

Napredni algoritmi i strukture podataka

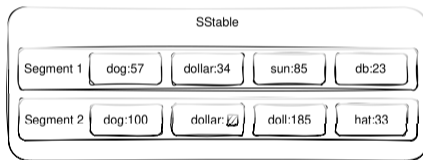
Put čitanja podataka, Keširanje, LRU, Prefix scan, Range scan, Paginacija



Univerzitet u Novom Sadu
Fakultet Tehničkih Nauka

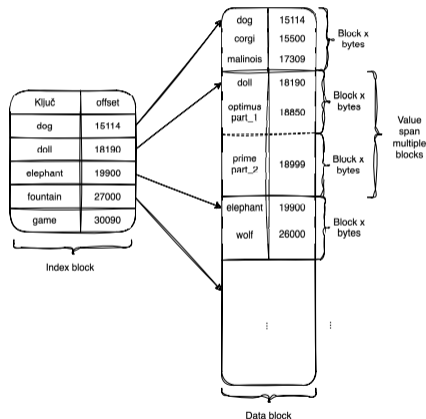
SSTable – blokovi podataka

- ▶ SSTable je niz parova **ključ:vrednost** koji su sortirani i zapisani na disk
- ▶ SSTable je nepromenljiva struktura (log based struktura)
- ▶ SSTable nastaje nakon što se Memtable popuni i zapiše na disk



SSTable – blokovi indeksa

- ▶ Ideja je da što pre stignemo do sadržaja
- ▶ Struktura je relativno jednostavna:
 1. **ključ** koji se nalazi u fajlu na disku (SStable)
 2. **offset** u fajlu, tj. pozicija u fajlu na disku
- ▶ Index bi bilo lepo sortirati (ne budite lenji, učinite sebi uslugu :))

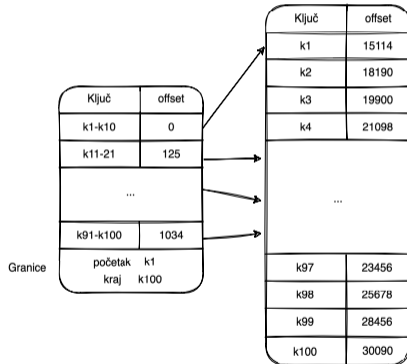


SSTable – Sampling indeksa

- ▶ Neki sistemi za skladištenje podataka (npr. Cassandra) koriste dodatan element za ubrzanje čitanja —

Summary

- ▶ Struktura je relativno jednostavna:
 1. **granice** povezanog index fajla — prvi i zadnji **ključ**
 2. **offset** ključa u index fajlu — poziciju u index fajlu



SSTable – Bloom Filter

- ▶ Bloom Filter-u trebamo da kažemo koliko elemenata se očekuje, ali to nije sada problem
- ▶ Ovaj podataka nam je unapred poznat, pošto tačno znamo koliko elemenata čuva Memtable
- ▶ SSTable isto zna tačno zna koliko će elemenata biti sačuvano
- ▶ Samim tim i za Bloom Filter imamo tu informaciju unapred poznatu!
- ▶ Stoga, sve elemente možemo formirati ispravno

SSTable – Metadata, blokovi podataka

- ▶ Dodatno formiramo *Metadata* fajl — Merkle stablo *Data* segmenta
- ▶ Nakon što je stablo formirano, stablo zapižemo u *Metadata* fajl
- ▶ Ovde je kraj pravljenja jenog SSTable-a
- ▶ Formiranje SSTable se obično odvija u pozadini, bez narušavanja rada sistema
- ▶ **Za vaš projekat ovo nije potrebno, može sve da se dešava i sinhrono**

SSTable – Struktura

```
<beginning_of_file>
[data block 1]
[data block 2]
...
[data block N]
[meta block 1]
...
[meta block K]
[metaindex block]
[index block]
[Footer]      (fixed size; starts at file_size - sizeof(Footer))
<end_of_file>
```

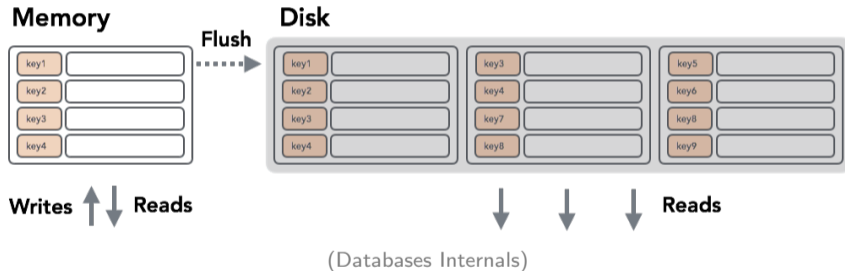
(LevelDB Format)

```
usertable-data-ic-1-CompressionInfo.db
usertable-data-ic-1-Data.db
usertable-data-ic-1-Filter.db
usertable-data-ic-1-Index.db
usertable-data-ic-1-Statistics.db
usertable-data-ic-1-Summary.db
usertable-data-ic-1-TOC.txt
```

(Cassandra format)

Put čitanja podataka — Read Path

Read path smo videli na nekom od prethodnih predavanja, ali jako malo



Pitanje 1

Ako pogledamo prethodnu sliku, vidimo da se svo čitanje odvija sa diska i iz SSTable-a. Zašto smo onda formirali onoliko elemenata prethodno predavanja (**HINT**: Nije da vas mučim :))...

ideje :)?

Put čitanja podataka

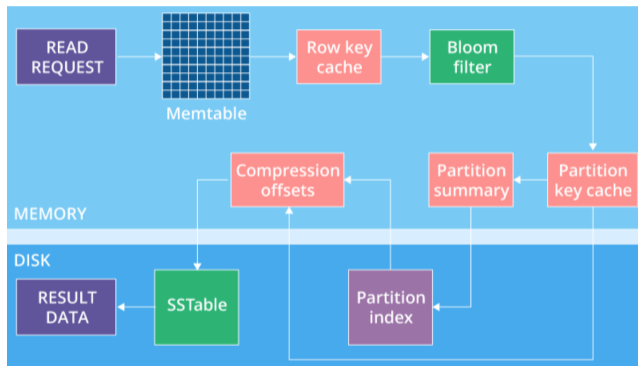
- ▶ Svi prethodno formirani elementi su nam potrebni, da bi što pre stigli do podatka koji tražimo, **AKO** je on tu
- ▶ Toliki broj fajlova nam treba zbog ovog **AKO**
 - ▶ Pravimo kompenzaciju za nesigurnost Bloom Filter-a
- ▶ **ALI** takodje da što je pre moguće da dobijemo informaciju nazada
 - ▶ Kada razmišljate o ovim sistemima, razmišljajte na nivou količine podataka Google-a, Facebook-a, Twitter-a, ...
- ▶ Ili bar da nam mašinerija kaže: **ključ koji tražis nije sigurno tu**
- ▶ **Naravno, nismo ograničeni na to, ove strukture su vrlo korisne za razne tipove aplikacija**

Put čitanja podataka

- ▶ 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 razumnom vremenu
- ▶ Pa hajde da vidimo kako to rade neki poznati sistemi —velikani (npr. Apache Cassandra)
- ▶ Videćemo kako taj sistem pravi koreografiju elemenata i šta nam još fali
- ▶ A možda i nešto možemo da izbacimo :)
- ▶ Pa da probamo da napravimo naš **Read Path**

Razlike

- ▶ Primer *read path-a* *Cassandra-e*
- ▶ Kompresiju ignorišemo **za sada**
 - ▶ Manje prostora na disku
- ▶ Imamo dva ključa :O – drugačiji model!!
 - ▶ Mi koristimo jedan :'(– gledamo *Row Key*
- ▶ **ALI**, imamo **cache** element
 - ▶ Pre Bloom filter-a, zanimljivo
 - ▶ Potencijalno ubrzanje



(Cassandra Performance: The Most Comprehensive Overview
You'll Ever See)

Pitanje 2

Slušali ste arhitekturu računara, šta je tamo radila **keš memorija**

ideje :)?

Keširanje podataka

- ▶ Keširanje podataka je proces koji skladišti više kopija podataka ili datoteka na privremenoj lokaciji za skladištenje — **cache**
 - ▶ Cache čuva podatke za softverske aplikacije, servere i web pretraživače, ali i za sistemski software
 - ▶ Ovaj proces osigurava da korisnici ne moraju da preuzimaju informacije svaki put kada pristupe web lokaciji ili aplikaciji
- ▶ Čitava ideja je nastala oko ideje: kako bi ubrzali učitavanje sajta, ili što pre pristupili podacima
 - ▶ Čitanje podataka iz memorije je (relativno) brzo
 - ▶ Ali opet, memorije nemamo toliko puno — ma šta neko rekao
 - ▶ U memoriji su obično najsvežiji podaci
- ▶ Čitanje podataka sa diska je (relativno) sporo, pogotov kada imate dosta podataka
 - ▶ Razmišljajte na nivou Facebook, Instagram, Amazon, ...
 - ▶ Sa druge strane disk je relativno jeftin, i ima ga prilično dosta

Keširanje podataka

- ▶ Cena nije ista kao cena diska, a ni kapacitet
- ▶ U memoriji čuvati samo podake kojima se (relativno) skoro pristupalo
 - ▶ Ovo će omogu'čiti brži pristup podacima
 - ▶ Ne čavi sve podatke!
 - ▶ Plus ako se sistem restartuje nema više podataka
- ▶ Ali moramo naći način da čuvamo najnovije informacije
- ▶ Postoje razni načini i strategije kako čuvati podatke u cache-u
 - ▶ Možemo da rotiramo sadržaj, ako je struktura fiksne dužine
 - ▶ Imamo nekakav fiksni skup, koliko elemenata može da uvek bude u cache-u
 - ▶ Noviji elementi izbacuju one starije
 - ▶ Možemo definisati život podataka u cache-u (time to live — TTL)
 - ▶ ...

Pitanje 3

Sada znamo šta je *Row cache*, ali kako ćemo čvati podatke u njemu...Koliko dugo, kada ih izbacivati...

ideje :)?

Keširanje podataka, zadržavanje stvari u kešu

- ▶ Tačan odgovor ne postoji, ili bar meni nije poznat :)
- ▶ Jedna strategija se ne može primeniti na sve slučajeve
- ▶ Zavisi od aplikacije
- ▶ Zavisi od slučaja korišćenja
- ▶ Zavisi od resursa
- ▶ ...

There are only two hard things in Computer Science: cache invalidation and naming things.

(Phil Karlton)

Cache – Zahtevi

- ▶ Cache sistemi obično imaju zahteve koje treba da isputene, 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 ograničenja je obično eksternizovano u nekakvu konfiguraciju — konfiguracioni fajl, ili neki drugi oblik

Pitanje 4

Kako da ispunimo ove zahteve...struktura, algoritam,...

ideje :)?

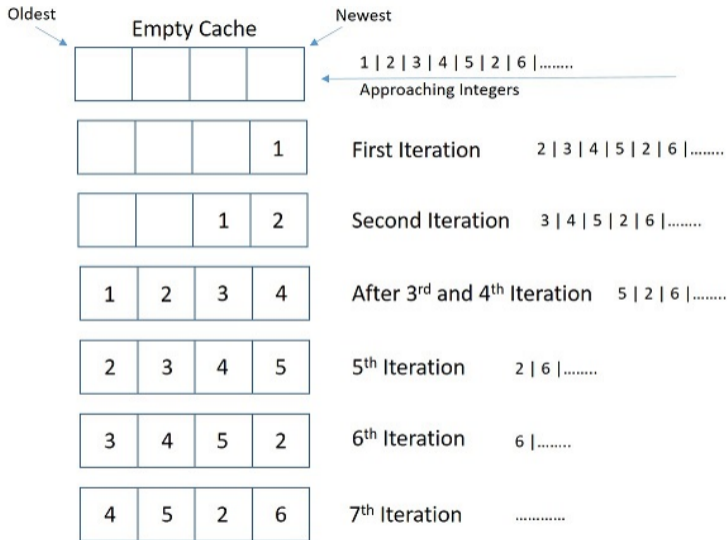
LRU, ideja

- ▶ Najmanje nedavno korišćen (LRU) 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 staka
- ▶ Ukratko, to je politika izbacivanja iz keša
- ▶ Navodimo kada se naš keš popuni i doda se novi element, uklanjamo najmanje nedavno korišćenu stavku iz keša
- ▶ Koristi **dve** strukture podataka u kombinaciji

Pitanje 5

A šta ako uzmemo neki element iz LRU...šta radimo sa pozicijama...

ideje :)?



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

LRU, osobine

- ▶ Efikasna implementacija:
 - ▶ Dvostruko spregnutu lista omogućava dodavanje na kraj u $\mathcal{O}(1)$ vremenu, i brisanje sa početka u $\mathcal{O}(1)$ vremenu – **zna** koje čvorove da izbacimo (najmanje korišćeni)
 - ▶ Hash mapa za skladištenje stvarnih parova podataka keša (ključ, vrednost), pri čemu su vrednost čvorovi u prethodnoj listi – brz pristup u $\mathcal{O}(1)$ vremenu
- ▶ 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)

Pitanje 6

Sada imamo cache, super, ali šta sa njim...šta on da čuva...

ideje :)?

Read Path – Blok cache

- ▶ Poslednji element da kompletiramo osnovni read path
 - ▶ Čuva **blokove podatke** kako im je korisnik **zadnje** pristupao, u memoriji u potpunosti – slično kao i buffer pool
- ▶ **Blok** u cache-u, vratimo korisniku **samo željnu vrednost** – brzo :D
- ▶ **Blok** nije u cache-u, moramo videti da li je prisutan uopšte – traje :(
- ▶ **ALI** imamo silne strukture da nam pomognu koliko jeto moguće
 - ▶ Ispitati **Bloom filter** sa diska i videti da li je ključ **možda** tu
 - ▶ Ako nije, odmah javimo korisniku – podatak nije sačuvan
 - ▶ Ako je **možda** tu, učitati **summry** i videti da li je ključ u tom opsegu
 - ▶ Ako nije, odmah javimo korisniku – podatak nije sačuvan
 - ▶ Ako **jeste**, pronaći ga u **index** strukturi, i uzmemo **offset**
 - ▶ Pozicioniramo se na **Data** blok, pročitamo, vratimo korisniku podatak

Pitanje 7

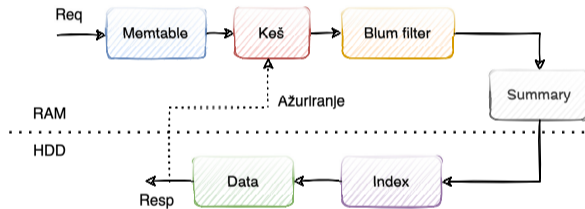
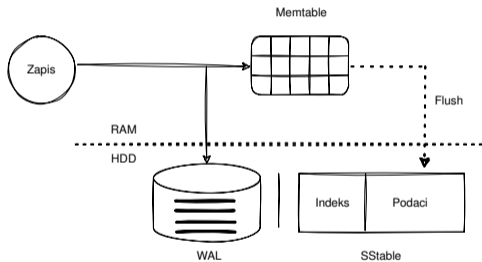
Ok, ali kako nam tu cache pomaže...

ideje :)?

Read path

- ▶ 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
- ▶ Promašaj keša je ishod u kojem sistem ili aplikacija postavlja zahtev za preuzimanje podataka iz keš memorije, ali ti specifični podaci trenutno nisu u keš memoriji
- ▶ Ovo se nekada beleži zarad nekakvih analitika i promene politike šta i kada keširati
- ▶ Kada se uputi zahtev za brisanje nekog podatka, **AKO** je on u cache-u, možemo ga obrisati

Write/Read path



Uvod

- ▶ Videli smo kako možemo pročitati nekakavu **vrednost** u sistemu koristeći **ključ**
 - ▶ Ova operacija se često zove i **GET** po uzoru na sličnu operaciju iz *hash map-a*
- ▶ Medjutim, ta operacija nije uvek dovoljna...
 - ▶ Pristup više **vrednosti** u isto vreme, treba nam više **ključeva**
 - ▶ Uradimo **GET** zahtev ka sistemu – NIJE OPTIMALNO, sistem se jako opterećuje
 - ▶ Pored toga, ako korisnik želi da prikaže ili obradi sve te podatke u celini (batch), potencijalno čeka dugo vremena
- ▶ I sami vidite koliko komponenti mora da se pozove da bi uradili jedno jedino čitanje podataka
- ▶ Dva dodatna načina pristupa **većem broju ključeva** su:
 1. Dobijanje vrednosti prefiksom – **prefix scan**
 2. Dobijanje vrednosti u opsegu – **range scan**

- ▶ Ova dva načina pristupa su vrlo slična u svojoj osnovi
- ▶ Oslanjaju se na ideju da su podaci u SSTable-u unapred **sortirani!**
- ▶ **AKO** to podržimo, implementacija ovih pristupa nije tako teška
- ▶ **ALI** moramo paziti da ne nasednemo na trivijalne provokacije, jednostavnosti pristupa
- ▶ Pre svega na situaciju kada je broj vrednosti koje treba da vratimo korisniku relativno velik
- ▶ Moramo voditi računa da ne opterećujemo sistem, a ni mrežu, predugačkim odgovorima
- ▶ Nego da nekako *iseckamo* podatke na delove fiksne veličine

Prefix scan

- ▶ Ideja iza ovog upida 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
- ▶ U ovom situaciji, sortiranje ključeva **jako** pomaže bržem dobijanju rezultata
- ▶ 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

- ▶ item Ideja iza ovog upida nad sistemom je relativno jednostavna
- ▶ 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
- ▶ U ovom situaciji, sortiranje ključeva **jako** pomaže bržem dobijanju rezultata
- ▶ 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

Uvod

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

Kursori

- ▶ Postoji nekoliko načina za rešavanje ovog problema
- ▶ Kotistićemo 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
- ▶ Veličinu stranice može da specificirira korisnik, ili sam sistem – zavisi od implementacije
- ▶ 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 :)

Kretanje kursorom

- ▶ Pokazivači na prethodnu i narednu stranicu nam trebaju zato što korisnik možda želi da se kreće kroz rezultate
- ▶ Možemo da idemo stranicu unapred, ili stranicu unazad i da dobijamo uvek **max** n elemenata – unapred očekivane performanse
- ▶ Nekoliko stranica možemo i ubaciti u keš, da bi odgovor bio još brzi – koliko stranica u kešu, to je podesivo
- ▶ Kada dolazimo do zadnje stranice, sistem mora nekako osvežiti vrednost keša – pozadinska operacija

Izmene stranica

- ▶ Pored toga, moramo voditi računa o još par stvari:
 - ▶ Neće sve stranice biti popunjene do fiksne veličine n – nema dovoljno podataka u zadnjoj stranici
 - ▶ Podaci se mogu izmeniti (dodati novi, obrisati/izmeniti postojeći) – sistem mora da odgovori na ove zahteve tako što ažurira podatke koji se nalaze u stranicama
- ▶ Kada dolazi do izmene stranica, moramo voditi računa da podaci koji se serviraju korisniku oslikavaju pravo stanje sistema
- ▶ Moramo voditi računa da podatke osvežimo blagovremeno – zato se operacija sinhronizacije obično obavlja u pozadini
- ▶ Podaci se mogu čuvati u kešu neko vreme – automatski se brišu nakon vremena t

Dodatni materijali

- ▶ A Low Overhead High Performance Buffer Management Replacement Algorithm
- ▶ Policy with Applications to Video Streaming
- ▶ <https://hazelcast.com/glossary/cache-miss/>
- ▶ Data Caching in Cassandra

Pročitati za narednu sedmicu

- ▶ LSM Trees paper
- ▶ Morning paper LSM Trees
- ▶ LSM-based storage techniques: a survey
- ▶ Designing Access Methods: The RUM Conjecture