoj upotrebi hash funkcije.

BRZINA

Kod Bloomova filtera brzina provjere je upravo smisao uporabe te strukture, kako bi Bloomov filter bio brz potrebna je brza hash funkcija. Brzina hash funkcije ovisi o broju i vrsti instrukcija kojima se hashiranje izvodi.
Poželjne su stoga funkcije s što manjim brojem instrukcija.

Najbrže od instrukcija su bitovne metode, zatim slijede metode koje se temelje na množenju, a najsporije su i najsloženije metode koje se temelje na dijeljenju.

Kriptografske hash funkcije

Kriptografske hash funkcije su posebna skupina hash funkcija koja se koristi u zaštiti podataka koji se pohranjuju. Zaštita se vrši tako da se podaci koji se pohranjuju, uvrštavaju se u hash funkciju, a dobiveni hashevi nisu "čitki vanjskom promatraču"[18] i kao takvi se pohranjuju.

Stoga su poželjna svojstva :

- •neinvertibilnost - vanjskom napadaču nije isplativo[19] generiranje izvornog podatka iz pohranjene hash vrijednosti
- •brzina - funkcija brzo računa hash vrijednost za bilo koju ulaznu vrijednost
- •jednoznačna - za istu ulaznu vrijednost svaki puta ispoljava istu hash vrijednost
- •otpornost na pronalazak praslike - za određenu vrijednost hasha h, napadaču je neisplativo pronaći vrijednost m koju hash funkcija preslikava u h
- •otpornost na sudare - napadaču nije isplativ pronalazak dviju ulaznih vrijednosti m1≠m2 takvih da hash funkcija i m1 i m2 preslikava u istu hash vrijednost tj. h(m1)=h(m2)

Kao kompromis za sigurnost, ove funkcije često odaju brzinu, npr. jedna od bržih kriptografskih funkcija Blake2b 32 puta je sporija od nekriptografske SeaHash funkcije.
Iz toga razloga za konstrukciju Bloomova filtera izbjegavaju se kriptografske hash funkcije jer je u tom slučaju prioritet brzina, a ne zaštita podataka.

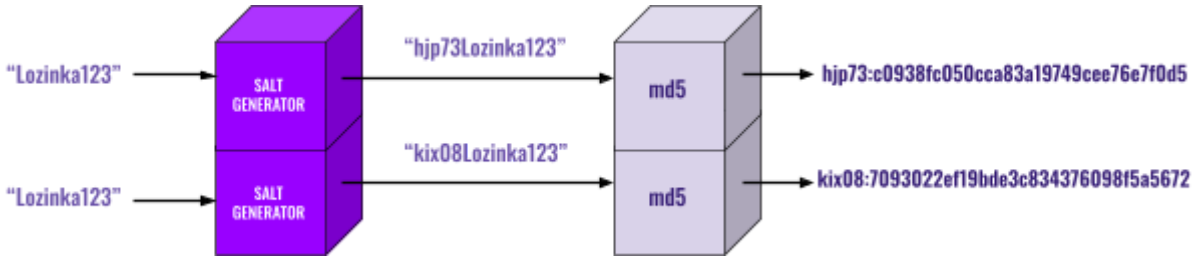
---

[18] nije moguće razumjeti značenje izvorno pohranjenog podatka iz hasha
[19] zbog nepostojanja egzaktne mjere, kao mjera za razinu zaštite neke kriptografske funkcije koristi se isplativost njene dekripcije

Salting

Salting je metoda koja se koristi kao dodatna mjera zaštite podataka pri kriptografskom hashiranju. Ulaznoj vrijednosti se dodaje nasumično generirana vrijednost - salt, koja se kasnije zajedno sa hashom trajno pohranjuje.
Salting je popularno korišten pri pohrani zaporki korisnika, kod prijava u neki sustav. Takav sustav onemogućava "brute force" napade na pohranjene podatke poput rainbow tablice[20].

Salting zapravo iskrivljuje jednoznačnost hash funkcije, tj. za istu ulaznu vrijednost u sustav neće biti pohranjena
ista hash vrijednost.



Kod kasnije prijave sustav dobavlja pohranjeni salt iz baze podataka, uvrštava unesenu lozinku i taj salt u hash funkciju te dopušta prijavu ako je generirani hash jednak onom pohranjenom kod izrade korisničkog računa.

[20] unaprijed izračunate hash vrijednosti za npr. milijardu najčešćih zaporki

Primjena

Bloomovi filteri imaju stvarno šarolike primjene, od marketinga, financijskog sektora i sigurnosti pa sve do korištenja za optimizaciju servera. Kako su primjene i načini korištenja jako široki te ih ne možemo sve zapisati i objasniti, to ćemo napraviti za neke zanimljive primjere, a linkovi na kojima se mogu naći detaljnija objašnjenja samih firmi se nalaze u literaturi ovog rada.

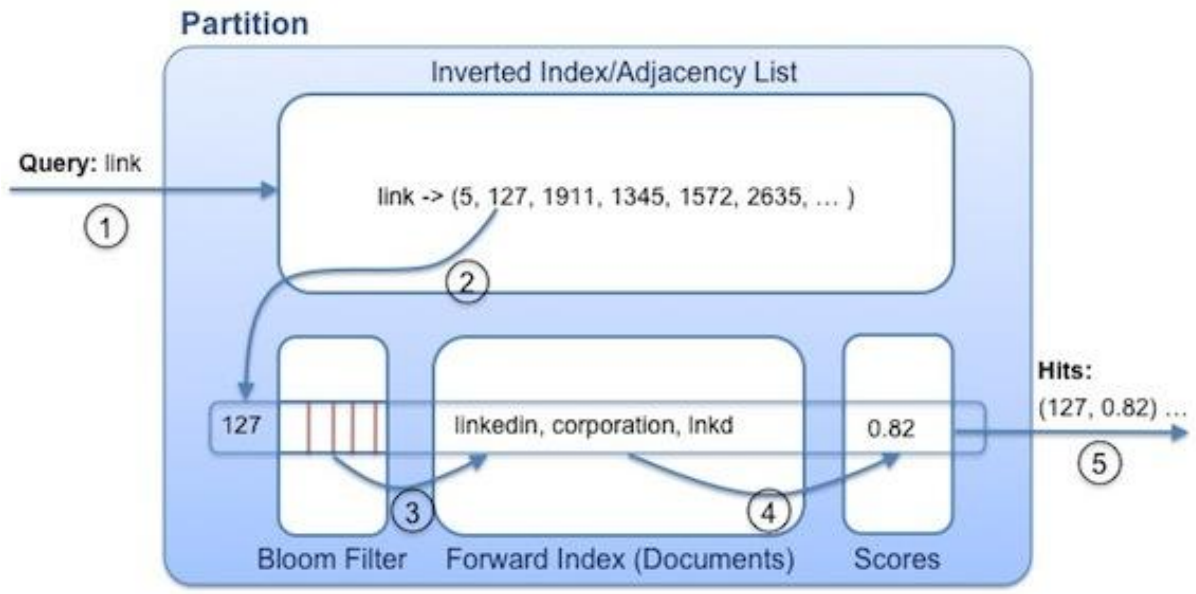
Otkrivanje malicioznih linkova - bitly kod skraćivanja linkova ne želi dopustiti da se skraćuju linkovi na neku od poddomena ili samu domenu koja je u njihovoj listi malicioznih stranica. Zato je bitly napravio svoj Bloomov filter koji nazivaju datablooms, a open-sourcali su ga da ga svatko može koristiti. Razvili su datablooms zato što im on omogućava da koriste brojač umjesto bita koji je aktivan ili neaktivan. Brojač omogućuje da se pri brisanju neke stranice ne briše cijelo mjesto već se smanjuje za 1, ali ostaje i dalje aktivno, čime ne dolazi do mogućnosti false negativa, ali cijela struktura postaje veća nego što je bila kada se koristi samo jedan bit, no to su ograničili koristeći samo 4 bita za brojač. Još jedna prednost databloomsa je što kako se skalira lista tih stranica dinamički se dodaju nove hash tablice tako da se ne povećaje vjerojatnost false positiva, već se dodaje nova tablica. Naravno, bitly nije jedina stranica koja koristi Bloomove filtere u tu svrhu, no najprominentnija je od viđenih.

Online oglašavanje - kod online oglašavanja javlja se problem nepotrebnog dupliciranja oglasa te postoji potreba da se prati tko je pogledao koji oglas. Tu Bloomovi filteri dolaze savršeno do izražaja, kod ovog tipa oglašavanja neka vjerojatnost ne postavljanja ili ponavljanja oglasa je jako dobar kompromis za značajnu uštedu memorije koju Bloomovi filteri nude. Agencija za oglašavanje Keep tako koristi Bloom filter za svaki oglas koji plasira na tržište gdje se korisnici dodaju u set nakon što su vidjeli taj oglas, a kada idući put dođe taj korisnik, nije moguće da opet pogleda isti oglas. No kako postoje različite metrike poput geografske lokacije korisnika koje isto ulaze u to koji će se oglas plasirati, nisu potrebni filteri jednake veličine za svaki oglas, ali i potreba skaliranja nekih setova, kao i bitly su napravili dinamičko skaliranje tablica, no na malo drugačiji način nego bitly. Tako umjesto 1.6TB RAMa koje bi agencija koristila da se setovi nisu dinamički realizirali, agencija je zapravo koristila samo 11GB RAMa. Različite izvedbe filtera se izrađuju ovisno o tome koje se metrike upotrebljavaju prilikom izrade tablice za istu svrhu. YouTube navodno koristi filtere na sličan način za prikaz preporučenih videa samo što on koristi račun kao tablicu i dodaje pogledana videa u set.

Otkrivanje prevara u bankarstvu - kako su u bankarstvu točne izvedbe alata za otkrivanje prevara povjerljive informacije ne možemo pisati o izvedbi Bloomovog filtera već o problemu koji rješava. Banke žele prepoznati i zabilježiti sve sumnjive transakcije s trgovcima s kojima korisnik nije prije poslovao. Kako banke imaju milijune korisnika i svaki korisnik posluje s desecima pa i stotinama subjekata, zapisivanje svega toga je jako zahtjevno na memoriju. Tu opet do izražaja dolaze Bloomovi filteri, jer uz vjerojatnost od 0.05% za false positive, ušteda u memoriji, koja je značajna kod projekata ovih veličina, je jedna prednost, dok je druga prednost to što se jako osjetljivi podaci ne zapisuju već samo dodaju u tablicu. IBM je za korisnike njihovog “IBM WebSphere eXtreme Scale” sustava napravio Java

implementaciju Bloomova filtera upravo da se slični sustavi, modificirani za potrebe banaka, mogu realizirati na njihovoj platformi, no i drugi scenariji koji koriste filter na sličan način.

Typeahead search - mnoge popularne društvene mreže koriste dinamične preporuke prilikom pisanja u tražilici te stranice, različite stranice koriste različite načine prikazivanja tih preporuka, neke traže najpopularnije korisnike s tim imenom dok druge traže povezanost s korisnikom, dok se većinom različitim formulama povezuje i jedno i drugo. Zato se koriste Bloomovi filteri da se provjeri je li ta preporuka koju aplikacija želi pokazati korisniku u setu i zato se koristi i korisnikov cache. Zašto Bloomovi filteri ovdje? Zato što su jako brzi, a kada pretražujemo nešto poput prijatelja od prijatelja i događaja na koji prijatelji idu, pojave se tisuće mogućih opcija, a pretražujemo u realnom vremenu i preporuke moraju biti jako brze. Bloom filteri pomažu ovdje zato što ako Bloomovi filteri odbiju nešto, aplikacija ne mora prefix matchat ostatak i time uštedi jako mnogo procesiranja za svaki search jer ne moramo pretraživati sve stranice i prijatelje koji su sam relevantni nego samo one koji počinju s tim slovom ili kompleksijom slova pa se onda sortira po relevantnosti. Ako ime ne počinje tim slovom Bloom filter odbacuje tu mogućnost, a ako Bloom filter da false positive on se ubacuje u listu kandidata no prefix matchanjem se odbacuje ta mogućnost pa stranica nikada neće dati netočnu preporuku. Naravno, ne pretražuje se dok korisnik piše već aplikacija čeka da korisnik prestane pisati, kod Facebooka je to vrijeme 17-100ms čime se opet štedi značajna količina procesiranja. Na taj način i Facebook, Twitter i LinkedIn daju preporuke prilikom pretraživanja. LinkedIn je izbacio Cleo, library koji koristi za realizaciju tih preporuka, čiji je graf rada ispod, na slobodno korištenje svima. Dok Facebook koristi čak dvije implementacije Bloomovih filtera za pretraživanje, s jednom hash funkcijom za prijatelje, a s dvije za stranice, događaje itd. koje pratite vi i prijatelji.



Otkrivanje duplih klikova na oglas - značajna količina oglasa koje vidimo na internetu su pay-per-click oglasi, što znači da vlasnik stranice dobije neku količinu novca tek nakon što korisnik klikne na taj oglas. Tu se javlja problem jer neki vlasnici pokušavaju multiplicirati taj klik čime bi zaradili više za samo jedan prikaz oglasa. Marketniške agencije zato razvijaju svoje algoritme koji služe da otkriju te prevare te isplate samo jednom za taj oglas. Ti algoritmi počivaju na korištenju Bloomovih filtera. Neki od korištenijih algoritama su GBF i TBF algoritmi. GBF algoritam izgrađen je na grupnim Bloom filterima koji mogu obrađivati klikove preko skočnih prozora s malim brojem potprozora, dok se TBF algoritam temelji na novoj strukturi podataka nazvanoj vremenski Bloom filter koji otkriva prevaru preko kliznih prozora i skočnih prozora s većim brojem potprozora.

Exactly-once processing - Google u svom Cloud Dataflowu treba provjeriti je li neki zapis duplikat, ali bez da pretražuje sve zapise kako bi optimizirao sustav. Većina zapisa neće biti duplikati pa su Bloomovi filteri idealno rješenje. Svaki worker ima Bloomov filter svakog identifiera kojeg je vidio, kada dođe novi id, provjeri u Bloomovom filteru postoji li taj id, ako ne postoji, worker je uštedio pretraživanje cijele memorije, a ako postoji, pretraživati će memoriju da potvrdi da postoji zbog vjerojatnosti za false positive, no kako je ta vjerojatnost prilično mala, ta struktura je idealna. Google koristi malo drugačiji pristup skaliranju od drugih kompanija o kojima smo pisali. Svakom zapisu pridružuje vrijeme, te umjesto stvaranja jednog velikog filtera, workeri stvaraju svako 10 minuta novi manji filter, pa kada zapis dođe, provjerava se filter aktivan u vremenu kada se tom zapisu pridružilo vrijeme. Slična problem pomoću Bloomovih filtera rješavaju Apache Cassandra i Couchbase, oni umjesto traženja elementa u stable storageu traže u Bloom filteru, ako dobiju pozitivan odgovor, provjeravaju postoji li na disku.

Spellcheck - Bloomovi filteri također imaju primjenu u softveru za spellchecking. Sve riječi nekog jezika se dodaju u jedan filter te se provjerava egzistencija kandidata u filteru. Sugestije se daju tako da se mijenja po pojedino slovo sve dok filter ne vrati pozitivan response. Slična primjena postoji i u odabiru lozinki gdje su sve najčešće lozinke dodane u filter koji javlja da je lozinka slaba ako se nalazi u filteru.

Ovo su bili neki od primjenjivijih primjera u kojima se koriste Bloomovi filteri i koji ne zahtjevaju neko dublje poznavanje polja (ili određenih pojmova u tom polju) u kojem se realizira za shvatiti primjenu. Neke još od primjena su u mrežnim komunikacijama za routing, onemogućavanje network loopova, IP traceback i multicast te za smanjivanje količine podataka koji prolazi kroz mrežu. Također postoje zanimljivi primjeri primjene u polju računalne sigurnosti za intrusion detection skeniranjem velikog broja paketa koji prolaze kroz mrežu, enkriptirano pretraživanje bez slanja onog što tražimo direktno preko mreže. Još jedna primjena se vidi u bazama podataka, poglavito distribuiranima, što možemo vidjeti u Oracleovim bazama 10 na više. Isto tako Googleov Guava Java library ima ugrađen Bloomov filter. Te primjer koji nas je zainteresirao za izbor Bloomovih filtera kao teme, a to je odabir korisničkih imena kod mnogih servisa kod kojih se registrirate, a kako se tu Bloomovi filteri koriste mislimo da je dosada prilično jasno.

Korištena literatura:

- https://www.eecs.harvard.edu/~michaelm/postscripts/rsa2008.pdf
- https://www.geeksforgeeks.org/what-are-hash-functions-and-how-to-choose-a-good-hash-function/
- https://www.cs.princeton.edu/courses/archive/spring04/cos226/lectures/hashing.4up.pdf
- Hash function - Wikipedia
- https://hbfs.wordpress.com/2015/10/13/testing-hash-functions-hash-functions-part-iii/
- https://www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation/
- https://llimllib.github.io/bloomfilter-tutorial/
- https://hackernoon.com/probabilistic-data-structures-bloom-filter-5374112a7832
- https://security.stackexchange.com/questions/11839/what-is-the-difference-between-a-hash-function-and-a-cryptographic-hash-function
- https://dadario.com.br/cryptographic-and-non-cryptographic-hash-functions/
- http://web.eecs.utk.edu/~jplank/plank/classes/cs494/494/notes/Bloom-Filter/index.html
- https://www.cs.princeton.edu/courses/archive/spr05/cos598E/bib/bloom_filters.pdf
- http://www.cs.utexas.edu/users/lam/396m/slides/Bloom_filters.pdf
- https://arxiv.org/pdf/1804.04777.pdf
- https://core.ac.uk/download/pdf/55607643.pdf
- https://word.bitly.com/post/28558800777/dablooms-an-open-source-scalable-counting
- https://paperswelove.org/2015/video/armon-dadgar-on-bloom-filters-and-hyperloglog/
- https://github.com/armon/bloomd
- https://nofluffjuststuff.com/blog/billy_newport/2010/01/bloom_filters_for_efficient_fraud_detection_with_ibm_websphere_extreme_scale
- https://engineering.linkedin.com/open-source/cleo-open-source-technology-behind-linkedins-typeahead-search
- https://www.facebook.com/Engineering/videos/432864835468/?v=432864835468
- https://ieeexplore.ieee.org/document/4595871
- https://cloud.google.com/blog/products/gcp/after-lambda-exactly-once-processing-in-cloud-dataflow-part-2-ensuring-low-latency ,
- https://docs.couchbase.com/server/4.5/architecture/bloom-filters.html
- http://cassandra.apache.org/doc/latest/operating/bloom_filters.html
- https://pdfs.semanticscholar.org/d899/05bdf1ff791bdddc7c471070f34f4da18844.pdf
- http://structureddata.org/2010/10/12/reading-parallel-execution-plans-with-bloom-pruning-and-composite-partitioning/