

Interna struktura i organizacija skladištenja kod Apache Cassandra baze podataka

Studijski program: Računarstvo i informatika

Modul: Softversko inženjerstvo

Predmet: Sistemi za upravljanje baze podataka

|  |  |
| --- | --- |
| Student: | Profesor: |
| Milica Sokolov,  broj indeksa 1655 | Prof. dr Aleksandar Stanimirović |

Niš, april 2024. godina

Sadržaj:

[Uvod 1](#_Toc164539114)

[Arhitektura Cassandra baze podataka 2](#_Toc164539115)

[Komponente Cassandre 2](#_Toc164539116)

[Čvor 2](#_Toc164539117)

[Virtuelni čvor 2](#_Toc164539118)

[Server 2](#_Toc164539119)

[Rack 2](#_Toc164539120)

[Data centri 3](#_Toc164539121)

[Klaster 3](#_Toc164539122)

[Struktura prstena 3](#_Toc164539123)

[Gossip protokol 4](#_Toc164539124)

[Partitioner 5](#_Toc164539125)

[RandomPartitioner 7](#_Toc164539126)

[ByteOrderPartitioner 7](#_Toc164539127)

[Murmur3Partitioner 8](#_Toc164539128)

[Replikacija 9](#_Toc164539129)

[R + W > N nejednakost 10](#_Toc164539130)

[Commit log 11](#_Toc164539131)

[MemTable 13](#_Toc164539132)

[SSTable 14](#_Toc164539133)

[Bloom filter 14](#_Toc164539134)

[Index files 15](#_Toc164539135)

[Data files 16](#_Toc164539136)

[Compaction 16](#_Toc164539137)

[Hinted handoff 18](#_Toc164539138)

[Zaključak 19](#_Toc164539139)

[Literatura 20](#_Toc164539140)

# 

# Uvod

Tradicionalni relationalni sistemi za upravljanje bazama podataka (RDBMS) su bili široko korišćeni za različite svrhe, kao što su vođenje finansijskih transakcija ili praćenje proizvodnje i zaposlenih. Međutim, iako su bili uspešni, oni se suočavaju sa izazovima skaliranja kako bi zadovoljili potrebe modernih aplikacija velikih razmera. Pokušaji skaliranja RDBMS rešenja putem deljenja baza podataka na više servera nisu uvek bili uspešni, a tradicionalni modeli nailazili su na probleme. U isto vreme, potreba za rešenjem koja bi pružila adekvatnu skalabilnost, dostupnost, performanse i doslednost podataka postajala je sve izraženija. Ovo je izazvalo trku u potrazi za boljim rešenjima, a Cassandra se pojavila kao odgovor na ove izazove.

Cassandra je besplatna baza podataka otvorenog koda koja je napisana u Javi. Ona se pojavila relativno kasno u poređenju sa drugim distribuiranim bazama podataka, ali je postala izuzetno popularna zbog svojih jedinstvenih karakteristika. Cassandra se svrstava u baze podataka koje koriste koncept "column family" modela za organizaciju podataka. Neki od najvažnijih karakteristika Cassandre uključuju geografsku distribuiranost, visoku dostupnost, otpornost na kvarove, linearno skaliranje i fleksibilan model podataka.

Međutim, uprkos širokoj prihvaćenosti i korišćenju, duboko razumevanje njenog unutrašnjeg mehanizma skladištenja i organizacije podataka od suštinskog je značaja za optimizaciju performansi i efikasnosti.

Ovaj rad istražuje internu strukturu i organizaciju skladištenja u Apache Cassandra bazi podataka. Fokusiran je na ključne koncepte i mehanizme koji omogućavaju Cassandri da ostvari visoke performanse i skalabilnost. Kroz analizu arhitekture skladištenja podataka, biče istraženo kako Cassandra upravlja podacima u distribuiranom okruženju i kako osigurava doslednost, raspoloživost i otpornost na kvarove.

Pored toga, biće istražene i napredne tehnike optimizacije skladištenja podataka u Cassandri, uključujući kompresiju podataka, particionisanje i replikaciju. Kroz ovo istraživanje, cilj je pružiti dublje razumevanje unutrašnjeg delovanja Apache Cassandra baze podataka i pružiti smernice za efikasno korišćenje i optimizaciju performansi ovog moćnog alata za upravljanje podacima.

# Arhitektura Cassandra baze podataka

## Komponente Cassandre

### Čvor

Čvor je osnovna komponenta infrastrukture Cassandre. Čvor je potpuno funkcionalna mašina koja se povezuje sa drugim čvorovima u klasteru pomoću Gossip protokola. Gossip protokol omogućava komunikaciju između čvorova. Čvor može biti fizički server, virtuelna mašina ili instanca u cloud okruženju poput EC2. Svi čvorovi su organizovani u topologiji prstenaste mreže, gde svaki čvor ima istu ulogu i nezavisan je od drugih čvorova. Cassandra koristi strukturu peer-to-peer za organizaciju čvorova. Svaki čvor sadrži stvarne podatke i može obraditi zahteve za čitanje i pisanje. Ovo znači da lokacija podataka unutar klastera nije od suštinskog značaja, jer će uvek biti dostupna najnovija verzija podataka bez obzira na to koji čvor obrađuje zahtev.

### Virtuelni čvor

Novije verzije Cassandre koriste virtuelne čvorove (vnodes). Virtuelni čvor je sloj za skladištenje podataka unutar servera. Po defaultu, postoji 256 virtuelnih čvorova po serveru. Svaki čvor ima opseg tokena koji su mu dodeljeni. Svaki virtuelni čvor koristi podopseg tokena od čvora kojem pripada. Ovi virtuelni čvorovi pružaju veću fleksibilnost u sistemu. Kada podaci imaju nejednako raspoređene tokene među čvorovima u klasteru, to može rezultirati neravnomernim opterećenjem čvorova. Međutim, korišćenjem virtuelnih čvorova, možemo lako proširiti kapacitet skladištenja tako što ćemo dodati više virtuelnih čvorova na čvorove koji su preopterećeni. Ovo omogućava bolje iskorišćenje resursa i ravnomerniju raspodelu opterećenja među čvorovima.

### Server

Kada koristimo termin server, mislimo na mašinu sa instaliranim Cassandra softverom. Svaki čvor ima jednu instancu Cassandre, što je tehnički server. Kao što smo ranije rekli, svaka instanca Cassandre može da sadrži 256 virtuelnih čvorova. Cassandra server pokreće osnovne procese. Na primer, procese kao što su raspoređivanje replika po čvorovima ili usmeravanje zahteva.

### Rack

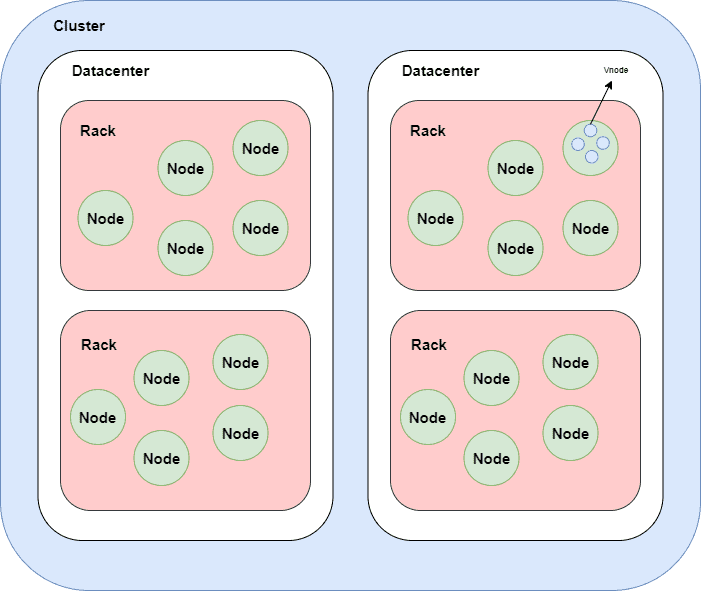
Cassandra rack predstavlja logičku grupu čvorova ili servera unutar klastera. Ovi čvorovi ili serveri se obično grupišu zajedno zbog fizičke blizine ili zbog slične namene. Rack omogućava Cassandra bazi podataka da organizuje raspodelu replika među različitim logičkim grupama, što doprinosi povećanoj otpornosti na greške i dostupnosti sistema. Na primer, ako se jedan rack nalazi u jednoj fizičkoj lokaciji, a drugi u drugoj, baza podataka će raspodeliti replike podataka između ovih rackova kako bi osigurala da gubitak jedne lokacije ne utiče na celokupnu dostupnost sistema.

### Data centri

Data centar se koristi za organizaciju i grupisanje rackova ili drugih logičkih grupa čvorova koji se često nalaze u istoj zgradi ili geografskoj lokaciji. Glavni cilj data centra je omogućavanje efikasne replikacije podataka i povećanje dostupnosti sistema. Na primer, Cassandra baza podataka može replikovati podatke između različitih data centara kako bi se osiguralo da gubitak jedne lokacije ne utiče na celokupnu dostupnost podataka. Takođe, data centri omogućavaju bržu i efikasniju komunikaciju između čvorova unutar istog data centra, što doprinosi poboljšanju performansi i smanjenju latencije.

### Klaster

Klaster je komponenta koja sadrži jedan ili više data centara. Na slici 1. možemo videti hijerarhiju elemenata u Cassandra klasteru: Prvo, imamo klastere koji se sastoje od data centara. Unutar data centara, imamo čvorove koji sadrže po defaultu 256 virtuelnih čvorova.



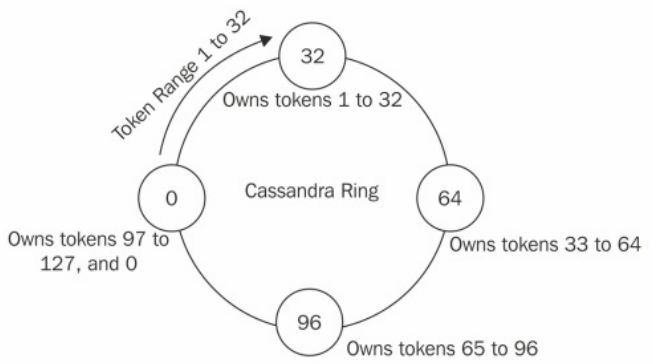
*Slika 1. Hijerarhijski elementi u Cassandra klasteru*

## Struktura prstena

Klaster Cassandre naziva se prstenom jer je organizovan u obliku prstena, što znači da svaki čvor u klasteru ima određenu ulogu u prstenu i komunicira sa susednim čvorovima. Ova terminologija je preuzeta iz Amazon Dynamo-a.

Cassandra 1.1 i ranije verzije obično su imale token dodeljen svakom čvoru. Svaki čvor ima svoj token, koji je u suštini heširana vrednost ključa reda. Kada se ti tokeni rasporede u prstenu, prvi čvor u prstenu (često nazvan "čvor 0") ima zadnji token vrednosti poslednjeg čvora (često nazvan "čvor n") i prvi token vrednosti prvog čvora (čvor 0). Ova struktura se naziva kružnom jer se nakon poslednjeg čvora nastavlja sa prvom vrednošću tokena, čineći tako kružni niz tokena. Dakle, ako skačete sa čvora na čvor, napravićete krug, i zbog toga se Cassandra klaster naziva prsten.

Na slici 2. Prikazan je primer ove strukture. Pretpostavimo da postoji algoritam heširanja (partitioner) koji generiše tokene od 0 do 127 i postoje četiri Cassandra mašine za kreiranje klastera. Da bi se postigla jednaka opterećenost, treba dodeliti svakom od četiri čvora jednaki broj tokena. Dakle, prva mašina će biti odgovorna za tokene od 1 do 32, druga će držati 33 do 64, treća 65 do 96, a četvrta 97 do 127 i 0. Ako obeležite svaki čvor sa maksimalnim brojem tokena koje može držati, klaster izgleda kao prsten.



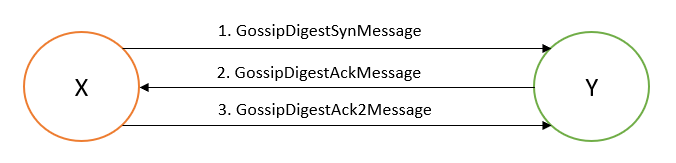
*Slika 2. Vlasništvo tokena i distribucija u izbalansiranom Cassandra prstenu*

## Gossip protokol

Cassandra koristi gosip protokol za komunikaciju između čvorova. Kao što samo ime sugeriše, protokol širi informacije na isti način kao što bi se širio trač. Takođe se može uporediti sa širenjem virusa. Ne postoji centralni emiter, ali informacije (virus) se prenose na celu populaciju.

Cassandra koristi gosip da sazna stanje i lokaciju drugih čvorova u prstenu (klasteru). Proces gosipovanja se izvršava svake sekunde i razmenjuje informacije sa, najviše, tri druga čvora u klasteru. Čvorovi razmenjuju informacije o sebi i drugim čvorovima o kojima saznaju putem neke druge sesije gosipovanja. To dovodi do toga da svi čvorovi na kraju saznaju o svim ostalim čvorovima. Kao i sve ostalo u Cassandri, poruke gosipovanja imaju povezan broj verzije. Dakle, svaki put kada dva čvora gosipuju, starije informacije o čvoru se prepravljaju novijim informacijama.

Što se tiče implementacije, zadatak gosipovanja se rukuje putem klase org.apache.cassandra.gms.Gossiper. Klasa Gossiper održava listu živih i mrtvih endpointa (nedostupni endpointi). Na jednu sekundu, ovaj modul pokreće kolo gosipovanja sa slučajno odabranim čvorom. Kompletno kolo gosipovanja sastoji se od tri poruke. Čvor X šalje syn poruku čvoru Y da inicira gosipovanje. Y, po prijemu ove syn poruke, šalje ack poruku nazad X-u. Da bi odgovorio na ovu ack poruku, X šalje ack2 poruku Y-u, završavajući tako celo kolo poruka. Slika 3 prikazuje dva čvora koji gosipuju:



*Slika 3. Gossip interakcija između dva čvora*

Syn i ack su takođe poznati kao handshake poruke. To je mehanizam koji omogućava dvema mašinama koje pokušavaju da komuniciraju jedna s drugom da pregovaraju o parametrima veze pre nego što prenesu podatke. Syn predstavlja "synchronize packet" (sinhronizacioni paket), a ack označava "acknowledge packet" (potvrdni paket).

Modul Gossiper je povezan sa detekcijom kvarova. Modul, kada čuje jednu od ovih poruka, ažurira detektor kvarova sa informacijama o tome da li je udaljeni čvor živ. Ako čuje GossipShutdownMessage, modul označava udaljeni čvor kao mrtav u detektoru kvarova.

Čvor s kojim će se obaviti razmena poruka biran je na osnovu sledećih pravila:

* Prvo, Cassandra bira nasumičan "živi" čvor u klasteru, tj. čvor koji je aktivan i dostupan za komunikaciju.
* Ako prvi korak nije uspeo ili je broj "živih" čvorova mali, Cassandra može odabrati nasumičan "nedostupan" čvor, tj. čvor koji nije aktivan ili je trenutno nedostupan za komunikaciju.
* Ukoliko ni prvi ni drugi korak nisu izvodljivi ili ukoliko je broj "živih" čvorova manji od broja seed čvorova u klasteru, Cassandra će umesto toga izabrati nasumičan "seme" čvor.

Seed čvorovi su posebni čvorovi u klasteru koji se koriste kao referenca za inicijalno otkrivanje i komunikaciju unutar klastera. Kada se novi čvor pridruži klasteru ili se čvor pokrene nakon pada, prvo pokušava da se poveže sa seed čvorom kako bi pronašao i upoznao ostale čvorove u klasteru. Seed čvorovi obično predstavljaju poznate i stabilne čvorove u klasteru. Predlaže se da u klasteru postoji više od jednog seed čvora. Seed čvor nije poput glavnog čvora u mehanizmu glavni-sluga (master-slave). To je samo još jedan čvor koji pomaže novo pridruženim čvorovima da pokrenu gossiper protokol.

## **Partitioner**

Cassandra je distribuirani sistem upravljanja bazama podataka. To znači da uzima jednu logičku bazu podataka i distribuira je preko jednog ili više mašina u klasteru baze podataka. Dakle, kada unesete neke podatke u Cassandru sa jedinstvenim ključem reda, na osnovu tog ključa reda Cassandra dodeljuje taj red jednom od čvorova koji je odgovoran za upravljanje njime.

Cassandra nasleđuje model podataka od Google-ovog BigTable-a što znači da grubo možemo da pretpostavimo da su podaci smešteni u nekoj vrsti tabele koja ima neograničen broj kolona (nije zaista neograničen; Cassandra ograničava maksimalni broj ćelija na 2 milijarde po particiji) sa redovima koji imaju jedinstveni ključ, tj. ključ reda.

Međutim, skladištenje velike količine podataka na jednoj mašini može biti problematično iz više razloga. Prvo, ograničen je prostor na disku koji jedna mašina može pružiti. Drugo, obrada podataka može biti ograničena, posebno ako se radi o složenim upitima ili operacijama nad velikim datasetima. Dodatno, ako podaci nisu duplicirani, postoji rizik od kvara sistema jer svi podaci zavise od jednog čvora.

Da bi rešila ove probleme, Cassandra definiše pravila kako će se podaci deliti na manje delove, poznate kao particije, i dodeljuje svaku particiju odgovarajućem čvoru u klasteru. Ovaj zadatak obavlja partitioner.

Kada govorimo o partitionerima u Cassandri, zapravo govorimo o mehanizmu koji određuje kako se podaci raspoređuju unutar klastera. Kada unesemo podatke u Cassandru, svaki red ima jedinstveni ključ koji se koristi za identifikaciju tog reda. Sada, ovde dolazi ključna uloga partitionera.

Partitioner generiše heš vrednost za ključ reda. Ova heš vrednost se koristi kako bi se odredilo u koj čvor u klasteru će se taj red smestiti. Ovo je bitno jer omogućava efikasno raspoređivanje podataka širom klastera.

Važno je napomenuti da Cassandra omogućava zamenu partitionera prema potrebama. Na primer, ako želite implementirati svoj sopstveni algoritam raspodele podataka, to možete uraditi implementiranjem org.apache.cassandra.dht.IPartitioner i spuštanjem .class ili .jar datoteke u Cassandra-ov lib direktorijum.

Kada se konfiguriše Cassandra, možemo navesti koji partitioner želimo da koristimo. Podrazumevano, Cassandra nudi nekoliko opcija particionera, od kojih svaki ima svoje prednosti i mane. Važno je pažljivo odabrati particioner jer to može imati značajan uticaj na performanse i raspodelu podataka u vašem klasteru.

Primer kako se unose postavke za particioner u cassandra.yaml: partitioner: org.apache.cassandra.dht.RandomPartitioner

Jedna od ključnih prednosti particioniranja podataka je što omogućava klasteru da raste inkrementalno. Ono što svaki algoritam za particioniranje radi je da daje doslednu podelu podataka preko svih dostupnih čvorova. Token koji je dodeljen čvoru od strane particionera takođe određuje položaj čvora u prstenu. Pošto je particioner globalno podešavanje, bilo koji čvor u klasteru može izračunati koje čvorove treba potražiti na osnovu datog ključa reda. Ova sposobnost da se izračunaju čvorovi koji drže podatke, bez ikakvog znanja osim ključa reda, omogućava bilo kom čvoru da izračuna koji čvor treba proslediti zahteve.

Cassandra (od verzije 1.2) nudi tri particionera, i to:

RandomPartitioner

Random particioner je podrazumevani mehanizam particionisanja pre verzije 1.2 u Cassandri. Koristi MD5 heširanje kako bi generisao heš vrednosti za ključeve redova. Budući da se heševi generišu na nasumičan način, ne garantuje se nikakav određeni redosled. To znači da dva ključa redova koji su leksički bliski mogu biti dodeljeni različitim čvorovima. Ova nasumična dodela tokena ključevima čini particioner pogodnim za ravnomernu distribuciju ključeva među čvorovima. Na taj način, veoma je malo verovatno da će balansirani čvor ikada imati tačke preopterećenja.

Broj ključeva generisanih od strane Random particionera varira od 0 do 2^127 - 1. Dakle, za i-ti čvor u klasteru sa ukupno N čvorova, početni token za taj čvor može se izračunati kao 2^127 \* (i - 1) / N.

Evo jednostavnog Python koda koji generiše kompletan niz početnih tokena za Random particioner u klasteru od osam čvorova:

čvorovi = 8

print("\n".join(["Čvor #" + str(i+1) + ": " + str((2\*\*127)\*i/čvorovi)

for i in xrange(čvorovi)]))

Čvor #1: 0

Čvor #2: 21267647932558653966460912964485513216

Čvor #3: 42535295865117307932921825928971026432

Čvor #4: 63802943797675961899382738893456539648

Čvor #5: 85070591730234615865843651857942052864

Čvor #6: 106338239662793269832304564822427566080

Čvor #7: 127605887595351923798765477786913079296

Čvor #8: 148873535527910577765226390751398592512

ByteOrderPartitioner

Byte-ordered partitioner generiše tokene za ključeve redova koji su uređeni prema heksadecimalnoj reprezentaciji ključa. Ovaj particioner omogućava redovima da budu poređani po heksadecimalnim vrednostima ključeva redova, što znači da će redovi biti sortirani po ključevima kada se iterira kroz njih. Međutim, ova prednost dolazi sa glavnim nedostatkom: hotspots. Razlog nastanka hotspots leži u neujednačenoj distribuciji podataka preko klastera. Na primer, ako imate klaster sa 26 čvorova i koristite ByteOrderedPartitioner, gde je svaki čvor odgovoran za jedno slovo abecede. Dakle, prvi čvor je odgovoran za sve ključeve koji počinju sa slovom A, drugi za B, i tako dalje. Kolona porodice koja koristi korisnička imena kao ključeve redova imala bi neujednačenu distribuciju podataka preko prstena. Distribucija podataka bi bila narušena sa čvorovima X, Q i Z koji su veoma slabo opterećeni, dok su čvorovi A i S veoma opterećeni.

Dakle, najbolji način da odredite početne tokene za klaster koji koristi ByteOrderedPartitioner jeste da analizirate uzorak podataka koji planirate da skladištite. Potrebno je identifikovati ključeve redova koji bi omogućili ravnomernu raspodelu podataka. To znači da treba izabrati ključeve koji će osigurati da svaki čvor u klasteru bude podjednako opterećen, kako bi se izbegla neravnoteža u opterećenju između čvorova.

Uzmimo hipotetički slučaj gdje se ključevi svih ključnih prostora mogu predstaviti petokarakternim nizovima od "00000" do "zzzzz". Evo kako generišemo početne tokene u Pythonu:

>>> start = int("00000".encode('hex'), 16)

>>> end = int("zzzzz".encode('hex'), 16)

>>> range = end - start

>>> nodes = 8

>>> print("\n".join(["Node #" + str(i+1) + ": %032x" % (start + range\*i/nodes) for i in xrange(nodes)]))

Node #1: 00000000000000000000003030303030

Node #2: 00000000000000000000003979797979

Node #3: 000000000000000000000042c2c2c2c2

Node #4: 00000000000000000000004c0c0c0c0b

Node #5: 00000000000000000000005555555555

Node #6: 00000000000000000000005e9e9e9e9e

Node #7: 000000000000000000000067e7e7e7e7

Node #8: 00000000000000000000007131313130

Kako bismo to postigli, koristimo Python kod koji koristi funkciju encode('hex') kako bismo pretvorili petokarakterne nizove u njihove heksadecimalne reprezentacije. Zatim, te heksadecimalne vrednosti pretvaramo u celi brojevi koristeći funkciju int(..., 16). Nakon toga, računamo opseg mogućih vrednosti za tokene, što je razlika između najmanje i najveće vrednosti

Zatim određujemo koliko čvorova imamo u klasteru (u ovom slučaju 8). Koristeći formulu (start + opseg\*i/čvorovi), izračunavamo početne tokene za svaki čvor. Na kraju, ispisujemo te početne tokene za svaki čvor.

Važno je napomenuti da je ovo samo primer kako se mogu generisati početni tokeni. U stvarnom slučaju, odluka o tome koje tokene dodijeliti treba biti donesena nakon analize i evaluacije podataka ili korištenjem drugih strategija

Murmur3Partitioner

Murmur3 particioner je novi podrazumevani alat za raspodelu podataka u Cassandra sistemu verzije 1.2 ili novije. Kada započinjete novi klaster, preporučljivo je koristiti Murmur3 particioner. Ovaj particioner ne čuva redosled i ima sve osobine Random particionera, ali je brži i pruža bolje performanse od Random particionera.

Da bismo generisali početne tokene, koristićemo poznatu formulu, ali ovaj put početna pozicija nije nula, pa je opseg tokena (kraj minus početak): +263 - (-263) = 264. Evo jednostavnog Python skripta za to:

>>> čvorovi = 8

>>> print("\n".join(["Čvor #" + str(i+1) + ": " + str(-(2\*\*63) + (2\*\*64)\*i/čvorovi) for i in range(čvorovi)]))

Čvor #1: -9223372036854775808

Čvor #2: -6917529027641081856

Čvor #3: -4611686018427387904

Čvor #4: -2305843009213693952

Čvor #5: 0

Čvor #6: 2305843009213693952

Čvor #7: 4611686018427387904

Čvor #8: 6917529027641081856

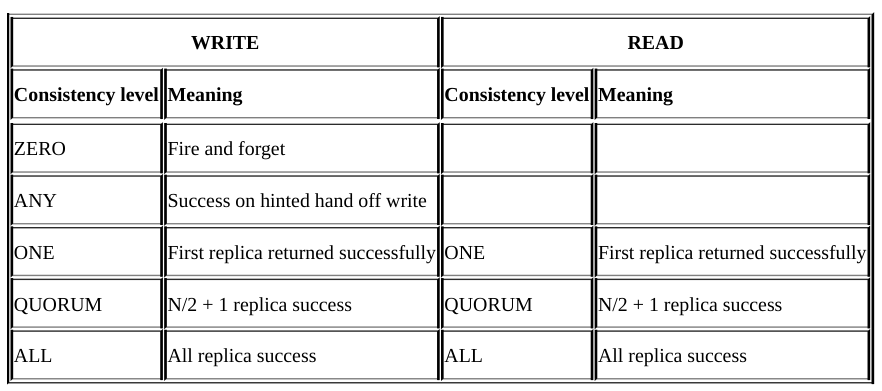
## Replikacija

Replikacija podataka u Cassandri je ključni mehanizam koji omogućava sistemu da bude otporan na greške i da nema tačku kvara. Kako bi se osigurala dostupnost podataka čak i ako neki čvor postane nedostupan ili prestane sa radom, podaci se replikuju na više od jednog čvora. Ovo pruža karakteristike kao što su tolerancija na greške i odsustvo jedne tačke otkaza.

Kada je u pitanju replikacija, Cassandra pruža nekoliko strategija, a faktor replikacije određuje koliko kopija podataka će biti napravljeno. Na primer, ako je faktor replikacije postavljen na 3, to znači da će svaki podatak imati tri kopije raspoređene na različitim čvorovima u klasteru. Prva replika će uvek biti smeštena na čvoru koji pokriva odgovarajući opseg tokena, dok se preostale replike smeštaju u skladu sa strategijom replikacije.

Važno je razumeti da nivo konzistentnosti (CL) u Cassandri definiše koliko replika mora odgovoriti da bi se operacija smatrala uspešnom. Na primer, ako je nivo konzistentnosti postavljen na 3 za čitanje, to znači da će klijent dobiti uspešan odgovor tek kada tri replike potvrde da imaju tražene podatke. Isto važi i za pisanje.

Treba napomenuti da veći nivo konzistentnosti obično rezultuje sporijim operacijama, ali istovremeno obezbeđuje veću pouzdanost i doslednost sistema. Cassandra nudi niz nivoa konzistentnosti, uključujući "fire and forget", CL ZERO, i garantuje sve operacije replikacije (čitanje i pisanje). Sledeća tabela prikazuje sve nivoe konzistentnosti:



Nivo konzistentnosti "Zero" ne zahteva nikakvu konzistentnost, što znači da Cassandra ne garantuje konzistentnost podataka nakon pisanja ili čitanja. Pisanje se smatra uspešnim čim se podaci pošalju čvoru, bez obzira na to da li su zapisani na disku. Ovo omogućava ekstremnu brzinu operacija, ali podaci mogu biti nekonzistentni između različitih čvorova u klasteru.

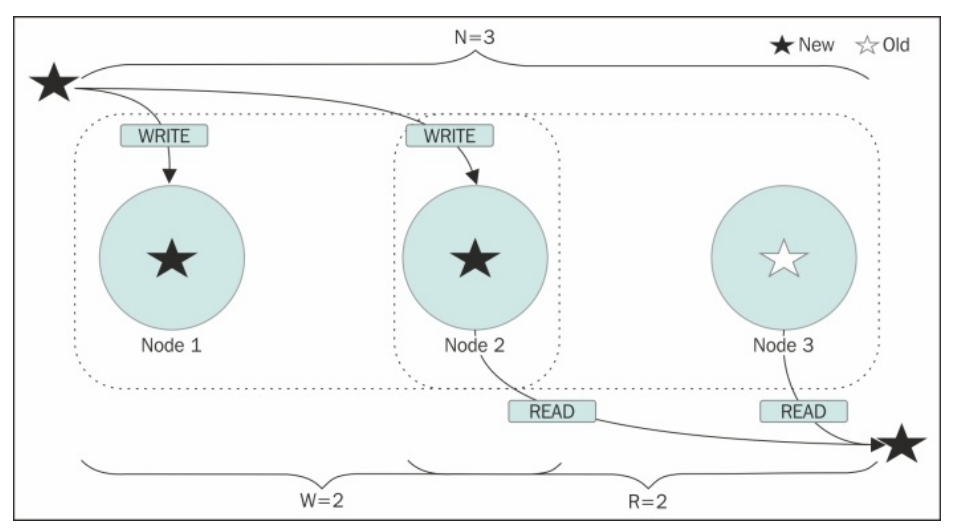
Nivo “Any" zahteva potvrdu samo od bilo kog dostupnog čvora u klasteru, dok "One" zahteva potvrdu samo od jednog određenog čvora. Oba nivoa pružaju bržu konzistentnost od "Zero", ali još uvek mogu dovesti do nekonzistentnosti podataka između čvorova.

Nivo "Quorum" zahteva potvrdu od većine replika u klasteru, što osigurava konzistentnost podataka i otpornost na kvarove. Ovo je popularan izbor za većinu aplikacija, pružajući balans između konzistentnosti, dostupnosti i otpornosti na kvarove.

Nivo "All" zahteva potvrdu svih replika u klasteru i pruža maksimalnu konzistentnost, ali može povećati latenciju i smanjiti dostupnost zbog potrebe da se svi čvorovi sinhronizuju pre nego što se upit smatra uspešnim.

### R + W > N nejednakost

Zamislite da je vrednost vašeg faktora replikacije tri. To znači da će vaši podaci biti smešteni na tri čvora. Ako imate nivo konzistentnosti pri pisanju jedan, a nivo konzistentnosti pri čitanju takođe jedan, oni mogu biti ili ne biti konzistentni. Evo zašto: kada se desi upis, informacije o mutaciji reda se šalju svim čvorovima, ali korisniku se vraća poruka o uspehu čim prva replika odgovori sa porukom o uspehu. U međuvremenu, podaci se pišu na još dva čvora. Ako zahtev za čitanje stigne na ta dva čvora sa nivoom konzistentnosti jedan, oni bi vratili zastarele podatke. Ili, ako je reč o scenariju intenzivnog pisanja i čitanja, sva tri čvora mogu imati različite podatke u nekom trenutku, a čitanje sa CL=1 može rezultirati nekonzistentnim čitanjima tokom veoma kratkog vremena. Sledeća slika prikazuje čitanja i upise, u sistemu R + W > N:



*Slika 4. Primer R + W > N nejednakost*

Ovde dolazi do koncepta slabe i jake konzistentnosti. Slaba konzistentnost je kada čitanja mogu biti netačna tokom kratkog vremenskog perioda, dok je jaka konzistentnost kada su rezultati uvek dosledni. U osnovi, slaba konzistentnost ponekad vraća netačne rezultate. Ako imate N replika, kako bi se osiguralo da čitanja uvek daju najnoviju vrednost, potrebno je pisati i čitati sa što više čvorova koji osiguravaju da se barem jedan čvor preklapa. Dakle, ako se piše na W čvorova i čita sa R čvorova tako da je R+W > N, mora postojati barem jedan čvor koji je zajednički za čitanje i pisanje. I to će osigurati da imate najnovije podatke.

Na slici 4, nivoi konzistentnosti ZERO i ANY su slaba konzistentnost. ALL je jaka konzistentnost. JEDAN za čitanje i ALL za pisanje, ili obrnuto, napraviće sistem koji je snažno dosledan. Sistem sa KVORUM-om za oba, čitanje i pisanje, je snažno dosledan sistem. Ponovo, ideja je osigurati da između čitanja i pisanja, barem jedan čvor se preklapa.

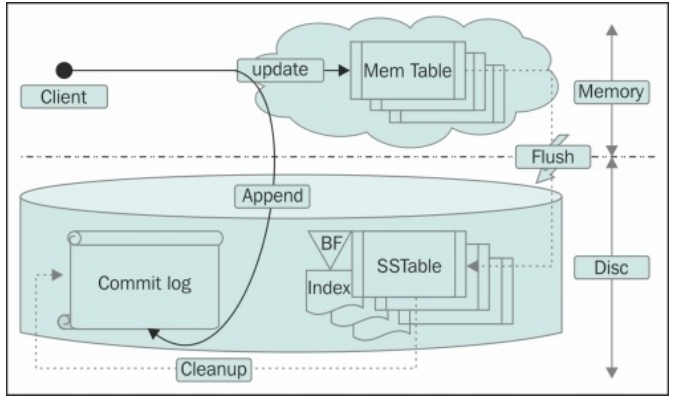
## Commit log

Jedno od osobina Cassandre je da krajnjim korisnicima garantuje trajnost. Trajnost osigurava da će svaka uspešna transakcija (poput upisa ili ažuriranja) biti trajno sačuvana, čak i ako dođe do sistemske greške. Drugim rečima, kada Cassandra potvrdi da je upis uspeo, to znači da su podaci sigurno sačuvani i da će preživeti eventualne probleme u radu sistema. Ovaj proces ostvaren je na isti način kao i kod drugih sistema za upravljanje bazama podataka koji garantuju trajnost: informacije o transakcijama se zapisuju u posebnu datoteku pre nego što se potvrdi da je upis uspešan. U Cassandra bazi podataka ta datoteka se naziva commit log.

Svaki put kada se izvrši upis na čvoru, taj proces prati org.apache.cassandra.db.commitlog.CommitLog. Taj softver zapisuje podatke zajedno sa određenim metapodacima u datoteku commit loga na način koji omogućava reprodukciju podataka. Osnovni cilj ovog postupka je da se garantuje da neće doći do gubitka podataka. Ukoliko, iz bilo kog razloga, podaci ne budu mogli da budu smešteni u MemTable ili SSTable, sistem može ponovo reprodukovati commit log kako bi rekonstruisao izgubljene podatke.

Commit log, MemTable i SSTable na čvoru su usko povezani. Svaka operacija upisa prvo se evidentira u commit logu, a zatim se MemTable ažurira. MemTable, po određenim kriterijumima, se zatim isprazni na disk u neizmenljive datoteke nazvane SSTable. Nakon što se podaci u MemTable-u isprazne u SSTable, podaci u commit logovima se brišu.

Takođe, svaki čvor servera ima samo jedan commit log. Kao i kod drugih mehanizama za zapisivanje, commit log je podešen da se "preliva" nakon što dostigne određenu veličinu.



*Slika 5. Commit log, MemTable i SSTable*

Sve klase koje upravljaju commit logom nalaze se u paketu org.apache.cassandra.db.commitlog. Singleton. Implementacije ICommitLogExecutorService su odgovorne za pisanje komandi u datoteku commit loga. Klasa CommitLogSegment predstavlja deo sistema koji upravlja pojedinačnom datotekom commit loga. Kada se vrše upisi u bazu podataka, ova klasa zapisuje te upise u commit log datoteku. Ono što je posebno interesantno kod ove klase je svojstvo cfLastWrite.

cfLastWrite je mapa koja čuva informacije o poslednjem zapisu (mutaciji) za svaku kolonu u bazi podataka. Svaki ključ u mapi predstavlja ime kolone, dok vrednost predstavlja celobrojnu vrednost koja označava poziciju (pomak) u datoteci commit loga gde je poslednja mutacija za tu kolonu zapisana. Ova mapa funkcioniše kao kursor koji prati gde se poslednji podaci za svaku kolonu nalaze u commit logu.

Kada se MemTable isprazni, odnosno kada se podaci iz MemTable-a prebace u SSTable, segmenti commit loga koji sadrže te mutacije označavaju se kao čisti za tu određenu kolonu. To znači da se više ne očekuje da se ovi podaci menjaju. Kada novi upis stigne, on se označava kao prljav sa pomakom na najnoviju mutaciju. Drugim rečima, novi upis se beleži kao "prljav" jer je nov i još uvek nije integrisan u glavne podatke, ali se zna gde se tačno u commit logu nalazi.

U slučaju neuspeha (kvara hardvera, iznenadnog gašenja), commit log pomaže sistemu da se oporavi na sledeći način:

1. Svaki segment commit loga se iterira u rastućem redosledu vremenskih oznaka kako bi se osiguralo da se pravilno reprodukuju podaci.
2. ReplayPosition je pozicija u commit logu koja označava tačku do koje su podaci već smešteni u SSTable-ima. Sistem bira najnižu ReplayPosition iz metapodataka SSTable-a kako bi odredio tačno gde se nalaze najnoviji podaci u commit logu.
3. Ulazi zapisa (mutacije) se reprodukuju za svaku kolonu ako je pozicija zapisa veća od pozicije reprodukcije u najnovijim metapodacima SSTable-a. Ovo osigurava da se samo najnoviji podaci integrišu u bazu podataka nakon oporavka.
4. Nakon što se reprodukcija zapisa završi, svi MemTable-i (memorijske tabele) se prinudno isprazne na disk kako bi se osiguralo da su svi podaci trajno sačuvani. Takođe, svi segmenti commit loga koji su korišćeni u procesu oporavka se recikliraju ili čiste kako bi se oslobodio prostor za nove zapise.

## MemTable

MemTable je struktura podataka koja postoji u radnoj memoriji računara i koristi se za brzo čuvanje i pristupanje podacima iz određene column family. Mogu se zamisliti kao keširani podaci koji su organizovani po ključu. Svaki unos u MemTable je sortiran po ključu reda. Za razliku od commit loga, koji samo dodaje nove unose, MemTable ne dozvoljava duplicirane unose. Ako se novi zapis sa istim ključem upiše u MemTable, on će zameniti stari zapis. Ova memorijska struktura je izuzetno brza i efikasna.

Na primer:

Upis 1:

{k1: [{c1, v1}, {c2, v2}, {c3, v3}]}

U CommitLog-u (novi unos, dodavanje):

{k1: [{c1, v1}, {c2, v2}, {c3, v3}]}

U MemTable-u (novi unos, dodavanje):

{k1: [{c1, v1}, {c2, v2}, {c3, v3}]}

Upis 2:

{k2: [{c4, v4}]}

U CommitLog-u (novi unos, dodavanje):

{k1: [{c1, v1}, {c2, v2}, {c3, v3}]}

{k2: [{c4, v4}]}

U MemTable-u (novi unos, dodavanje):

{k1: [{c1, v1}, {c2, v2}, {c3, v3}]}

{k2: [{c4, v4}]}

Upis 3:

{k1: [{c1, v5}, {c6, v6}]}

U CommitLog-u (stari unos, dodavanje):

{k1: [{c1, v1}, {c2, v2}, {c3, v3}]}

{k2: [{c4, v4}]}

{k1: [{c1, v5}, {c6, v6}]}

U MemTable-u (stari unos, ažuriranje):

{k1: [{c1, v5}, {c2, v2}, {c3, v3}, {c6, v6}]}

{k2: [{c4, v4}]}

## SSTable

SSTable je način čuvanja podataka na disku. Kada MemTable prebacimo na disk, ona postaje neizmenljiva SSTable. Podaci iz MemTablea se smeštaju u pojedinačne SSTable datoteke, a sve operacije upisa se izvršavaju jedna za drugom, što čini proces brzim. Drugim rečima, što je brži disk, to je brže ispraznjenje MemTablea na disk.

Kasnije, SSTable datoteke se spajaju u procesu kompaktiranja, pri čemu se podaci organizuju u jednu datoteku. Ovaj dodatni korak kompaktiranja ima koristi prilikom čitanja podataka.

SSTable datoteke se sastoje od tri osnovne komponente: bloom filtera, index files i data files.

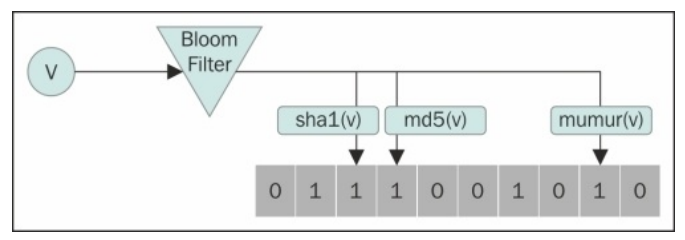
### Bloom filter

Bloom filter je test koji se koristi da bi se proverila dostupnost određenih podataka u skladištu. Za razliku od klasičnog testa, bloom filter može rezultirati lažno pozitivnim rezultatima; to jest, može tvrditi da podaci postoje u kolekciji povezanoj sa bloom filterom, iako zapravo ne postoje. Međutim, bloom filter nikada ne rezultira lažno negativnim rezultatima; to jest, nikada ne tvrdi da podaci nisu prisutni kada jesu. Razlog za korišćenje bloom filtera, uprkos njegovim nedostacima u lažno pozitivnim rezultatima, je što je veoma brz i njegova implementacija je veoma jednostavna.

Cassandra koristi bloom filtere kako bi odredila da li određeni ključ reda ima podatke u određenoj SSTable datoteci. Bloom filteri su efikasni za ove provere prisustva, ali nisu pogodni za precizno određivanje opsega ili za detaljne pretrage podataka unutar tog opsega. Ovo smanjuje broj ulazno-izlaznih operacija na disku koje bi bile potrebne za potpuno skeniranje SSTable-a, što je spor proces. Zato ih Cassandra koristi kako bi se izbeglo čitanje mnogih SSTable datoteka, što bi moglo dovesti do uskog grla.

Kako radi bloom filter:

Bloomov filter je struktura podataka koja se koristi za brzu proveru prisustva određenih elemenata u skupu podataka. U osnovi, Bloomov filter je niz bitova dužine "l", gde su svi bitovi inicijalno postavljeni na nulu. Pored toga, Bloomov filter koristi "k" predefinisanih heš funkcija.



*Slika 6. Bloom filter*

Kada se ključ dodaje u Bloomov filter, koristi se svih "k" heš funkcija da se izračuna heš vrednost ključa. Zatim se za svaku dobijenu heš vrednost uzima ostatak pri deljenju sa dužinom niza "l", i ta pozicija u nizu se postavlja na 1. Ovaj proces se ponavlja za sve "k" heš funkcije i rezultuje postavljanjem određenih bitova na vrednost 1 u nizu.

Sledeći pseudokod objašnjava šta se dešava:

//calculate hash, mod it to get location in bit array

arrayIndex1 = md5(v) % arrayLength

arrayIndex2 = sha1(v) % arrayLength

arrayindex3 = murmur(v) % arrayLength

//set all those indexes to 1

bitArray[arrayIndex1]=1

bitArray[arrayIndex2]=1

bitArray[arrayIndex3]=1

Da bi se proverili postojanje određenog ključa u Bloomovom filteru, sledi se sličan proces kao prilikom dodavanja ključa. Prvo, se primenjujemo sve predefinisane heš funkcije na tom ključu. Zatim, se uzima ostatak dobijenih heš vrednosti pri deljenju sa dužinom niza bitova. Nakon toga, se gleda na tim pozicijama u nizu.

Ako se ispostavi da makar jedna od tih lokacija u nizu ima vrednost nula, to znači da ovaj ključ sigurno nikada nije bio dodat u ovaj Bloomov filter, i stoga ne postoji u povezanoj kolekciji podataka. Sa druge strane, ako su sve vrednosti na tim lokacijama 1, to znači da taj ključ možda postoji u kolekciji podataka povezanoj sa ovim Bloomovim filterom. Međutim, nije moguće garantovati da se ključ zaista nalazi u toj kolekciji, jer je moguće da postoje drugi ključevi koji su popunili iste pozicije u nizu bitova koristeći druge heš funkcije.

U originalnoj verziji Bloomovog filtera, nije moguće direktno ukloniti ključ iz filtera. To je zbog toga što više ključeva može rezultirati istim indeksom bita postavljenim na 1 u nizu, koristeći različite heš funkcije. Ovo može dovesti do situacije u kojoj uklanjanje jednog ključa može uticati i na druge ključeve koji dele isti indeks bita.

Međutim, ovaj problem se može rešiti upotrebom niza celobrojnih vrednosti za Bloomov filtera. Dakle, umesto niza bitova, koristi se niz celih brojeva gde svaki element funkcioniše kao brojač. Kada se ključ dodaje, brojač se povećava, a kada se ključ uklanja, brojač se smanjuje. Ovo omogućava preciznije upravljanje ključevima i smanjuje mogućnost neželjenih efekata prilikom brisanja ključeva iz Bloomovog filtera.

Efikasnost bloom filtera zavisi od veličine kolekcije na koju se primenjuje. Što je kolekcija veća, to će biti veća verovatnoća lažno pozitivnih rezultata, jer će niz bitova biti gušće popunjen jedinicama. Još jedan faktor koji utiče na performanse bloom filtera je kvalitet heš funkcije koja se koristi. Dobra heš funkcija će ravnomerno rasporediti heš vrednosti u nizu i biti brza. U ovom slučaju, ne uzima se u obzir kriptična snaga heš funkcije, pa će Murmur3 heš funkcija biti više preferirana od SHA1 heš funkcije.

### Index files

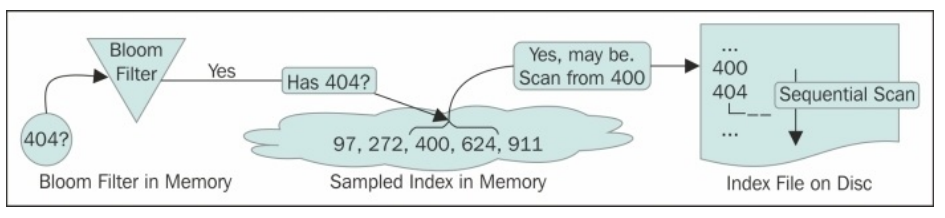
Index datoteke su prateće datoteke SSTabela. Slično kao i bloom filteru, postoji jedna index datoteka po SSTabeli. Ona sadrži sve ključeve redova u SSTabeli, a njen ofset je tačka gde počinje red u data fajlu.

Prilikom pokretanja, Cassandra čita svaki 128. ključ (konfigurabilno) u memoriju (sampled indeks). Kada index traži redni ključ (nakon što je bloom filter dao naznaku da se redni ključ možda nalazi u ovoj SSTabeli), Cassandra vrši binarno pretraživanje sampled indeksa u memoriji. Nakon pozitivnog rezultata binarnog pretraživanja, Cassandra će morati da pročita blok u index datoteci sa diska, počevši od najbliže vrednosti manje od vrednosti koju tražimo.

Uzmimo primer u kojem Cassandra traži redni ključ 404. Kada Cassandra prvo proveri bloom filter za određenu SSTabelu, dobije pozitivan signal, što znači da ta SSTabela možda sadrži traženi redni ključ. Nakon toga, umesto da odmah pretraži SSTabelu ili index datoteku, Cassandra koristi sampled indeks koji se nalazi u memoriji kako bi dobila neku vrstu predstave o sadržaju SSTabele.

Pregledanjem sampled indeksa, Cassandra primeti da postoji redni ključ 400 i još jedan ključ, 624. To sugeriše da fragmenti redova mogu biti prisutni u ovoj SSTabeli. Ali, još važnije, sampled indeks daje informaciju o ofsetu za unos koji počinje sa ključem 400 u index datoteci.

Nakon toga, Cassandra skenira SSTabelu počevši od ofseta za ključ 400 i pronalazi unos za ključ 404. To omogućava Cassandri da locira tačan ofset unosa za ključ 404 u SSTabeli i da odatle pročita željene podatke.



*Slika 7. Primer korišćenja index fajla*

### Data files

Data datoteke su zapravo podaci. One sadrže redne ključeve, metapodatke i kolone (delimične ili potpune). Čitanje podataka iz data datoteka zahteva samo jedno traženje na disku, nakon čega sledi sekvencijalno čitanje, jer je ofset ka rednom ključu već dobijen iz pripadajuće index datoteke.

### Compaction

Zahtev za čitanjem može zahtevati od Cassandre da čita preko više SSTabela kako bi dobila rezultat. Ovo je neefikasno, košta više (diskovnih) traženja, može zahtevati rešavanje konflikata, i ako ima previše SSTabela, može usporiti čitanje. Da bi rešila ovaj problem, Cassandra ima proces nazvan kompaktiranje (Compaction).

Kompaktiranje spaja više SSTabela u jednu. Cassandra nudi dva tipa mehanizama za kompaktiranje: strategiju kompaktiranja po veličini nivoa i strategiju kompaktiranja po nivoima. Ovaj odeljak ostaje fokusiran na mehanizam kompaktiranja po veličini nivoa radi boljeg razumevanja.

Proces kompaktiranja počinje kada broj SSTabela na disku dostigne određeni prag (koji se može konfigurisati). Iako je proces spajanja malo intenzivan u pogledu I/O operacija, dugoročno donosi koristi smanjenjem broja traženja na disku tokom čitanja. Pored toga, postoji nekoliko drugih koristi od kompaktiranja, kao što su:

* Uklanjanje isteklih tombstone-ova (Cassandra v0.8+)
* Spajanje fragmenata redova
* Rekonstrukcija primarnih i sekundarnih indeksa

Kada se govori o spajanju SSTabela, važno je imati na umu da su SSTabele već sortirane po rednim ključevima unutar sebe. Kada se SSTabele spajaju, ova sortiranost omogućava efikasno spajanje podataka. Pretpostavimo da je prag za kompaktiranje postavljen na četiri, što znači da će Cassandra započeti proces spajanja kada broj SSTabela dostigne ovaj prag. Kada se to dogodi, Cassandra uzima četiri SSTabele koje su jednake veličine i spaja ih u jednu veću SSTabelu.

Kratkoročno, dok se spajaju, na disku će privremeno biti dva puta više podataka nego što je ukupan zbir podataka u tim SSTabelama. Ovo se dešava zato što se podaci iz četiri manje SSTabele kombinuju u jednu veću, pa je prostor koji zauzimaju privremeno dupliran.

Važno je napomenuti da sve SSTabele koje se spajaju imaju istu veličinu kako bi proces spajanja bio konzistentan. Dakle, kada se četiri manje SSTabele spoje u jednu veću SSTabelu, rezultujuća nova SSTabela će biti iste veličine kao i zbir veličina svih četiri manjih SSTabela. To znači da će naredni proces kompaktiranja zauzeti još veći prostor prilikom spajanja, jer će se koristiti veća SSTabela kao osnova za spajanje novih podataka.

Nakon što se SSTabele spoje, obeležavaju se kao izbrisive, što znači da su spremne za brisanje. Ovo brisanje se obavlja tokom ciklusa sakupljanja smeća u JVM-u ili prilikom ponovnog pokretanja Cassandrea. Proces kompaktiranja se dešava na svakom čvoru i ne utiče na ostale čvorove. Ova vrsta kompaktiranja naziva se manje kompaktiranje, i automatski se pokreće, sistem ga kontroliše i dešava se redovno. Pored manjeg kompaktiranja, postoji još vrsta postavki za kompaktiranje u Cassandri, kao što je veliko kompaktiranje.

Veliko kompaktiranje je proces spajanja svih SSTabela u jednu jedinu SSTabelu. Iako može izgledati zbunjujuće što i manje kompaktiranje spaja SSTabele, postoji razlika. Na primer, ako koristimo strategiju kompaktiranja po veličini nivoa, ta strategija spaja tabele koje su iste veličine. Dakle, ako je prag postavljen na četiri, Cassandra će započeti spajanje kada identifikuje četiri SSTabele iste veličine. Ako vaš sistem počne sa četiri SSTabele veličine X, nakon kompaktiranja završićete sa jednom SSTabelom veličine 4X. Sledeći put kada imate četiri SSTabele veličine X, završićete sa dve SSTabele veličine 4X, i tako dalje. Nakon nekog vremena, imaćete nekoliko velikih SSTabela, nekoliko srednjih i mnogo manjih. Ovo je posledica stalnog manjeg kompaktiranja. Dakle, možda će biti potrebno pretraživanje više SSTabela da biste dobili podatke za upit. Kada pokrenete veliko kompaktiranje, sve velike i male SSTabele se spajaju u jednu. To je osnovna korist velikog kompaktiranja.

Nakon verzije Cassandre v0.8+, veliko kompaktiranje možda nije najbolja opcija iz nekoliko razloga. Jedan od razloga je taj što automatsko manje kompaktiranje više neće automatski pokrenuti nakon izvršenja velikog kompaktiranja. To znači da je potrebno ručno izvršavanje ili dodatni rad, kao što je postavljanje cron zadatka, kako bi se redovno obavljalo veliko kompaktiranje. Takođe, performanse se mogu pogoršati s vremenom nakon velikog kompaktiranja. Ovo je verovatno zbog činjenice da veće SSTabele, koje dobijamo nakon velikog kompaktiranja, imaju veću verovatnoću da dobiju više lažno pozitivnih rezultata bloom filtera. To može rezultirati dužim vremenom izvršavanja binarnog pretraživanja na indeksu, koji postaje veoma velik.

## Hinted handoff

Razmotrite sledeći scenario: zahtev za pisanje je poslat Cassandri, ali replika čvora gde pisanje pravilno pripada nije dostupna zbog particionisanja mreže, kvara hardvera ili nekog drugog razloga. Da bi se osigurala opšta dostupnost prstena u takvoj situaciji, Cassandra implementira funkciju nazvanu hinted handoff. Možete zamisliti hinted handoff kao malu belešku koja sadrži informacije iz zahteva za pisanje. Ako je replika čvora gde pisanje pripada nedostupna, koordinator će kreirati hinted handoff, koji je mali podsetnik koji kaže: "Imam informacije za pisanje koje su namenjene čvoru B. Zadržaću ovo pisanje, i primetiću kada čvor B ponovo postane dostupan; kada to učini, poslaću mu zahtev za pisanje." Drugim rečima, kada detektuje putem gossip protokola da je čvor B ponovo dostupan, čvor A će "predati" čvoru B "hinted handoff" u vezi sa pisanjem. Cassandra čuva posebnu hint za svaku particiju koja treba da bude zapisana.

Ovo omogućava Cassandri da uvek bude dostupna za pisanja, i generalno omogućava klasteru da podnese isto opterećenje pisanja čak i kada su neki čvorovi nedostupni. Takođe, smanjuje vreme koje će proći pre nego što se neispravan čvor vrati u doslednost nakon što ponovo postane dostupan. Uopšteno, naznake se ne računaju kao pisanja u svrhu nivoa doslednosti. Izuzetak je nivo doslednosti ANY, koji je dodat u verziji 0.6. Ovaj nivo doslednosti znači da će samo naznačena predaja biti dovoljna za uspeh operacije pisanja. Drugim rečima, čak i ako je zabeležena samo naznaka, pisanje se i dalje smatra uspešnim. Napomena da se pisanje smatra trajnim, ali podaci možda neće biti čitljivi dok se naznaka ne dostavi ciljnoj repliki.

Postoji praktičan problem sa hinted handoff (i pristupima zagarantovanoj dostavi, u stvari): ako je čvor neko vreme nedostupan, hintovi mogu značajno narasti na drugim čvorovima. Zatim, kada drugi čvorovi primete da se neispravan čvor ponovo vratio, obično preplave taj čvor zahtevima, baš u trenutku kada je najranjiviji (kada se muči da se vrati u igru nakon kvara). Da bi se rešio ovaj problem, Cassandra ograničava skladištenje hintova na konfigurabilni vremenski prozor. Takođe je moguće potpuno onemogućiti hinted handoff.

Kao što njeno ime sugeriše, org.apache.cassandra.hints.HintsService je klasa koja interno implementira hinted handoff.

Iako hinted handoff pomaže povećanju dostupnosti Cassandre, zbog pomenutih ograničenja sama po sebi nije dovoljna da garantuje doslednost podataka među replikama.

# Zaključak

U ovom radu smo detaljno istražili internu strukturu i organizaciju skladištenja u Apache Cassandra bazi podataka. Shvatili smo da je razumevanje ključnih komponenti kao što su čvorovi, virtuelni čvorovi, serveri, rackovi i klasteri, kao i procesi kao što su replikacija, kompaktiranje i nadomjestni prijenos, od suštinskog značaja za efikasno korišćenje i optimizaciju performansi Cassandra sistema.

Kroz analizu različitih particionera kao što su RandomPartitioner, ByteOrderPartitioner i Murmur3Partitioner, stekli smo uvid u načine kako Cassandra distribuira podatke i osigurava skalabilnost i doslednost. Takođe smo istražili ključne strukture podataka kao što su commit log, MemTable i SSTable, i razumeli kako ove komponente igraju ključnu ulogu u procesu zapisivanja i čitanja podataka.

Predstavili smo i napredne tehnike poput kompresije podataka, particioniranja i replikacije kao načine za optimizaciju performansi Cassandra baze podataka.

Na kraju, zaključujemo da je dublje razumevanje interne strukture i organizacije skladištenja u Apache Cassandra bazi podataka ključno za efikasno upravljanje podacima, unapređenje performansi i osiguranje pouzdanog rada u distribuiranim okruženjima.

# Literatura

1. Cassandra The Definitive Guid (Third edition), Jeff Carpenter and Eben Hewitt  
[9781492079514 (2).pdf (live-datastaxd8.pantheonsite.io)](https://live-datastaxd8.pantheonsite.io/sites/default/files/content/ebook/2020-04/9781492079514%20%282%29.pdf)

2. Mastering Apache Cassandra Second Edition, Nishant Neeraj

[Mastering-Apache-Cassandra-2nd-(2015).pdf (hoclaptrinhdanang.com)](https://hoclaptrinhdanang.com/downloads/pdf/devops/Mastering-Apache-Cassandra-2nd-(2015).pdf)

3. Apache Cassandr Architecture, Inside DataStax Distribution of Apache Cassandra

[DataStax-WP-Apache-Cassandra-Architecture\_Technical\_4\_2.pdf (apexassembly.com)](https://apexassembly.com/wp-content/uploads/2020/08/DataStax-WP-Apache-Cassandra-Architecture_Technical_4_2.pdf)

4. Cluster, Datacenters, Racks and Nodes in Cassandra

[Cluster, Datacenters, Racks and Nodes in Cassandra | Baeldung](https://www.baeldung.com/cassandra-cluster-datacenters-racks-nodes)