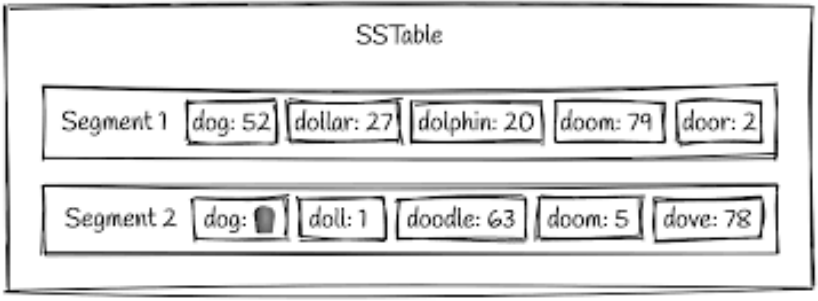
Napredni algoritmi i strukture podataka

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

Univerzitet u Novom Sadu
Fakultet Tehničkih Nauka

SSTable — Data

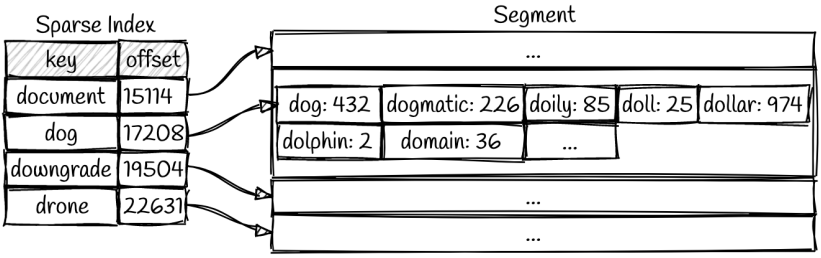
- SSTable je niz parova **ključ:vrednost** koji su sortirani i zapisani na disk
- SStable je nepromenljiva struktura (log based struktura)
- SSTable nastaje nakon što se Memtable popuni i zapiše na disk



(LSM — Write Heavy Databases)

SSTable — Index

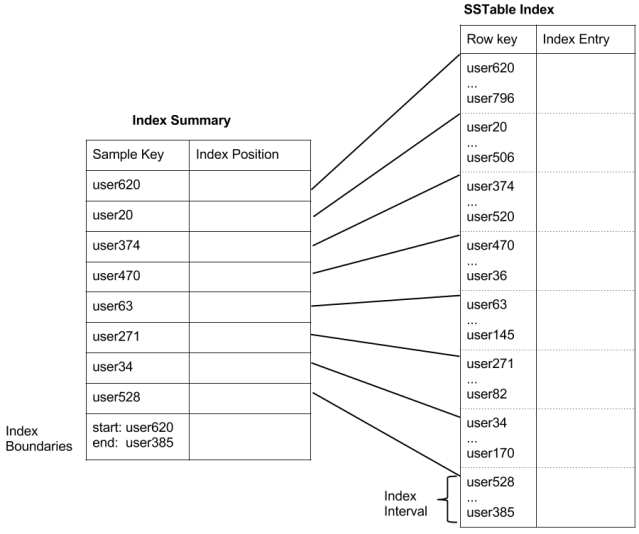
- Ideja je da što pre stignemo do sadržaja
- Struktura je relativno jednostavna:
  1. **ključ** koji se nalazi u fajlu na disku (SStable)
  2. **offset** u fajlu, tj. pozicija u fajlu na disku
- Index bi bilo lepo sortirati (ne budite lenji, učinite sebi uslugu :))



(LSM — Write Heavy Databases)

SSTable — Summary

- 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



(Distributed Datastore, Summary)

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

- 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>
```

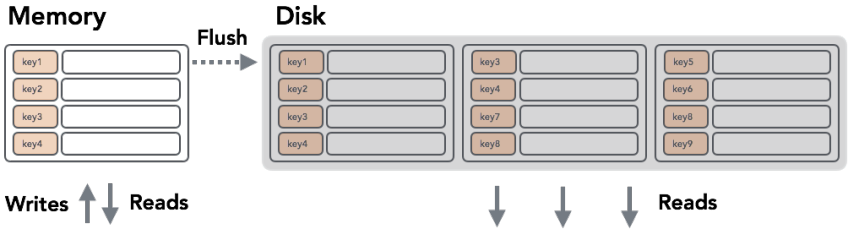
(LevelDB Format)

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

Write path smo videli na nekom od prethonih predavanja, Read path ukratko prošlo predavanje, ali jako prosto



(Databases Internals)

Pitanje 1

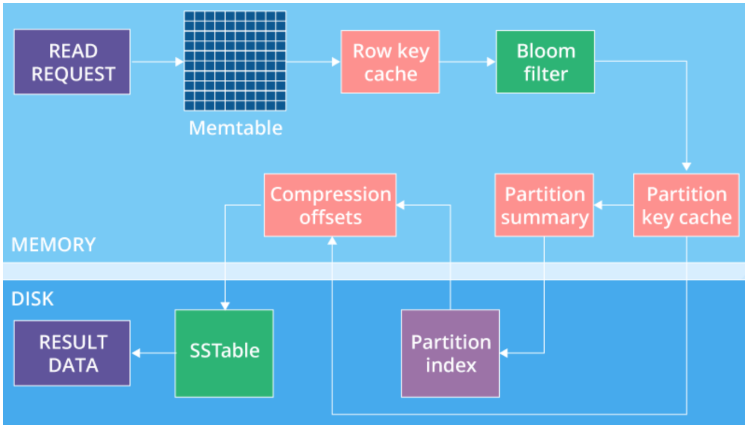
Ako pogledamo prethodnu sliku, vidimo da se svo čitanje odvija sa diska i iz SSTable-a. Zašto smo onda formirali onoliko elemenata prethodno predavanja (**HINT:** Nije da vas mučim :))...

ideje :)?

- 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** i što je pre moguće da dobijemo informaciju nazada
- Ili bar da nam mašinerija kaže: **ključ koji tražis nije sigurno tu**
- Kada razmišljate o ovim sistemima, razmiľjajte na nivou količine podataka Google-a, Facebok-a, Twitter-a, ...
- **Naravno, nismo ograničeni na to, ove strukture su vrlo korisne za razne tipove aplikacija**

- Svi formirani elemnti 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**

Apache Cassandra — Read Path



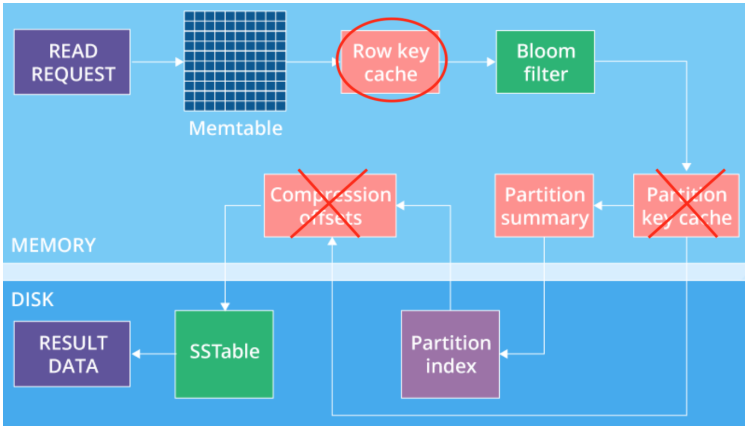
(Cassandra Performance: The Most Comprehensive Overview You'll Ever See)

Razlike

- Ako pogledamo prethodni dijagram
- Možda vam deluje komplikovano :), ali nije
- Deo oko kompresije smo rekli prošli put da ćemo ignorisati
- On služi da zauzmemo još manje resursa na disku
- Razlika je što ovde imamo dva ključa :O — Cassandra ima drugačiji model
- Naš model ima samo jedan :'(— mi koristimo opšti model podataka

- Pošto naš model koristi jedan ključ, drugi ćemo prosto ignorisati :D
- Ostaje nam samo jedan — *Row key*
- **ALI**, na dijagramu postoje elementi koji se zovu **cache**
- I još se nalazi se pre Bloom filter-a
- Pa ovo je vrlo zanimljivo....
- Znači da možda može dodatno da ubrza posao :D

Cache element



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 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
- Čitanje podataka iz memorije je (relativno) brzo
- Ali opet, memorije nemamo toliko puno — ma šta neko rekao
- U memoriji su obično najsvežiji podaci

- Cena nije ista kao cena diska, a ni kapacitet
- Plus ako se sistem restartuje nema više podataka
- U memoriji čuvati samo podake kojima se (relativno) skoro pristupalo
- Ne čavi sve podatke!
- Ovo će omogu'citi brži pristup podacima
- 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 :)?

- 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
- ...

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 ograniučenja je obično eksternizovano u nekakvu konfiguaraciju — konfiguracioni fajl, ili neki drugi oblik

Pitanje 4

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

ideje :)?

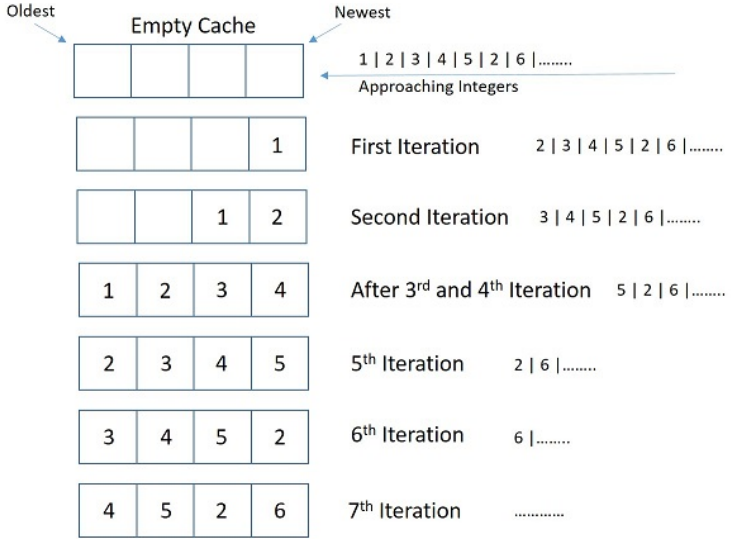
LRU

- 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 stalka
- Ukratko, to je politika izbacivanja iz keša
- Navodimo kada se naš keš popuni i doda se novi element, uklanjamo najmanje nedavno korišc'cenu stavku iz keša

Pitanje 5

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

ideje :)?



(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)

Pitanje 6

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

ideje :)?

Read Path — keširanje

- Cache je poslednji element koji nam fali da kompletiramo read path
- Cache čuva ključeve i vrednosti kako im je korisnik **zadnje** pristupao
- Cache se nalazi u memoriji u potpunosti
- **AKO** je podatak u cache-u, vratite ga korisniku — jako brzo :D
- **AKO** podatak nije u cache-u, onda moramo da vidimo da li je prisutan uopšte — taj put traje :(
- **ALI**, zato imamo sve one silne strukture da nam pomognu i ubrzaju koliko jeto moguće

- To znači da 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 **summry** 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 uzeto **offset**
- Kada imamo **offset**, možemo da se pozicioniramo na **Data** deo i da pročitamo podatak i vratimo korisniku

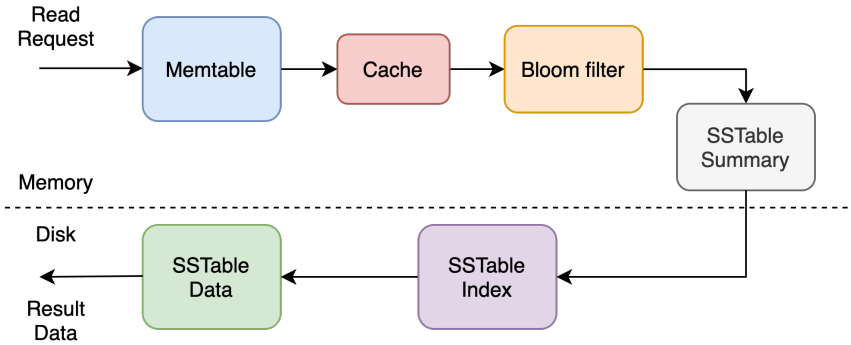
Pitanje 7

Ok, ali kako nam tu cache pomaže...

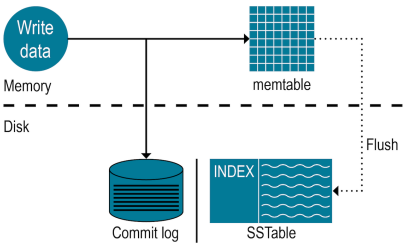
ideje :)?

- Podatke u cache-u moramo da ažurirmao 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

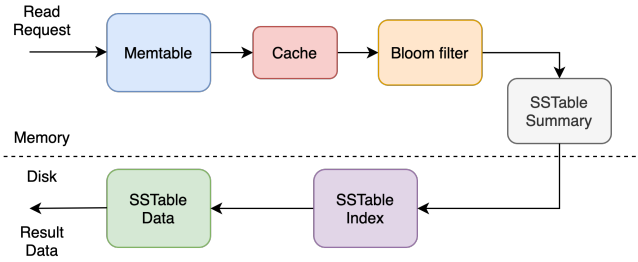
Read path


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

Read/Write path



(Distributed Datastore, SSTable format)



Uvod

- Kroz prethodne mehanizme 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 za zapis naših podataka...
- Ako korisnik želi da pročita više **vrednosti** u isto vreme, treba nam više **ključeva**
- Ovo možemo da rešimo tako što za svaki uradimo **GET** zahtev ka sistemu
- OVO NIJE OPTIMALNO – sistem se jako opterećuje

Dva dodatna pristupa

- I sami vidite koliko komponeti mora da se pozove da bi uradili jedno jedino čitanje podataka
- Ali ako treba da dobijemo **niz** vrednosti koji su povezani na nekakav način, ovo će biti skupa operacija
- Pored toga, ako korisnik želi da prikaže ili obradi sve te podatke u celini (batch procesuiranje), potencijlano čeka dugo vremena
- 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 provocije, jednostavnosti pristupa
- Pre svega na situaciju kada je broj vrednosti koje treba da vratimo korisniku raltivno 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 korsiniku
- 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, odfgovor 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 $\mathfrak{n}$
- Veličinu stranice može da specificiria korisnik, ili sam sistem – zavisi od implementacije
- Prilikom vraćanja rezultat korisniku, sada vraćamo dve vrednosti:
  1. Stranicu podataka veličine $\mathfrak{n}$
  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

Pitanja

Pitanja :) ?