UNIVERZITET U NIŠU

ELEKTRONSKI FAKULTET

Katedra za računarstvo

PostgreSQL High-Availability Cluster

Sistemi za upravljanje bazama podataka

Student:

Andrija Petrović 1387

Mentor:

Doc. dr Aleksandar Stanimirović

Sažetak

Instanciranje kontejnerskih aplikacija pomoću *Docker*-a, *Kubernetes*-a i *RedHat OpenShift*-a je veoma dobro uspostavljen način implementacije i upravljanja softverom u kompanijama koje su već usvojile *Could native* pristup životnom ciklusu softvera. Relativno zreli alati, dostupni su za usluge bez stanja, kao što su web aplikacije i proksiji kojima upravlja automatizovana platforma za upravljanje kontejnerima. Nije uobičajeno da se sistemi sa podacima o stanju, kao baze podataka, postavljaju u isto okruženje. Ovaj rad je baziran na problematici kreiranja High-Availability (veoma dostupnog) klastera *PostgreSQL* baze podataka. Rešenje se sastoji od slika (*images*) kontejnera koje sadrže odabrane *PostgreSQL* alate za visoku dostupnost i *middleware*-a (operatera) koji upravlja ovim kontejnerima u virtualnom *Docker* okruženju. Operater može da inicijalizuje klaster, otkrije padove i automatski izvrši prelazak na rezervnu repliciranu instancu baze. Za izradu na *Ubuntu 21.10* operativnom sistemu, ali i Digitalocean platformi, korićena je poslednja dostupna *PostgreSQL 14.4* verzija. *Docker-compos*e u sprezi sa servisima: *postgresql*, *repmgr*, *pgadmin4* i *pgpool* su korišćeni na lokalnom nivou.

Sadržaj

1 Highly-Available DBMS

Problemi redundantnosti podataka, konzistentnosti i atomičnosti operacija, doveli su do nastojanja da se razviju namenski sistemi za skladištenje i upravljanje podacima. Različiti tipovi sistema baza podataka su evoluirali tokom vremena, kako bi zadovoljili različite tehničke i poslovne slučajeve upotrebe. Gubitak podataka u većini slučajeva značajno utiče na poslovanje. Veliki značaj podataka dovodi do neophodnosti implementacije visoke dostupnosti sistema baza podataka. Postizanje visoke dostupnosti podataka je teže u poređenju sa sistemima bez stanja. U ovom poglavlju predstavljena je teorijska pozadina sistema baza podataka visoke dostupnosti.

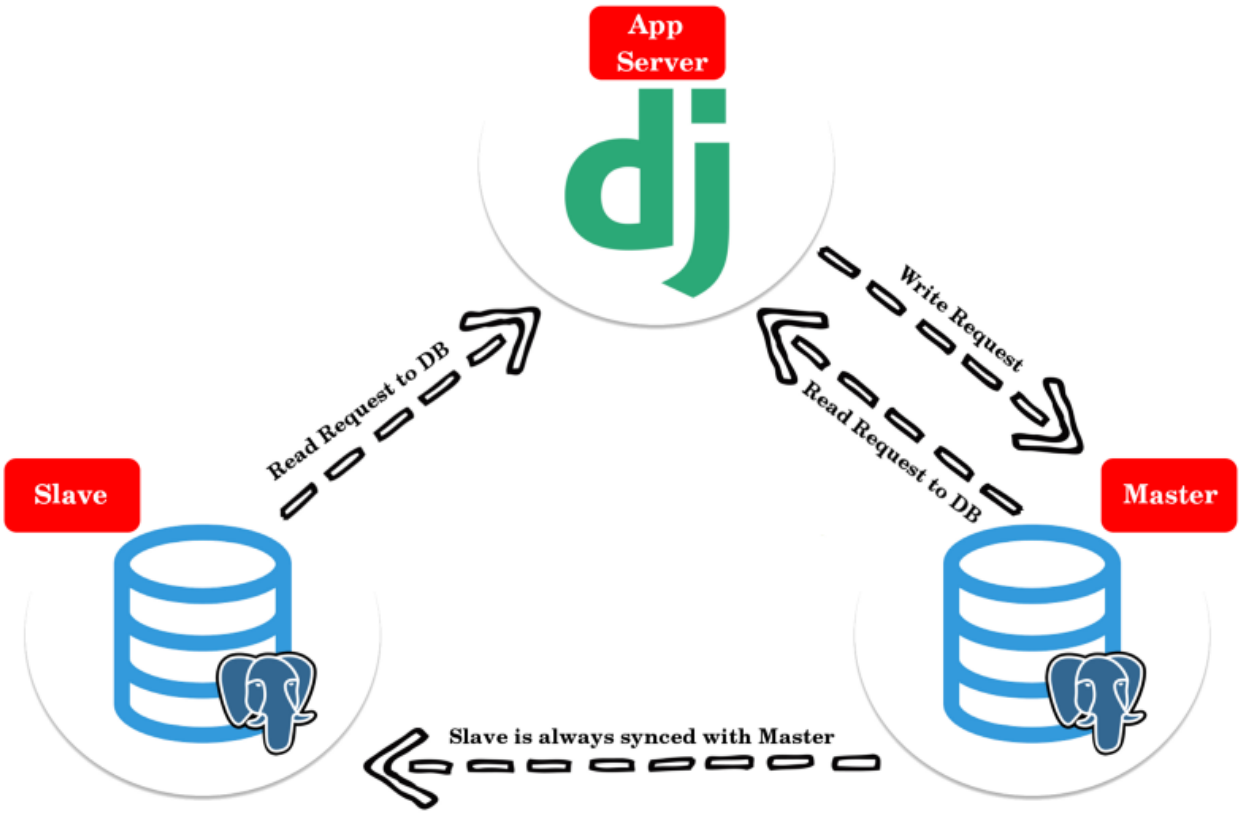
1.1 Replikacija

Kada se relacione baze podataka koriste za skladištenje važnih podataka, podržavajući neke kritične poslove ili istraživanja, sistem baze podataka mora biti visoko dostupan i imun na preopterećenje. Ovi zahtevi su slični ključnim karakteristikama *Kubernetes*-a. Rešenja koja donose ove mogućnosti u sisteme baza podataka su takođe zasnovana uglavnom na klasteru baze podataka sa više čvorova (servera). U slučaju da se sistem baze podataka sastoji od više servera, podaci moraju biti sinhronizovani između svih čvorova. Trajno kopiranje promena podataka iz jednog čvora u drugi naziva se replikacija baze podataka.

Postoji više pristupa replikaciji baze podataka, svaki od njih ima neke prednosti i nedostatke. Postoje i različiti kriterijumi, koji se mogu koristiti za kategorizaciju tipova replikacije. Možemo razlikovati replikaciju sa jednim i više *master*-a.

Češći i tradicionalniji način dizajna klastera baze podataka je sa samo jednim *master*-om, koji može da obrađuje zahteve za pisanje i čitanje. Ažuriranja iz master-a i obrada upita za čitanje, distribuiraju se replikama. Replike rade u režimu *read-only* (samo čitanje). Da bi se sprečilo preopterećenje *master*-a, zahtevi za čitanje mogu biti preusmereni na *slave* (read-only) replike.

Druga opcija je dozvoliti zahteve za pisanje, balansiranjem opterećenja, svim čvorovima u klasteru. Distribucija svih zahteva može biti prednost, međutim, ovaj pristup dodaje mnogo složenosti rešenju. *Multi-master* replikacija je manje transparentna i lako se mogu javiti neki sukobi u konzistentnosti podataka.



Slika 1. Master(read/write)-slave(read-only)

Drugi način klasifikacije je pravljenje razlike između sinhrone i asinhrone replikacije. U slučaju da se transakcija može obaviti na *master*-u i kopirati na *slave* sa malim zakašnjenjem, naziva se asinhrona replikacija. *Slave* čvorovi obično malo kasne, ali transakcije su brze i trenutne. Sinhrona replikacija je striktnija u pogledu doslednosti. Transakcija se ne može izvršiti pre nego što se ažuriranje kopira na najmanje dva čvora u repliciranom klasteru. Ovaj pristup garantuje bolju doslednost, ali transakcije traju duže.

Postoje i alternativne implementacije kopiranja podataka između čvorova. Koncept fizičke replikacije znači da se podaci kopiraju u binarnom formatu. Sadržaj podataka na *master* čvoru se replicira na *slave* čvorove, bajt po bajt. Logička replikacija, nasuprot tome, prenosi logičke promene na osnovu identiteta replikacije.

Fizička replikacija je zreliji i standardni način razmene podataka među čvorovima. Obično je lakše podesiti i dobro funkcioniše u slučajevima kada je potrebno skaliranje veličine klastera. Ograničenje je da se obično ista verzija softvera mora instalirati na svim čvorovima. Logička replikacija je, s druge strane, fleksibilnija, omogućava repliciranje upravo izabranih tabela i slanje različitih skupova izmena različitim pretplatnicima. Podešavanje logičke replikacije je

obično složenije, a brzo skaliranje je teže. Logička replikacija je moguća u svim verzijama baze podataka, što može biti korisno posebno za nadogradnju baze podataka.

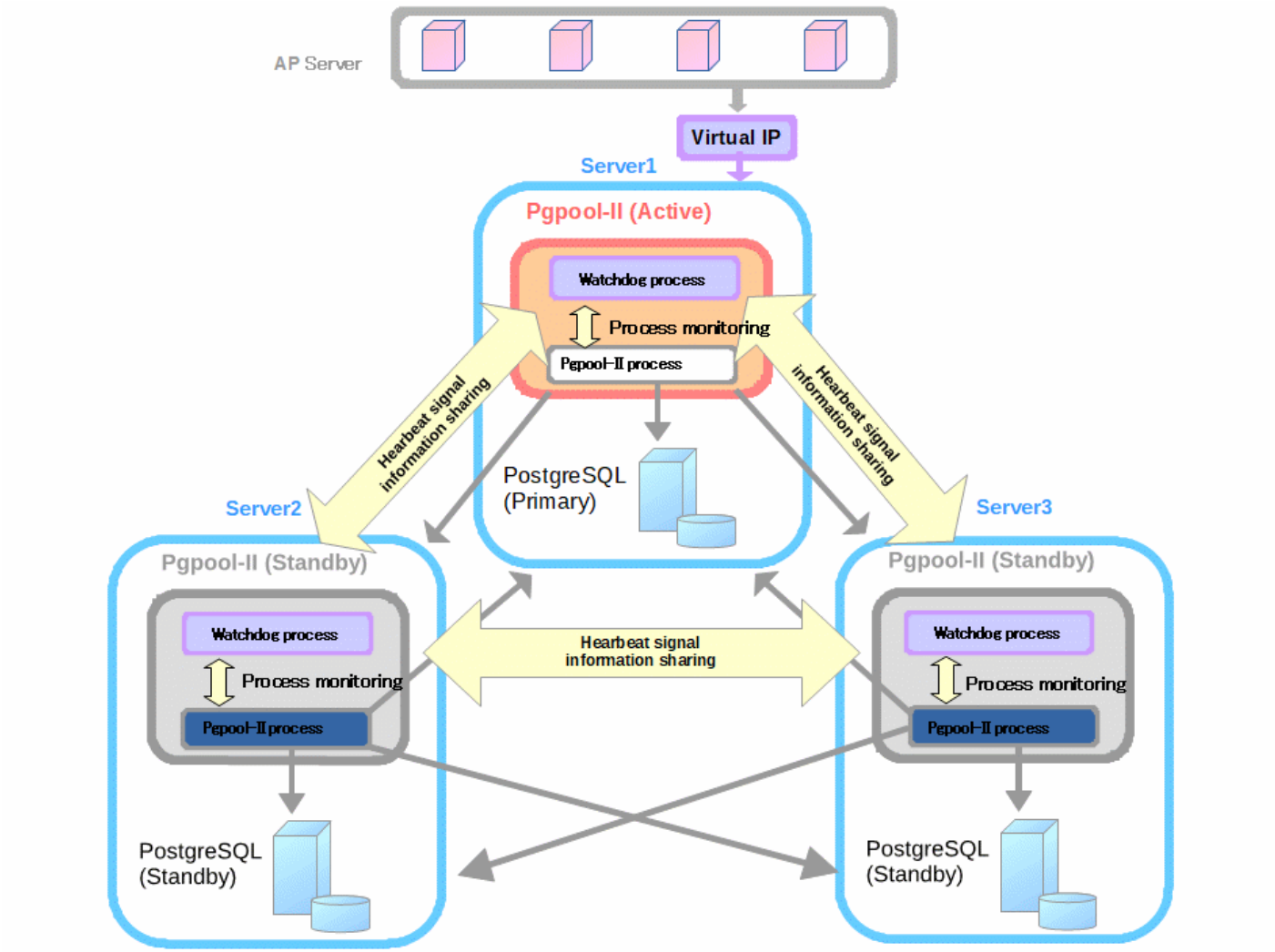
1.2 Failover

Visoka dostupnost baze podataka je jedan od glavnih motiva za korišćenje replikovanog klastera baze podataka. Postoji više čvorova koji čuvaju trenutno stanje podataka, a mogu se nalaziti u različitim data centrima ili čak i lokalno. U slučaju da dođe do kvara softvera ili hardvera na jednom od čvorova, još uvek postoji nekoliko drugih čvorova, koji nose identične podatke i mogu da obrađuju dolazne zahteve.

Postoje dva različita scenarija katastrofe kada se koristi replikacija *single-master*, koja se popularnije naziva *master-slave* replikacija. Kada se kvar pojavi na jednom od podređenih čvorova, može se ponovo pokrenuti ili isključiti i zameniti novim čvorom. Pogođena *slave* replika neće prihvatati povećanje podataka neko vreme, ali može da povuče sve promene i da se sinhronizuje sa glavnim čvorom nakon što se oporavi. To ni malo ne utiče na funkcionalnost i performanse klastera.

Najgora situacija jeste, kada dođe do oštećenja podataka ili neke vrste softverskog kvara na glavnom čvoru. Uobičajeno rešenje takvog scenarija je odabir jedne od dostupnih *slave* replika i promovisanje iste u novi *master* čvor. Ova replika počinje da obrađuje zahteve za pisanje nakon unapređenja (nije više u read-only modu). Svi ostali čvorovi, uključujući i stari *master* čvor, u slučaju da je ponovo dostupan, moraju biti obavešteni o novom *master* čvoru. Opisana procedura promovisanja *slave* čvora u *master* se naziva *failover*.

Ipak nije tako jednostavno ispravno podesiti *failover*. U slučaju da se promocija dogodi odmah nakon što trenutni *master* postane nedostupan, neki mali problem sa mrežom ili kašnjenje paketa, mogu pokrenuti *failover*, čak i kada to nije neophodno. S druge strane, dug period čekanja takođe produžava rezultirajuću agoniju nedostupnosti klastera. U slučaju da komunikacija između čvorova ne funkcioniše dobro, može se desiti da više čvorova radi u glavnom režimu istovremeno. Pravilno konfigurisan i funkcionalan *failover* je u osnovi jedini način da se postigne razuman nivo visoke dostupnosti.



Slika 2. Pgpool middleware failover control

2 Cloud rešenja

Novi pristup dizajniranju, kreiranju i primeni aplikacija je mnogo drugačiji od tradicionalnog načina, koji je pešačkog tipa. Glavni ciljevi su brže izvođenje novih usluga na tržište, minimiziranje troškova i povećanje skalabilnosti usluga. Ključ za postizanje ovih ciljeva je automatizacija svih faza životnog ciklusa usluge (*CI/CD pipeline*). Principi poput arhitekture mikro-servisa, platforme za kontinuiranu isporuku i orkestraciju kontejnera koriste se za automatizaciju procesa životnog ciklusa softvera. Ovaj poseban pristup životnom ciklusu aplikacije u kombinaciji sa novim alatima se obično naziva "*Cloud native*". Ovo poglavlje opisuje osnovne komponente i platforme za orkestraciju kontejnera i srodne *Cloud native* tehnologije.

2.1 Orkestracija kontejnera

Kontejneri kao standardizovani format koji nosi u sebi sve što aplikacija zahteva, je efikasan osnovni element za izgradnju skalabilnih usluga. Međutim, ovo je samo prvi korak na dugom putu ka rešavanju problema pokretanja velikog broja usluga u velikom obimu.

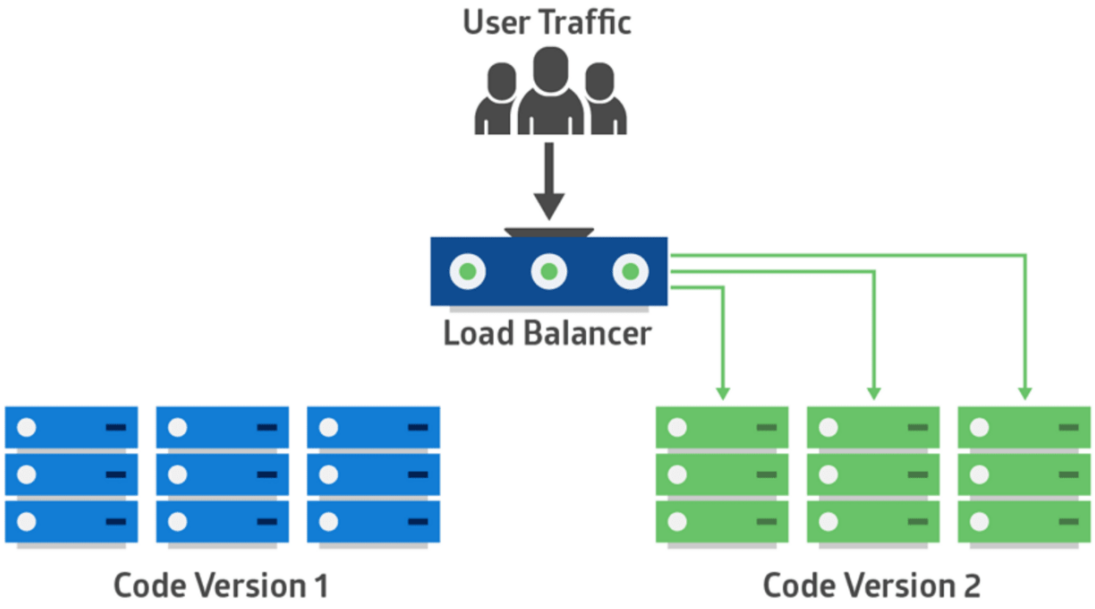
Klaster je često sastavljen od mnogih instanci aplikacije, za rukovanje hiljadama zahteva svakog minuta, koje bi trebalo da prežive greške ili otkazivanja nekih od instanciranih kontejnera. Sama usluga je obično podeljena na više jedinica koje međusobno komuniciraju. Aplikaciju treba da podržava skladište podataka ili baza podataka, a takođe i koncept keš memorije.

Upravljanje i koordinacija takvog klastera uključuje mnogo različitih aktivnosti. Svi ovi kontejneri moraju biti zakazani i pokrenuti u pravom trenutku, ponekad moraju početi određenim redosledom, tačnije zavise jedan od drugog. U slučaju da jedan od kontejnera prestane da radi, drugi kontejner iste vrste mora biti pokrenut kao zamenski. Krajnje tačke aplikacije moraju biti izložene spoljašnjem svetu (*public IP*). U slučaju da se primenjuje nova verzija aplikacije, stari kontejneri moraju biti zamenjeni na pametan način kako bi se smanjilo vreme zastoja aplikacije. Termin orkestracija kontejnera se koristi za koordinaciju i sekvenciranje ove vrste aktivnosti u klasteru.

2.2 Blue green deployment

Blue green deployment je model izdavanja aplikacije, koji postepeno prenosi korisnički saobraćaj sa prethodne verzije aplikacije ili mikro-servisa na skoro identično novo izdanje. Stara verzija se može nazvati plavim okruženjem,

dok se nova verzija naziva zelenim okruženjem. Kada se proizvodni saobraćaj u potpunosti prenese iz plave u zelenu, plava može biti u stanju pripravnosti u slučaju vraćanja ili povučena iz proizvodnje i ažurirana da postane šablon na osnovu kojeg se vrši sledeće ažuriranje.



Slika 3. Blue green deployment

Na ovaj način, nova verzija koda se može izdati bez ikakvog zastoja. To je moguće jer je uzet mikro-servis u proizvodnom okruženju (plavo) i kopiran u identičan — ali poseban — kontejner (zeleno). Kod prolazi kroz fazu automatskog testiranja, uglavnom uz pomoć *Jenkins*-a, pre nego što je gurnut u proizvodno okruženje pored aktivnog plavog okruženja. Operativni tim može da iskoristi *load balancer* da preusmeri sledeću transakciju svakog korisnika sa plavog na zeleno okruženje, a kada se sav proizvodni saobraćaj filtrira kroz zeleno okruženje, plavo okruženje se stavlja van mreže. Plavo okruženje može ostati u stanju pripravnosti, kao opcija oporavka od katastrofe ili može postati kontejner za sledeće ažuriranje.

2.3 Docker Compose

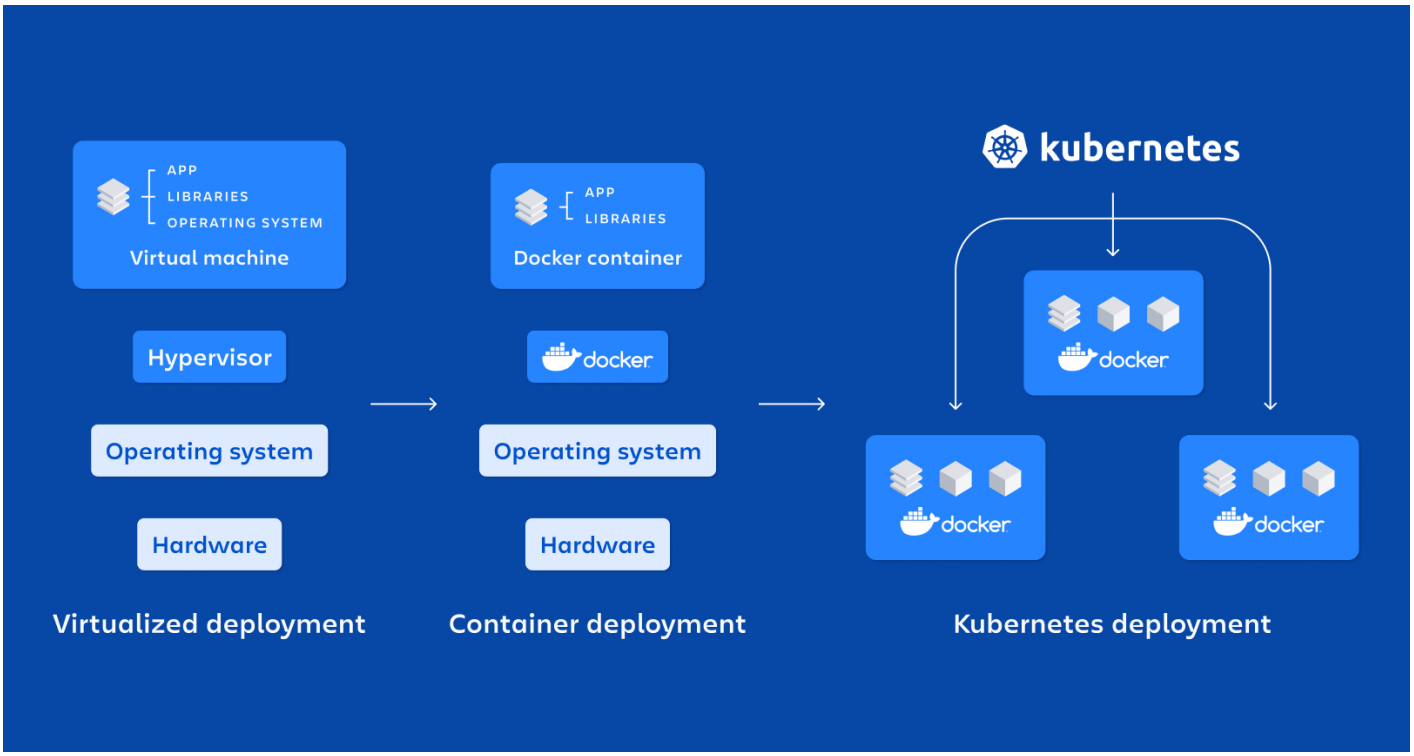
Docker Compose je alatka koja omogućava pokretanje više međusobno komunicirajućih kontejnera na jednostavan način, koristeći manifeste, u *YAML* formatu. Manifest je mesto gde se definiše željeni status. *Docker Compose* će se pobrinuti za sve radnje, koje su neophodne za održavanje tog statusa, a još bolje, to se može uraditi

jednom komandom. *YAML* format, je veoma čest format i u drugim tipovima orkestratora, koji su takođe veoma rasprostranjeni, kao što su *Kubernetes* ili *Docker Swarm*.

Swarm je *Docker*-ov izvorni alat za orkestraciju kontejnera. Može da pakuje i pokreće aplikacije kao kontejnere. *Swarm* može pronaći relevantne slike kontejnera i postaviti kontejnere na laptopove, servere, javne oblake i privatne oblake. *Docker Swarm* je lakši za konfigurisanje i korišćenje od *Kubernetes*-a, što ga čini atraktivnom opcijom za male scenarije korišćenja i razvojne timove bez *Kubernetes* stručnosti.

2.4 Kubernetes

Kubernetes uspon bio je meteorski. Dok su drugi sistemi orkestracije kontejnera postojali pre *Kubernetes*-a, oni su bili komercijalni proizvodi vezani za prodavca, a to je uvek bilo prepreka njihovom širokom usvajanju. Sa dolaskom zaista besplatne orkestracije kontejnera otvorenog koda, usvajanje i kontejnera i *Kubernetesa* raslo je fenomenalnom brzinom. Do kraja 2017., orkestracijski ratovi su bili gotovi, a *Kubernetes* je pobedio. Dok su drugi sistemi još uvek u upotrebi, od sada kompanije koje žele da premeste svoje infrastrukture do kontejnera treba da ciljaju samo jednu platformu: *Kubernetes*.



Slika 4. VM vs Docker vs Kubernetes

Glavne prednosti *Kubernetes*-a, pored automatizacije operativnih tokova rada, su visoka dostupnost, balansiranje opterećenja, automatsko skaliranje, samonadgledanje i samoizlečenje. Visoka dostupnost se postiže zahvaljujući distribuciji važnih delova sistema na više čvorova. Balansiranje opterećenja zajedno sa automatskim skaliranjem sprečava preopterećenje usluga koje rade na *Kubernetes*-u. Dolazni zahtevi se distribuiraju po skupu kontejnera i u slučaju da skup nije dovoljno velik prema nekim praćenim metrikama, više kontejnera se dodaje automatski. Kontejneri koji rade na *Kubernetes* klasteru se nadgledaju i zamenjuju ili ponovo pokreću u slučaju da se otkriju neki kvarovi. Ovo čini platformu samoisceljujućom.

Kubernetes se fokusira na upravljanje primenama kontejnera u računarskom klasteru, uključujući njihovu međusobnu komunikaciju. Tipično, format kontejnera koji se koristi je *Docker* format kontejnera. Podržano je više formata kao što je *containerd*, a novi se mogu priključiti po potrebi koristeći *Kubernetes*-ov interfejs za izvršavanje kontejnera (CRI). Najmanja operativna jedinica *Kubernetes*-a je *Pod*. *Pod*-ovi sadrže jedan ili više kontejnera. *Pod*-ovi se obično ne kreiraju ručno, već takozvanim *Deployment*-om.

Glavni razlog za uspeh *Kubernetes*-a je dostupnost u modernom oblaku. Svi glavni dobavljači oblaka (*Amazon AWS*, *Microsoft Azure*, *Google Cloud*) imaju dostupnu *Kubernetes* ponudu. *Kubernetes* se lako može integrisati sa implementacijama za skladištenje i umrežavanje bilo kog dobavljača oblaka.

2.5 OpenShift

OpenShift je platforma koja pruža *Kubernetes* podršku sa plaćenom podrškom. *OpenShift* koristi *Kubernetes* kao osnovu i gradi dodatne slojeve polisa, tokova slika kontejnera i podrške softverske komponente na vrhu. *OpenShift* pruža strožu sigurnosnu politiku i integrisanu *CI/CD* i web konzolu za upravljanje klasterima. Od verzije *OpenShift 4* postoji i integrisani *OperatorHub*, skladištenje operatora zajednice. Ovi operateri postaju jedan od uobičajeni načina za *deployment* na *OpenShift* i *Kubernetes* platformi. Korišćenje *OpenShift* umesto čistog otvorenog koda *Kubernetes* projekta donosi i neka ograničenja. *Kubernetes* podržava mnoge *Linux* distribucije, dok se *OpenShift* može lako instalirati samo na distribucije kao što su *Red Hat Enterprise Linuk* (RHEL), *CentOS* i još nekoliko *Red Hat* alternativa.

2.6 Minikube

Podešavanje potpuno funkcionalnog distribuiranog *Kubernetes* klastera zahteva mnogo resursa, konfigurisanja i rada na održavanju. Kada se *Kubernetes* aplikacije razvijaju lokalno ili treba da se testiraju, izgradnja visoko

dostupnog distribuiranog klastera obično nije neophodna. *Minikube* je alatka za lokalno pokretanje *Kubernetes* klastera sa jednim čvorom. Podržava sve glavne funkcije *Kubernetes*-a i može da radi na prosečnom personalnom računaru ili laptopu. Postoji i opcija za lokalno pokretanje *OpenShift* klastera. *Minishift* je dizajniran slično kao *Minikube*, to je lagano rešenje koje omogućava pokretanje klastera jednog čvora koji sadrži i sve *OpenShift* slojeve na vrhu *Kubernetes*-a.

3 PostgreSQL High-Availability arhitektura

Na mnogo načina, arhitektura servera baze podataka se tretira samo kao naknadna misao. Često je mnogo lakše jednostavno kreirati jedan čvor, instalirati neki softver i smatrati da je cela stvar rešena. Ako je kompanija posebno paranoična, možda čak pomisle o ugradnji replike, ili možda o nekoj vrsti rezervne kopije. U ovom poglavlju će biti pretstavljeni arhitektonski *PostgreSQL High-Availability* principi.

Arhitektura baze podataka definiše šta ulazi u klaster servera baze podataka i razlog za svaki gradivni element. Kako komunicira? Koliko čvorova je potrebno? Gde staviti čvorove, i zašto? Koji su zajednički problemi uzrokovani tim odlukama? Kako će odluke uticati na osnovne troškove? Koje kompromise je bezbedno napraviti, s obzirom na neka važna ograničenja? Kako sve ovo utiče na dostupnost podataka? Postoji mnogo važnih razmatranja prilikom instanciranja visoko dostupnog *PostgreSQL* klaster-a.

Zašto je onda tako uobičajeno da se kritične aplikacije i korisnički podaci koji pokreću čitav niz aplikacija iza same kompanije tretiraju tako bezobzirno? Mnogo pažnje i fokusa usmereno je na aplikaciju, sa njenim različitim slojevima indirektnosti, keš memorije, automatizacijom kontejnera i mikro-arhitekturom, dok je sloj podataka totalno zanemaren. Ovo je zapravo veoma razumljivo, jer u većini slučajeva, sloj baze podataka *PostgreSQL* zahteva potpuno drugačiji pristup, sa kojim razvoj, administracija sistema i druga polja informacionih tehnologija možda nisu u potpunosti upoznati. Čak i iskusni administratori baza podataka možda neće razumeti razmere i neophodne teorijske koncepte koji pokreću visoku dostupnost baza podataka.

3.1 Biranje redundantnih kopija

Broj redundantnih kopija podataka na kraju određuje koliko čvorova mora postojati, bez obzira na to da li nam je potrebno više centara podataka, bez obzira na to da li treba da uzmemo u obzir kašnjenje, itd. Katastrofalni kvar je neosporiva činjenica i veoma moguć, stoga za takav događaj treba biti spreman. Čak i ako zasebni server nije potpuno operativan *PostgreSQL* čvor, on mora postojati i treba da bude deo referentnog dizajna. Isto tako, moramo imati najmanje jednu *PostgreSQL* repliku.

Jednostavno, svaki visoko aktivan *PostgreSQL* klaster mora imati najmanje dva čvora koji mogu da ispune ulogu primarne baze podataka. Za vraćanje rezervnih kopija potrebno je vreme, dok se replike generalno mogu omogućiti za pisanje jako brzo.

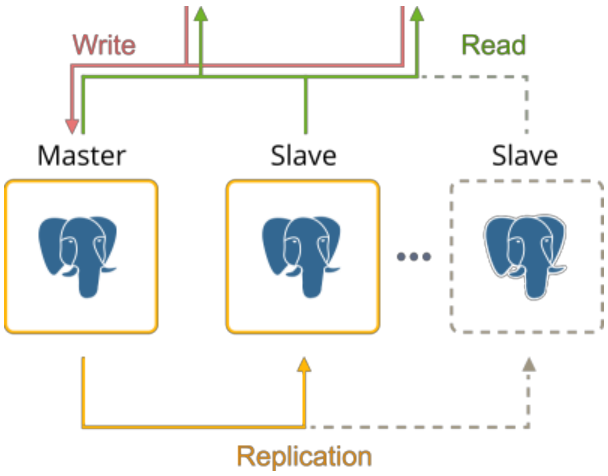
Jedna replika zaista pokriva samo slučaj kada je prelazak sa jednog *PostgreSQL* čvora na alternativni ručna procedura. Potpuno automatizovani mehanizmi za otkrivanje grešaka zahtevaju neparan broj čvorova za potrebe glasanja. Ovaj treći čvor može biti ili običan entitet za glasanje, ili puna *PostgreSQL* replika.

3.2 Kvorum

Kvorum se najbolje može objasniti zamišljanjem bilo kog sistema glasanja. To je rezultat pouzdanog konsenzusa i oslanja se na višestruke implementacije podržane disertacijom i kvantitativnom studijom. Najčešći način da se garantuje kvorum za *PostgreSQL* klaster, jeste korišćenje čvora *witness* (svedoka). On postoji samo da bi glasao i posmatrao stanje klastera. Ovo pomaže u postizanju maksimalne dostupnosti, garantujući da uvek postoji aktivan primarni čvor.

3.3 Master-Slave arhitektura

PostgreSQL dozvoljava replikaciju na čvorove koji mogu da pokreću upite samo za čitanje. Ako primarni server postane neaktivan, sekundarni server je moguće promovisati u primarni i pritom omogućiti upis podataka. *PostgreSQL Master-Slave* replikacija omogućava kompanijama da održavaju svoje poslovne aktivnosti čak i kada jedan server nije u pogonu, jer web lokacija može da pristupi repliciranom serveru. *PostgreSQL Master-Slave* replikacija pomaže u skaliranju aplikacije bez brige o otkazu sistema.



Slika 5. Master-slave arhitektura

Vrlo je verovatno da aplikacija ima mnogo upita za čitanje i vrlo malo upita za pisanje. U takvom slučaju, idealna strategija bi bila implementacija *Master-Slave* replikacije, što znači replikaciju glavnog servera na nekoliko *slave* servera. *Slave* serveri se koriste samo za upite za čitanje i kada dođe do *failover*-a.

Ovaj tip replikacije omogućava da se promene koje se dešavaju u klasteru primarne baze podataka kopiraju u rezervnu bazu podataka na drugom serveru. Na taj način se baza podataka distribuira na nekoliko različitih mašina, čime dobijamo rezervnu kopiju, a upiti se mogu rešavati bez zastoja.

Kada je sistem u problemu sa porastom saobraćaja ili kada se dođe do velikog porasta pretplatnika na uslugu, baza podataka može početi da usporava. Kada baza podataka postane slaba karika u lancu, postaje ranjiva na razne vrste neočekivanih grešaka. Jedno rešenje bi bila nadogradnja servera, kako bi postao moćniji i sposobniji da obrađuje više upita. Ovaj pristup rešava trenutni problem uskog grla baze podataka, ali je moguće doći u situaciju, da nadogradnja više nije moguća u bliskoj ili daljoj budućnosti. Bolje rešenje bi bilo dodavanje više servera paralelno. Na taj način se postiže skalabilnost, pri čemu se takođe osigurava, da neuspeh jedne instance ne uruši ceo sistem.

3.4 Multi-Master arhitektura

Neki dobavljači *PostgreSQL*-a obezbeđuju proširene funkcionalnosti koje omogućavaju da klaster sadrži više primarnih čvorova za pisanje istovremeno. Korisnici ove vrste softvera mogu očekivati određene poboljšane mogućnosti, iako su ustupci često neophodni. Očigledna korist koja proizilazi iz korišćenja višestrukih *master PostgreSQL* čvorova je smanjenje latencije prilikom upisivanja u bazu.

Ako je klaster podešen sa serverima koji se nalaze na različitim kontinentima, svako upisivanje mora prvo da pređe hiljade kilometara, pre nego što se izvrši. Zbog načina na koji replikacija funkcioniše, lokalne replike u tim regionima će morati da sačekaju da se transakcija ponovo reprodukuje pre nego što bude vidljiva u tom regionu. Vreme latencije može itekako biti značajno.

Svako upisivanje može zahtevati više od nekoliko stotina *ms* samo da bi se došlo do primarnog čvora. Zatim, isti podaci moraju da se prenesu sa primara na svaki *standby* čvor, udvostručujući tako vreme potrebno da transakcija bude vidljiva na kontinentu odakle potiče. Pošto mnoge radnje aplikacije mogu pozvati više transakcija, to može izazvati efekat vremenskog pojačanja koji bi u ekstremnim slučajevima mogao trajati i nekoliko minuta.

Zbog toga je bitno, da li aplikacija obavlja više radnji po zadatku. Za prikazivanje web stranice može biti potrebno i više desetina upita. Podnošenje transakcionih zahteva, može značiti nekoliko pisanja i zahtevanja rezultate. Svaka

sekunda čekanja povećava šanse da korisnik jednostavno pređe na drugu aplikaciju, bez problema sa kašnjenjem. Ako bi svaki od tih čvorova bio primarni, troškovi pisanja transakcije bi bili efektivno svedeni na nulu. Ova funkcionalnost može eliminisati vreme promocije čvora i omogućiti potpuno aktivan *stek* aplikacija na svim uređajima.

4 PostgreSQL High-Availability

PostgreSQL je široko prihvaćen kao moderna transakciona baza podataka odličnih performansi. Visoko dostupan *PostgreSQL* klaster može izdržati kvarove uzrokovane prekidima mreže, zasićenjem resursa, kvarovima hardvera, rušenjem operativnog sistema ili neočekivanim ponovnim pokretanjem. Takav klaster je često kritična komponenta okruženja poslovnih aplikacija, gde je četiri devetke dostupnosti minimalni zahtev.

Dostupnost se obično izražava kao procenat radnog vremena u toku godine. Ugovori o nivou usluge često se odnose na mesečne zastoje ili dostupnost, kako bi se izračunala cena za uslugu, koja odgovara mesečnim ciklusima naplate. Sledeća tabela prikazuje prevod sa datog procenta dostupnosti na odgovarajući vremenski period kada sistem ne bi bio dostupan.

Dostupnost %	Downtime po godini
90% ("one nine")	36.53 dana
99% ("two nines")	3.65 dana
99.9% ("three nines")	8.77 sati
99.99% ("four nines")	52.60 minuta
99.999% ("five nines")	5.26 minuta

Tabela 1. Dostupnost

U nastavku poglavlja, biće pretstavljena neka od dostupnih rešenja za *PostgreSQL High-Availability* klaster.

4.1 PostgreSQL izvorna replikacija

PostgreSQL pruža izvornu podršku za podešavanje replikacije, a samim tim i kreiranje klastera visoke dostupnosti. *PostgreSQL* koristi nešto drugačije terminologija. Termin *primarni* se koristi za *master* čvor, a *slave* čvor

se naziva *standby* čvor. Pored matične replikacije, postoji više alata i sistema za visoku dostupnost. Ovi alati pružaju dodatne funkcije i zahtevaju manje ručnog rada i konfigurisanja.

Ugrađena *PostgreSQL* replikacija je sasvim dovoljna za izgradnju klastera, koji je spreman za proizvodno okruženje. Podržava režim visoke dostupnosti toplog stanja pripravnosti/log Shipping (*Warm Standby/Log Shipping High-Availability mode*), što znači da se podaci repliciraju na rezervne čvorove, ali nisu podešeni za rukovanje upitima. Rezervne replike su samo rezervna kopija, pripremljena da se unapredi u glavnu repliku u slučaju failover-a. U verziji 9.0 dodata je *Hot Standby/Streaming* replikacija. Dolazni upiti samo za čitanje, mogu se izbalansirati na replike u stanju pripravnosti u režimu replikacije strimovanja i sprečiti lako preopterećenje glavnog čvora.

Pre nego što bilo šta kopiramo, moramo da odredimo šta da kopiramo. U nekim slučajevima, možda će biti potrebno kopirati celu bazu podataka u svrhe oporavka od katastrofe. U drugim slučajevima, takvo kopiranje bi trošilo resurse. Moramo da napravimo razliku između ova dva scenarija. Kada to uradimo, trebalo bi da odlučimo šta da radimo kada ne želimo da kopiramo celu bazu podataka. Moramo da znamo koje tabele da kopiramo i gde da ih pošaljemo.

Primarni mehanizam koji *PostgreSQL* koristi da obezbedi garanciju trajnosti podataka je *Write-Ahead Log* (*WAL*). Svi transakcioni podaci se zapisuju na ovu lokaciju pre nego što se unesu u datoteke baze podataka. Jednom kada *WAL* datoteke više nisu neophodne za oporavak od pada, *PostgreSQL* će ih ili izbrisati ili arhivirati. Za visoko dostupan server, preporučeno je čuvanje ove važne datoteke što je duže moguće. Postoji nekoliko razloga za to:

- Arhivirane *WAL* datoteke se mogu koristiti za *Point-In-Time Recovery* (*PITR*).
- Ako se koristi *streaming* replikacija, prekinuti tokovi se mogu ponovo uspostaviti primenom *WAL* datoteka dok se replika ne podigne.
- *WAL* datoteke se mogu ponovo koristiti za servisiranje više kopija servera.

Za korišćenje ovih prednosti, neophodno je omogućiti *PostgreSQL WAL* arhiviranje i sačuvati ove datoteke sve dok više ne budu potrebne.

Veoma je dobra praksa, ako ne i direktan zahtev, imati drugu onlajn kopiju *PostgreSQL* servera u klasterima visoke dostupnosti. Bez takvog servera na mreži, oporavak od prekida može zahtevati sate odgovora na incidente, oporavak rezervne kopije i obezbeđivanje servera. Pored toga, proces postavljanja *hot standby* služi kao osnova za kreiranje *PostgreSQL streaming* replika.

Replika serveri mogu da se povežu na *upstream PostgreSQL* server i direktno konzumiraju modifikacije podataka. Sa malim kašnjenjem mreže i brzim transakcijama, to znači da je prilično uobičajeno da replike za strimovanje zaostaju za *master*-om samo nekoliko mili-sekundi. U kontekstu visoke dostupnosti, to znači da možemo horizontalno skalirati

kopiranjem baze podataka na više servera. Naravno, to znači da moramo da kopiramo celu bazu podataka na svaki server. Za male do srednje instance baze podataka, ovo je relativno mali zahtev. To takođe znači da možemo da proizvodimo najnovije rezervne kopije, vršimo ad-hoc upite na praktično ažurnim podacima i agregiramo informacije u izveštaje bez ometanja primarne baze podataka.

4.2 Slony

Iako postoji nekoliko logičkih asinhronih sistema replikacije za *PostgreSQL*, *Slony* je bio prvi koji je široko prihvaćen. Zašto bismo koristili *Slony* kada *PostgreSQL* već ima fizičku i logičku replikaciju? Verzije *PostgreSQL*-a starije od *v10*, mogle su da kopiraju samo celu instalaciju. Jedina opcija za te sisteme je kopiranje svake baze podataka, šeme, tabele i korisnika na binarnom nivou. Neki od glavnih ciljeva dizajna kod *Slony* alata, bili su proširivost sistema replikacije i koncept kaskadnih slave čvorova. Ideja kaskadnih čvorova je da svaki čvor može proslediti ažuriranja drugim čvorovima. Ako podređeni čvorovi mogu da šalju podatke drugim slave replikama, to sprečava preopterećenje glavnog čvora.

4.3 Bucardo

Slučajevi korišćenja *Bucardo* programa replikacije su slični kao kod *Slony* alata. Osnovni deo *Bucarda* je demon napisan u *Perl* programskom jeziku. Informacije o replikaciji potrebne za demon se čuvaju u glavnoj *Bucardo* bazi podataka. Ova baza podataka sadrži listu baza podataka uključenih u replikaciju, informacije o tabelama i režimu replikacije. Demon osluškuje zahteve za obaveštavanje, povezuje se sa udaljenim bazama podataka i kopira podatke. *Bucardo* demon, koji kontroliše replikaciju, može da radi na jednom od servera uključenih u replikaciju ili na zasebnom serveru. Jedna od prednosti koju *Bucardo* pruža je podrška za režim replikacije sa više mastera.

4.4 Pglogical

PostgreSQL verzija 10 uvela je početnu podršku za logičku replikaciju, što je alternativni pristup standardnoj *streaming* (fizičkoj) replikaciji u *PostgreSQL*-u. *Pglogical* proširenje je razvijeno da koristi najnovije *PostgreSQL* karakteristike i podržava nadogradnje između glavnih verzija baze podataka, selektivnu replikaciju skupova tabela i spajanje podataka sa više servera uzvodno. Ovo u suštini omogućava dekodiranje evidencije transakcija i izdvajanje saobraćaja pisanja baze podataka za daljinsko ponavljanje na logičkom nivou. Za razliku od standardne replikacije,

koja zahteva da primarni čvor i replika budu identični, slotovi se mogu izvući za specifične informacije relevantne za potrebe korisnika.

4.5 Repmgr High-Availability

Repmgr je alatka otvorenog koda koja poboljšava ugrađenu *PostgreSQL* sposobnost replikacije. *Repmgr* je lagan alat, implementiran kao potpuno *PostgreSQL* proširenje. *Repmgr* pojednostavljuje inicijalizaciju klastera tako što pakuje delove podešavanja i operacija replikacije u skup jednostavnih komandi. U velikoj meri pojednostavljuje proces postavljanja i upravljanja bazama podataka sa zahtevima visoke dostupnosti (*HA*) i skalabilnosti. Pomaže *DBA* i sistem administratorima da upravljaju klasterom *PostgreSQL* baza podataka, koristeći prednosti i mogućnosti *Hot Standby* režima. *Repmgr* pojednostavljuje proces administracije i svakodnevnog upravljanja, poboljšava produktivnost, dopunjuje ugrađene mogućnosti replikacije i smanjuje ukupne troškove *PostgreSQL* klastera kroz:

- praćenje procesa replikacije.
- pružanje podrške za *HA* operacije, kao što je prebacivanje na *failover*.

Prva važna komponenta *Repmgr*-a je alatka komandne linije koja služi za administrativne radnje kao što je podešavanje čvora, promovisanje odabranog *slave*-a u *master* i prikazivanje statusa klastera. Druga komponenta je demon koji se zove *repmgrd*. Administrator može opciono da pokrene ovaj demon za upravljanje i nadgledanje na svakom čvoru u klasteru. *Repmgrd* je dizajniran da automatizuje mnoge radnje uključujući *failover* i usmeravanje na druge *slave* čvorove.

4.6 Patroni High-Availability

Patroni je softver za upravljanje visoke dostupnosti koji je razvio *Zalando*, za orkestriranje i automatizaciju nekoliko aspekata *PostgreSQL* klastera. Za razliku od *repmgr*-a, on zahteva niz drugih komponenti i komunicira sa njima kao niz slojeva da bi proizveo krajnji rezultat visoke dostupnosti. Razlog zašto *Patroni* funkcioniše kao serija softverskih slojeva, jeste smanjenje oslanjanja na bilo koju potencijalnu pojedinačnu tačku kvara. Uvek postoji element upravljanja konsenzusom i resurs za udruživanje. *Patroni* bi trebalo da isporuči klaster sa sledećim mogućnostima:

- Može automatski izabrati zamenu u slučaju *failover*-a.
- Može da preusmeri veze sposobne za pisanje na novoizabrani primarni čvor.
- Novo obezbeđeni čvorovi mogu da se dodaju u klaster.

- Oporavljeni primarni čvorovi mogu ponovo da se pridruže klasteru kao replike.

Za izgradnju klastera, neophodan je pouzdan sloj za prenošenje poruka. Neki preduzimljivi studenti na Univerzitetu *Stanford* smislili su konsenzus algoritam, koji su nazvali *Raft*. Postoji mnogo teorija o tome kako to funkcioniše, ali krajnji rezultat je da par ključ-vrednost uskladišten u sloju zasnovanom na *Raft*-u ostaje interno konzistentan na svim serverima i generalno je otporan na mrežne particije.

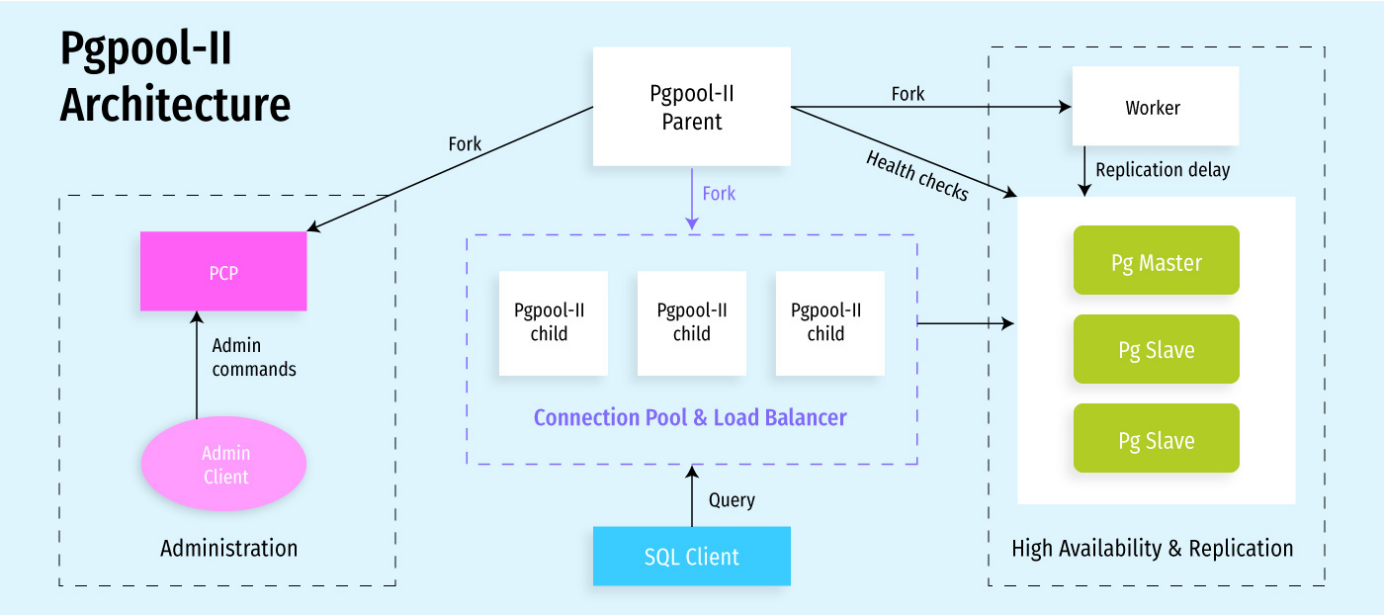
Za razliku od *repmgr*-a, koji direktno implementira algoritam, *Patroni* ovo čini modularnom komponentom. Podrazumevani softver koji *Patroni* koristi za *Raft* je *etcd*. Ovo je presudno važno jer se koristiti za čuvanje lokacije primarnog Postgres servera. Pod uslovom da imamo uslugu koja se može povezati na *etcd*, bilo koji od naših Postgres servera će odmah znati lokaciju primarnog sistema. Zbog toga je trivijalno menjati izvore replikacije kada se promeni primarni sistem.

Partoni koristi *HAProxy*, kod koga se svaka IP adresa ponaša kao da je primarni čvor. Dakle, sve dok se povezujemo preko porta dodeljenog *HAProxy* alatu, komuniciramo sa bilo kojim čvorom koji je primarni u tom trenutku. *HAProxy* ima ulogu sloja za rutiranje.

4.7 Pgpool-II

Iako se skup funkcija koje pruža *Pgpool-II* preklapa sa prethodno objašnjenim alatima, njegov dizajn i namena su drugačiji. *Pgpool-II* služi kao *proxy*, preusmerava zahteve klijenta *PostgreSQL* baze podataka na servere i šalje rezultirajuće podatke nazad klijentu.

Objedinjavanje veza je osnovna karakteristika *Pgpool-II*. Uspostavljene veze se efikasnije koriste zahvaljujući ovom mehanizmu. Još jedna korisna karakteristika je balansiranje opterećenja upita (*load balancing*) samo za čitanje na više *slave* servera. Sama replikacija i prelazak na failover se takođe mogu rukovoditi na nivou *Pgpool-II* alata. Podaci se ne strimuju od glavnog čvora do replika, nego bivaju *backup*-ovani na replike od strane *Pgpool-II* alata koji posreduje između njih.



Slika 6. Pgpool-II

5 Praktična primena

Sloj podataka je suštinska komponenta aplikacija u stvarnom svetu, posebno u poslovnom sektoru. Podaci koji se čuvaju u sistemima baza podataka sa statusom su kritični deo sistema i ne mogu se udvostručiti ili zameniti tako lako kao komponente bez stanja. Pokretanje sistema baza podataka na kontejnerski način može biti veoma komplikovano. Skaliranje ili ponovno pokretanje čvorova baze podataka može dovesti do kršenja integriteta podataka ili gubitka podataka ako se ne radi pažljivo. Uobičajena praksa je da su aplikacije koje rade na *Kubernetes* klasterima zajedno sa drugim komponentama bez stanja povezane sa spoljnim sistemom baze podataka. Ovaj sistem baze podataka se obično replicira, ali nije kontejnerizovan i njime upravlja platforma za upravljanje kontejnerima. U ovom poglavlju, biće demonstrirana praktična primena tehnologija *Kubernetes* klastera i lokalnih kontejnera za instanciranje *PostgreSQL High-Availability* klastera.

5.1 PostgreSQL izvorna replikacija

Ovim rešenjem ću prikazati izvornu replikaciju *PostgreSQL*-a, u *master-slave* modu. Najpre kreiramo novu *docker* mrežu u *bridge* modu, kako bi kontejneri mogli da komuniciraju:

```
docker network create bridge-docker
```

Kod 1. Kreiranje bridge-docker mreže

Zatim gradimo kontejnere i pokrećemo ih sledećom komandom:

```
docker-compose up --build
```

Kod 2. Izgradnja i pokretanje kontejnera

Kontejneri se grade i pokreću uz pomoć predefinisanog *docker-compose.yml* fajla:

```
version: "3"
services:
  pg_master_1:
     build: ./master
     ports:
         - "5445:5432"
     volumes:
         - mdata:/var/lib/postgresql/data
```

```
    environment:
        - POSTGRES_USER=admin
        - POSTGRES_PASSWORD=root
        - POSTGRES_DB=postgres
        - PG_REP_USER=rep
        - PG_REP_PASSWORD=root
    networks:
          - bridge-docker
    restart: always
  pg_slave_1:
    build: ./slave
    ports:
        - "5446:5432"
    volumes:
        - sdata:/var/lib/postgresql/data
    environment:
        - POSTGRES_USER=admin
        - POSTGRES_PASSWORD=root
        - POSTGRES_DB=postgres
        - PG_REP_USER=rep
        - PG_REP_PASSWORD=root
    networks:
          - bridge-docker
    restart: always
    depends_on:
     - pg_master_1
  pgadmin:
    container_name: pgadmin4_container
    image: dpage/pgadmin4
    restart: always
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@admin.com
      PGADMIN_DEFAULT_PASSWORD: root
    ports:
      - "5050:80"
volumes:
  mdata:
  sdata:
networks:
  bridge-docker:
    external:
      name: bridge-docker
```

Kod 3. PostgreSQL izvorna replikacija - docker-compose.yml

Na samom startu fajla definišemo *docker-compose* verziju koju koristimo, a zatim definišemo pojedinačne servise koje ćemo koristiti, uz mrežu i volumen-e koje koristimo. Prvi definisani servis, *pg_master_1*, predstavlja *master* čvor, a drugi, *pg_slave_1*, predstavlja *slave* čvor. *PostgreSQL* standardni port 5432, se mapira na portove hosta, *Ubuntu OS* u ovom slučaju. *Master* i *slave* čvorove mapiramo na portove 5445 i 5446 respektivno. Definišemo kredencije za oba čvora, ali i volumen-e, na kojima će *PostgreSQL* čuvati podatke, kako bi isti opstali nakon restartovanja kontejnera. Za oba kontejnera definisana su dva zasebna *Docker* fajla, koji pozivaju zasebne *shell* skripte u svrhu podešavanja baze podataka u *master* i *slave* mod. *Docker* fajl *master* čvora izgleda ovako:

```
FROM postgres:latest
EXPOSE 5432
COPY ./setup-master.sh /docker-entrypoint-initdb.d/setup-master.sh
RUN chmod 0666 /docker-entrypoint-initdb.d/setup-master.sh
RUN apt-get update && apt-get --assume-yes install iputils-ping && apt-get install
--assume-yes ssh
```

Kod 4. Master Docker fajl

Shell skripta koja vrši parametriranje *master* čvora izgleda ovako:

```
#!/bin/bash
echo "host replication all 0.0.0.0/0 md5" >> "$PGDATA/pg_hba.conf"
set -e
psql -v ON_ERROR_STOP=1 --username "$POSTGRES_USER" --dbname "$POSTGRES_DB" <<-EOSQL
CREATE USER $PG_REP_USER REPLICATION LOGIN CONNECTION LIMIT 100 ENCRYPTED PASSWORD
'$PG_REP_PASSWORD';
CREATE USER postgres;
EOSQL
cat >> ${PGDATA}/postgresql.conf <<EOF
wal_level = hot_standby
archive_mode = on
archive_command = 'cd .'
max_wal_senders = 10
max_replication_slots = 10
wal_keep_size = 8
hot_standby = on
EOF
```

Kod 5. Shell Master

Docker fajl *slave* čvora izgleda ovako:

```
FROM postgres:latest
EXPOSE 5432
RUN apt-get update && apt-get --assume-yes install iputils-ping && apt-get install
--assume-yes ssh && apt-get install --assume-yes gosu
COPY ./setup-slave.sh /setup-slave.sh
RUN chmod +x /setup-slave.sh
ENTRYPOINT ["/setup-slave.sh"]
CMD ["gosu","postgres","postgres"]
```

Kod 6. Slave Docker fajl

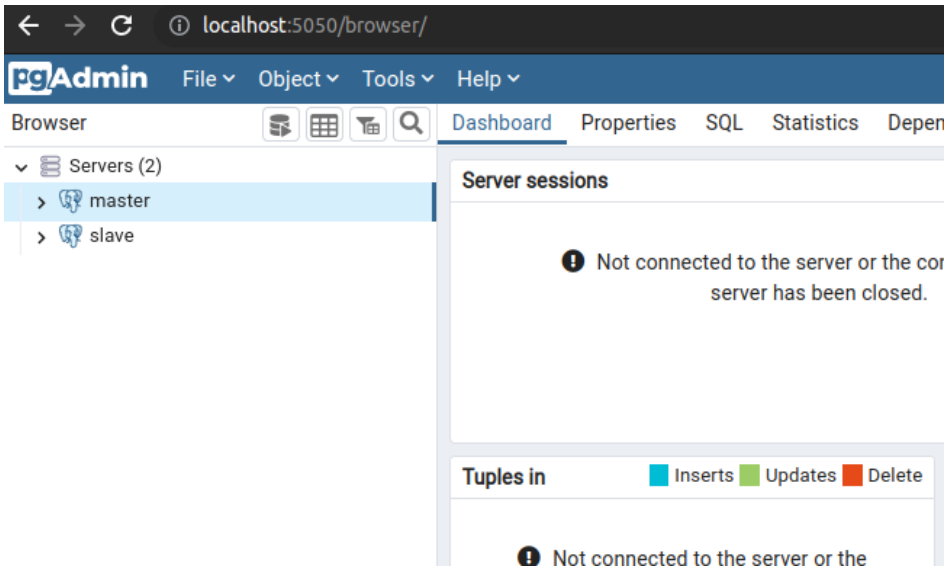
Shell skripta koja vrši parametriranje *slave* čvora izgleda ovako:

```
#!/bin/bash

if [ ! -s "$PGDATA/PG_VERSION" ]; then
      echo "*:*:*:$PG_REP_USER:$PG_REP_PASSWORD" > ~/.pgpass
      chmod 0600 ~/.pgpass
      until ping -c 1 -W 1 pg_master_1
      do
            echo "Waiting for master to ping..."
            sleep 1s
      done
      until pg_basebackup -h pg_master_1 -D ${PGDATA} -U ${PG_REP_USER} -vP -W
      do
            echo "Waiting for master to connect..."
            sleep 1s
      done
      echo "host replication all 0.0.0.0/0 md5" >> "$PGDATA/pg_hba.conf"
      set -e
      cat > ${PGDATA}/standby.signal <<EOF
      standby_mode = on
      primary_conninfo = 'host=pg_master_1 port=5432 user=$PG_REP_USER
password=$PG_REP_PASSWORD'
      trigger_file = '/tmp/touch_me_to_promote_to_me_master'
EOF
      chown postgres. ${PGDATA} -R
      chmod 700 ${PGDATA} -R
fi
sed -i 's/wal_level = hot_standby/wal_level = replica/g' ${PGDATA}/postgresql.conf
exec "$@"
```

Kod 7. Shell Master

Servis *pgadmin4* koristimo za grafički prikaz i manipulaciju bazama podataka u pretraživaču, na putanji: http://localhost:5050/.



Slika 7. Master-slave pgadmin4

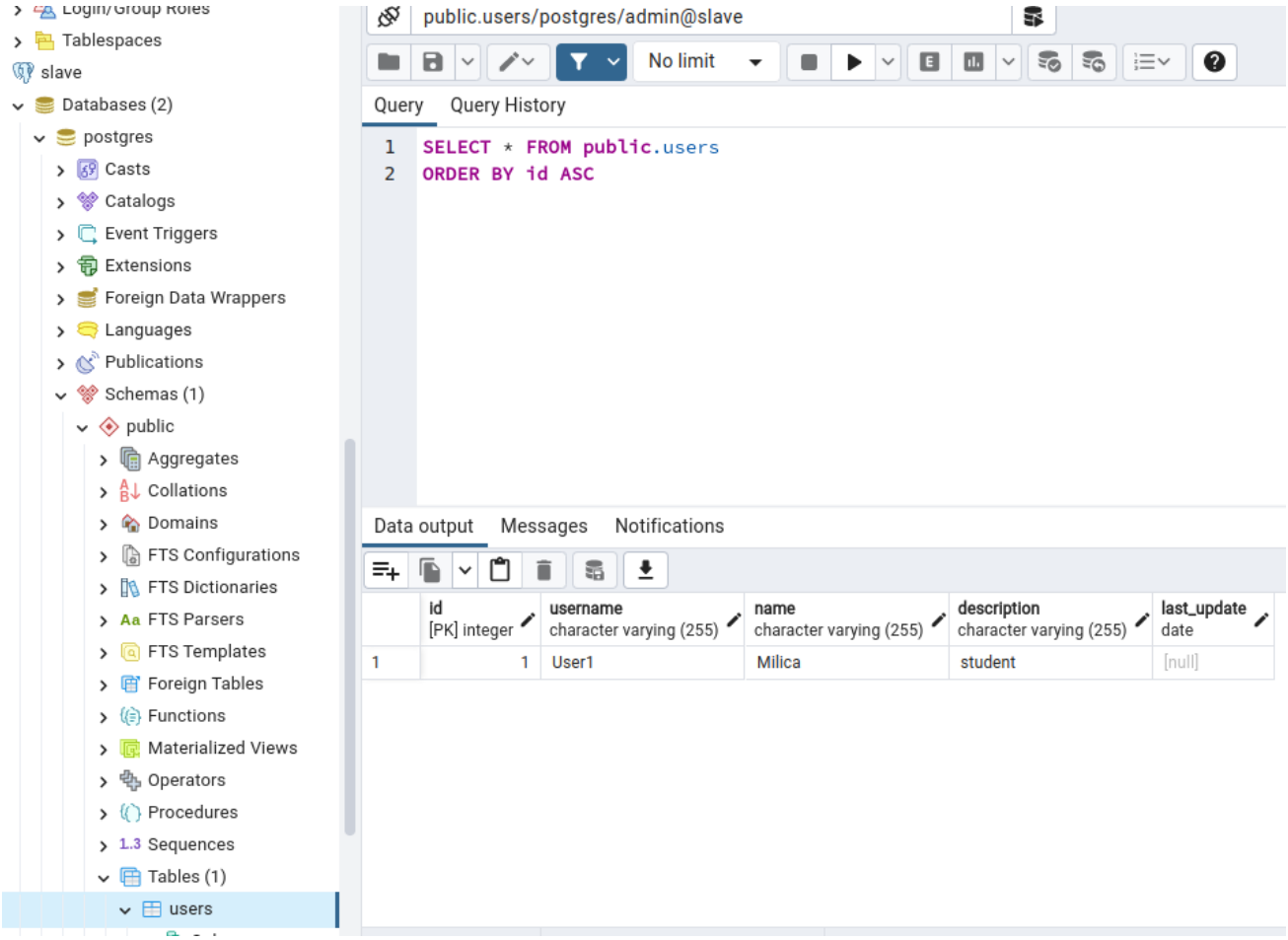
Kako bismo proverili klaster, generisaćemo tabelu u *master* bazi i proveriti da li će automatski biti replikovana na *slave* bazu.

```
CREATE TABLE users (
      id SERIAL PRIMARY KEY,
      username VARCHAR(255) NOT NULL,
      name VARCHAR(255) NOT NULL,
      description VARCHAR (255),
        last_update DATE
);

INSERT INTO users (username, name, description)
VALUES('User1', 'Milica', 'student');
```

Kod 8. SQL kreiranje tabele

Kada je *slave* čvor podešen da radi u *slave* modu, neće nam biti dozvoljena operacija kreiranja tabela na njemu, jer bi zapravo bio u *read-only* modu. Proverom je utvrđeno da je kreirana tabela uspešno replikovana sa *master*-a na *slave* čvor, što pokazuje sledeća slika:



Slika 8. Slave replikacija

5.2 Repmgr i Pgpool-II

Korišćenjem *Repmgr* i *Pgpool-II* alata, demonstriraću *failover* princip, kada u arhitekturi imamo jedan *master* čvor i dva *slave* čvora. *Docker-compose* će biti korišćen i u ovom primeru, ali će sada svi čvorovi, tj. *docker* kontejneri biti na istom portu.

Proces kreiranja klastera je identičan procesu u prošlom primeru. Najpre kreiramo *bridge* mrežu, a zatim instanciramo *docker* servise, s tim da je *docker-compose.yml* faj sada znatno izmenjen:

```
version: "3"
services:
  pg-0:
    image: bitnami/postgresql-repmgr:latest
    ports:
      - 5432
    volumes:
      - pg_0_data:/bitnami/postgresql
    environment:
      - POSTGRESQL_POSTGRES_PASSWORD=root
      - POSTGRESQL_USERNAME=admin
      - POSTGRESQL_PASSWORD=root
      - POSTGRESQL_DATABASE=postgres
      - REPMGR_PASSWORD=root
      - REPMGR_PRIMARY_HOST=pg-0
      - REPMGR_PARTNER_NODES=pg-0,pg-1,pg-2
      - REPMGR_NODE_NAME=pg-0
      - REPMGR_NODE_NETWORK_NAME=pg-0
    networks:
      - bridge-docker
  pg-1:
    image: bitnami/postgresql-repmgr:latest
    ports:
      - 5432
    volumes:
      - pg_1_data:/bitnami/postgresql
    environment:
      - POSTGRESQL_POSTGRES_PASSWORD=root
      - POSTGRESQL_USERNAME=admin
      - POSTGRESQL_PASSWORD=root
      - POSTGRESQL_DATABASE=postgres
      - REPMGR_PASSWORD=root
      - REPMGR_PRIMARY_HOST=pg-0
      - REPMGR_PARTNER_NODES=pg-0,pg-1,pg-2
      - REPMGR_NODE_NAME=pg-1
      - REPMGR_NODE_NETWORK_NAME=pg-1
    networks:
      - bridge-docker
  pg-2:
    image: bitnami/postgresql-repmgr:latest
    ports:
      - 5432
    volumes:
      - pg_2_data:/bitnami/postgresql
    environment:
```

```
      - POSTGRESQL_POSTGRES_PASSWORD=root
      - POSTGRESQL_USERNAME=admin
      - POSTGRESQL_PASSWORD=root
      - POSTGRESQL_DATABASE=postgres
      - REPMGR_PASSWORD=root
      - REPMGR_PRIMARY_HOST=pg-0
      - REPMGR_PARTNER_NODES=pg-0,pg-1,pg-2
      - REPMGR_NODE_NAME=pg-2
      - REPMGR_NODE_NETWORK_NAME=pg-2
    networks:
      - bridge-docker
  pgadmin:
    container_name: pgadmin4_container
    image: dpage/pgadmin4
    restart: always
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@admin.com
      PGADMIN_DEFAULT_PASSWORD: root
    ports:
      - "5050:80"
  pgpool:
    restart: 'always'
    image: bitnami/pgpool:latest
    ports:
      - 5432:5432
    volumes:
      - /home/aback/docker-pg-cluster/pgpool.conf:/config/pgpool.conf
    environment:
      - PGPOOL_USER_CONF_FILE=/config/pgpool.conf
      - PGPOOL_BACKEND_APPLICATION_NAMES=0,1,2
      - PGPOOL_BACKEND_NODES=0:pg-0:5432,1:pg-1:5432,2:pg-2:5432
      - PGPOOL_SR_CHECK_USER=admin
      - PGPOOL_SR_CHECK_PASSWORD=root
      - PGPOOL_ENABLE_LDAP=no
      - PGPOOL_POSTGRES_USERNAME=admin
      - PGPOOL_POSTGRES_PASSWORD=root
      - PGPOOL_ADMIN_USERNAME=admin
      - PGPOOL_ADMIN_PASSWORD=root
      - PGPOOL_AUTO_FAILBACK=yes
      - PGPOOL_CLIENT_IDLE_LIMIT=5
    healthcheck:
      test: ["CMD", "/opt/bitnami/scripts/pgpool/healthcheck.sh"]
      interval: 10s
      timeout: 5s
      retries: 5
```

```
    networks:
      - bridge-docker
    depends_on:
      - pg-0
      - pg-1
      - pg-2
volumes:
  pg_0_data:
  pg_1_data:
  pg_2_data:
networks:
  bridge-docker:
    external:
      name: bridge-docker
```

Kod 9. Repmgr i Pgpool-II - docker-compose.yml

Dodatak *Repmgr* i *Pgpool-II* servisa, omogućava automatski failover, u trenutku kada se detektuje da master čvor nije dostupan. Definisana je skripta *healthcheck*, koja u podešenom intervalu proverava servere. U slučaju da *master* čvor ne odgovori u zadatom intervalu, jedan od *slave* čvorova će automatski biti promovisan u *master* čcor. Svi ostali *slave* čvorovi, bivaju obavešteni da postoji novi *master* čvor. Nakon što se *master* čvor oporavi, biva vraćen na mesto *master*-a, pri čemu će sve izmene, dok je bio van pogona, biti replikovane na njemu.

Master čvor ćemo izbaciti iz pogona standardnom *docker* komandom:

```
docker stop docker-pg-cluster_pg-0_1
```

Kod 10. Stopiranje docker kontejnera

Lista svih aktivnih *docker* kontejnera se može videti komandom:

```
docker ps
```

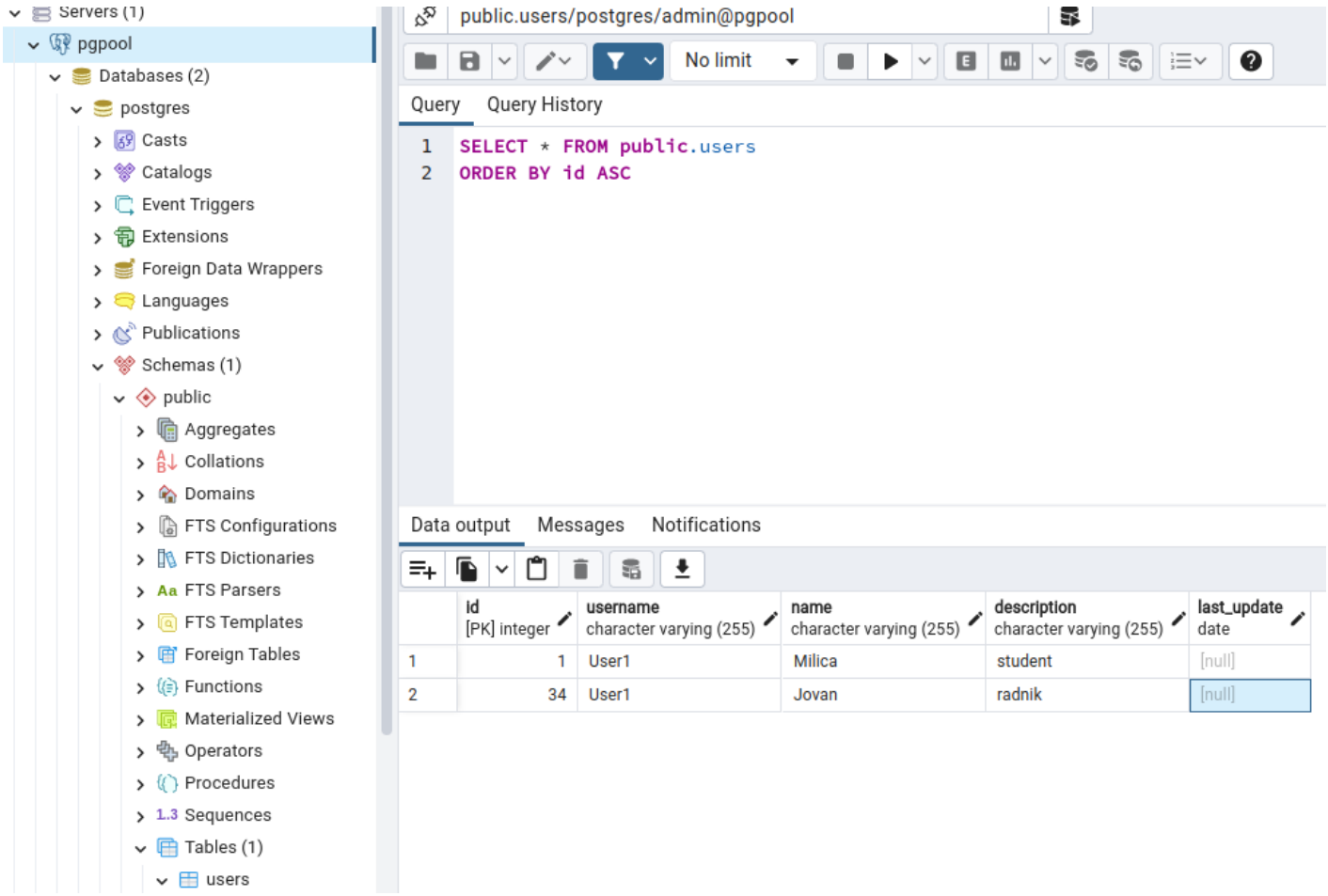
Kod 11. Lista aktivnih docker kontejnera

Nakon što *Pgpool-II* izvrši *failover* sledeća poruka se može uočiti u konzolnom log-u:

```
pgpool_1    | 2022-06-23 13:58:44.498: main pid 1: LOG:  find_primary_node: primary node is 1
pgpool_1    | 2022-06-23 13:58:44.498: main pid 1: LOG:  find_primary_node: standby node is 2
pgpool_1    | 2022-06-23 13:58:44.498: main pid 1: LOG:  failover: set new primary node: 1
pgpool_1    | 2022-06-23 13:58:44.498: main pid 1: LOG:  failover: set new main node: 1
```

Slika 9. Pgpool-II failover

Dok je *master* čvor van pogona, dodat je novi red u kreiranoj tabeli, kako bismo testirali replikaciju nakon vraćanja čvora u pogon. Unutar *pgadmin4* interfejsa, konektujemo se na samo jedan server, zapravo *Pgpool-II* koji se sada ponaša kao *ruter* ili *proxy* i nalazi se između čvorova. Kada stopiramo master čvor i odmah pokušamo da osvežimo *pgadmin4* interfejs, uočićemo da baza nije odmah dostupna, jer postoji malo zakašnjenje. Nakon par sekundi kada opet osvežimo interfejs, *slave* čvor će biti promovisan u *master*, čime nam je omogućena interakcija sa bazom. Nakon ručnog startovanja *master*-a i provere, tabela je zaista ažurirana:



Slika 10. Pgpool-II failover replikacija

Pgpool-II balansiranje opterećenja upita radi u *master-slave* režimu i u režimu replikacije. Kada je omogućeno, *Pgpool-II* šalje upite za pisanje primarnom čvoru u *master-slave* režimu. Na koji čvor mehanizam za balansiranje

opterećenja šalje upite za čitanje odlučuje se u vreme početka sesije i neće se menjati sve dok se sesija ne završi osim ako nije naveden *statement_level_load_balance*.

5.3 Kubernetes

Deployment *PostgreSQL High-Availability* klaster-a korišćenjem *Kubernetes*-a, demonstriraću na Digitalocean cloud infrastrukturi. Sama procedura je u velikoj meri olakšana korišćenjem Kubegres *Kubernetes* operator-a, koji omogućava *deployment PostgreSQL* instanci sa replikacijom i *failover*-om. Sve što treba da uradimo je da primenimo manifeste, a *Kubegres* će se pozabaviti replikacijom, failover-om i rezervnim kopijama umesto nas.

Nakon kreiranja *Kubernetes* klaster-a na *Digitalocean* platformi, moramo instalirati *kubectl* i *doctl* na lokalnoj *Ubuntu* mašini. Koristimo *doctl* da se povežemo sa kreiranim klasterom (*Digitalocean* daje jenu liniju koda za konekciju). Nakon uspešnog povezivanja, sledećom komadom možemo izlistati kreirane pod-ove u klasteru:

```
kubectl get nodes
```

Kod 12. Listanje pod-ova u klasteru

Možemo videti, da su uspešno pokrenuta 3 pod-a:

```
NAME                   STATUS   ROLES    AGE   VERSION
pool-0uhg4nzs7-c5gvb   Ready    <none>   4m    v1.22.8
pool-0uhg4nzs7-c5gvj   Ready    <none>   4m    v1.22.8
pool-0uhg4nzs7-c5gvr   Ready    <none>   4m    v1.22.8
```

Kod 13. Pokrenuti pod-ovi

Nakon uspešnog kreiranja *Kubernetes* klastera, nastavljamo sa instalacijom *Kubegres* operator-a. Instalaciju vršimo sledećom komandom:

```
kubectl apply -f
https://raw.githubusercontent.com/reactive-tech/kubegres/v1.15/kubegres.yaml
```

Kod 14. Kubegres operator instalacija

Kako je *PostgreSQL High-Availability* klaster, koji želimo da instanciramo *stateful* sistem, znači da je neophodno obezbediti volumen za čuvanje podataka. Da bismo proverili trenutni volumen klastera koristimo komandu:

```
kubectl get sc
```

Kod 15. Provera volumena

Ako su komande koje smo uneli do sada uspešno realizovane, kao odgovor na upit volumena, treba da dobijemo sledeći odgovor:

```
NAME                         PROVISIONER
do-block-storage (default)   dobs.csi.digitalocean.com
```

Kod 16. Digitalocean volumen

Zatim ćemo kreitati *postgres-secret.yaml* fajl sa kredencijama Postgres superuser-a i korisnika za replikaciju, a zatim ga primeniti, komandom *kubectl apply -f postgres-secret.yaml*:

```
apiVersion: v1
kind: Secret
metadata:
  name: mypostgres-secret
  namespace: default
type: Opaque
stringData:
  superUserPassword: root
  replicationUserPassword: root
```

Kod 17. postgres-secret.yaml

Ostaje još da primenimo novo kreirani *Kubegres resource* fajl, komadnom *kubectl apply -f postgres.yaml*:

```
apiVersion: kubegres.reactive-tech.io/v1
kind: Kubegres
metadata:
  name: mypostgres
  namespace: default
spec:
   replicas: 3
   image: postgres: 14.4
   database:
      size: 1Gi
   env:
      - name: POSTGRES_PASSWORD
        valueFrom:
```

```
          secretKeyRef:
             name: mypostgres-secret
             key: superUserPassword
       - name: POSTGRES_REPLICATION_PASSWORD
         valueFrom:
             secretKeyRef:
                name: mypostgres-secret
                key: replicationUserPassword
```

Kod 18. postgres.yaml

Konačno možemo pokrenuti pod-ove:

```
kubectl get pods -o wide -w
```

Kod 19. Pokretanje podova

Pokrenute pod-ove možemo videti sledećom komadnom:

```
kubectl get pod,statefulset,svc,configmap,pv,pvc -o wide
```

Kod 20. Pregled podova

Detaljne informacije o pod-ovima izgledaju ovako:

```
NAME                 READY   STATUS    NODE
pod/mypostgres-1-0   1/1     Running   worker1
pod/mypostgres-2-0   1/1     Running   worker2
pod/mypostgres-3-0   1/1     Running   worker3

NAME                            READY
statefulset.apps/mypostgres-1   1/1
statefulset.apps/mypostgres-2   1/1
statefulset.apps/mypostgres-3   1/1

NAME                         TYPE
service/mypostgres           ClusterIP
service/mypostgres-replica   ClusterIP

NAME
configmap/base-kubegres-config
```

```
NAME                          CAPACITY
persistentvolume/pvc-729...   1Gi
persistentvolume/pvc-6ba...   1Gi
persistentvolume/pvc-tr4...   1Gi

NAME                                                 CAPACITY
persistentvolumeclaim/postgres-db-mypostgres-1-0     1Gi
persistentvolumeclaim/postgres-db-mypostgres-2-0     1Gi
persistentvolumeclaim/postgres-db-mypostgres-3-0     1Gi
```

Kod 21. Detaljan pregled pod-ova

Sada možemo lako proveriti kojem od pod-ova je dodeljena *master* uloga:

```
$ kubectl get pods --selector replicationRole=primary

NAME             READY   STATUS    RESTARTS   AGE
mypostgres-1-0   1/1     Running   0          12m51s
```

Kod 22. Master pod

a kojim pod-ovima je dodeljena *slave* uloga:

```
$ kubectl get pods --selector replicationRole=replica

NAME             READY   STATUS    RESTARTS   AGE
mypostgres-2-0   1/1     Running   0          13m14s
mypostgres-3-0   1/1     Running   0          13m14s
```

Kod 23. Slave pod-ovi

Ako obrišemo *master* pod, jedan od *slave* pod-ova će automatski biti promovisan u *master* pod:

```
kubectl delete pod mypostgres-1-0
```

Kod 24. Brisanje master pod-a

Proveru novog *master* pod-a vršimo na već poznati način, s tim što je sada *master* pod, promovisani *slave* pod sa rednim brojem 2:

```
$ kubectl get pods --selector replicationRole=primary

NAME             READY   STATUS    RESTARTS   AGE
mypostgres-2-0   1/1     Running   0          24s
```

Kod 25. Promovisani master pod

Ono što je veoma interesantno kod ove implementacije, jeste da *Kubernetes* klaster automatski kreira novi *slave* pod nakon što smo obrisali *master* pod:

```
$ kubectl get pods --selector replicationRole=replica

NAME             READY   STATUS    RESTARTS   AGE
mypostgres-3-0   1/1     Running   0          15m42s
mypostgres-4-0   1/1     Running   0          44s
```

Kod 26. Kubernetes self-healing

6 Zaključak

Glavni cilj ovog rada, bio je da se dizajnira i implementira *PostgreSQL* klaster koji obezbeđuje visoku dostupnost, uz minimalan napor koji je potreban od administratora. Mnoge postojeće alternative za *PostgreSQL* replikaciju su upoređene i analizirane. Bilo je neophodno identifikovati alate koji ispunjavaju zahteve okruženja u oblaku kao što su jednostavno podešavanje, skaliranje i mogućnost preživljavanja restartovanja. Rešenje se sastoji od *Kubernetes* klastera i lokalnih kontejnerizovanih *PostgreSQL* čvorova sa ekstenzijom *Repmgr* podržanih upornim skladištem i *Pgpool-II* kontejnera koji služi kao *proxy*. Visoka dostupnost se postiže *striming* replikacijom i automatskim prelaskom *failover*, koji se pokreće ako se glavni čvor sruši. Pokretanje različitih vrsta baza podataka, u potpuno automatizovanim okruženjima u oblaku zajedno sa aplikacijama, sigurno će postati standardno rešenje budućnosti.

Reference

1. https://www.postgresql.org/docs/current/index.html
2. https://www.postgresql.org/docs/12/high-availability.html
3. https://www.redhat.com/en/topics/devops/what-is-blue-green-deployment
4. https://kubernetes.io/docs/home/
5. https://www.kubegres.io/doc/getting-started.html
6. https://github.com/CrunchyData/postgres-operator
7. https://github.com/zalando/postgres-operator
8. https://www.kubegres.io/doc/getting-started.html
9. https://www.slony.info/
10. https://bucardo.org/Bucardo/
11. https://repmgr.org/docs/4.0/using-repmgrd.html
12. https://www.pgpool.net/docs/latest/en/html/index.html
13. Shaun Thomas - PostgreSQL 12 High Availability Cookbook Third Edition
14. Hans-Jurgen Schonig - PostgreSQL Replication - Second Edition 2nd Edition
15. Simon Riggs, Gianni Ciolli - PostgreSQL 14 Administration Cookbook