Bloomove filtre, Podielové filtre

©kuko 10.4.2018

1 Filtrovanie

- problém:
  - chceme reprezentovať množinu S s operáciami insert a isMember
  - chceme minimalizovať použitú pamäť
  - stačí nám, ak množinu reprezentujeme približne (DŠ môže dať s malo pravdepodobnosťou zlú odpoveď)
- motivácia:
  - máme databázu, pričom vyhľadávanie je drahé – napr. dáta sú uložené na disku alebo na inom počítači (možno nevieme presne na ktorom a potrebujeme sa spýtať viacerých)
  - nemalá časť dotazov je na dáta, ktoré databáza neobsahuje
  - chceli by sme odfiltrovať väčšinu týchto dotazov, ktoré zlyhajú
  - je OK, ak sa pomýlime a dotaz vykonáme aj tak, ale nesmieme odfiltrovať prvok, ktorý sa v DB nachádza
- problém množiny sa dá riešiť exaktne pomocou vyvážených stromov alebo hešovania
- na tejto prednáške nás budú zaujímať aproximačné riešenia: prvkov je veľa, pamäte málo, vieme reprezentovať množinu s oveľa menšou pamäťou, pričom nám stačí približné riešenie?
- povoľujeme „falošne pozitívne výsledky", t.j.
  - ak algoritmus povie, že $x \notin S$, tak zaručene $x \notin S$
  - ak algoritmus povie, že $x \in S$, tak možno $x \in S$, ale s malou pravdepodobnosťou sa môže mýliť a $x \notin S$
- aplikácie:
  - Akamai cacheuje iba web objekty, ktoré boli vyžiadané aspoň 2× – zistili, že 3/4 objektov, ktoré cachovali, boli vyžiadané len raz, takže sa cacheovali zbytočne; nové riešenie: pri prvom prístupe ulož do filtra, pri ďalších, ak už filter objekt obsahuje, ulož do cache
  - databázy ako BigTable, Postgresql, HBase, Cassandra – odfiltrujú dotazy na neexistujúce riadky/stĺpce
  - Google Chrome – kedysi ukladal filter so škodlivými URL lokálne
  - Bitcoin – synchronizácia peňaženiek
  - SPIN model checker – množina stavov modelu
  - Medium
  - ...

2 Bloomov filter

- $S = \{x_1, \dots, x_n\}$
- DŠ použije m bitov
- k nezávislých hešovacích fn. $h_1, \dots, h_k : \mathcal{U} \to [m]$
- označme δ pravdepodobnosť falošne pozitívneho výsledku
- implementácia:
  - insert(x): nastav bity $B[h_i(x)] = 1$ ($\forall i$)
  - isMember(y): skontroluj, či $\forall i : B[h_i(y)] = 1$

- predpokladajme úplne náhodné h_i
- aká je $\Pr[B_i = 0]$? $\Pr[B_i = 1]$? $\Pr[FP] = \delta$?
- pre dané m, n, aké k máme zvoliť pre min δ?
- $p = \Pr[B_i = 0] = (1 - 1/m)^{kn} = \left((1 - 1/m)^m\right)^{kn/m} \approx e^{-kn/m}$
- $\delta = \Pr[FP] = (1 - p)^k = \exp(k \ln(1 - p))$

- ak fixneme m/n=počet bitov na jeden prvok (t.j. ak sa rozhodneme, koľko pamäte chceme použiť), potom δ sme vyjadrili ako funkciu k, t.j. vieme dopočítať, aké k máme zvoliť tak, aby pp. chyby δ bola čo najmenšia
  - jedna možnosť je zderivovať $\delta = \exp(k \ln(1 - e^{-kn/m}))$ podľa k a nájsť minimum
  - druhá možnosť je všimnúť si, že ak dosadíme $p = e^{-kn/m}$, t.j. $k = -\frac{m}{n} \ln p$ do vzorca pre δ, dostaneme $\delta = \exp(-\frac{m}{n} \cdot \ln p \cdot \ln(1 - p))$
  - toto je symetrická funkcia vzhľadom na p, ktorá nadobúda minimum pre $p = 1/2$
- inými slovami, pri optimálnom k je v BF každý bit 0 alebo 1 s pp. 50%
- keďže $p = 1/2 = e^{-kn/m}$, dostávame optimálne $k = \ln 2 \cdot (m/n) \approx 0.693(m/n)$
- pre takto zvolené k je $\delta = 1/2^k \approx 0.619^{m/n}$, resp. $m/n \approx 1.445 \lg(1/\delta)$
- napr. ak chceme $\delta = 10\%$, zvolíme $m/n \approx 4.8 - 5$bitov na jeden prvok a $k = 3$ alebo 4
- ak chceme $\delta = 1\%$, zvolíme $m/n \approx 9.6$, $k = 6$ alebo 7
- všimnite si, že m/n nezávisí od veľkosti univerza (je jedno, či do BF ukladáme 64-bitové čísla alebo 1kB-ové stringy)
- upozornenie: pri analýze sme používali úplne náhodné hešovacie funkcie a pre jednoduchosť sme brali k, m, n ako reálne čísla; samozrejme, $k, m, n \in \mathbb{N}$; navyše sme použili $(1-1/x)^x \approx 1/e$; presnejšou analýzou sa dá dokázať $\delta \leq (1 - e^{-k(n+0.5)/(m-1)})^k$, takže náš odhad nie je úplne mimo
- dá sa dokázať dolný odhad, že ak chceme pp. chyby δ, každá DŠ potrebuje aspoň $m/n \geq \lg(1/\delta)$ bitov na jeden prvok; BF potrebuje $\approx 1.445\times$viac

- ďalšie operácie:
  - ak máme 2 BF s rovnakými parametrami m, p, reprezentujúce množiny S_1, S_2, vieme spočítať BF pre $S_1 \cup S_2$ a $S_1 \cap S_2$
  - dá sa odhadnúť počet prvkov v BF ($|S|$): $n^{\cdot} = m/k \cdot \ln(m/\#_0)$, kde $\#_0$ je počet nulových bitov

- nevýhody:
  - maximálne n treba poznať dopredu (BF nevieme dynamicky zväčšovať)
  - BF nepodporuje operáciu delete (ale dá sa upraviť za cenu väčšej pamäte: tzv. Počítajúci BF – namiesto každého bitu máme počítadlo; 4-bitové stačí pre väčšinu aplikácií)
  - BF nie je cache-friendly – 1 operácia = k prístupov na náhodné miesto v pamäti

3 Podielové filtre

- myšlienka: $h : \mathcal{U} \to [2^p]$, budeme *exaktne* reprezentovať množinu $h(S) = \{h(x_1), h(x_2), \ldots, h(x_n)\}$
- tzn. jediná chyba, ktorá môže vzniknúť je, keď zobrazíme S na $h(S)$ a pre niektoré prvky dostaneme kolízie, avšak množinu $h(S)$ už reprezentujeme presne
- pre pp. chyby δ stačí zvoliť $p = \lg(n/\delta)$
- ako reprezentovať $h(S)$?
- a) zoznam $[h(x_1), \ldots, h(x_n)]$ – p bitov/prvok, $O(n)$ čas
- b) hešovacia tabuľka, kolízie riešime zreťazením – aká veľkosť? čo do nej uložíme?
- idea: ak máme n prvkov, stačí nám tabuľka veľkosti $\Theta(n)$; z celého p-bitového hešu teda stačí zobrať prvých q bitov a hešovať do tabuľky veľkosti $2^q = \Theta(n)$
- idea 2: do heš-tabuľky netreba uložiť celý heš – stačí zvyšných $r = p - q$ bitov (prvých q bitov je jasných z pozície)
- teda: $h(x)$ – p bitov; horných q bitov – „podiel“ označuje pozícia v tabuľke, dolných r bitov – „zvyšok“ uložíme
- b') takto upravená hešovacia tabuľka so zreťazením stále zaberá dosť miesta – potrebujeme aspoň $\log n$ bitov na smerníky

- c) hešovacia tabuľka s lineárnym sondovaním? cache friendly, stačí q bitov/prvok

- problém: ako zistíme, kam sa prvok zahešoval pôvodne? (ale bol odsunutý, lebo dané miesto bolo obsadené)
- invariant: prvky s rovnakým podielom budú vždy tvoriť jeden súvislý úsek – jeden beh (na rozdiel od lineárneho sondovania, kde prvok pridáme vždy na koniec klastra a prvky zahešované na rôzne miesta môžu byť pomiešané)
- presnejšie: ak je lokácia x ¡ lokácia y, potom zvyšok x bude uložený pred zvyškom y a ak je zvyšok x uložený na pozícií p, tak medzi „domovskou“ pozíciou x a p je všetko obsadené
- pre každé domovské políčko si potrebujeme nejak zapísať kde začína príslúchajúci beh a kde končí
- táto informácia sa dá zakódovať úsporne: stačí pre každé políčko 1 bit: či sa na dané políčko zahešoval nejaký prvok a 1 bit: či na tejto pozícií končí nejaký beh
- tzn. pre každý beh vieme, kde by začínal (keby nebol vytesnený inými behmi) a kde naozaj končí
- isMember(x):
  - nech i je podiel a v zvyšok $h(x)$
  - najskôr sa pozrieme, či sa na domovské i-te políčko niečo zahešovalo – ak nie, odpoveď je nie
  - ak áno, zistíme, kde končí zodpovedajúci beh (pozícia ℓ)
  - prehľadáme pozície od ℓ smerom naspäť ku i, končíme ak nájdeme hodnotu v (odpoveď je áno), alebo ak narazíme na koniec predchádzajúceho behu (odpoveď je nie), alebo ak prejdeme až za domovské políčko i)
- insert(x):
  - nech i je podiel a v zvyšok $h(x)$
  - ak i-te políčko nie je obsadené, vložíme tam v
  - v opačnom prípade nájdeme koniec behu pre i (resp. koniec predchádzajúceho behu, ak sa zatiaľ nič nezahešovalo do i) a posunieme všetky prvky (behy) za tým o 1 doprava
  - v vložíme na uvoľnené miesto
  - upravíme metadáta o koncoch behov a nastavíme bit, že na i-te políčko sa niečo zahešovalo

- ostáva vyriešiť, ako pre dané políčko nájsť zodpovedajúci koniec behu
- c_1) ak si neuložíme žiadne iné dáta, dostávame riešenie s $r+2$ bitmi/prvok, ale hľadanie konca behu trvá $O(n)$
- c_2) ak si predpočítame koniec behu pre každé políčko, stačí $O(1)$ čas, ale potrebujeme $r + 10$ bitov/prvok (netreba absolútnu pozíciu, stačí relatívnu vzhľadom na i – čo bude malé číslo, 1 bajt)
- ak by sme vedeli podporovať operácie rank&select, stačí zavolať rank a zistiť, koľko jednotiek je v bitvektore s domovskými políčkami (koľkatý beh hľadáme) a následne použiť select a nájsť príslušný koniec behu v druhom bitvektore
- rank&select je klasická úloha, ktorá má úsporné riešenie v konšt. čase, avšak nie je veľmi praktické
- c_3) praktické riešenie: ako c_2, ale koniec behu si nepredpočítame pre každé políčko, ale pre každé 64-té
- úlohu rank&select na bitvektore dĺžky 64 vieme riešiť nielen v $O(1)$ čase (lebo vstup má dĺžku $O(1)$), ale v skutočnosti aj veľmi rýchlo pomocou inštrukcií, ktoré dnešné procesory podporujú:
  - rank(v,i)=počet 1 od 0 po i-tu pozíciu=PopCount(v & (2^i-1)) (číslo 2^i-1 je v binárnom zápise i jednotiek, pomocou ANDu všetky ostatné pozície vynulujeme a PopCount je inštrukcia, ktorá vráti počet 1 v binárnom zápise)
  - select(v,i)=pozícia i-tej jednotky=TZCnt(PDep$(2^i,v)$) (inštrukcia TZCnt vráti počet núl na konci (po prvý 1 bit), PDep(a,b) umiestni bity čísla a na pozície jednotiek v b, t.j. číslo PDep$(2^i,v)$ bude mať iba jednu jednotku na pozícií i-tej jednotky vo v)
- v našom konečnom riešení teda zlomok odpovedí predpočítame a zvyšné hodnoty dopočítame, keď treba
- potrebujeme 8 bitov na 64 prvkov, t.j. +0.125bitu na prvok – celkovo $r + 2.125$ bitu

- navyše časová zložitosť je rovnaká ako klasické hešovanie s lineárnym sondovaním, t.j. očakávane $O(1)$
- ďalšie vylepšenie – layout: dáta rozdelíme na bloky a v rámci jedného bloku uložíme metadáta aj políčka heš-tabuľky pre 64 prvkov – takto namiesto prístupov na 4 rôzne lokácie v pamäti stačí načítať s vysokou pravdepodobnosťou 1 alebo pár susedných blokov, čo je dobré kvôli cachovaniu

- ďalšie operácie:
  - všimnite si, že z daného podielového filtra vieme zrekonštruovať celé $h(S)$
  - z toho napr. vyplýva, že vieme spočítať prienik alebo zjednotenie dvoch filtrov
  - veľkosť heš-tabuľky vieme dynamicky meniť s rastúcim n: ak sa tabuľka preplní, prerozdelíme p: q zväčšíme o 1 a r zmenšíme o 1; na-alokujeme filter dvonásobnej veľkosti a všetky prvky doň vložíme
  - tzn., že môžeme začať s $q = 0$, $r = p$: heš-tabuľka bude obsahovať len jednu hodnotu, celé $h(x)$; potom sa tabuľka zdvojnásobí, bude mať 2 hodnoty, posledných $p - 1$ bitov

4 Počítajúce podielové filtre

- vedeli by sme podporovať delete?
- áno, namiesto množiny $h(S)$ budeme reprezentovať multimnožinu (vrátane duplikácií)
- takéto riešenie však nie je efektívne; ak sa hodnota z vyskytuje v $h(S)$ 7×, namiesto siedmich kópií by bolo dobre poznamenať si ich počet
- je viacero možností ako reprezentovať počet kópií – my by sme chceli, aby (za predpokladu, že väčšina prvkov sa v tabuľke vyskytuje iba raz) ak sú všetky prvky (heše) rôzne, DŠ nezaberala viac miesta ako obyčajný podielový filter
- spôsob, ktorý zvolili autori: počet kópií budeme ukladať priamo v heš-tabuľke (s r-bitovými hodnotami)
- problém: ako budeme vedieť, či je daná hodnota zvyšok alebo počet kópií?
- v rámci jedného behu uložíme zvyšky utriedené od najmenšieho po najväčší
- ak v rámci jedného behu po hodnote nasleduje menšie číslo, vieme, že znamená počet kópií a nie hodnotu zvyšku
- presnejšie: ak x je hodnota zvyšku, ktorú chceme uložiť a c je počet kópií x:
- ak $c = 1$, uložíme len x
- ak $c = 2$, uložíme 2 kópie: x, x
- ak $c > 2$, uložíme $x, \langle\text{zakódovaný počet kópií } c - 2\rangle, x$; tzn. hodnotu x použijeme ako ukončovací znak; kód pre $c-2$ musí začínať cifrou $< x$ (ak nezačína, pripíšeme pred číslo ešte 0); $c-2$ zapíšeme v sústave so základom $2^r - 2$: políčka heš-tabuľky sú r-bitové, takže každá cifra má 2^r možností okrem dvoch špeciálnych hodnôt: 0 a samotného x
- špeciálny prípad je $x = 0$, pre ktorú neexistuje menšie číslo – pre $c = 3$ uložíme 3 kópie: $0, 0, 0$ a pre $c > 3$ použijeme dve nuly ako ukončovací znak, t.j. $0, \langle\text{zakódovaný počet kópií } c - 3\rangle, 0, 0$ (číslo $c - 3$ zapíšeme v $(2^r - 1)$-tkovej sústave, pričom nepoužívame cifru 0)
- dá sa ukázať, že ak do takejto DŠ s kapacitou n a chybovosťou δ vložíme M hodnôt, pričom z týchto M je k rôznych (začneme s heš-tabuľkou veľkosti 1 a zdvojnásobíme ju vždy, keď sa naplní na $\geq 95\%$), tak DŠ bude zaberať $O(k \log(nM/\delta k^2))$ bitov.
- špeciálne ak $k = 1$, tzn. M-krát vložíme ten istý prvok, pamäť bude $O(\log(n/\delta) + \log M)$ (čo je pamäť, ktorú zaberá 1 heš + 1 počítadlo)
- ak sú všetky prvky rôzne, $k = M$, zložitosť je $O(M \cdot (\log(n/\delta) - \log M))$ (t.j. $O(M)$ políčok heš-tabuľky, každé má $r = O(\log n/\delta - \log M)$ bitov)