Napredni algoritmi i strukture podataka

LRU cache, Read Path, Range/Prefix Scan

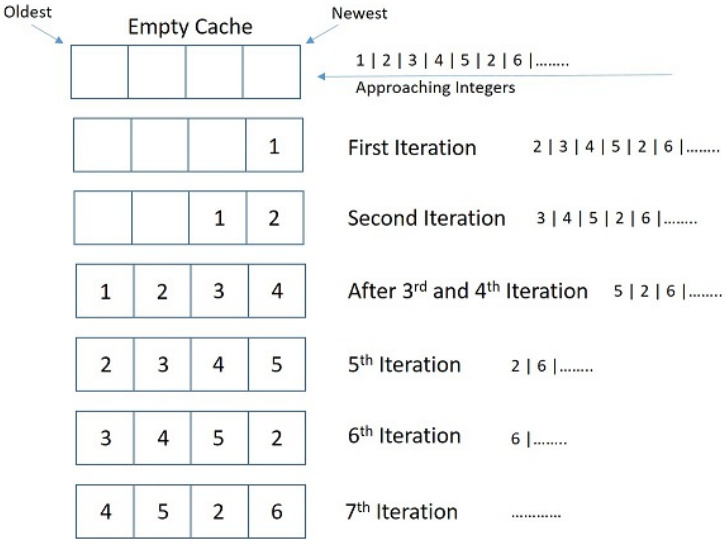
Univerzitet u Novom Sadu
Fakultet Tehničkih Nauka

Cache — Zahtevi

- Cache sistemi obično imaju zahteve koje treba da ispune, i neki od njih su:
  - Fiksna veličina: Cache memorija treba da ima neke granice, tj. da ograniči upotrebu memorije
  - Brz pristup: Operacija dodavanja elementa u cache i traženja treba da bude brza, poželjno $\mathcal{O}(1)$ vreme
  - Zamena unosa: U slučaju da je dostignuto ograničenje memorije, cache treba da ima efikasan algoritam za izbacivanje unosa kada je memorija puna
- Dodavanje ograniučenja je obično eksternalizovano u nekakvu konfiguaraciju — konfiguracioni fajl ili neki drugi oblik konfiguracije

LRU

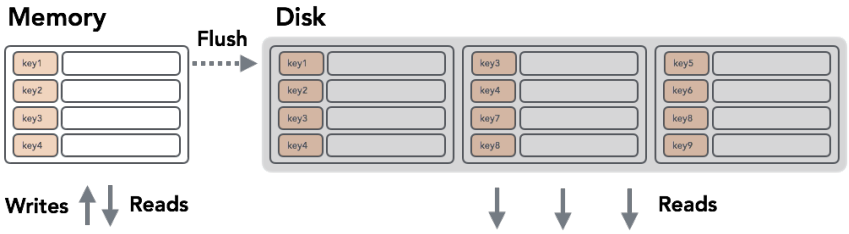
- Najdavnije korišćen (LRU) keš organizuje elemente po redosledu korišćenja
- Omogućavajući da brzo identifikujemo koja stavka nije korišćena najduže vreme
- Zamislite stalak za odeću, gde je odeća uvek okačena na jednoj strani
- Da bi pronašli predmet koji smo najmanje koristili, pogledate predmet na drugom kraju stalka
- Ukratko, to je politika izbacivanja iz keša
- Kada se naš keš popuni i doda se novi element, uklanjamo najdavnije korišćenu stavku iz keša



(Opengenus, Implement Least Recently Used (LRU) Cache)

- Prednosti:
  - Super brzi pristupi: LRU keš čuva stavke po redosledu od nedavno korišćenih do najmanje korišćenih — oba mogu pristupiti u $\mathcal{O}(1)$ vremenu
  - Super brza ažuriranja: Svaki put kada se pristupi stavci, ažuriranje keša traje $\mathcal{O}(1)$ vremena
- Mane:
  - LRU keš koji čuva n stavki zahteva spregnutu listu dužine n i hash mapu koja sadrži n stavki — to je $\mathcal{O}(n)$ prostor, ali to su i dalje dve strukture podataka (za razliku od jedne)

Put čitanja podataka — Read Path



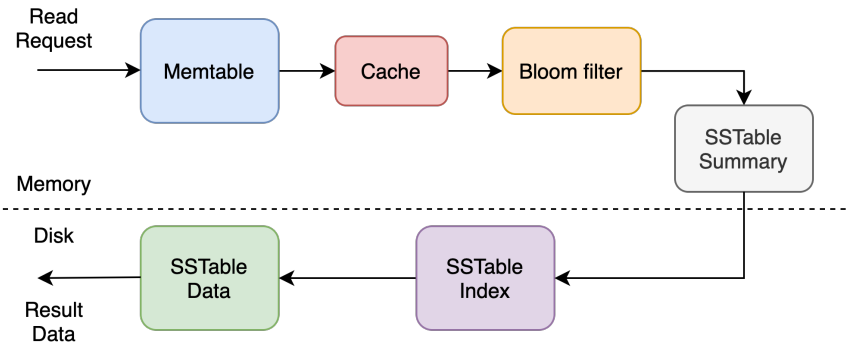
(Databases Internals)

- Proverimo **Memtable**, ako je podatak tu vratimo odgovor korisniku
- Ako ga nismo pronašli, treba da proverimo **cache** i ako tu pronađemo element, ažuriramo njegovu poziciju u kešu i vratimo odgovor korisniku
- Ako element nije u kešu, moramo učitati **Bloom filter** sa diska i videti da li je ključ **možda** tu
- Ako nije, odmah javimo korisniku da ključ nije prisutan — podatak nije sačuvan
- Ako je **možda** tu, učitati **summary** i videti da li je ključ u tom opsegu
- Ako nije, odmah javimo korisniku da ključ nije prisutan — podatak nije sačuvan
- Ako **jeste**, pronaći ga u **index** strukturi i uzeti **offset**
- Kada imamo **offset**, možemo da se pozicioniramo na **Data** deo i pročitamo podatak, dodajemo ga u keš i vratimo korisniku

- Svi pretohodno formirani elementi su nam potrebni kako bismo što pre stigli do podatka koji tražimo, **AKO** je on tu
- Toliki broj struktura nam treba zbog ovog **AKO**
- Pravimo kompenzaciju za nesigurnost Bloom Filter-a
- Što je pre moguće da dobijemo informaciju nazad
- Ili bar da nam mašinerija kaže: **ključ koji tražiš nije sigurno tu**
- Svi formirani elementi čine put čitanja podataka – **Read Path**

- I svi oni moraju da se povežu u sinhronu celinu, **AKO** želimo da dobijemo informaciju u razumenom vremenu
- Podatke u cache-u moramo da ažuriramo svaki put kada se korisnički podatak locira!
- Kada dobijemo podatak, pre nego što ga vratimo korisniku prvo ga zapišemo u cache
- Ova prosta strategija nam omogućava da kod sledeće pretrage **MOŽDA** ne moramo da idemo po disku
- Kada se uputi zahtev za brisanje nekog podatka, **AKO** je on u cache-u, možemo ga obrisati

Read path


Read
Request
Memtable
Cache
Bloom filter
SSTable
Summary
Memory
Disk
SSTable
Data
SSTable
Index
Result
Data

Prefix scan

- Ideja iza ovog upita nad sistemom je relativno jednostavna
- Prosledimo **prefix** po kom želimo da se uradi pretraga
- Prolazimo kroz naše podatke i pravimo odgovor od **svih** onih vrednosti čiji ključ **počinje** sa **prefixom** koji je korisnik prosledio
- Moramo voditi računa o par stvari:
  - Svi podaci se nalaze u jednoj SSTable-i – trivijalna situacija
  - Podaci se nalaze u n SSTable-a – komplikovanija situacija zahteva agregaciju
- Iskoristiti indeksne strukture za brže nalaženje podataka

Range scan

- Prosledimo upit u obliku $[k_1, k_2]$
- Prolazimo kroz naše podatke i pravimo odgovor od **svih** onih vrednosti čiji ključ **upada** u specificirani interval
- Moramo voditi računa o par stvari:
  - Svi podaci se nalaze u jednoj SSTable-i – trivijalna situacija
  - Podaci se nalaze u n SSTable-a – komplikovanija situacija zahteva agregaciju
- Iskoristiti indeksne strukture za brže nalaženje podataka

Paginacija

- Prethodna dva upita su vrlo slična
- Iako je način odabira podataka drugačiji, rezultat je sličan – niz vrednosti koje treba vratiti korsiniku
- U ovim situacijama moramo jako voditi računa o samom odgovoru – količina podataka može *potencijalno* biti velika!
- Vratiti korisniku sve podatke ako može, **AKO** ih nema previše
- U protivnom, odgovor treba **iseckati** tako da vraćamo deo po deo podataka – paginacija sadržaja ili straničenje

Kursori/iteratori

- Postoji nekoliko načina za rešavanje ovog problema
- Koristićemo mehanizam koji se dosta koristi u raznim softverskim sistemima – *cursor*
- Ideja iza ovog pristupa je takodje vrlo jednostavna – podelimo ukupan skup podataka D na *stranice* **fiksne** veličine n
- Prilikom vraćanja rezultat korisniku, sada vraćamo dve vrednosti:
  1. Stranicu podataka veličine n
  2. Pokazivače na prethodnu i narednu stranicu – izuzetak su prva i zadnja stranica :)
- Pokazivači na prethodnu i narednu stranicu nam trebaju zato što korisnik možda želi da se kreće kroz rezultate
- Iteratori u našem projektu su specijalan slučaj kursora, kada je veličina stranice jednaka 1

Zadaci

- Implementirati LRU algoritam
- Za listu koristiti *container/list* paket u go-u (dokumentacija)
- Implementirati **Read Path** strategiju, povezati sve prethodno kreirane elemente u jednu sinhronu celinu
- Dodati GET operaciju koja će inicirati **Read Path** strategiju
- Definisati algoritme pretrage za range i prefix scan operacije