

Napredni algoritmi i strukture podataka

LSM Stabla, Kompakcije, Amplifikacije, RUM pretpostavke



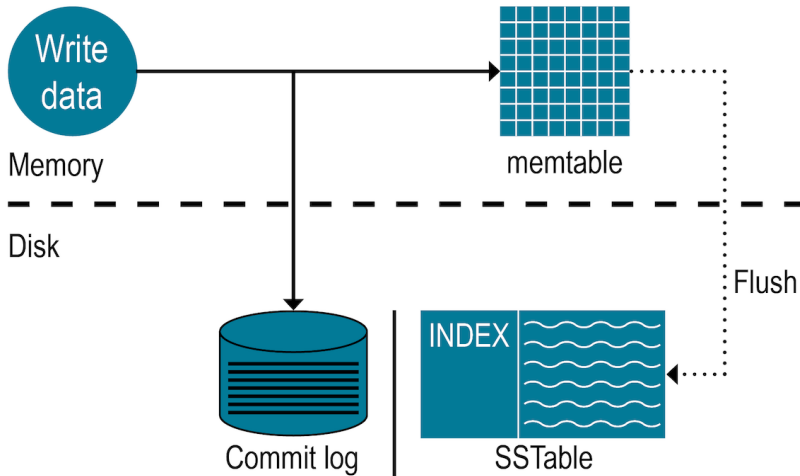
Univerzitet u Novom Sadu
Fakultet Tehničkih Nauka

Memtable — rekapitulacija

- ▶ Memtable ideja je relativno jednostavna — zapisati podatke u memoriju i čitati podatke iz memorije
- ▶ Memorija je brza, memorija je super, memorija je kul
- ▶ **ALI** nemamo beskonačno memorije (recite to matematičarima :))
- ▶ **AKO** se podaci nalaze u memoriji, sve operacije su relativno brže nego da su podaci **striktno** na disku
- ▶ **ALI** memorija nije sigurna :/

- ▶ Restart sistema i naših podataka više nema (objasnite to korisnicima:))
- ▶ Iz tog razloga nam treba snažna garancija trajnosti podataka
- ▶ Iz tog razloga Memtable komunicira sa WAL-om, koji nam daje ove garancije
- ▶ Ovaj princip se pokazuje jako korisno kod **write-heavy** problema
- ▶ Kada se Memtable struktura popuni, ona se perzistira na disk (Flush) i pravi se **SSTable** koja je neprimenljiva
- ▶ Veličinu memtable-a, možemo podešavati

Write path



(Cassandra write path)

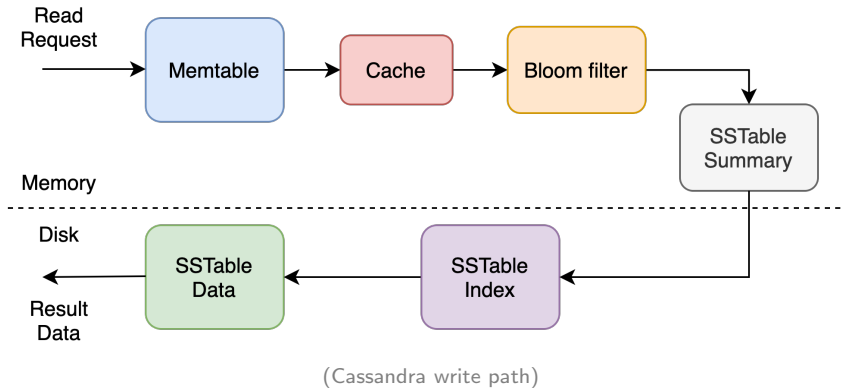
SSTable — rekapitulacija

1. Niz parova **ključ:vrednost** koji su sortirani, i zapisani na disk kao nepromenljiva struktura – struktura tipa log-a
2. SSTable se sastoji od nekoliko elemenata
3. Sve ove elemente je potrebno formirati kada se formira SSTable
4. kada se Memtable napuni, podaci se sortiraju i zapisuje na disk formirajući SSTable
5. **Voditi računa da je potrebno i blagovremeno izmeniti stanje cache-a!**

- ▶ SSTable (između ostalog) sadrži
 1. Bloom Filter sa dostupnim ključevima za taj SSTable
 2. SSTable Data segment – konkretni podaci koje tablea čuva
 3. SSTable Index pošto znamo na kojoj poziciji u Data fajlu je koji ključ
 4. SSTable Summary, pošto je SSTable Index već formiran i znamo pozicije ključeva
 5. Potrebno je da formiramo Merkle stablo od podataka iz Data segmenta
- ▶ **Napomena: organizacija zavisi od toga da li je sve u jednoj datoteci, ili više njih!**

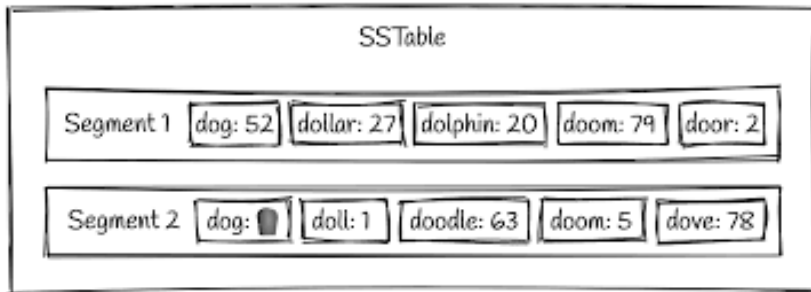
- ▶ Ova struktura nam omogućava da učitamo jako malo podataka u memoriju da bi proverili da li podatak postoji
- ▶ Uvek prvo proverimo **cache**
- ▶ Ako ključ nije tu, proveravamo da li je **možda** zapisan koirsteći **Bloom filter** koji učitavamo u memoriju
- ▶ Ako možda jeste, u memoriju učitavamo **deo Summary-a**, da bi našli gde **Index** čuva tu vrednost — **Index** ne učitavamo u memoriju
- ▶ Kada dobijemo **offset** iz **Index** fajla koristeći **Seek** operaciju
- ▶ Iz **Data** fajla možemo dobiti podatak koristeći **Seek** operaciju
- ▶ Podatak zapišemo u **Cache** za naredni put!

Read path



SSTable — Put brisanja

- ▶ Kada se obriše podatak on nije momentalno uklonjen sa diska
- ▶ Sistem zapisuje specijalan podatak *Tombstone* da je neki ključ obrisani — markira ga za brisanje
- ▶ Brisanje elemenata je zapravo **nov zapis** u SSTable — SSTable je nepromenljiva struktura
- ▶ Fizičko brisanje sa diska se odvija kada se desi specifičan proces **kompakcije**



(SSTable structure with tombstone)

Pitanje 1

Da li vidite problem koji može nastati kod **Write path, Delete path**

ideje :)?

Pitanje 2

Da li vidite problem koji može nastati kod **Read path**

ideje :)?

Read path vs Write path \wedge Delete path

- ▶ Write/Delete path može stvoriti previše fragmentisanih delova — SSTable-a
- ▶ Svaki put kada se Memtable popuni, uradimo Flush na disk praveći SSTable
- ▶ Time ne samo da fragmentišemo disk, nego Read path ima problem da ispita sve te delove — čitanja postaju jako skupa vremenom
- ▶ Pored toga SSTable je nepromenljiva struktura, šta ako su podaci obrisani...
- ▶ Brisanje je zapravo dodavanja, što znači da kada nesto obrišemo **dodajemo** nov element — pravimo dodatne zapise
- ▶ Ako podatak više nije aktivan i validan, trebalo bi da ga ulonimo sa diska — **problem ako je podatak u više fajlova**

- ▶ Dolazimo u situaciju da potencijalno imamo jako puno skeniranje podataka, nekih i više puta
- ▶ Moramo nekako ustanoviti šta je istina — koji podaci su **aktivni**
- ▶ Moramo nekako izbeći preveliku fragmentaciju podataka i ukloniti ono što nije aktivno
- ▶ Ovo pogotovo može biti nezgodno kod sistema sa velikim brojem čvorova — distribuiran sistem
- ▶ Dobacujemo se bespotrebnim podacima — podacima koji nemaju važnost
- ▶ **Ova stvari se mora rešiti, i to se moraju rešavati brzo!**

Pitanje 3

Moramo nekako ustanoviti šta je istina, moramo nekako izbeći preveliku fragmentaciju podataka

ideje :)?

LSM Stabla — ideja

- ▶ Ideja za rešavanje problema je relativno jednostavna – dodati nekakvu strukturu u naše podatke :)
- ▶ Omogućiti da SSTable elementi ne postoje tako sami za sebe i ukloniti višak
- ▶ Velika istina je, SSTable ne postoje tako sami za sebe na disku bez nekog reda
- ▶ One su (često) deo veće strukture koja se zove **Log structured Merge Trees** ili **LSM stabla**
- ▶ SSTable je struktura tipa log-a, SSTable grade LSM stabla, LSM stabla su strukture tipa log-a

- ▶ Samim tim i SSTable, i LSM stabla možemo čitati sekvencijalno
- ▶ Obe strukture su optimizovane za sekvencijalni rad
- ▶ Pokazuju dobre performanse na svim tipovima diskova
- ▶ Ova stabla su srce manje više svakog sistema za skladištenje velike količine podatka, pa čak i obradu
- ▶ Manje više svaki veliki sistem za obradu/skladištenje podataka danas koirsti LSM stabla (ili varijaciju) u svojoj osnovi
- ▶ Lako se prilagode drugačijim formatima podataka

LSM Stabla — uvod

- ▶ Osnovna ideja LSM stabla je implementacija rezidentnih stabala diska sličnih *B-stablina*
- ▶ Razlika je u tome što je LSM stablo optimizovano za sekvencijalni pristup disku i čvorovi mogu imati punu zauzetost
- ▶ Vredi napomenuti da se LSM stabla često porede sa B-stablina — nije striktno tačno ni korektno
- ▶ Naglasak LSM stabla je na dozvoljavanju nepromenljive, spojive datoteke
- ▶ Druga ideja je praviti hijerarhije u podacima zarad lakšeg rada
- ▶ Ovde dve osobine čine ova stabla **izuzetno** korisnim za razne primene

- ▶ Priroda primarnog indeksa za tabelu je stvar implementacije
- ▶ Izjava da je nešto implementirano kao LSM stablo ne govori nužno ništa, o složenosti pretraživanja
- ▶ Čak ne govori ni o unutrašnjem izgledu datoteke, već samo o **konceptualnoj strukturi!**
- ▶ Treba istaći da mnoge moderne LSM implementacije u polju baze podataka imaju nešto zajedničko — upotrebu sortirane tabele stringova ili SSTable
- ▶ LSM stabla se jako oslanjaju na SSTable, i to im je jedinica zapisa sa kojom dalje stabla operišu

LSM Stabl — ideja

- ▶ LSM stablo je struktura podataka dizajnirana da obezbedi jeftino indeksiranje za datoteke koje imaju visoku stopu dodavanja (i brisanja) tokom dužeg perioda
- ▶ Dobar moderan primer može biti dolazni **stream** podataka koji se zapisuje u nekakvu tabelu
- ▶ Jedina stvar koja je potrebna da bi LSM imala prednost izbora, spram drugih struktura je visoka stopa ažuriranja naspram stope čitanja
- ▶ U isto vreme pretraga mora biti dovoljno česta, da bi se održavala neka vrsta indeksa
- ▶ Sekvencijalna pretraga kroz sve zapise ne dolazi u obzir!

LSM Stabl — struktura

- ▶ Pošto govorimo o stablu, ono sigurno ima nekakve nivoe
- ▶ LSM stablo se sastoji od dva ili više nivoa $C_i = (C_0, \dots, C_k)$
- ▶ Ove nivoe čine strukture podataka u **obliku stabla**
- ▶ Na primer, dvokomponentno LSM stablo ima:
 1. Manju komponentu koja je u potpunosti rezidentna u memoriji, poznata kao C_0 stablo — C_0 komponenta
 2. Veću komponentu koja je rezidentna na disku, poznatu kao C_1 stablo — C_1 komponenta

- ▶ Podaci se prvo zapisuju u C_0 , a odatle migriraju/zapisuju u C_1
- ▶ Nivo C_0 služi kao bafer za zapise
- ▶ Jeftino dodati unos u C_0 stablo rezidentno u memoriji
- ▶ **Ali** cena/kapacitet memorije u spram diska ograničava veličinu onoga što ima smisla držati u memoriji!
- ▶ Broj nivoa je konfigurabilan, možemo specificirati koliko god nam je potrebno
- ▶ Ova stvar ima smisla da bude deo konfiguracije
- ▶ Obično je stvar potrebe sistema koriste ovu strukturu – stvar optimizacije

Pitanje 3

Podaci se prvo ubacuje u C_0 , a odatle migriraju/zapisuju u C_1 ...Hmmmm ovo zvuči poznato...

ideje :)?

- ▶ U našem slučaju nivo C_0 je Memtable, ono se nalazi u memoriji, fiksne je dužine
- ▶ Sa memorijom se ne razbacujemo — ma šta neko rekao vi ste inženjeri!
- ▶ Primenljivo je i služi kao bafer — da amortizuje sporost diska
- ▶ Drugi nivo C_1 čini SSTable koje nastaje kada se Memtable (nivo C_0) popuni i zapisujemo na disk
- ▶ Neprimenljivo je
- ▶ Imamo ga znatno više

Pitanje 4

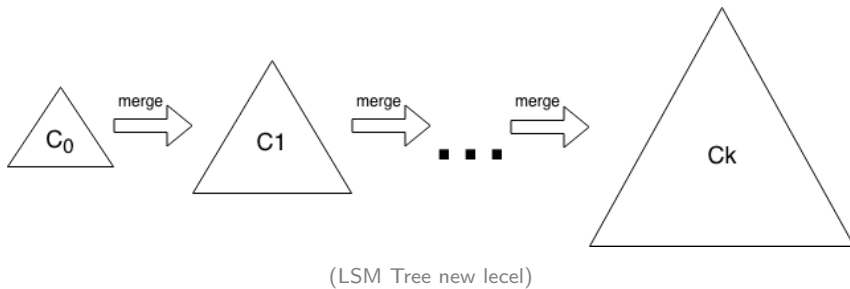
Sve je ovo lepo, ali kako ćemo rešiti problem sa početka...imamo puno fajlova koje napravimo svaki put kada se Memtable zapiše na disk...

ideje :)?

Kompakcije — ideja

- ▶ Pošto broj SSTable-a na disku stalno raste, nešto se mora uraditi po tom pitanju
- ▶ Podaci se nalaze u nekoliko datoteka: više verzija istog zapisa, zapisi koji su za brisanje, ...
- ▶ Čitanja će vremenom postajati sve skuplja operacija
- ▶ Ideja rešenja je zapravo jako jednostavna :)
- ▶ U srcu LSM strukture je proces spajanja tabela — **kompakcija**
- ▶ Ovi algoritmi ne moraju nužno biti jednostavni, ali generalna ideja jeste

- ▶ Iako mnogi primeri uključuju podatke vremenskih serija, ovo nije neophodna karakteristika!
- ▶ LSM stabla su (uglavnom) sačinjena od SSTable-a
- ▶ Svaki nivo C_k LSM stabla, sadrži n SSTable-a
- ▶ Spajati SSTable i obrisati nepotrebne podatke! (how cool :))
- ▶ Ovaj proces kreira veću SSTable, samim tim kreira i novi C_k nivo u LSM stablu
- ▶ Briše nepotrebne podatke
- ▶ I to je sve
- ▶ Vrste:
 - ▶ Minor — Memtable u SSTable
 - ▶ Major — Spajanje više SSTable-a



Kompakcije — plan

- ▶ Zbog strukture SSTable-a, ova operacija je veoma efikasna – zato smo sortirali podatke :)!
- ▶ Koriste algoritam koji podsecća na sortiranje spajanjem — **merge sort**
- ▶ Zapisi se čitaju iz nekoliko izvora uzastopno i mogu se odmah dodati u izlaznu datoteku
- ▶ Pošto su svi ulazi sortirani i spojeni, rezultirajući fajl će imati isto svojstvo (sweet :))
- ▶ Proces pravljenja indeksne datoteke može biti skuplja operacija u smislu složenosti
- ▶ Kada napravimo novu SSTable, moramo napraviti i sve prateće elemente za nju!

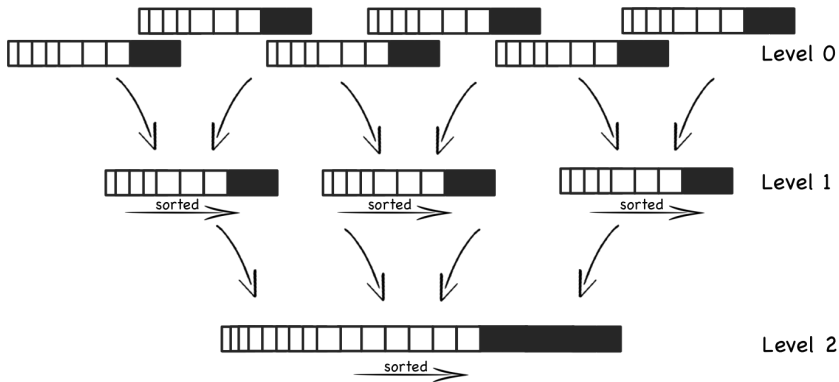


Kompakcije — proces spajanja

- ▶ Proces spajanja je **jako** bitna stvar kod formiranja LSM stabla
- ▶ Sa njim, čistimo disk od bespotrebnih podataka
- ▶ Proces čitanja postaje dosta brži
- ▶ **Ali** moramo biti mudri, da ne zapadnemo u probleme!
- ▶ Postoje tri stvari koje su važne kod procesa spajanja
 1. Garancije složenosti
 2. Logika brisanja
 3. Amplifikacije

Kompakcije — garancije složenosti

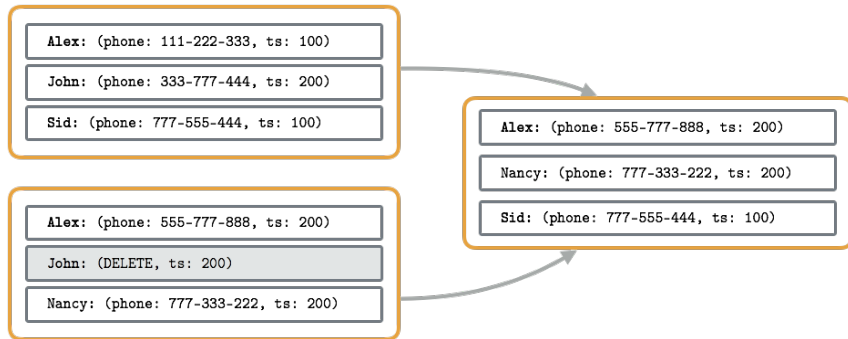
- ▶ U smislu složenosti, spajanje SSTable-a je isto kao i spajanje sortiranih kolekcija
- ▶ Ima $\mathcal{O}(N)$ overhead memorije, gde je N količina SSTable-a koje se spajaju
- ▶ Iteratori moraju pokazivati na korespondentne elemente iz obe SSTable-e!!
- ▶ Na svakom koraku, stavka se uzima iz sortirane kolekcije i ponovo popunjava iz odgovarajućeg iteratora
- ▶ Tokom kompakcije, sekvencijalno čitanje i sekvencijalno pisanje pomažu u održavanju dobrih garancija performansi



(LSM Tree compaction)

Kompakcije — logika brisanja (Shadowing)

- ▶ Shadowing je neophodno da bi se osiguralo da ažuriranja i brisanja funkcionišu
- ▶ Brisanje u LSM stablu se izvodi tako što se doda specijalna oznaka (Tombstone), navodeći koji ključ je označen za brisanje
- ▶ Slično tome, ažuriranje je samo zapis sa većom vremenskom oznakom
- ▶ Tokom čitanja, zapisi koji su označe brisanje neće biti vraćeni klijentu
- ▶ Ista stvar se dešava i sa ažuriranjima
- ▶ Od dva zapisa sa istim ključem, vraća se onaj sa kasnijom vremenskom oznakom

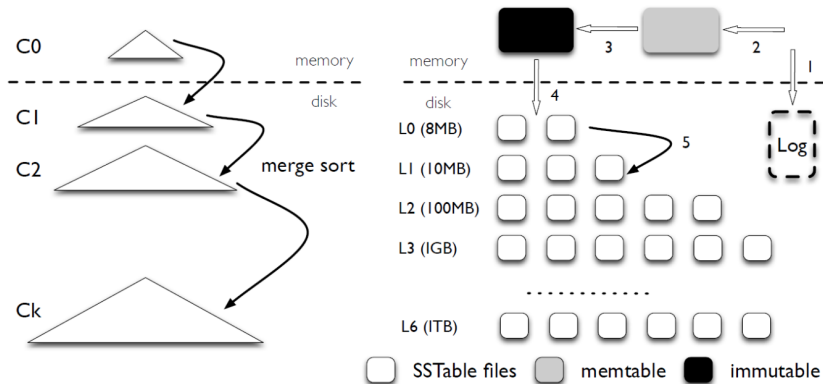


(LSM Tree compaction)

Pitanje 5

Gde se nalaze novije informacije, u manjim ili većim blokovima...

ideje :)?



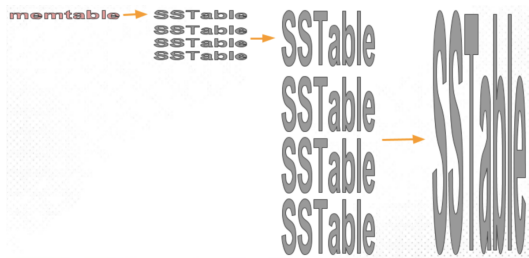
(LSM Tree compaction)

Kompakcije — algoritmi i amplifikacije

- ▶ Postoji nekoliko algoritama za kompakciju podataka
 - ▶ *Size-tiered* kompakcija
 - ▶ Leveled kompakcija
 - ▶ Hibridna kompakcija — kombinacija prethodna dva algoritma
- ▶ Ovi algoritmi imaju različite amplifikacione osobine koje se dešavaju tokom ovog procesa:
 - ▶ Amplifikacija čitanja označava broj operacija koje se dešavaju na disku pri zahtevu za čitanje
 - ▶ Amplifikacija pisanja n je definisano kao bajtovi podataka koji su stvarno upisani na disk kada je potrebno upisati jedan bajt podataka
 - ▶ Amplifikacija prostora se uglavnom odnosi na količinu nesakupljenih isteklih podataka, koji su ili stare verzije podataka ili izbrisani unosi

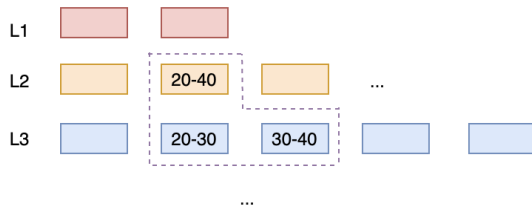
Kompakcije — Size-tiered kopakcija

- ▶ Prednost: nizak nivo amplifikacije pisanja je pogodno za sisteme koja zahtevaju intenzivno pisanje
- ▶ Nedostatak: amplifikacija čitanje i prostora je relativno visoko
- ▶ Kada na nekom nivou C_i nakupimo n SSTable-a, spajamo ih da bi dobili novi nivo C_{i+1}
- ▶ **Spajanje na nivou C_i može da izazove spajanje na višim nivoima lančano**
- ▶ Kada uradimo spajanje, obrišemo nepotrebne SSTable-e



Kompakcije — Leveled kompakcija

- ▶ LSM-stablo se sastoji od više nivoa — naredni nivo je T puta veći od prethodnog nivoa
- ▶ Svaki nivelisani nivo je **run** koji se sastoji od više SSTable-a — naredni **run** (obično) 10x veći od prethodnog
- ▶ Kada veličina podataka svakog nivoa dostigne gornju granicu, ovaj nivo će se spojiti sa **run**-om sledećeg nivoa
- ▶ Pošto je sve sortirano, prilikom kompakcije gledamo gde se ključ nalazi i spajamo sa tom SSTable-om
- ▶ Problem sa kojim se suočava ovaj algoritam je amplifikacija pisanja



Kompakcije — mane

- ▶ Proces kompakcije je konstantan posao
- ▶ Pozadinski procesi su dužni da nadgledaju sadržaj i da primene izabrani proces kompakcije
- ▶ Ovaj proces je **zahtevan** i može dodatno da optereti sistem – prilično!
- ▶ Opterećenje se odnosi kako na procesne resurse, tako i na sam disk
- ▶ Zbog svojih osobina, može dodatno da uspori sistem — Java GC
- ▶ Treba voditi računa o implementaciji i pokretanju u pravo vreme

LSM stabla — zaključak

► Prednosti

- LSM stabla mogu da podnesu veoma visok protok pisanja
- LSM stabla se mogu bolje kompresovati i tako rezultirati manjim datotekama segmenta loga
- Samim time potencijalno mogu skladištiti više informacija

► Mane:

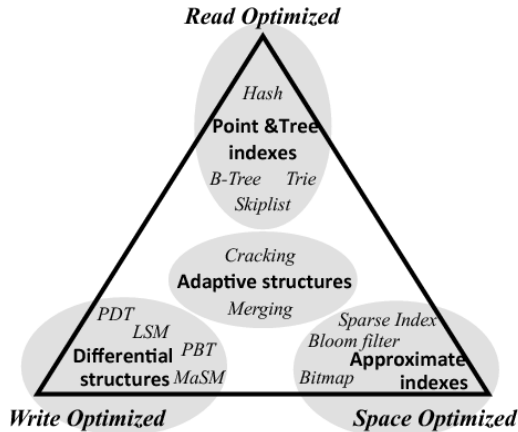
- Proces kompakcije ponekad ometa performanse tekućeg čitanja i pisanja sadržaja
- Svaki ključ može postojati na više mesta samim time zauzima se dodatan prostor za podatke koji se ne koriste — amplifikacija

Uvod

- ▶ Sada ste čuli razne strukture podataka
- ▶ Videli ste razne organizacije podataka na disku – ima ih još, ovo je samo elementarno :)
- ▶ Postavlja se pitanje kada se koja koristi?
- ▶ Postoje li situacije kada je jedna bolja od druge?
- ▶ Na ova pitanja postoji odgovor, i on je definisan u okviru **RUM** pretpostavki
- ▶ **RUM** je definisan od strane istraživača sa Harvarda

Metode pristupa – RUM pretpostavke

- ▶ Tri ključna aspekta za šta su strukture podataka optimizovana
 - ▶ Čitanje – **R**
 - ▶ Pisanje/Izmena – **U**
 - ▶ Memorija/Prostor – **M**)
- ▶ Uzimamo u razmatranje kada biramo strukturu podataka za neki posao
- ▶ Uzimamo u obzir kada konsultujemo domen problema
- ▶ Kada radite sa velikim podacima ili optimizujete sistem **izuzetno koristan vodič**



(Designing Access Methods: The RUM Conjecture)

Dodatni materijali

- ▶ LSM Trees paper
- ▶ Morning paper LSM Trees
- ▶ LSM Trees
- ▶ TiKV LSM Trees
- ▶ Alibabacloud LSM compactions
- ▶ Scylladb compactions
- ▶ LSM-based storage techniques: a survey
- ▶ Fast Compaction Algorithms for NoSQL Databases
- ▶ Re-enabling high-speed caching for LSM-trees
- ▶ Database Internals: A Deep Dive into How Distributed Data Systems Work
- ▶ Designing Access Methods: The RUM Conjecture

Pitanja

Pitanja :) ?