

Napredni algoritmi i strukture podataka

Streaming podataka, Count-min sketch, HyperLogLog, Lookup tabele



**Univerzitet u Novom Sadu
Fakultet Tehničkih Nauka**

Streaming podatak

- ▶ Izraz *streaming* koristi se za opisivanje neprekidnih tokova podataka koji se kontinuirano generiše $[A, B, C, \dots, \infty]$
- ▶ Ovaj pojam donosi konstantan niz podataka koji se mogu koristiti (obično) bez **prethodnog skladištenja** - bitna informacija!
- ▶ Podatke možemo da transformišemo, da ih skladištimo ili reagujemo **kako dolaze**
- ▶ Ovakve skupove podataka generišu razne vrste izvora, u različitim **formatima i obimu**
- ▶ To je dodatan problem prilikom rada sa ovakvim grupama problema

Count–min sketch - problem 1

Zaposlili ste se u Twitteru (jeee), vaš prvi zadatak je da napravite sistem za evidenciju hash tagova u objavama, da bi sledeći tim mogao da implementira bolji *trending* feature. Od vas se očekuje da napravite evidenciju frekfencije *hash tagova*, i pred vas su stavljeni sledeća ograničenja i zahtevi:

- ▶ Sistem mora da radi sa *streaming* podacima
- ▶ Sistem mora da koristi malo resursa
- ▶ Sistem treba da omogući laku paralelizaciju
- ▶ 100 % preciznost nije obavezna

Predlozi :)?

Count–min sketch - problem 2

Zaposlili ste se u Youtube-u (opaaa), vaš prvi zadatak je da napravite sistem za evidenciju pregleda video-a, da bi sledeći tim mogao da implementira bolji *recommender* sistem. Od vas se očekuje da napravite evidenciju frekfencije pregleda videa, i pred vas su stavljeni sledeća ograničenja i zahtevi:

- ▶ Sistem mora da radi sa *streaming* podacima
- ▶ Sistem treba da koristi malo resursa
- ▶ Sistem treba da omogući laku paralelizaciju
- ▶ 100 % preciznost nije obavezna

Predlozi :)?

Count-min sketch - Uvod

- ▶ *Count-min sketch* je probabilistička struktura podataka koja služi kao **tabela učestalosti dogadjaja** u *stream-u* podataka
- ▶ Ova struktura koristi *hash funkcije* za **preslikavanje dogadjaja na frekvencije**
- ▶ Za razliku od *hash* tabele koristi **dosta manje prostora**, na račun **precenjivanja nekih** dogadjaja zbog **kolizija**
- ▶ Jednom kreirana, struktura **ne raste**, ma šta radili sa njom – zgodna osobina
- ▶ Zbog ovih osobina, često se koristi u sistemima koji rade za izuzetno velikom količinom podataka
- ▶ Druga vrlo zgodna primena su strimovi podataka – nema kraja podacima :)

- ▶ Ova struktura koristi **k** hash funkcija, slično kao i Bloom Filter
- ▶ Count-min sketch predstavlja tabelu gde registrujemo učestalost dogadjaja
- ▶ Svaka *hash* funkcija h_i se koristi za **korespondentni red** u tabeli
- ▶ Tabela ima **m** kolona, a vrednosti **nećemo birati nasumično**
- ▶ Preciznost ove strukture **zavisi od toga koliko redova dodajemo**, tj. koliko **hash funkcija koristimo**
- ▶ Više redova veća preciznost, više redova veća struktura i potrošnja resursa
- ▶ Potrebno je naći **balans**

Inicijalizacija tabele

- ▶ Inicijalno svaka ćelija unutar Count-min sketch (CMS) tabele se postavlja na vrednost **0**
- ▶ Zbog daljih operacija, ovo će biti neutralni element
- ▶ Ako imamo CMS sa **k** redova i **m** kolona onda je proces inicijalizacije sledeći:
 - ▶ $\forall i \in \{0, 1, \dots, k\}$
 - ▶ $\forall j \in \{0, 1, \dots, m\}$
 - ▶ $CMS[i, j] = 0$
- ▶ Dakle prodjemo kroz tabelu, i u svaku ćeliju postavimo 0
- ▶ Može se izvesti **relativno brzo**

Count-min sketch - Dodavanje

Ako dobijemo element iz *stream-a* sa ključem **K**, postupak dodavanja je sledeći:

- ▶ Propustimo element **K** kroz **svaku hash funkciju**: $\forall h_i \in \{1, \dots, k\}$
- ▶ Dakle svaka *hash* funkcija h_i je **red** u tabeli
- ▶ Dobijemo vrednost kolone: $j = h_i(K) \% m$ – slično kao i kod Bloom Filter-a
- ▶ Na preseku reda i kolone povećamo vrednost za **1**: $CMS[i, j] += 1$
- ▶ Može se izvesti **relativno brzo**

Count-min sketch - Dobijanje vrednosti

Ako želimo da vidimo učestalost elementa **K** u tabeli, postupak je sledeći:

- ▶ Propustimo element **K** kroz **svaku hash funkciju**: $\forall h_i \in \{1, \dots, k\}$
- ▶ Dobijemo vrednost kolone: $j = h_i(K) \% m$ – slično kao i kod Bloom Filter-a
- ▶ Formiramo **niz** vrednosti sa odgovarajućih pozicija $R[i] = CMS[i, j]$, $i \in \{0, \dots, k\}$
- ▶ Uzmemmo minimum iz niza i to je **procena učestalosti** dogadjaja **K**
 $E(K) = \min(R[i]), i \in \{1, \dots, k\}$
- ▶ Može se izvesti **relativno brzo**
- ▶ Zašto/Kako su prethodne tri operacije relativno brze?

Primer, formiranje

- ▶ Prepostavimo da imamo tok podataka od samo 4 elementa $D = \{4, 4, 2, 3\}$;
- ▶ Prepostavimo da imamo CMS sa parametrima $m = 4$ i $k = 2$ (niz od četiri registra i dve heš funkcije);
- ▶ Prvi element iz strima (broj 4) propustimo kroz algoritam, dobijamo $j_1 = k_1(4) \bmod m$ i $j_2 = k_2(4) \bmod m$
- ▶ Recimo da smo dobili vrednost $j_1 = 3$, a vrednost $j_2 = 3$, dobijamo u tabelu
- ▶ Nadalje nastavimo proces za svaki broj iz strima

Primer, procena

- ▶ Zelena boja prvi i drugi, plava treći, zuta četvrti element
- ▶ Prepostavimo da je bilo i kolizija (pozicija CMS[k₁, 3] zuta, ne zelena!!)

	0	1	2	3
k ₁	0	1	0	3
k ₂	1	0	1	2

- ▶ Ako želimo da **procenimo** koliko puta se broj 4 pojavljuje u tabeli:
- ▶ Ponovimo dobijanje indeksa (j₁, j₂) kao i za dodavanje
- ▶ $x = \min(2, 3)$
- ▶ Vrednost 2 je **procenjena**
- ▶ Procenjena vrednost za broj 2 je?

Count-min sketch - Izbor parametara

- ▶ Parametre k i m nećemo nasumično birati
- ▶ Kao i kod Bloom Filtera možemo da se oslonimo na malo matematike
- ▶ Ako hoćemo da definišemo tabelu veličine $k \times m$ treba da izaberemo preciznost (ϵ) koju želimo da postignemo, kao i sigurnost sa kojom dolazimo do tačnosti (δ)
- ▶ Dobijamo $k = \lceil \ln \frac{1}{\delta} \rceil$ i $m = \lceil \frac{\epsilon}{\delta} \rceil$, gde je ϵ Ojlerov broj

ϵ	$1 - \delta$	m	k	$k \times m$
0.1	0.9	28	3	84
0.1	0.99	28	5	140
0.1	0.999	28	7	196
0.01	0.9	272	3	816
0.01	0.99	272	5	1360
0.01	0.999	272	7	1940
0.001	0.999	2719	7	19033

HyperLogLog - Problem 1

Zaposlili ste se u Facebook-u (you rocks), i od vas se traži da izračunate broj različitih korisnika koji su posetili Facebook u datoj nedelji, gde se svaka osoba prijavljuje više puta dnevno. Ovo rezultuje velikim skupom podataka sa mnogo duplikata. Od vas se zahteva da:

- ▶ Ne potrosite previse resursa
- ▶ 100 % tačan podatak nije obavezan
- ▶ Lako paralelizujemo proces
- ▶ Sistem treba da radi i sa *streaming* podacima

Predlozi :)?

HyperLogLog - Problem 2

Zaposlili ste se u Google-u (bravo majstori), i od vas se traži da izračunate broj različitih stvari koje su korisnici pretraživali svaki dan. Ovo rezultuje velikim skupom podataka sa mnogo duplikata. Od vas se zahteva da:

- ▶ Ne potrosite previse resursa
- ▶ 100 % tačan podatak nije obavezan
- ▶ Lako paralelizujemo proces
- ▶ Sistem treba da radi i sa *streaming* podacima

Predlozi :)?

HyperLogLog - Uvod

- ▶ HyperLogLog (HLL) je probabilistička struktura podataka koja se koristi za izračunavanje **kardinalnosti** velikih skupova podataka (broj različitih elemenata u skupu) – *Count-distinct problem*
- ▶ Kao i Bloom Filter i Count-min sketch, i on se oslanja na *hash funkcije*
- ▶ Za razliku od prethodne dve strukture, **on ne skladišti** hash funkcije
- ▶ HLL u se memoriji reprezentuje kao fiksna struktura koja **neće rasti** sa dodavanjem elemenata
- ▶ HLL rešava problem pronalaženja kardinalnosti masovnog skupa podataka koji koristi manje od **1,5 KB** memorije i sa procenom greške manjom od **2 %**

- ▶ Sam algoritam je relativno jednostavan, ali matematika i dokazi u pozadini nisu baš :/
- ▶ Kao i prethodna dva algoritma, danas se prilično intenzivno koristi u raznim aplikacijama sa velikim skupovima podataka
- ▶ Dosta se koristi kod *streaming* aplikacija
- ▶ Pogotovo je koristan u Big Data i Cloud aplikacijama gde su skopovi podataka jako veliki
- ▶ Zbog svojih osobina (kao i prethodne struture) mogu se čak koristiti i na sistemima sa ograničenim resursima, sa (skoro) identičnim performansama

Intuicija Flajolet i Martin

- ▶ Metrika Flajolet i Martin broji *nula* bitove na početku heširanih vrednosti
- ▶ Kod nasumičnih skupova podataka, sekvenca od k **uzastopnih** *nula* bitova će se pojaviti **jednom u svakih** 2^k **elemenata**, u proseku
- ▶ Potražimo sekvence, i zabeležimo najdužu sekvencu *nula* bitova da bismo procenili ukupan broj **jedinstvenih elemenata**
- ▶ Međutim, ovo još uvek nije sjajna procena
 - ▶ U najboljem slučaju može nam dati procenu broja elemenata stepena dvojke uz ogromnu varijansu
 - ▶ Sa pozitivne strane, da bismo zabeležili sekvencu vodećih nula bitova u 32 bita, potreban nam je broj od 5 bita

HyperLogLog - Ideja

- ▶ Ako imamo dovoljno veliku kolekciju brojeva fiksne veličine (npr. 32/64/128 bita) i pronadjemo broj koji ima maksimlanih k **uzastopnih nula bitova**, možemo biti **gotovo** sigurni da postoji **najmanje 2^k brojeva** u toj kolekciji
- ▶ HLL prvo primenjuje *hash* funkciju na sve vrednosti i predstavlja ih kao **cele brojeve iste veličine**
- ▶ Zatim ih pretvara u **binarne vrednosti** i procenjuje kardinalnost iz **heširane vrednosti, umesto** iz samih zapisa
- ▶ Izlaz *hash* funkcije je podeljen na dva dela:
 - ▶ Bakete na osnovu **vodećih** (*leading*) bitova
 - ▶ Vredosti najveći mogući broj **krajnjih** uzastopnih (*consecutive*) nula +1

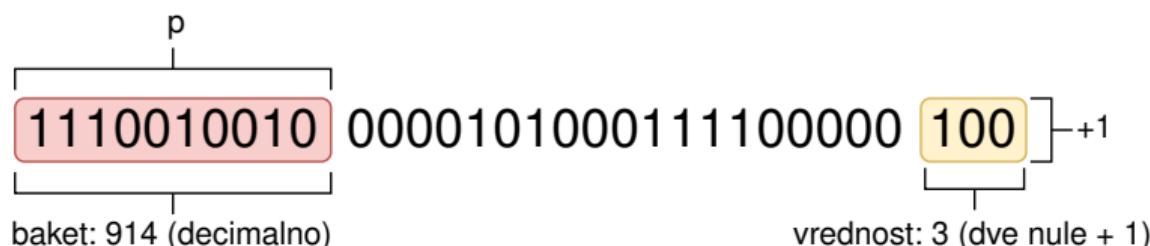
- ▶ Ako dobijemo više uzastopnih nula iz krajnjeg desnog bita za isti baket, ažuriraćemo taj baket.
- ▶ Oslanjamо se na nekoliko parametara:
 - ▶ **p** koliko vodećih bitova koristimo za baket
 - ▶ **m** veličina seta
- ▶ Prvo moramo da odredimo koliko vodećih bitova korisitmo za baket **p** – kolika je **preciznost** strukture
- ▶ Vrednost **p** je obično u intervalu [4, 16]
- ▶ Veća vrednost **p** smanjuje grešku u brojanju, koristeći više memorije
- ▶ Nakon toga treba da izračunamo koliki nam set **m** treba, koristeći formulu
$$m = 2^p$$

HyperLogLog - dodavanje

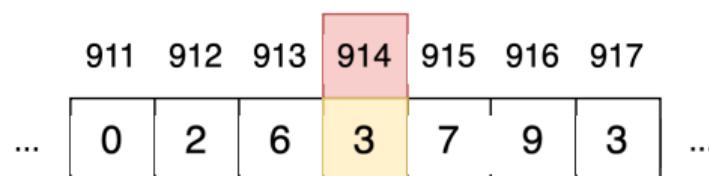
- ▶ Prepostavimo da imamo HLL definisan sa preciznošću 10 ($p = 10$)
- ▶ Kao rezultat toga, znamo da je veličina seta $m = 2^{10}$ (po formuli $m = 2^p$)
- ▶ Ako korisnik hoće da doda vrednost **X** u HLL, vrednost treba da heširmao i pretvorimo u binarni oblik
- ▶ Recimo da dobijamo vrednost: 11100100100000101000111100000100
- ▶ Iz dobijene binarne vrednosti zaključujemo da je vrednost bucket-a gde ćemo upisati vrednost *1110010010* tj. **914** – preciznost $p = 10$
- ▶ Vrednost koju upisujemo dobijamo tako što prebrojimo broj **uzastopnih** nula sa kraja niza i na to dodamo **+1**
- ▶ Sada znamo gde upisujemo **bucket**: **914**, i koja je **vrednost**: **3** koja se upisuje

HyperLogLog - dodavanje slikovito

- ▶ $X | f_{\text{binary}}(h(X)) \rightarrow 11100100100000101000111100000100$



- ▶ if $3 > \text{HLL}[914]$
 $\text{HLL}[914] = 3$



HyperLogLog – kardinalnost

- ▶ $\text{CARDINALITY}_{\text{HLL}} = \text{constant} * \frac{m^2}{\sum_{j=1}^m 2^{-R_j}}$
- ▶ R_j označavaju baket koji sadrži najduži niz uzastopnih nula
- ▶ Izraz $\sum_{j=1}^m 2^{-R_j}$ se naziva *harmonijska sredina*, smanjenje greške bez povećanja potrebne memorije (Za dokaz konsultovati originalan rad)
- ▶ Durand-Flajolet je izveo constant da ispravi pristrasnost ka većim procenama (LogLog).

m	const
2^4	0.673
2^5	0.697
2^6	0.709
$\geq 2^7$	$\frac{0.7213*m}{m+1.079}$

Opažanje

- ▶ Ako pogledamo prethodna dve strukture, vidimo da postoje **tabele**, gde **unapred** imam definisane **[podrazumevane]** vrednosti...
- ▶ Da li ih je potrebno **uvek** računati, ako već **znamo vrednosti** za **česte** kombinacije?
- ▶ Kako računanje možemo **izbeći**, i kako možemo **poboljšati** naše performanse (npr. uređaji sa ograničenim resursima)?

Ideje :)?

Lookup tabele – ideja

- ▶ Ideja iza Lookup tabela (LUT) je relativno jednostavna
- ▶ To je je niz ili tabela koja zamenjuje računanje određenih parametara jednostavnijom operacijom **indeksiranja**
- ▶ Ideja je da, **zakucamo** u niz ili tabelu **unapred poznate/podrazumevane vrednosti** koje se često koriste
- ▶ Na taj način ne moramo **stalno da ih računamo**
- ▶ Možemo dodatno *ubrzati* program ili vreme potrebno za instanciranje komponente žrtvovanjem **malo prostora**
- ▶ *"Svaki program je balans izmedju složenosti algoritma i složenosti strukture podataka"* (prof. Branko Perišić)

Prednosti LUT-a

- ▶ **Direktno adresiranje** vrednosti koristeći indeks ili nekakav ključ – brzo
- ▶ Iako jednostavna, ova ideja se pokazuje **vrlo** korisna u raznim problemima
- ▶ Izbor boja iz nekakve matrice, predefinisani parametri za Count-min sketch, HyperLogLog itd.
- ▶ Dodatno nam omogućva da **odredjene** strukture, algoritme ili proračune možemo da vršimo i na uređajima sa **ograničenom** količinom resursa
- ▶ Zbog svoje jednostavnosti, mogu se često spustiti i u sam *hardware* čine dodatno možemo ubrzati izvršavanje
- ▶ "*Nema tog hardvera kog mi softveraši ne možemo da zagušimo*" (prof. Dušan Malbaški)

Count-min sketch - Dodatni materijali

- ▶ Count-Min Sketch
- ▶ An improved data stream summary: the count-min sketch and its applications
- ▶ An Improved Data Stream Summary: The Count-Min Sketch and its Applications
- ▶ Algorithms and Data Structures for Massive Datasets
- ▶ Live example
- ▶ Probabilistic Data Structures and Algorithms for Big Data Applications

HyperLogLog - Dodatni materijali

- ▶ HyperLogLog Paper
- ▶ HyperLogLog playground
- ▶ Facebook engineering HyperLogLog
- ▶ Algorithms and Data Structures for Massive Datasets
- ▶ Probabilistic Data Structures and Algorithms for Big Data Applications

Pročitati za narednu sedmicu

- ▶ Skip list
- ▶ Similarity Estimation Techniques from Rounding Algorithms
- ▶ Detecting Near-Duplicates for Web Crawling

Važna napomena

Formule za Bloom filter, Count-min sketch HyperLogLog ne trebate da učite napamet!!!

Nemojte to sebi raditi!

Zanimljivo

Bioinformatika, medicina i probabilističke strukture :)

To Petabytes and beyond: recent advances in probabilistic and signal processing algorithms and their application to metagenomics