
# Apache Cassandra

## HIGH AVAILABILITY REŠENJA

Đorđe Nikolić | Sistemi za upravljanje bazama podataka | jun 2022.

# Sadržaj

# Uvod

U ovom radu će biti diskutovan koncept sistema baza podataka za visokom dostupnošću (engl. *High Availability*, u daljem tekstu HA). Biće govora o samoj definiciji datog koncepta, problema sa implementiranjem velike dostupnosti kod standardnih relacionih baza podataka, kao i načina putem kojih je moguće rešiti te probleme. Zatim, biće predstavljan sistem Apache Cassandra i njegove osnovne odlike, nakon čega će ostatak rada biti posvećen načinima i mehanizmima putem kojih ovaj sistem realizuje visoku dostupnost.

# Visoka dostupnost

Dostupnost predstavlja mogućnost korisnika da pristupi i koristi određeni servis ili resurs. [1] U slučaju da korisnik ne može koristiti servis u datom trenutku, on se smatra nedostupnim. Visoka dostupnost je sposobnost sistema da kontinualno funkcioniše, tj. bude dostupan korisnicima, bez prekida tokom određenog vremenskog perioda. Drugim rečima, prilikom dizajniranja HA sistema, fokus je na garantovanju visokog procenta dostupnosti tokom određenog vremenskog perioda, a zlatan standard u industriji je poznat kao „dostupnost pet devetki“, tj. sistem koji je dostupan 99.999% vremena. Modernizacija je dovela do toga da postoji veliko oslanjanje korisnika na pojedine servise, te je visoka dostupnost od veće važnosti nego ikada pre.

## PRINCIPI

Prilikom dizajniranja HA sistema postoje tri glavna principa koji se prate kako bi se postigla najviša moguća dostupnost [2]:

- **Eliminisanje tački propasti**: Tačka propasti je jedna komponenta koja bi u slučaju da otkaže, dovela do toga da ceo sistem propadne. Ako se servis pokreće samo sa jednog servera, onda je taj server tačka propasti. U slučaju da taj server prestane da radi, pada ceo servis.
- **Pouzdan prelaza**: Kreiranje redundantnosti u ovim sistemima je od velike važnosti. Redundantnost omogućava da rezervna komponenta preuzme u slučaju da glavna otkaže. U slučaju da dođe do toga, obavezno je obezbediti pouzdan prelaz sa komponente koja je otkazala, na rezervnu, bez gubitka podataka ili smanjenja dostupnosti.
- **Detekcija propasti**: Sistem mora biti sposoban da detektuje komponente koje otkažu, i pokrene određene procedure tako da nastavi da funkcioniše u potpunosti. Korisniku kvar ne sme biti vidljiv.

## PROBLEMI SA TRADICIONALNIM SISTEMIMA BAZA PODATAKA

Tradicionalno, bilo je teško razviti sisteme baza podataka koje odlikuje visoka dostupnost, a to je posebno tačno za relacione sistemi baza podataka koji su proteklih nekoliko decenije činili većinu tržišta [3]. Ovi sistemi su uglavnom dizajnirani da funkcionišu na jednoj mašini.

### ACID

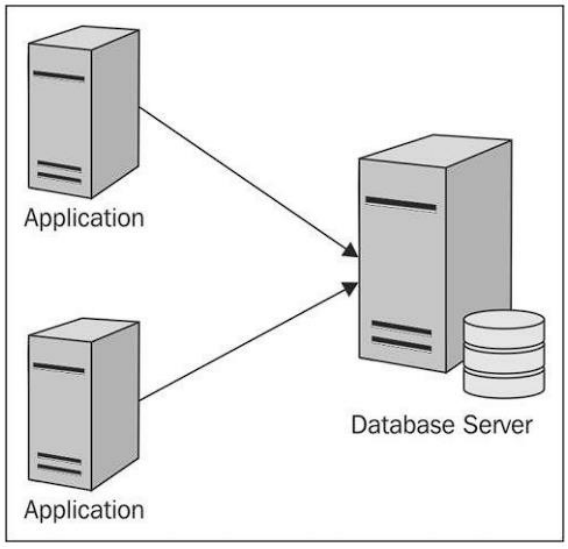
Jedna od najvećih prepreka ka ostvarivanju visoke dostupnosti kod tradicionalnih sistema baza podataka je to što pokušavaju da striktno garantuju ACID svojstva:

- **Atomičnost** - izvršavanje transakcije po metodi „sve ili ništa“
- **Konzistentnost** – održavanje integriteta podataka u svim kopijama
- **Izolovanost** – održavanje pravilnog redosleda izvršenja transakcija
- **Trajnost** – svi upisi se čuvaju na trajnom medijumu podataka

Arhitekte uglavnom ostvaruju ova svojstva kroz komplikovane mehanizme koji dovode do žrtvovanja dostupnosti. Kao rezultat toga, pokušaji postizanja visoke dostupnosti zahtevaju implementaciju dodatnih mehanizma preko samog sistema, sa ciljem da se originalna svojstva očuvaju.

### Monolitska arhitektura

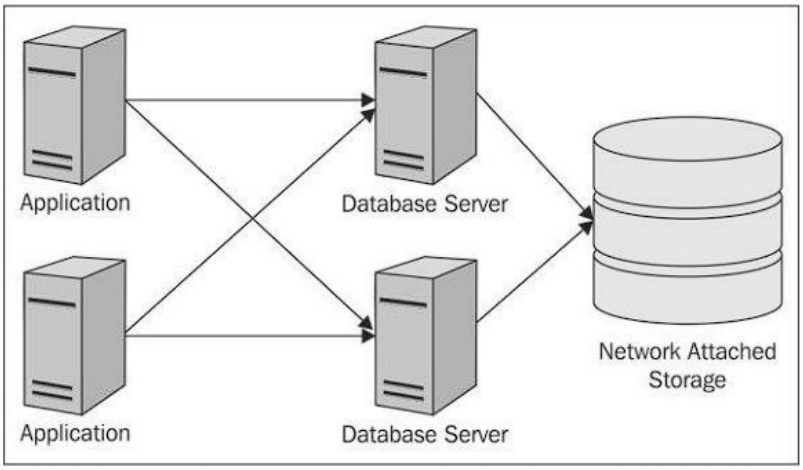
Najprostiji dizajn koji može da garantuje ACID svojstva uključuje monolitsku arhitekturu kod koje sve funkcionalnosti postoje samo na jednoj mašini. Sa obzirom na to da ne postoji nikakva komunikacija između više čvorova, poštovanje ACID pravila nije komplikovano.




Slika 1. Primer proste monolitske arhitekture servisa [3]

Povećavanje dostupnost u takvim sistemima se često postiže uvođenjem poboljšanja na hardverskom nivou, kroz RAID diskove, više mrežnih interfejsa, itd. Doduše, ništa od toga ne menja činjenicu da server ostaje tačka propasti. Ako server otkaže, servis postaje nedostupan.

Jedna od čestih metoda povećanja kapaciteta obrade zahteva kod ovakve arhitekture je prebacivanje sloja skladištenja na jednu deljenu komponentu, kao što su NAS (engl. *network attached storage*) i SAN (engl. *storage attached network*) mreže. Takve komponente su uglavnom veoma pouzdane, sa velikim brojem diskova i mrežnih interfejsa. Kao što se da primetiti, iako ovakav pristup omogućava veći kapacitet obrade zahteva, i samim tim povećava dostupnost u slučaju velikog priliva zahteva, tačka propasti i dalje postoji u formi sloja skladištenja.



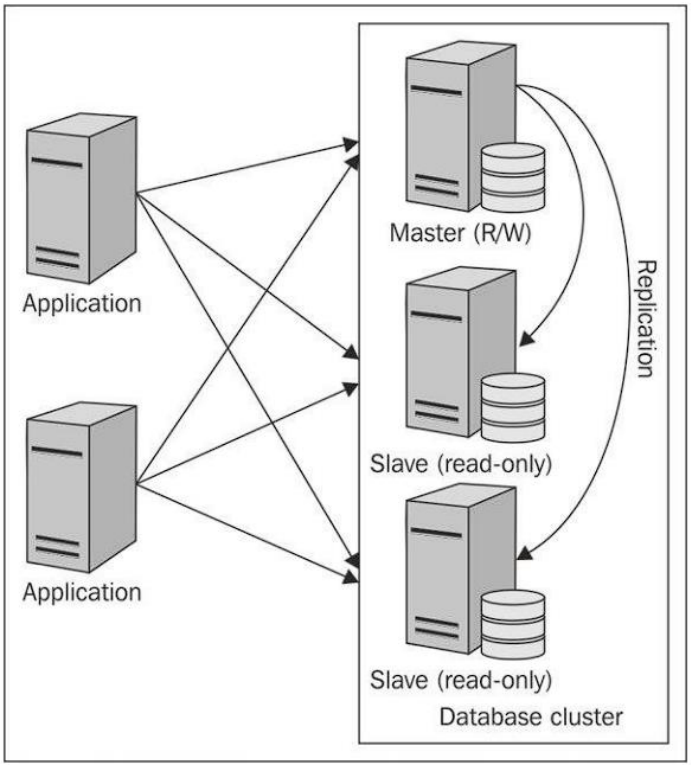

Slika 2. Primer više servera sistema baza podataka povezanih na NAS [3]

## Master-slave arhitektura

Kako su distribuirani sistemi postavi sve više prisutni, potreba za distribuiranim sistemima visokog kapaciteta je porasla. Mnoge distribuirane baze podataka i dalje pokušavaju da održe ACID svojstva (a u nekim slučajevima, samo konzistentnost, koje je i najteže svojstvo za postići u distribuiranom okruženju). Ovo je dovelo do kreiranje master-slave arhitektura.

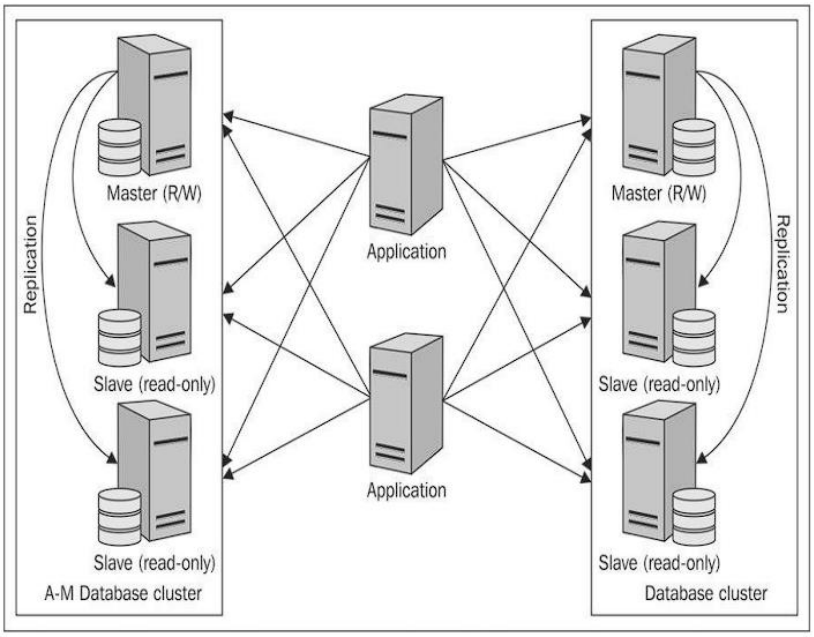
Kod ovakvih sistema, iako mogu postojati mnogo servera koji obrađuju zahteve, samo jedan server može vršiti upise, kako bi podaci ostali konzistentni. Ovim izbegavamo scenario u kome isti podatak može biti izmenjen kroz više čvorova istovremenim zahtevima.

Ovim pristupom i dalje nije rešen problem dostupnosti, jer ako server preko koga se vrše upisi otkaže, dostupnost servisa pada. Takođe, obrada velikog broja upisa postaje potencijalno neperformantna, jer se svi upisi šalju na jedan server.




Slika 3. Primer master-slave arhitekture [3]

## Sharding




Slika 4. Primer master-slave arhitekture sa primenjenom *sharding* tehnikom [3]

Varijacija na master-slave arhitekturu koja omogućava bolju obradu velike količine zahteva za upis je tehnika podele podataka na delove gde svaki deo pripada jednoj grupi servera koji prate master-slave arhitekturu (engl. *sharding*). Na primer, baza korisničkih profila može biti podeljena po prezimenu korisnika, tako da svi korisnici čija prezimena počinju slovima A-M pripadaju jednoj grupi servera, tj. jednom klasteru, dok svi korisnici čija prezimena počinju slovima N-Z pripadaju drugom klasteru.

Mane ovog pristupa su brojne, a prva je to što se unose više tački propasti, gde svaki master server predstavlja jednu od njih. Takođe, odgovornost upravljanja kom klasteru treba poslati neki zahtev pada na aplikacioni sloj. Dodavanjem novih klastera, podaci se moraju izmešati tako da se ispoštuje nova podela.

Neki sistemi koji koriste ovakvu tehniku, imaju dodatan sloj apstrakcije između aplikacije i fizičkih klastera koji odgovornost upravljanja zahtevima ka odgovarajućim klasterima prebacuje sa aplikacije na sebe.

### Prelaz u slučaju otkazivanja master čvora

Jedan od čestih načina povećanja dostupnosti u slučajevima otkazivanja master čvora (koji predstavljaju tačke propasti kod ovog tipa arhitekture) je prebacivanje master odgovornosti na neki novi čvor, tj. tehnika izbora vođe. Tačna implementacija ovog algoritma se razlikuje od sistema do sistema ali je koncept isti. Uglavnom, primena ovog algoritma dovodi do povećanja dostupnosti sistema sa master-slave arhitekturom.

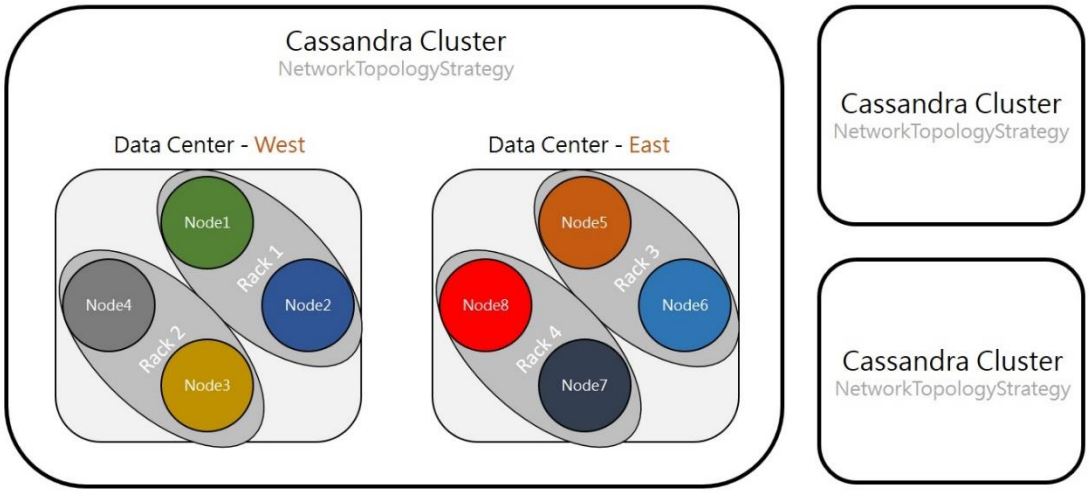
Čak i sistemi koji implementiraju gorepomenute tehnike, imaju veći broj mana:

- Aplikacije moraju da poznaju topologiju baze (ova mana se otklanja unosom novog sloja apstrakcije)
- Particije podataka se moraju pažljivo planirati
- Skaliranje obrade upisa je loše
- Prebacivanje master odgovornosti unosi dodatnu kompleksnost u sistem, posebno kod klastera koji se nalaze na više lokacija
- Dodavanje novih klastera zahteva novo razmeštanje podataka

# Apache Cassandra

Apache Cassandra je NoSQL (nerelaciona) distribuirana baza podataka koda otvorenog tipa čiji je glavni fokus na odličnoj skalabilnosti, otpornosti na greške, i dostupnosti. Bazirana je na Amazon Dynamo, i Google Bigtable sistemima, i originalno je razvijana od strane Facebook-a, iako je od jula 2008. godine celokupni kod sistema potpuno dostupan javnosti.

## ARHITEKTURA




Slika 5. Arhitektura sistema Apache Cassandra na visokom nivou [4]

Cassandra je organizovana po distribuiranim klasterima koji sadrže više homogenih čvorova (*nodes*), i po njima su podaci raspoređeni i organizovani tako da nikad ne postoji takozvana „jedna tačka propasti“ [5]. Čvorovi su u stalnoj međusobnoj komunikaciji i razmenjuju informacije o trenutnom stanju koristeći *peer-to-peer* komunikacioni protokol. Na svakom čvoru se održava sekvencijalni *commit log* koji beleži svaki upis podataka, koji se nakon toga indeksiraju i upisuju u strukturu u memoriji, nazvanu *memtable* (jedna za svaku porodicu kolona). Svaki put kada se ova struktura popuni, podaci se upisuju na disk u formi *SSTable* fajlova. Svaki proces upisa u bazu je praćen automatskim particionisanjem i replikacijom na čvorovima u klasteru. Sistem periodično konsolidira podatke u SSTable fajlovima kroz specijalan proces kompakcije (engl. compaction) putem kog takođe briše zastarele podatke označene za brisanjem *tombstone* objektom. Kako bi se osiguralo stanje konzistentnosti svih podataka, sistem koristi različite *repair* mehanizme.

Cassandra je particionisana baza podataka koji svoje podatke čuva u redovima. Redovi su organizovani u tabele (porodice kolona) sa obaveznim primarnim ključem. Pristup bazi se vrši kroz CQL jezik, koji koristi sličnu sintaksu kao i SQL, jezik popularan među relacionim bazama podataka. Jedna od najvećih razlika između ova dva jezika, kao i

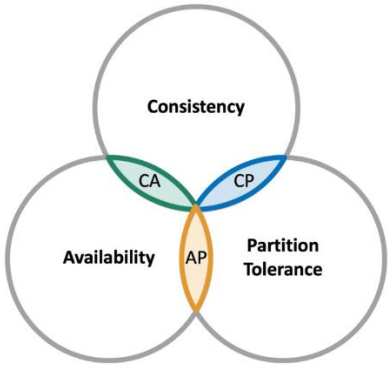
odlika Apache Cassandra sistema koji ga najviše razlikuje od relacionih sistema baza podataka, je to što koncept spajanja tabela ne postoji. Takođe, nemoguće je pisati pod-upite. Podaci se čuvaju u denormalizovanom formatu, i prilikom dizajniranja baze podataka za neku aplikaciju, mora se pristupiti drugačije nego prilikom dizajniranja relacione baze podataka za istu namenu.

Zahtevi za čitanje ili upis podataka mogu biti poslati na bilo koji čvor u klasteru. Kada se klijent poveže na neki od čvorova, taj čvor postaje koordinator za interakciju sa tim klijentom. Preko koordinatora se vrši komunikacija između klijenta i čvorova koji zapravo sadrže potraživane podatke. On takođe određuje, na osnovu konfiguracije klastera, kom čvoru treba da prosledi zahtev na procesiranje.

## ŠTA CASSANDRA GARANTUJE?

Apache Cassandra je sistem baza podataka koji je izuzetno skalabilan i pouzdan. Koristi se u veb aplikacijama koje uslužuju veliki broj korisnika i gde je red veličine podataka u petabajtima. Cassandra garantuje određena svojstva o skalabilnosti, dostupnost i pouzdanosti, i kako bi razumeli ta svojstva, a i ograničenja koja se rađaju sa takvim dizajnom baze podataka, bitna je CAP teorema. [6]

### CAP teorema




Slika 6. Vizualizacija CAP teoreme [7]

Po CAP teoremi, distribuirana baza podataka može maksimalno garantovati 2 od 3 navedena svojstva:

- **Konzistentnost**: Svaki zahtev čitanja dobija podatke od poslednjeg upisa
- **Dostupnost**: Svaki zahtev dobija odgovor (ne nužno konzistentan).
- **Particiona tolerancija**: Tolerancija sistema na otkazivanje određenog dela mreže. Čak iako su neke poruke izgubljene ili odgovori na zahteve odloženi, sistem nastavlja da funkcioniše.

Sa obzirom na to da je dostupnost od visokog značaja za moderne veb aplikacije, Cassandra je dizajnirana tako da su dostupnost i particiona tolerancija na prvom mestu. Ta svojstva se garantuju, dok je konzistentnost kompromitovana na neku ruku.

Sistem Apache Cassandra garantuje sledeće stavke:

- Visoku skalabilnost
- Visoku dostupnost
- Trajnost
- Eventualnu konzistentnost upisa u jednu tabelu
- Lagane transakcije
- Grupni upisi u više tabela se obrađuju po pravilu „sve ili ništa“
- Sekundarni indeksi su konzistentni sa podacima u lokalnoj replici

## KAKO CASSANDRA POSTIŽE HA?

Postoji više mehanizma implementiranih u sistemu Apache Cassandra koji zajedno funkcionišu da postignu visoku dostupnost. Sa obzirom na to da bi detaljno zalaženje u implementacije svih ovih mehanizma daleko izašlo iz opsega ovog rada, u nastavku će biti reči o većini tih mehanizama, gde će fokus biti na sažimanju njihove uloge u celom sistemu, a negde će akcenat takođe biti i na osnovnom opisu njihove implementacije.

### Gossip protokol

*Gossip* je peer-to-peer komunikacioni protokol putem kog čvorovi u određenom klasteru međusobno razmenjuju podatke o sebi i drugim čvorovima sa kojima su prethodno komunicirali. Ovaj proces se okida svake sekunde i čvorovi razmenjuju informacije sa maksimalno tri druga čvora u klasteru. Na ovaj način, svi čvorovi ubrzo dobijaju znanje o stanju svih ostalih čvorova u klasteru. Poruka poslata ovim protokolom ima verziju, tako da prilikom komunikacije, najnovije informacije o stanju čvora menjaju one koje su zastarele. [8]

### Seed čvorovi

*Seed* čvorovi su oni čvorovi koji se koriste prilikom prvobitnog učitavanja novog čvora u klaster. Ovi čvorovi se suštinski ne razlikuju od ostalih i samim tim ne predstavljaju tačku propasti. Svaki čvor ima listu seed čvorova u svojoj konfiguracionoj datoteci, i prilikom svog prvog priključivanja klasteru, on bira jedan od tih seed čvorova i putem njega dobavlja informacije o ostatku klastera, tj. dobavlja informacije o topologiji klastera. Takođe, seed čvorovi pomažu sprovođenje gossip protokola kroz klaster. Preporuka je da lista seed čvorova bude mala, da bude ista u svim čvorovima u klasteru, i da ih ne bude više od 3 po centru podataka (engl. *data center*). [8]

### Detektovanje otkazivanja

Detektovanje otkazivanja čvorova se u suštini vrši kroz normalno funkcionisanje prethodno opisanog gossip protokola. Prethodno sačuvana stanja čvorova se čuvaju i upoređuju sa novim kako bi se utvrdilo da li je došlo do nekog otkazivanja. Ove informacije se koriste kako bi sistem prosledio zahteve čvorovima koji zapravo jesu dostupni.

Gossip protokol prati stanje drugih čvorova direktno (direktna komunikacija sa tim čvorom) i indirektno (priliv informacija o datom čvoru preko čvorova koji su komunicirali sa njim). Sistem nema jedan statičan prag vrednosti po kome se određuje da li je čvor otkazao ili nije, već se koristi mehanizam postepene detekcije, tj. prirast sumnje, koji pri određivanju praga za dati čvor obraća pažnju na detalje kao što su performanse same mreže, opterećenje, istorija stanja, itd. Moguće je podesiti parametar koji utiče na to koliko je klaster osetljiv na detektovanje otkazivanja, i njegovim podešavanjem se može smanjiti broj lažnih detektovanja otkazivanja u određenim situacijama.

Sa obzirom na to da su detekcije nedostupnosti ili otkazivanja nekog čvora vrlo retko trajne, iako nekad mogu trajati duži period, takva detekcija ne pokreće otklanjanje čvora iz klastera. Ostali čvorovi periodično pokušavaju da stupe u kontakt sa „propalim“ čvorom kako bi proverili da li je on opet dostupan.

Kada čvor eventualno postane dostupan, postoji mogućnost da je propustio neke upise koji su relevantni za kopije podataka koje on poseduje. U tom slučaju, postoje mehanizmi popravke, kao što je *hinted handoff* ili manuelna popravka upotrebom komande *nodetool repair*, koji se koriste kako bi se opet postigla konzistentnost podataka. U zavisnosti od dužine perioda nedostupnosti čvora, bira se odgovarajući mehanizam popravke. [9]

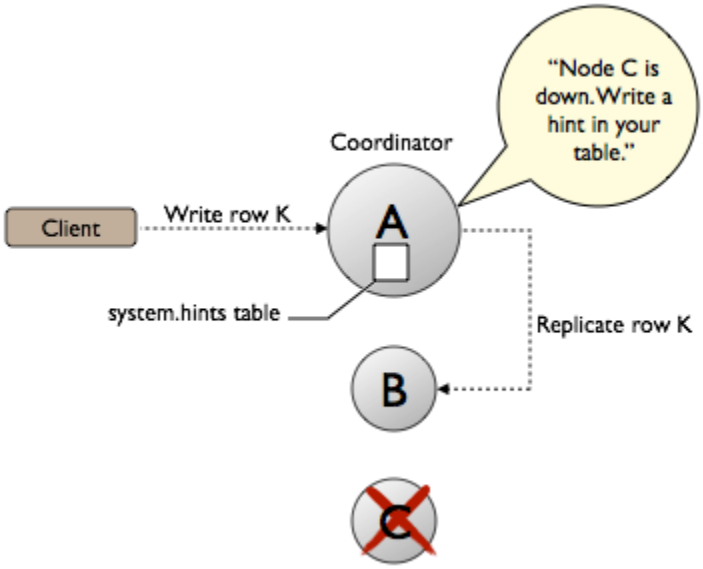
### Hinted handoff

*Hinted handoff* je mehanizam koji optimizuje održavanje konzistentnosti podataka kada čvor koji sadrži repliku nije dostupan, kako bi sistem prihvatio upis podataka, i eventualno taj upis prosledio nedostupnom čvoru kada on opet postane dostupan. Ovo je automatski mehanizam koji se može omogućiti kroz konfiguracionu datoteku.

Prilikom zahteva za upis, u slučaju da je *hinted handoff* mehanizam omogućen, i da nivo konzistentnosti od strane korisnika može biti zadovoljen (o ovome će više reči biti u nastavku rada), čvor koordinator čuva *hint* o upisu namenjen nedostupnim čvorovima, ako važi da:

- Čvor sa replikom datog podatka je već poznat kao nedostupan
- Čvor sa replikom ne odgovori na zahtev za upis

Ako klaster ne može da upotpuni traženi nivo konzistentnosti, *hint* se ne čuva.




Slika 7. Vizualizacija hinted handoff procesa [10]

*Hint* označava da određeni upis treba piti ponovljen određenim nedostupnim čvorovima. On sadrži podatke o lokaciji čvora koji je nedostupan, tj. relevantne replike na tom čvoru, podatke verzije, i same podatke koji se upisuju. Po podrazumevanim podešavanjima, *hint* se čuva maksimalno 3 sata, jer ako nedostupnost čvora traje duže od toga, on se smatra trajno nedostupnim. Ako ovo vreme istekne, nužno je pokrenuti ručni mehanizam popravke konzistentnosti.

Kada čvor sazna putem *gossip* protokola da je neki čvor, za koji on sadrži *hint*, opet dostupan, on njemu prosleđuje sačuvane podatke o upisima.

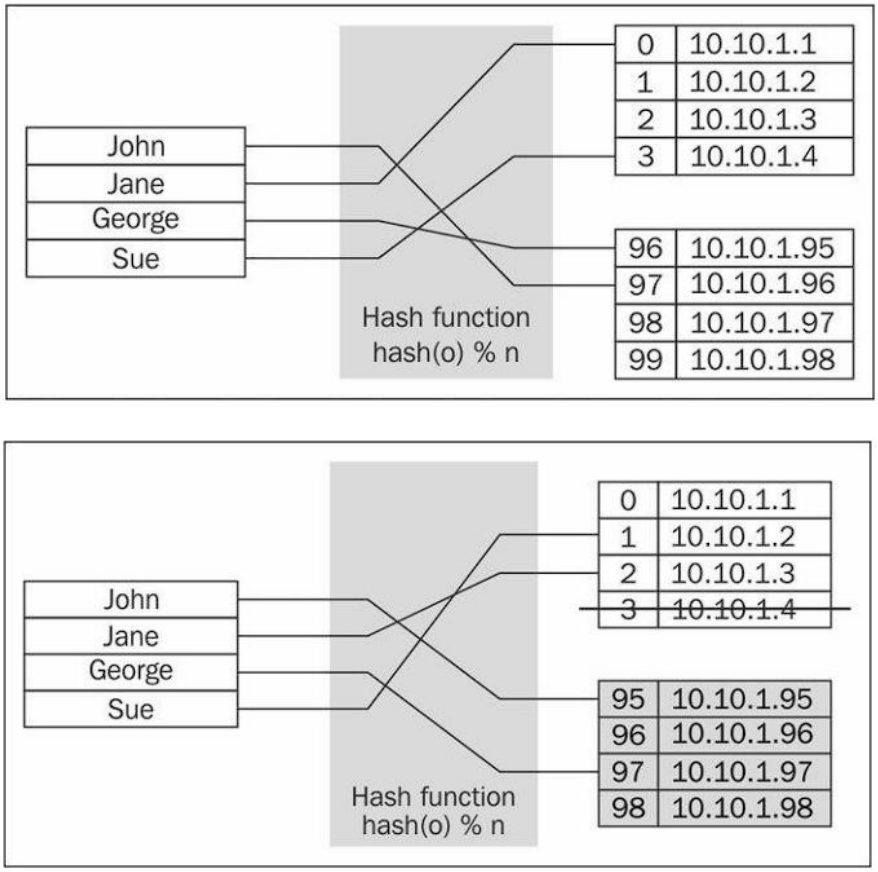
Kao što je i ranije pomenuto, *hinted handoff* ne garantuje prihvatanje nekog upisa, jer klaster i dalje mora zadovoljiti traženi nivo konzistentnosti. Ako su svi čvorovi sa replikama podatka nedostupni, upis može biti prihvaćen samo ako je korisnik za nivo konzistentnosti specificirao *ANY*. U tom slučaju se upis prihvata i na koordinatoru se čuvaju *hint*-ovi za sve nedostupne čvorove sa replikama. [10]

### Raspodela podataka po čvorovima

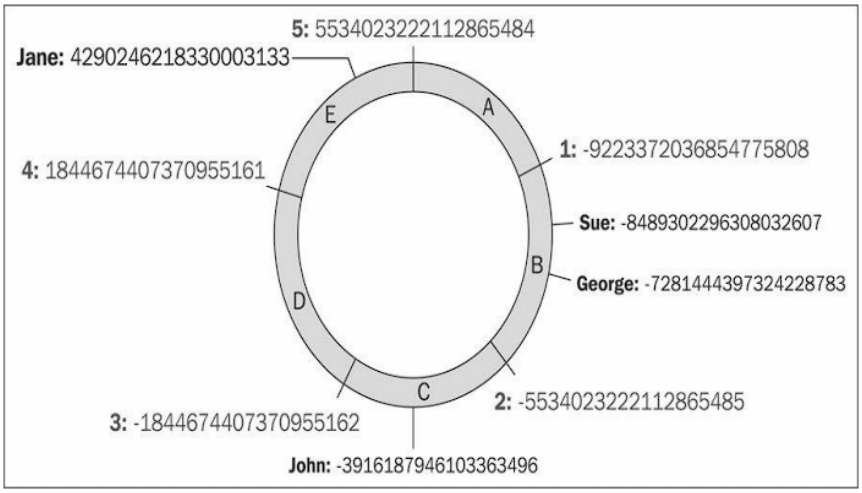
Kako bi sistem Apache Cassandra postigao visoku dostupnost i skalabilnosti, koristi se posebna struktura koja pomaže svim čvorovima da lakše pronađu lokaciju datog podatka. Ta struktura je distribuirana heš tabela (DHT). Kako bi se ovaj proces poboljšao, koriste se nekoliko tehnika raspodele podataka po čvorovima. [3]

Bitno je znati da su čvorovi u klasteru raspoređeni po topologiji prstena. Kako svrstati podatke u te čvorove se određuje heširanjem. Kada bi se koristile tradicionalne funkcije heširanja, imali bi problem sa reorganizacijom podataka prilikom dodavanja ili izbacivanja čvorova iz klastera. Kako bi se rešio taj problem, koristi se tehnika

konzistentnog heširanja. Svakom čvoru pripada određeni opseg vrednosti. Vrednost po kojima se podaci svrstavaju u te opsege je zapravo rezultat funkcije heširanja vrednosti particionog ključa za dati red podataka.




Slika 8. Problemi raspodele podataka ako se koristi normalno heširanje [3]

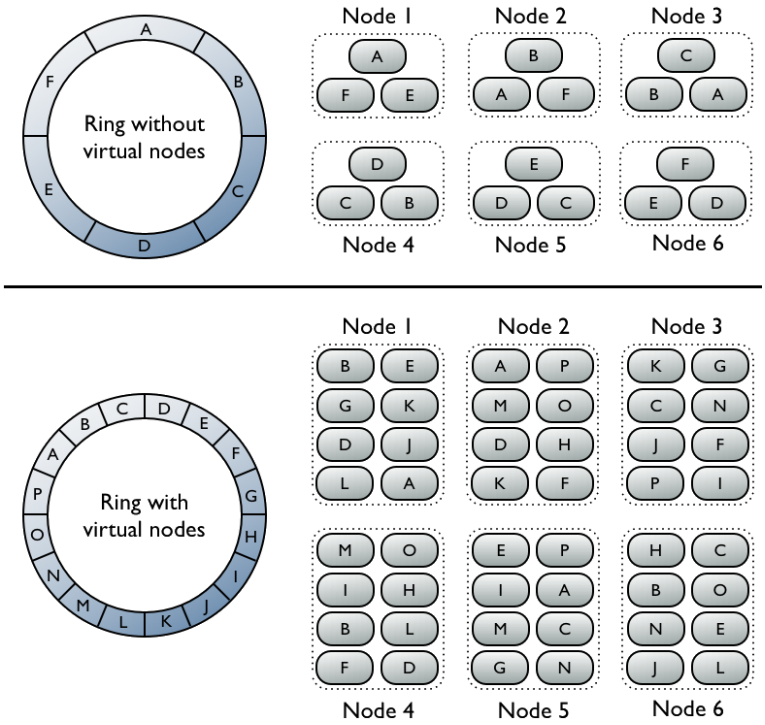



Slika 9. Raspodela podataka kada se koristi konzistentno heširanje [3]

### *Virtuelni čvorovi*

Kako bi se odredilo kom čvoru pripada koji opseg podataka, pre verzije sistema 1.2 bilo je nužno izračunati *token* (vrednost koja određuje početak opsega vrednosti koje pripadaju čvoru) i dodeliti ga čvoru. *Token*-i takođe određuju poziciju čvora u topologiji klastera.

Od verzije sistema 1.2 pa nadalje, prešlo se na automatsko generisanje *token*-a putem paradigme virtuelnih čvorova (engl. *vnodes*). Ova tehnika omogućava dodeljivanje velikog broja manjih opsega vrednosti svakom čvoru. [11]



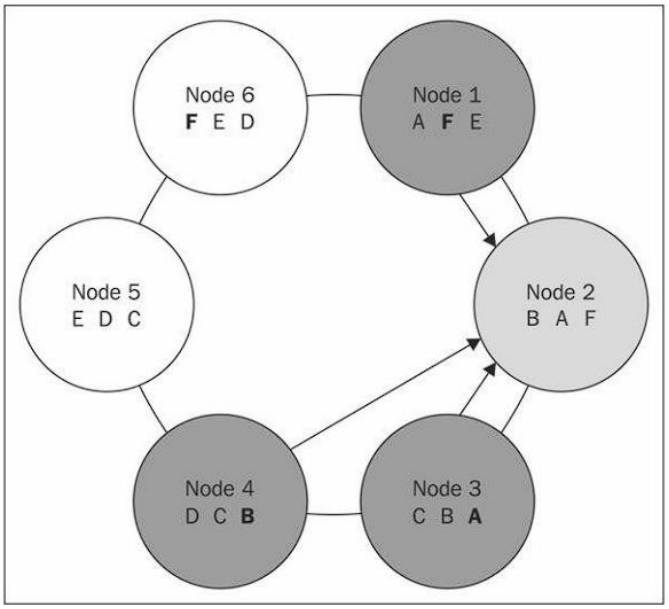

Slika 10. Arhitektura virtualnih čvorova naspram arhitekture jednog *token*-a [11]

Način na koji *vnode* tehnika poboljšava dostupnost u slučajevima kada prilikom popravke podataka čvora, ili izgradnje podataka čvora koji je novo dodat, sistem ostaje pod velikim opterećenjem.

Kada svaki čvor sadrži jedan *token*, celokupni podaci tog čvora se repliciraju na broj čvorova koji je jednak replikacionom faktoru minus 1 (više reči o replikaciji u nastavku rada). Dakle, ako je replikacioni faktor 3, dva čvora sadrže replike koje se mogu koristiti za mehanizam popravke, iako će samo jedan od ta dva čvora biti izabran za pružanje pomoći prilikom popravke. Bitno je napomenuti da u ovom slučaju svaki čvor, pored replike svog glavnog, dodeljenog *token*-a, takođe sadrži još dve replike. [3]

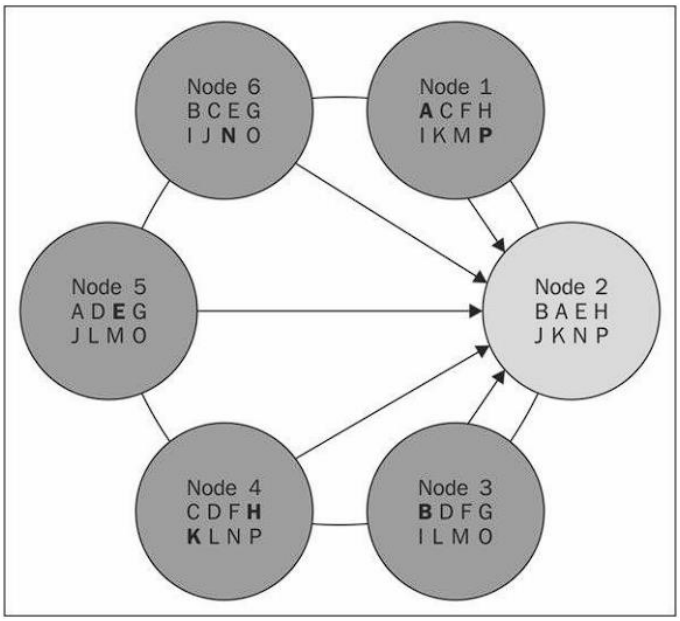
Ako imamo klaster sa 6 čvorova, i podešen je replikacioni faktor 3, u slučaju da je potrebno izvršiti mehanizam popravke nad jednim od tih čvorova, proces popravke će

celokupno pasti na 3 dostupna čvora (od ukupno 5). U ovom slučaju, samo dva čvora rade pod punim kapacitetom, dok su ostali pod potencijalno velikim opterećenjem. Takođe, neki od zauzetih čvorava su jedini koji sadrže određene opsege podataka.




Slika 11. Popravka podataka čvora kod arhitekture jednog *token*-a [3]

Kod tehnike virtuelnih čvorova, zbog veće granularnosti raspodele opsega po klasteru, moguće je ravnomernije raspodeliti opterećenje koje nastaje prilikom popravke podataka nekog čvora.




Slika 12. Popravka podataka čvora kod arhitekture sa virtuelnim čvorovima [3]

### *Replikacija*

Kao što smo ranije pomenuli, kroz prosto podešavanje faktora replikacije (RF) moguće je konfigurisati broj replika koje će biti kreirane za određeni set podataka. One su shodno podešavanjima raspoređene po klasteru (a potencijalno i šire, na drugim geografskim lokacijama). Takođe je moguće postići „pametno“ raspoređivanje replika putem određenih mehanizma pod imenom *snitches*.

```
CREATE KEYSPACE AddressBook
  WITH REPLICATION = {
    'class' : 'NetworkTopologyStrategy',
    'dc1' : 3,
    'dc2' : 2
  };
```

Slika 13. Konfigurisanje broja replika *keyspace*-a kroz dva *datacentre*-a, gde je RF jednog od njih jednak 3, a kod drugog jednak 2 [3]

Iako je moguće ući u sitne detalje samog procesa replikacije i načine na koje taj proces utiče na druge mehanizme u sistemu, u ostatku ovog rada fokus će biti na mehanizmu podešavanja nivoa konzistentnosti, koji je verovatno i najbitniji za garantovanje visoke dostupnosti. Takođe, logično je da i sam proces replikacije uveliko povećava nivo dostupnosti sistema.

Sistem Apache Cassandra omogućava da se za svaki upit definiše nivo konzistentnosti koji se traži. U zavisnosti od toga da li upit predstavlja zahtev za čitanje ili upis, nivo konzistentnosti ima drugačiji efekat. Kada je u pitanju čitanje, nivo konzistentnosti čitanja (R) predstavlja broj kopija koje će biti pročitane, i biće uzeta vrednost one koja predstavlja najnoviju verziju. Kod upisa, nivo konzistentnosti upisa (W) predstavlja broj kopija od kojih čekamo povratnu informaciju o uspešnom upisu, pre nego što koordinator proglasi celokupan zahtev za upisom uspešnim. [12] Kao što je bilo reči i ranije, ako koordinator zna da zbog nedostupnih čvorova ne postoji dovoljan broj dostupnih čvorova da se postigne odgovarajući nivo konzistentnosti, on će odmah proglasiti operaciju nemogućom. Ali, ukoliko se desi da koordinator ne dobije dovoljan broj odgovora od strane replika, on će korisniku vratiti grešku pod imenom *TimedOutException*. Veoma je bitno napomenuti da kod upisa podataka, ovo ne znači da upis u potpunosti nije bio uspešan. Ovo samo znači da upis nije prošao na dovoljno velikom broju replika. [13]

Podešavanjem vrednosti R i W, određuje se šta je bitnije za dati zahtev. Što su vrednosti R i W manje, to je bitnija dostupnost, tj. što brži odziv, dok konzistentnost nije u prvom planu. Što su te vrednosti veće, to je konzistentnost bitnija, a odziv potencijalno duži. Kod sistema gde se traži neki balans, uglavnom se gleda da važi $R + W > RF$. [12]

| Level | Description | Usage |
|---|---|---|
| ALL | Returns the record after all replicas have responded. The read operation will fail if a replica does not respond. | Provides the highest consistency of all levels and the lowest availability of all levels. |
| EACH_QUORUM | Not supported for reads. | |
| QUORUM | Returns the record after a quorum of replicas from all datacenters has responded. | Used in either single or multiple datacenter clusters to maintain strong consistency across the cluster. Ensures strong consistency if you can tolerate some level of failure. |
| LOCAL_QUORUM | Returns the record after a quorum of replicas in the current datacenter as the coordinator has reported. Avoids latency of inter-datacenter communication. | Used in multiple datacenter clusters with a rack-aware replica placement strategy ( NetworkTopologyStrategy) and a properly configured snitch. Fails when using SimpleStrategy. |
| ONE | Returns a response from the closest replica, as determined by the snitch. By default, a read repair runs in the background to make the other replicas consistent. | Provides the highest availability of all the levels if you can tolerate a comparatively high probability of stale data being read. The replicas contacted for reads may not always have the most recent write. |
| TWO | Returns the most recent data from two of the closest replicas. | Similar to ONE. |
| THREE | Returns the most recent data from three of the closest replicas. | Similar to TWO. |
| LOCAL_ONE | Returns a response from the closest replica in the local datacenter. | Same usage as described in the table about write consistency levels. |
| SERIAL | Allows reading the current (and possibly uncommitted) state of data without proposing a new addition or update. If a SERIAL read finds an uncommitted transaction in progress, it will commit the transaction as part of the read. Similar to QUORUM. | To read the latest value of a column after a user has invoked a lightweight transaction to write to the column, use SERIAL. Cassandra then checks the inflight lightweight transaction for updates and, if found, returns the latest data. |
| LOCAL_SERIAL | Same as SERIAL, but confined to the datacenter. Similar to LOCAL_QUORUM. | Used to achieve linearizable consistency for lightweight transactions. |

Slika 14. Različiti nivoi konzistentnosti kod procesa čitanja, kao i njihovi opisi. [14]

| Level | Description | Usage |
|---|---|---|
| ALL | A write must be written to the commit log and memtable on all replica nodes in the cluster for that partition. | Provides the highest consistency and the lowest availability of any other level. |
| EACH_QUORUM | Strong consistency. A write must be written to the commit log and memtable on a quorum of replica nodes in each datacenter. | Used in multiple datacenter clusters to strictly maintain consistency at the same level in each datacenter. For example, choose this level if you want a write to fail when a datacenter is down and the QUORUM cannot be reached on that datacenter. |
| QUORUM | A write must be written to the commit log and memtable on a quorum of replica nodes across *all* datacenters. | Used in either single or multiple datacenter clusters to maintain strong consistency across the cluster. Use if you can tolerate some level of failure. |
| LOCAL_QUORUM | Strong consistency. A write must be written to the commit log and memtable on a quorum of replica nodes in the same datacenter as the coordinator. Avoids latency of inter-datacenter communication. | Used in multiple datacenter clusters with a rack-aware replica placement strategy, such as NetworkTopologyStrategy, and a properly configured snitch. Use to maintain consistency locally (within the single datacenter). Can be used with SimpleStrategy. |
| ONE | A write must be written to the commit log and memtable of at least one replica node. | Satisfies the needs of most users because consistency requirements are not stringent. |
| TWO | A write must be written to the commit log and memtable of at least two replica nodes. | Similar to ONE. |
| THREE | A write must be written to the commit log and memtable of at least three replica nodes. | Similar to TWO. |
| LOCAL_ONE | A write must be sent to, and successfully acknowledged by, at least one replica node in the local datacenter. | In a multiple datacenter clusters, a consistency level of ONE is often desirable, but cross-DC traffic is not. LOCAL_ONE accomplishes this. For security and quality reasons, you can use this consistency level in an offline datacenter to prevent automatic connection to online nodes in other datacenters if an offline node goes down. |
| ANY | A write must be written to at least one node. If all replica nodes for the given partition key are down, the write can still succeed after a hinted handoff has been written. If all replica nodes are down at write time, an ANY write is not readable until the replica nodes for that partition have recovered. | Provides low latency and a guarantee that a write never fails. Delivers the lowest consistency and highest availability. |

Slika 15. Različiti nivoi konzistentnosti kod procesa upisa, kao i njihovi opisi. [14]

# Citirana dela


1] [Na mreži]. Available: https://en.wikipedia.org/wiki/High_availability.

2] [Na mreži]. Available: https://www.techtarget.com/searchdatacenter/definition/high-availability.

3] R. Strickland, Cassandra High Availability, Packt, 2014.

4] [Na mreži]. Available: https://jiankaiwang.gitbooks.io/itsys/content/cassandra/architecture.html.

5] [Na mreži]. Available: https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/architecture/archIntro.html.

6] [Na mreži]. Available: https://cassandra.apache.org/doc/latest/cassandra/architecture/guarantees.html.

7] [Na mreži]. Available: https://hazelcast.com/glossary/cap-theorem/.

8] [Na mreži]. Available: https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/architecture/archGossipAbout.html.

9] [Na mreži]. Available: https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/architecture/archDataDistributeFailDetect.html.

10] [Na mreži]. Available: https://docs.datastax.com/en/cassandra-oss/2.1/cassandra/dml/dml_about_hh_c.html.

11] [Na mreži]. Available: https://www.sciencedirect.com/science/article/pii/B9780128191545000187.

12] [Na mreži]. Available: https://www.javatpoint.com/query-processing-in-dbms.

13] [Na mreži]. Available: https://logicalread.com/sql-server-query-processing-w01/.

14] [Na mreži]. Available: https://www.sqlshack.com/sql-server-execution-plans-overview/.