REPUBLIKA SLOVENIJA
MINISTRSTVO ZA IZOBRAŽEVANJE, ZNANOST IN ŠPORT

Univerza *v Ljubljani*

EVROPSKA UNIJA
EVROPSKI
SOCIALNI SKLAD
NALOŽBA V VAŠO PRIHODNOST

Management podatkovnih baz v resničnem/poslovnem okolju - 3. del

Tadej Borovšak, XLAB d.o.o.

Uvod

V zadnjem delu delavnice se bomo posvetili reševanju težave, s katero se sreča večina sistemov, ki služi velikemu številu uporabnikov: zagotavljanju hitre obdelave zahtev. Del rešitve smo si ogledali že pri optimizaciji poizvedb, danes pa si bomo ogledali še nadaljnje ukrepe, ki jih lahko izvedemo, če prejšnji ne zadostujejo. Zaključili pa bomo s pregledom noSQL tehnologij, ki pristopajo k reševanju težave na drugačen način, in si malo bolj podrobno ogledali Apache Cassandra-o.

CAP izrek

Eric Brewer je leta 2000 na ACM simpoziju o distribuiranih sistemih predstavil tri osnovne zahteve, katerim naj bi ustrezali veliki, distribuirani sistemi:

1. **Konsistentnost (Consistency)** nam zagotavlja, da vsi uporabniki sistema v istem trenutno vidijo enako vrednost.
2. **Dostopnost (Availability)**, ki zagotavlja hipen dostop do sistema za vse uporabnike.
3. **Odpornost na izpade (Partition tolerance)**, ki predpisuje, da izpad enega ali večih delov sistema ne vpliva na pravilnost delovanja sistema.

Na žalost pa je takoj zatem predstavil svoj izrek, ki pravi, da ima lahko poljuben sistem največ dve izmed treh lastnosti. Formalno je bil izrek dokazan leta 2002 na MIT-u, s čimer so pokopali vse upe o obstoju idealnega sistema. Grafično lahko bistvo izreka prikažemo z Venn-ovim diagramom s slike 1.

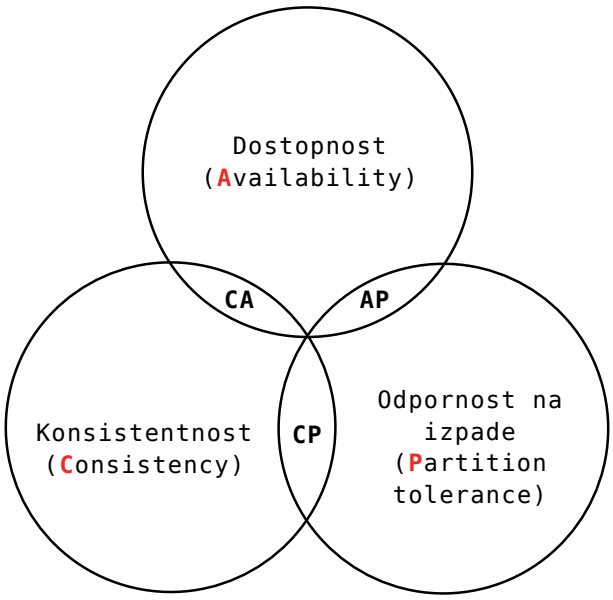

Figure 1: Lastnosti sistemov.

Ker lahko na osnovi CAP izreka za poljuben sistem izberemo le dve lastnosti, katerima bo ustrezal, sisteme v praksi delimo v tri kategorije:

1. **CA sistemi** so konsistentni in dostopni, ob izpadu ene izmed komponent distribuiranega sistema pa bo sistem vsaj nekaj časa nedosegljiv za uporabnike.
2. **CP sistemi** so konsistentni in odporni na izpade, vendar pa za doseganje teh lastnosti žrtvujemo del učinkovitosti delovanja, kar se kaže kot daljši odzivni čas poizvedb oz. upočasnitev delovanja pri sočasnem dostopu več uporabnikov.

3. **AP sistemi** pa so odporni na izpade komponent in uporabnikom strežejo podatke z visokimi hitrostmi, vendar pa so ti sistemi lahko v določenem trenutku nekonsistentni, npr. dve enaki poizvedbi ob istem času vrneta različen rezultat, ker se distribuirane komponente še niso medsebojno uskladile.

Sistemi, s katerimi smo se seznanili do sedaj (PostgreSQL in njemu sorodni sistemi za upravljanje relacijskih podatkovnih baz) spadajo v skupino CA sistemov, ker večinoma niso distribuirani in za popravilo pokvarjenega sistema potrebujemo čas, med katerim storitev ni dosegljiva. Na prejšnji delavnici smo si ogledali, kako lahko z uvedbo toplih oz. vročih replik sistema uvedemo določeno mero tolerance na odpoved posamezne komponente, ampak smo morali pri tem žrtvovati del dostopnosti sistema, ker so postale posamezne modifikacije podatkov počasnejše.

Distribuirani sistemi, s katerimi se bomo seznanili v zadnjem delu delavnice, pa večinoma spadajo v CP (Neo4J, Google Bigtable, MongoDB, Redis) ali AP (Cassandra, CouchDB, Riak) kategoriji. Shematsko lahko kategorije prikažemo z preseki dveh lastnosti s slike 1.

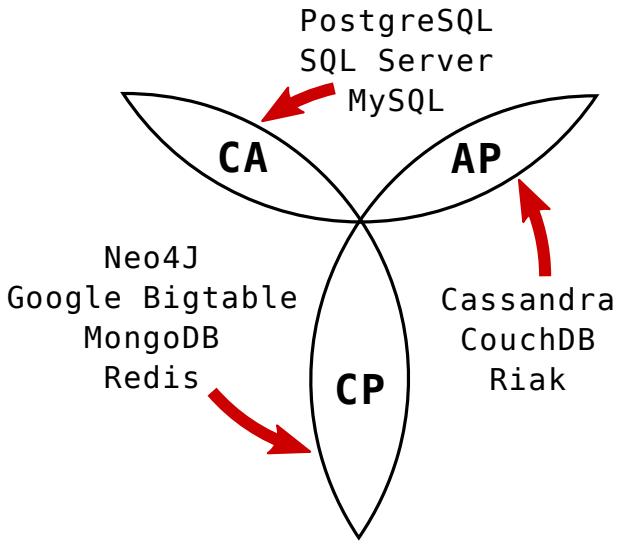

Figure 2: Kategorije sistemov.

Ker so sistemi za delo z relacijskimi bazami trenutno najpogosteje uporabljeni sistemi v poslovnem okolju, si bomo v naslednjem razdelku ogledali, kako lahko izboljšamo hitrost poizvedb končnega uporabnika.

Uporaba predpomnjenja

Ker večina sistemov za delo z relacijskimi bazami ne podpira popolnoma distribuiranega modela (vedno samo en strežnik komunicira z uporabniki, ostali pa služijo kot rezerva), se pri povečevanju števila sočasnih uporabnikov slej ko prej doseže meja, nad katero strežnik ne zmore več sproti opraviti vseh zahtev.

Ena izmed rešitev, ki bi razbremenila glavni strežnik, je preusmeritev uporabniških zahtev, ki izvajajo le branje podatkov, na eno izmed toplih replik. Na ta način bi se zmanjšal promet na glavnem strežniku, vendar pa bi lahko nekateri uporabniki občasno prejeli nekoliko zastarele podatke (CAP izrek pravi, da ni zastonj kosila). V določenih primerih je tak kompromis mogoč, seveda pa obstajajo situacije, kjer si nekonsistentnosti ne moremo privoščiti.

Količino prometa na glavnem strežniku pa lako zmanjšamo tudi z dodatkom predpomnilniškega

sistema kot je npr. Memcached[1]. Takšna sprememba pa ni več povsem transparentna za končnega uporabnika, ker je potrebno spremeniti program, ki dostopa do podatkovne baze.

Spremembe programa najlažje prikažemo z uporabo psevdokode. Če predpostavimo, da smo v osnovni različici programa do podatkov dostopali s pomočjo funkcije `get_data(query)`, bi njena implementacija lahko izgledala nekako takole:

```
function get_data(query)
  data = dbserver.get(query)  // dbserver je objekt, ki predstavlja bazo
  return data
end
```

V tem primeru vsaka poizvedba potuje neposredno do strežnika, ki jo sprocesira in vrne rezultat. Če želimo uporabiti še sistem za predpomnjenje, potem bi se funkcija za dostop do podatkov spremenila v nekaj podobnega temu:

```
function get_data(query)
  key = get_cache_key_from_query(query)
  data = cache.get(key)  // cache je objekt, ki predstavlja predpomnilnik
  return data if data

  data = dbserver.get(query)
  cache.set(key, data)
  return data
end
```

Dostop do podatkov se sedaj opravi v dveh korakih. Najprej se skuša pridobiti predpomnnjene podatke in vrniti te. V koliko pa to ne uspe, se podatke pridobi neposredno iz podatkovne baze in se jih pred vrnitvijo shrani še v sistem za predpomnjenje.

Na sliki 3 so prikazani trije poteki poizvedb. V rdeči barvi je označen potek poizvedbe brez uporabe predpomnjenja, v modri je označen potek poizvedbe, ki še ni shranjena v sistemu za predpomnjenje, v vijolični barvi pa je prikazan potek poizvedbe, ki je že predpomnjena.

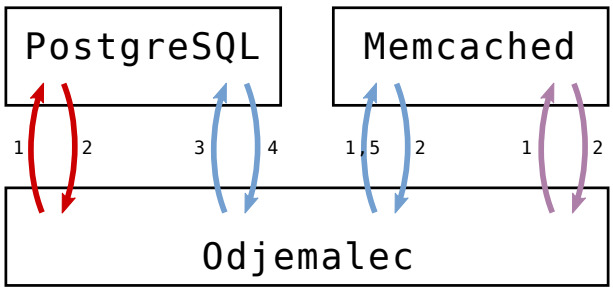

Figure 3: Potek poizvedb s predpomnjenjem.

Zakaj je takšen način dostopa do podatkov hitrejši? Glavni razlog je razlika v hitrosti delovanja sistema za predpomnjenje in podatkovne baze. Podatkovna baza mora ob vsaki poizvedbi izvesti planiranje poizvedbe, dejansko poizvedbo in vrniti rezultate, medtem ko sistemi za predpomnjenje delujejo kot veliko kazalo, ki ob zadetku le vrnejo podatke, ki so shranjeni v delovnem pomnilniku, kar je neprimerno cenejša operacija v smilu porabe procesorske moči in časa.

Če spremenimo le funkcijo za pridobivanje podatkov, potem bomo v večini primerov v klientu prejemali podatke, ki so rahlo zastareli, ker sistem za predpomnjenje ne ve, kdaj se spremeni vsebina podatkovne baze. Predpostavimo, da je osnovna funkcija za posodobitev podatkov izgledala takole:

```
function set_data(query)
  dbserver.set(query)
end
```

[1] https://memcached.org/

Najlažji način za zagotavljanje konsistentnosti podatkov v predpomnilniškem sistemu je preprosto zbrisanje ustreznega vnosa iz predpomnilnika:

```
function set_data(query)
  dbserver.set(query)
  key = get_cache_key_from_query(query)
  cache.delete(key)
end
```

Na ta način bo naslednji dostop do podatkov predpomnilnik napolnil s svežimi podatki in nekonsistence ne bo. Še boljše pa je, če lahko ob posodobitve podatkovne baze posodobimo tudi predpomnilnik.

```
function set_data(query)
  dbserver.set(query)
  key = get_cache_key_from_query(query)
  data = get_data_from_query(query)
  cache.set(key, data)
end
```

Ker funkcija `get_data_from_query` deluje lokalno in ne dostopa do baze, takšen način posodabljanja ni vedno mogoč (ker npr. aplikacija ne ve, kakšno stanje bo ob koncu posodibitve). V kolikor pa je to mogoče, bodo prihranki toliko večji, ker bo podatkovna baza obdelovala le zapisovanje in pa poizvedbe, ki se niso izvedle že nekaj časa in so bile izbrisane iz predpomnilnika zaradi poteka časa veljavnosti.

Do te točke nam je uspelo rešiti veliko večino problemov, ki smo jih imeli s skalabilnostjo naših rešitev, obstaja pa scenarij, v katerem nobena izmed obstoječih rešitev ne poveča odzivnosti sistema: velika količina pisanja v podatkovno bazo. Ta scenarij pa si bomo ogledali v naslednjem razdelku.

NoSQL in Apache Cassandra

V zadnjem delu pa bomo svojo pozornost posvetili novejšim sistemom za upravljanje s podatkovnimi bazami, ki so za razliko od prej spoznanih rešitev, od samega začetka zasnovani za obdelavo (in predvsem za zapisovanje) velike količine podatkov.

Ker so si sistemi za hranjenje velike količine podatkov med seboj precej različni, je skoraj nemogoče opisati arhitekturo vseh. Zato se bomo v nadaljevanju osredotočili na opis arhitekture Apache Cassandra-e, ki je tipičen, v industriji preizkušen sistem.

Arhitektura sistema

Arhitekturo Cassandre bi lahko radelili na pet nivojev:

1. **Gruča** je najvišji nivo in predstavlja skupek strežnikov, na katerih je nameščena Cassandra in deluje usklajeno, obenem pa gruča služi kot vsebovalnik za imenske prostore.
2. **Imenski prostor** ustreza podatkovni bazi v klasičnih (relacijskih) sistemih za upravljanje baz. Imansko prostor vsebuje tabele, podobno kot podatkovna baza vsebuje relacije.
3. **Tabela** je preprost vsebovalnik za vrstice.
4. **Vrstica** je zbirka stolpcev, ki imajo enako vrednost primarnega ključa.
5. **Stolpec** je osnovna enota podatkov, ki jih lahko hranimo v bazi. V Cassandri je stolpec preprost par podatkov, kjer je prvi podatek ključ, drugi pa vrednost, ki ustreza ključu.

Na sliki 4 je prikazana tipična gruča, ki je zaradi zagotavljanja tolerance na izpade fizično locirana v dveh različnih podatkovnih centrih. Strežniki znotraj posameznega podatkovnega centra pa so v večini primerov postavljeni v vsaj dve različni omari z ločenim napajanjem in dostopom do omrežja.

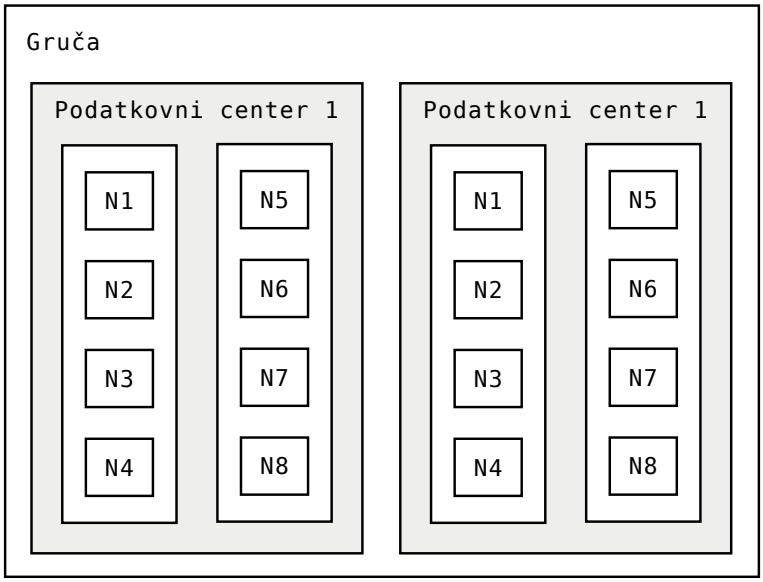

Figure 4: Tipična Cassandra gruča

Na logičnem nivoju pa so vsi strežniki, ki gostijo Cassandro, povezani v enoten prostor, ki ga okarakterizirajo števila med 0 in $2^127 - 1$. Vsakemu izmed strežnikov pripada interval števil iz prostora, ki določa, kateri podatki se bodo shranjevali na izbranem strežniku. Pa si oglejmo, kaj vse se izvede ob zapisovanju podatkov.

Zapisovanje podatkov

Zahteva za zapisovanje podatkov lahko pride do poljubnega strežnika, ki gosti Cassandro, in ta strežnik postane **lokalni koordinator**. Njegova prva naloga je poiskati podatkovne centre, ki bodo gostili podatke. V vsakem podatkovnem centru koordinator izbere **oddaljenenega koordinatorja**, ki bo skrbel za zapis podatkov v določenem podatkovnem centru. Nato pa vsem vozliščem in koordinatorjem pošlje podatke, ki jih je potrebno zapisati. Shematsko je proces prikazan na sliki 5, kjer sta strežnika N2 in N4 lokalni oz. oddaljeni koordinator, puščice pa ponazarjajo tok podatkov, ki jih zapisujemo.

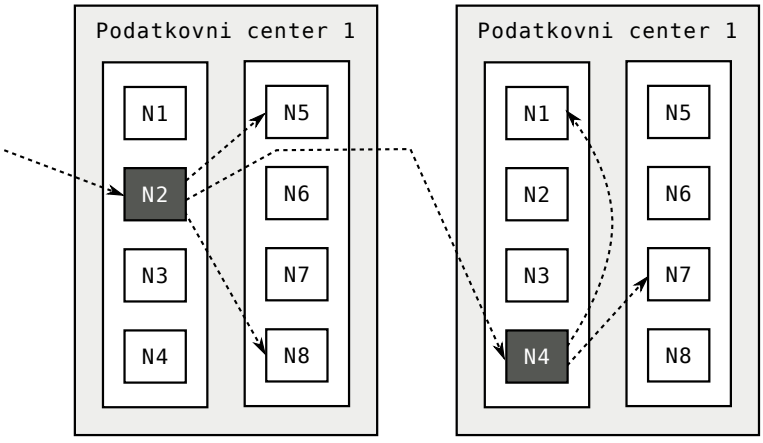

Figure 5: Potek zapisa podatkov.

Kot je iz slike 5 jasno razvidno, je replikacija podatkov v Cassandri privzeto aktivirana in nam je, za razliko od npr. PostgreSQL-a, ni potrebno posebej aktivirati.

Postopek zapisovanja, ki smo ga predstavili zgoraj, je idealna pot. Seveda pa se lahko zaradi nestabilnosti internetnih povezav zgodi, da določeno zapisovanje ne uspe. Natančen postopek obravnave takšnih situacij je izven okvirov te delavnice, ima pa Cassandra vgrajene mehanizme za detekcijo takšnih težav in njihovo odpravljanje.

Branje podatkov

Za zaključek pa si oglejmo še postopek branja podatkov iz Cassandre in mehanizme za zagotavljanje konsistentnosti podatkov, ki jih prejme klient. Postopek se prične z zahtevo klienta po podatkih. Podobno kot pri branje se strežnik, ki je prejel zahtevo, trenutno označi kot lokalni koordinator, ki nato v vseh ostalih podatkovnih centrih izbere oddaljene koordinatorje. Koordinatorji nato iz najhitrejše replike preberejo podatke, iz zadostnega števila ostalih replik pa zahtevajo kontrolno vsoto podatkov. Če so podatki in kontrolne vsote konsistentne, se podatki vrnejo uporabniku, sicer pa se aktivirajo mehanizmi za vzpostavitev konsistence podatkov.

Shematično je postopek branja prikazan na sliki 6, kjer puščice znova predstavljajo tok podatkov. Oznaka R na puščici pomeni branje podatkov, medtem ko oznaka D označuje prenos le kontrolne vsote.

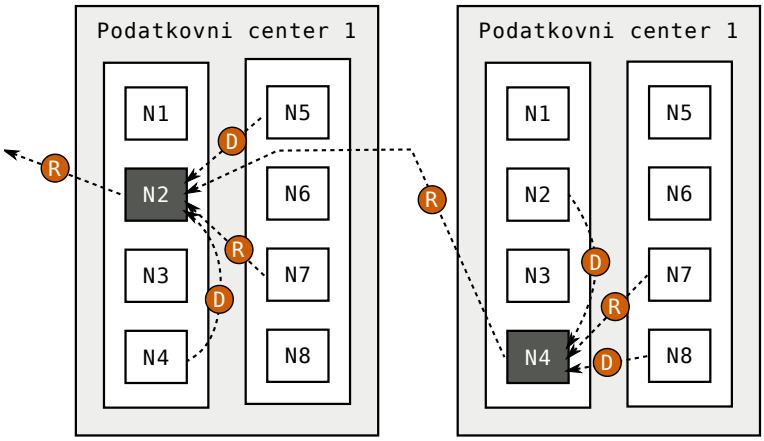

Figure 6: Potek branja podatkov.

Pri branju smo kontrolno vsoto prebrali iz zadostnega števila replik. Koliko natančno kontrolnih vsot potrebujemo je odvisno od nastavitev naše gruče, ker Cassandra nima fiksne meje konsistentnosti ampak jo nastavi končni uporabnik sam.