Izrada aplikacije primjenom sustava NuoDB

Danko, Bukovac

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Danko Bukovac

Izrada aplikacije primjenom sustava NuoDB

ZAVRŠNI RAD

Varaždin, 2018.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Danko Bukovac

Matični broj: 44021/15-R

Studij: Informacijski sustavi

Izrada aplikacije primjenom sustava NuoDB

ZAVRŠNI RAD

Mentor/Mentorica:

Prof. dr. sc. Mirko Maleković

Varaždin, rujan 2018.

Danko Bukovac

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

NewSQL sustavi za upravljanje bazama podataka (u daljnjem tekstu SUBP) glavna su tema ovog rada u kojem su opisane karakteristike tih sustava. Objašnjene su sličnosti i razlike NewSQL SUBP-a u usporedbi s relacijskim i NoSQL SUBP-ovima. Odabrani predstavnik NewSQL SUBP-a je NuoDB, koji je opisan i ukratko uspoređen s ostalim NewSQL SUBP-ovima, te je time opravdan izbor istog. Rad također obuhvaća i dizajniranje baze podataka korištenjem alata MySQL Workbench, koja je potom implementirana u NuoDB SUBP-u. Bazu podataka koristit će jednostavna aplikacija koja je izrađena u programskom jeziku C# korištenjem razvojnog okruženja Visual Studio 2017 s fokusom na interakciju baze podataka i aplikacije, a kao domena aplikacije odabrana je veleprodajna trgovina.

Ključne riječi: implementacija, aplikacija, dizajn, modeliranje, sustavi NewSQL, sustav NuoDB, aplikacijska domena

Sadržaj

1. Uvod

Temeljni razlog zbog kojeg su ljudi počeli koristiti računala je obrada podataka koje je računalo moglo obavljati puno brže od čovjeka. Podaci su središnji element koji predstavlja i ulaz i izlaz računalne obrade (Fortune, 2014). Razvojem obradnog kapaciteta računala nastala je i potreba spremanja podataka te popratni razvoj medija za spremanje podataka . Od velikog značaja je i definiranje relacijskih baza podataka, što je 1970. godine učinio Edgar F. Codd (Codd, 1970). Relacijske baze podataka predstavljaju tip baza podataka u kojem se podaci spremaju u tablice u kojima se jedan objekt upisuje u jedan red tablice, a njegovi atributi određeni su stupcima tablice . Uz relacijske baze podataka danas je usko vezan upitni jezik SQL (engl. *Standard Query Language)* kao standardni jezik koji se koristi u svim relacijskim sustavima (Date, 1990). Relacijske baze podataka s relativno neizmjenjenim načinom rada su i danas dominantna vrsta baza podataka na tržištu .

2000-ih godina javljaju se NoSQL baze podataka koje ne spremaju podatke prema relacijskom modelu podataka (Carvelli, 2015). Podaci se spremaju na više različitih načina, ali su generalno NoSQL baze podataka dizajnirane za horizontalno skaliranje te podržavaju visoku razinu particioniranja i omogućuju korištenje distribuiranog sustava poslužitelja. Nedostatak ovakvih sustava je što se navedene prednosti ostvaruju smanjenjem konzistentnosti podataka, što je naravno jedan od temeljnih zahtjeva koji bi baza podataka uvijek trebala ispuniti .

NewSQL baze podataka nastale su kao pokušaj ispunjavanja svih potreba današnjeg tržišta. NewSQL baze podataka su relacijske baze podataka koje zadržavaju sve dobre strane relacijskih baza podataka, kao što je njihov model spremanja podataka te ACID (engl. *Atomicity, Consistency, Isolation, Durability*) garancije, tj. ispunjavaju zahtjeve atomnosti, konzistencije, izolacije i trajnosti podataka. Dobra strana NoSQL baza podataka koje se zadržavaju je horizontalno skaliranje potrebno kod obrade internetskih transakcija (Pikeos, n.d.). Od više NewSQL SUBP-a za potrebe ovog rada kao predstavnik NewSQL SUBP-a odabran je NuoDB zbog podrške za Windows operativni sustav i razine razvijenosti. Ovaj rad odnosit će se većinom na karakteristike NewSQL baza podataka i, konkretno, NuoDB SUBP-a te usporedbe s relacijskim i NoSQL bazama podataka, ali i s drugim NewSQL SUBP-ovima. Za demonstraciju njegovih karakteristika bit će napravljena aplikacija u jeziku C# koja će komunicirati s NuoDB bazom podataka.

2. Metode i tehnike rada

Veći dio ovog rada posvećena je bazama podataka i sustavu u kojem će se ona realizirati. Odabrani sustav za izradu baze podataka je NuoDB SUBP, međutim, taj sustav podržava samo izradu baze podataka preko komandnog sučelja. Izradi baze podataka uvijek prethodi njeno modeliranje i dizajniranje koje ću u ovom slučaju raditi u MySQL Workbench alatu izabranom zbog mogućnosti brze i kvalitetne izrade ERA (engl. *Entity-Relationship-Attributes*) modela i prijašnjeg iskustva u korištenju tog alata. NuoDB će također biti sustav preko kojeg će se kontrolirati pristup podacima.

Drugi dio rada uključuje izradu aplikacije koja će komunicirati s bazom podataka i time pokazati način njihove interakcije. Aplikacija za veleprodajnu trgovinu bit će izrađena u jeziku C# korištenjem razvojnog okruženja Visual Studio 2017. Interakcija s bazom podataka ostvarit će se ADO.NET upravljačkim programom i korištenjem Entity Framework izvora podataka.

3. Nerelacijske tehnologije za upravljanje podacima (NoSQL)

Popularno zvane NoSQL, nerelacijske tehnologije za upravljanje podacima temelje se, kao što im ime govori, na drugim načinima upravljanja koji nisu relacijski. Ova tehnologija postoji od 60-ih godina, ali je relacijska tehnologija prevladavala do 2000-ih. NoSQL tehnologija postala je popularna zbog svoje koristi u Web aplikacijama, s obzirom da nudi brzu obradu transakcija i podržava distribuirane baze podataka. Potrebno je napomenuti da nerelacijski SUBP-ovi ne koriste SQL kao jezik za upite jer je SQL prilagođen za podatke spremane u relacijama (tablicama). U ovom poglavlju bit će napravljen opći pregled nerelacijskih tehnologija kao svojevrsnih prethodnika NewSQL tehnologiji uz relacijsku tehnologiju.

3.1. Podjela NoSQL SUBP-ova

NoSQL SUBP-ovi dijele se u kategorije prema formatu u koji spremaju podatke. Tako ih možemo podijeliti na one u kojima se podaci spremaju u stupce tablice, u dokumente, parove ključ-vrijednost i u grafove.

3.1.1. Stupčasto-orijentirani SUBP-ovi

Spremanje podataka u stupce slično je spremanju u redove, ali je u principu obrnuto (Gobla, n.d.). Prilikom spremanja u redove navode se svi atributi za jedan objekt i ti se atributi grupiraju u red prema stupcima. Važno je napomenuti da se svaki red onda sprema kao jedan objekt u memoriju i prepoznaje se preko svog identifikatora. Baze podataka koje spremaju podatke u stupce grupiraju, recimo, sva imena korisnika u jedan stupac, koji se kao objekt sprema u memoriju i prepoznaje se preko svog sadržaja. Spremanje podataka u stupce zapravo poprilično nalikuje spremanju u redove jer se zadržava oblik tablice (Database Guide, n.d.). Shemi relacijskog oblika spremanja podataka odgovara prostor ključeva stupčasto-orijentiranih SUBP-ova. Prostor ključeva sadrži stupčane obitelji koje pak sadrže redove koji se sastoje od stupaca, tj. to su tablice u relacijskim sustavima. Redovi su građeni od svog ključa i određenog broja stupaca koji sadrže podatke. Stupci se sastoje od svog imena, vrijednosti i vremenske oznake, te su vezani samo uz svoj red. Vremenska oznaka koristi se za određivanje zadnje verzije podatka .

Prednosti ovakvog sustava su: brza obrada upita koji traže samo određene podatke iz „klasičnog“ reda, efikasna kompresija podataka i pogodnost za implementaciju distribuirane baze podataka. Nedostatak je vrijeme potrebno za obavljanje transkacija pisanja.

3.1.2. Dokumentno-orijentirani SUBP-ovi

SUBP-ovi ove vrste spremaju svoje podatke u jednu od više vrsta dokumenata, obično u XML ili JSON obliku (Database Guide, n.d.). Svima je zajednička činjenica da postoji standardan format zapisa ili kodiranja podataka kako bi se moglo isprogramirati čitanje i pisanje. Podaci se u dokumente zapisuju na način da svaki dokument predstavlja jedan objekt, kao što bi to bio red tablice u relacijskom sustavu. Jedino što je propisano u dokumentu je oblik zapisa, a sve drugo je proizvoljno. Tako može postojati više dokumenata za korisnike, gdje jedan dokument sadrži broj telefona, a drugi ne ili se može pojaviti dokument s jedinstvenim poljem. Baza podataka evidentira dokumente pod jedistvenim identifikatorom, a unutar dokumenta podaci se dohvaćaju prema oznakama koje su zapravo ono što bi u stupčasto-orijentiranom SUBP-u bili nazivi stupaca. Ovaj tip SUBP-a nudi svoje API-je (engl. *Application Programming Interface*) i upitne jezike za dohvaćanje podataka, a to rade preko metapodataka koji su zapisani u dokumentu.

Prednosti ovog SUBP-a su pogodnost za distribuirane baze podataka zbog spremanja u dokumente koje je lako držati na više računala i brzo pretraživanje podataka. Nedostaci su, kao i kod većina NoSQL baza podataka, spora obrada akcija pisanja u bazu podataka.

3.1.3. Ključ-vrijednost SUBP-ovi

Ključ-vrijednost način spremanja temelji se na sustavu gdje jedan ključ pokazuje na jednu vrijednost (Mendis, n.d.). Ovaj način jedan je od najjednostavnijih načina spremanja podataka jer ne uključuje nikakva ograničenja nad podacima. Podaci se pronalaze po ključevima koji su građeni prema izgledu pristupanja podacima u tablici (npr. ključ najprije sadrži ime tablice, zatim šifru reda te ime atributa, a vrijednost je vrijednost atributa). Zbog ovakve organizacije podataka SUBP nema svoj upitni jezik, nego se upiti moraju kreirati u jednom od programskih jezika koje SUBP podržava (npr. LevelDB koristi Javu).

Jedna od prednosti ovog načina spremanja podataka je jako malo dodatnih podataka potrebnih da se podaci zapišu u memoriju. Kod relacijskih baza potrebne su informacije o nazivima stupaca i određeni prostor koristi se čak i kad je ćelija tablice prazna, a prilikom spremanja podataka na ključ-vrijednost način toga nema. To, naravno, rezultira boljim preformansama prilikom izvršavanja određenih upita, uglavnom jednostavnih. Nedostatak je što nema povezanosti podataka kao kod relacijskih baza, iako se na tom problemu posljednjih nekoliko godina radi i neke baze podataka ovog tipa nude mogućnost vanjskih ključeva i jače

povezanosti. Također, kao nedostatak se ističe što se upiti moraju pisati kao funkcije u nekom programskom jeziku, s obzirom na to da se podaci u ključ-vrijednost bazu podataka mogu spremati na različite načine te se upitni jezik tu ne može koristiti.

3.1.4. Grafovski SUBP-ovi

Grafovi su organizirani u čvorove i veze između tih čvorova. Prilikom spremanja podataka u obliku grafova čvorovi sadrže podatke o entitetima (npr. korisnici, računi, tvrtke itd) (Neo4j, n.d.-b)nn. Veze povezuju čvorove koji su u nekakvom odnosu i sadrže dodatne informacije o tom odnosu. Tablica korisnika u relacijskom sustavu se u kontekstu grafovskog sustava raščlanjuje na svakog pojedinog korisnika koji je predstavljen svojim zasebnim čvorom. Čvor sadrži informacije o korisniku i informacije o vezama s ostalim čvorovima. Veze mogu, primjerice, povezivati korisnika s odjelom u kojem on radi i sadrže informaciju da on pripada tom odjelu ili više njih (u tom slučaju postoji više izlaznih veza iz čvora tog korisnika). Veze su u grafovskom sustavu usmjerene, a to je bitno zbog navigacije prilikom izvršavanja upita. Grafovske baze podataka nisu toliko različite od relacijskih prema načinu spremanja podataka, već je razlika u načinu obrade upita (Neo4j, n.d.-a).

Prednost ovog sustava je što se osnovni podaci o entitetu spremaju u čvorove, a veze eliminiraju potrebu za slabim entitetima, tj. spojnim tablicama ili vanjskim ključevima. U relacijskoj bazi podataka obrada upita koji moraju spajati tablice korištenjem JOIN SQL naredbe može potrajati jako dugo prilikom izvršavanja. Grafovske baze podataka ostvaruju svoju brzinu obrade zbog činjenice da zbog semantičkih veza između čvorova koje su ostvarene pomoću fizičkih pokazivača u čvorovima nije potrebno izvršavati JOIN naredbe i time se složenost upita višestruko smanjuje. Nedostatak je što ovakvi SUBP-ovi koriste druge upitne jezike, koji se međusobno razlikuju, iako se najčešće koriste Gremlin, SPARQL i Cypher upitni jezici.

4. NewSQL sustavi za upravljanje bazama podataka

U ovom poglavlju definirat će se NewSQL SUBP-ovi i prikazat će se njihove opće karakteristike. Zatim će se te karakteristike usporediti s karakteristikama relacijskih i NoSQL SUBP-ova. Neki od najčešće korištenih NewSQL SUBP-ova su: Google Spanner, CockroachDB, Altibase, Apache Ignite, GridGain, TiDB, Clustrix, VoltDB, MemSQL, NuoDB, Trafodion, ScaleArc, MariaDB MaxScale, Aurora, ClearDB ("13 NewSQL Databases - Compare Reviews, Features, Pricing in 2018 -PAT RESEARCH: B2B Reviews, Buying Guides & Best Practices," n.d.).

4.1. Kategorizacija NewSQL sustava za upravljanje bazama podataka

NewSQL SUBP-ovi temelje se na zadržavanju ACID garancija i istovremenom održavanju brzine transakcija uz horizontalno skaliranje. Ispunjavanjem tih zahtjeva SUBP se može svrstati u NewSQL. S obzirom na implementaciju, postoje tri kategorije NewSQL SUBP-ova: sustavi za particioniranje baza podataka, NewSQL baze podataka u oblaku i NewSQL SUBP-ovi temeljeni na novim arhitekturama (Amitai, n.d.).

4.1.1. Sustavi za particioniranje baze podataka

Jedna od vrsta SUBP-ova koji spadaju u NewSQL jesu sustavi za particioniranje baza podataka (Pavlo & Aslett, 2016). Ova vrsta SUBP-ova veže se uz NewSQL zato jer se koristi za kreiranje distribuirane baze podataka iz baze podataka pohranjene na jednom čvoru, a da nova baza podataka koristi isti SUBP. Za komunikaciju između novostvorenih čvorova stvara se posrednički program koji usklađuje rad distribuirane baze podataka. Prednosti ovoga sustava jesu da on omogućuje stvaranje distribuirane baze podataka, a da ona i dalje koristi istu tehnologiju i pritom nije potrebno mijenjati aplikaciju jer posrednički program aplikaciji i dalje prikazuje jednu logičku bazu podataka. Nedostatak ovoga sustava je to što baza koja se pretvara i dalje radi preko SUBP-a koji je dizajniran za rad na jednom čvoru, te nema prednosti optimizacije podataka i upita na svakom čvoru koje nude potpuno novi SUBP-ovi. Ovaj SUBP nema mogućnosti kreiranja baze podataka, ali se svrstava u NewSQL jer omogućuje da se relacijska baza podataka pretvori u onu koja svojim svojstvima spada pod NewSQL. Primjeri takvih sustava su ScaleArc i MariaDB MaxScale .

4.1.2. NewSQL baze podataka u oblaku

Ova kategorija NewSQL SUBP-ova razlikuje se od novih arhitektura po svojoj implementaciji koja je ostvarena po principu „baza podataka kao usluga“ (engl. *Database as a Service, DBaaS)* (Pavlo & Aslett, 2016). Korisnici ovakvih usluga uz plaćanje dobivaju poveznicu za spajanje na bazu podataka i način kontrole preko API-ja ili kontrolne ploče te ne moraju održavati SUBP na privatnom poslužitelju ili iznajmljenom poslužiteljskom prostoru. Korisnici plaćaju usluge baze podataka prema očekivanom korištenju resursa baze podataka na način da poslužitelj definira maksimalne razine korištenja resursa koje može garantirati. Poslužitelj ove usluge ima obavezu održavanja fizičke konfiguracije baze podataka, uključujući replikaciju i sigurnosnu kopiju baze podataka. Primjer NewSQL baze podataka u oblaku je Aurora tvrtke Amazon. Aurora je temeljena na NewSQL SUBP-u i funkcionira na prethodno opisani način baze podataka kao usluge. Drugi primjer NewSQL baze podataka u oblaku je ClearDB koji funkcionira na način da se ClearDB prodaje kao programski proizvod (softver) koji korisnik onda postavlja na javni oblak. Prednost ovog načina je da korisnik može koristiti različite pružatelje usluga oblaka kako bi izbjegao prestanak rada sustava zbog poslužiteljeve nemogućnosti pružanja usluge.

4.1.3. NewSQL SUBP nove arhitekture

Ova kategorija NewSQL SUBP-ova podrazumijeva SUBP-ove koji su od temelja izgrađeni kao NewSQL SUBP (Kumar, Gupta, Maharwal, Charu, & Yadav, 2014). Oni se razlikuju od sustava za particioniranje po tome što omogućavaju kreiranje potpuno nove NewSQL baze podataka. Ovakvi SUBP-ovi u koje spada NuoDB su na razini Microsoft SQL Servera iz klase relacijskih SUBP-ova i MongoDB-a iz klase NoSQL SUBP-ova s obzirom na mogućnosti kreiranja baze podataka i upravljanja njome. Temeljne odrednice ove vrste SUBP-a su distribuirane arhitekture poslužitelja, podrška za istodobnu kontrolu više čvorova, replikacija baze podataka, kontrola toka i distribuirano izvršavanje upita. Prednost koju imaju SUBP-ovi koji su temeljeni na distribuiranom načinu rada jesu ugrađeni mehanizmi za optimiziranje rada sustava s više čvorova. Prilikom obrade velikog broja operacija čitanja i pisanja veliku prednost nad SUBP-ovima druge vrste daju im ugrađeni sustavi optimiziranja upita i komunikacije između čvorova. Primjer toga je da SUBP-ovi nove arhitekture omogućavaju čvorovima međusobnu razmjenu podataka vezanu uz upit, dok sustavi za particioniranje koriste centralizirani sustav u kojem svi čvorovi šalju podatke na središnju lokaciju. SUBP-ovi ove kategorije sami upravljaju primarnim spremnikom i koriste namjenski kreirane sustave za upravljanje podacima, čime se ostvaruje bolji raspored resursa i omogućuje smanjeno korištenje mreže. Mreža se manje koristi jer opisani sustav šalje upite

manjeg formata čvorovima, dok drugi sustavi za upravljanje podacima, koji nisu izgrađeni za distribuirane sustave, moraju sakupljati podatke iz čvorova. Time ova kategorija SUBP-ova postiže bolje performanse od sustava koji se integriraju s postojećim sustavima, primjerice sustav za particioniranje.

4.2. Karakteristike NewSQL SUBP-a

Karakteristike preko kojih ću predstaviti NewSQL SUBP-ove vezane su uz NewSQL i pojavljuju se kao nove tehnologije i koncepti, dok su neke već otprije prisutne u relacijskim i NoSQL SUBP-ovima.

4.2.1. Pohrana podataka

Svi SUBP-ovi dosada koristili su sustav spremanja podataka gdje je primarno spremište bio tvrdi disk, tj. arhitektura adresabilnih blokova na SSD-u (engl. *Solid State Disk*) ili tvrdom disku. Neki NewSQL sustavi i dalje koriste takav način spremanja podataka, dok, na primjer, VoltDB i MemSQL koriste spremanje u glavnu memoriju (Pavlo & Aslett, 2016). Taj način razvio se zbog pada cijena i povećanja kapaciteta memorije do razine na kojoj je isplativo cijelu bazu podataka držati u glavnoj (radnoj) memoriji. Prednosti su brži rad radne memorije i izbjegavanje prebacivanja podataka u pričuvnu memoriju te korištenje privremenog spremnika zato što se sad svim podacima koji se koriste pristupa direktno. Ovaj način rada znatno je brži od tradicionalnog, što čini SUBP-ove koji ga koriste puno boljima u obradi online transakcija. Problem kod ovog načina spremanja podataka može nastati zbog manjka prostora u memoriji, za koji neki sustavi imaju riješenje. Svaka baza podataka ima dio koji se koristi redovito (često) i dio koji se nalazi u trajnom spremištu kojem se pristupa rjeđe. Kako bi se smanjila veličina baze podataka koja se nalazi u memoriji, stvoreni su mehanizmi koji prepoznaju podatke koji se ne koriste tako često kao neki drugi i izbacuju ih iz memorije. Ti podaci su i dalje vezani za podatke u memoriji, pa se na njihovo mjesto stavlja oznaka gdje su izbačeni podaci spremljeni. H-Store SUBP, na primjer, prilikom pokušaja dohvata izbačenih podataka prekida dohvaćanje i asinkronom dretvom pokreće vraćanje tih podataka u memoriju. MemSQL smanjuje veličinu baze u memoriji spremanjem tablica u stupčastom obliku, pri čemu se u bloku memorije spremaju podaci jednog stupca iz više redova.

4.2.2. Particioniranje

Distribuirane baze podataka jedan su od temeljnih odrednica NewSQL SUBP-a. Takve baze podataka podijeljene su u više disjunktnih skupova koji se nazivaju particije (Kerstiens, n.d.). Podaci se u NewSQL bazama podataka spremaju u tablice koje se horizontalno dijele u

fragmente, a oni su određeni prema vrijednostima jednog ili više stupca. N-torke podataka raspoređuju se u fragmente koristeći rasponsko ili *hash* particioniranje. Povezani fragmenti više tablica spajaju se i formiraju particiju kojom upravlja jedan čvor zadužen za izvršavanje upita nad podacima u toj particiji. SUBP proces izvršavanje upita prepušta svakom čvoru koji mu vraća podatke na sintezu za formiranje odgovora (Pavlo & Aslett, 2016). Baze podataka koje su orijentirane obradi online transakcija organiziraju podatkovnu shemu u obliku stabla gdje su listovi vezani s korijenom preko vanjskog ključa. Tablice se prema tom načinu particioniraju prema tim vezama tako da se svi podaci jednog entiteta nalaze u jednoj particiji. Na primjeru izmjene podataka vezanih uz jedan korisnički račun, transakcija koja se mora provesti bit će odrađena samo na jednoj particiji i time se smanjuje mrežni promet i potreba za kontrolom ispravnosti transakcije. Takvim načinom particioniranja podataka stvara se homogena arhitektura čvorova. Ovdje treba napomenuti da se govori o homogenosti ili heterogenosti čvorova baze podataka, a ne generalno o bazi podataka.

Za razliku od ostalih NewSQL SUBP-ova, NuoDB i MemSQL koriste heterogenu arhitekturu. Temelj heterogene arhitekture je podjela rada među čvorovima na način da postoje čvorovi specijalizirani za izvršavanje i za pohranu. Kod ove arhitekture cilj je smanjiti količinu podataka koju je potrebno dohvatiti iz čvorova s podacima. MemSQL SUBP to radi na način da čvorovi za pohranu izvršavaju dio upita u svrhu smanjenja količine podataka poslanih u čvorove za izvršavanje. NuoDB također ima heterogenu arhitekturu čvorova, ali o tome će biti detaljnije napisano u poglavlju od NuoDB. Heterogena arhitektura ima mogućnost dodavanja resursa za obradu, tj. čvorove za izvršavanje, bez potrebe ponovnog particioniranja baze podataka, što homogena arhitektura nema.

Arhitekture i sustavi koji ih koriste još nisu dovoljno dugo u upotrebi da se može prosuditi koja je arhitektura bolja. Particioniranje baze podataka omogućuje migraciju te baze podataka bez prekida usluge. Migracija tj. prijenos podataka kroz čvorove, radi se zbog smanjenja opterećenja nekih čvorova ili promijene kapaciteta SUBP-a. SUBP-ovi koriste jedan od dva načina izvedbe migracija: premještanjem virtualnih particija ili premještanjem n-torki podataka. Podjelom baze podataka u virtualne particije one se sele po čvorovima, a precizniji način je premještanjem n-torki gdje se mogu premještati manje količine podataka i time se preciznije raspodjeljuje teret po čvorovima.

4.2.3. Replikacija

Replikacija baze podataka je proces kopiranja baze podataka s jednog računala ili poslužitelja na drugi (Rouse, n.d.). Na taj način stvaraju se kopije baze podataka za trajnu pohranu ili dodatne baze podataka koje omogućuju bolje posluživanje korisnika. NewSQL, kao i svi moderni SUBP-ovi, podržava replikaciju baze podataka. Replikacijom baze podataka stvara se distribuirana mreža replika koje treba održavati konzistentnima. Postoje strogo

konzistentni SUBP-ovi u kojima transakcija pisanja mora biti potvrđena od strane svih replika prije negoli se smatra izvršenom. Prednosti ovog načina replikacije je što replike odražavaju istu sliku (*screenshot*) stanja sustava kao i glavna baza podataka, tako da se mogu koristiti za izvršavanje upita. U slučaju da jedna od replika postane neiskoristiva, zbog konzistentnosti replika nisu izgubljeni podaci. Nedostatak je stroga potreba potvrđivanja transakcije od strana svih replika, što zahtijeva razmjenu poruka između replika koja može dovesti do zastoja ukoliko je mreža spora ili replika ne radi (Pavlo & Aslett, 2016). Alternativa strogoj konzistenciji je eventualna konzistencija gdje nije potrebna potvrda svih replika za izvršavanje transakcije i usklađivanje se odrađuje nakon određenog perioda. Svi NewSQL SUBP-ovi koriste strogu konzistenciju jer je to jedna od njihovih temeljnih karakteristika.

Drugi aspekt replikacije je način obnavljanja replika. Aktivno-aktivna replikacija podrazumijeva istovremeno izvršavanje transakcije na svim replikama. Aktivno-pasivna replikacija izvršava transakciju na jednom čvoru i stanja, nakon izvršavanja, šalje u sve ostale replike. Većina NewSQL SUBP-ova koristi aktivno-pasivni pristup jer se transakcije ne označavaju vremenski, što znači da bi se na čvorovima izvele u različitim redoslijedima. Deterministički SUBP-ovi označavaju transkacije vremenskim oznakama i na taj način garantiraju da se one izvrše u istom redoslijedu na svih čvorovima.

4.2.4. Mehanizmi osiguravanja konzistentnosti baze podataka

Jedan od mehanizama osiguravanja konzistentnosti baze podataka je kontrola transakcija koje se trebaju izvršiti kako bi se osigurala atomnost i izolacija podataka (Pavlo & Aslett, 2016). Ova kontrola omogućava pristupanje bazi podataka iz više aplikacija, a da svaka aplikacija ima dojam da jedino ona komunicira s bazom podataka u tom trenutku. Ta činjenica čini kontrolu transakcija jednom od važnijih karakteristika SUBP-a. Koordinacija transakcija spada pod mehanizme osiguravanja konzistentnosti jer je transakcija osnovni način izmjene podataka tijekom korištenja baze podataka, a podrazumijeva propuštanje (uspješno izvršavanje) ili odbijanje (neuspješno izvršavanje) transakcije. Jedan od dva načina koordinacije transakcija je centralizirani način, gdje postoji centralni kooordinator koji provjerava sve transakcije. Drugi je decentralizirani način gdje svaki čvor provjerava transakcije koje mijenjaju podatke na tom čvoru. Čvorovi međusobno provjeravaju eventualne konflikte istodobnih transakcija. Takav se način bolje skalira zbog velikog broja transakcija koje će čvorovi sami provjeriti, dok za onaj manji dio zajedničkih transakcija satovi čvorova moraju biti jako usklađeni, kako bi se zadržalo ispravno stanje baze podataka.

NewSQL sustavi koriste viševerzijsku kontrolu konkurentnosti (engl. *Multi-version Concurrency Control*, MVCC) koja je trenutno najpogodniji mehanizam za particionirane baze podataka. MVCC kreira novu n-torku podataka koje mijenja transakcija. Više verzija n-torke podataka omogućava nekim transakcijama da se izvrše unatoč tome što je originalna n-torka

već izmijenjena te omogućava da transakcije čitanja, koje dugo traju, ne blokiraju transakcije pisanja. Neki NewSQL SUBP-ovi koriste kombinaciju MVCC-a i dvofaznog zaključavanja (engl. *two-phase locking*, 2PL). Transakcije koje se provode s 2PL zahtijevaju zaključavanje podataka koje mijenjaju kao prvu fazu, dok se druga faza sastoji od otključavanja izmijenjenih podataka. Kombinacija 2PL-a i MVCC-a traži od transakcija zaključavanje podataka. Nakon izmjene podataka stvara se nova verzija, slično kao kod MVCC-a. Transakcije koje čitaju u ovoj kombinaciji ne moraju zatražiti zaključavanje i tako čitanjem ne blokiraju transakcije pisanja. VoltDB SUBP osigurava konzistentnost rasporedom vremenskih oznaka (engl. *Time Ordering*, TO). Korištenjem TO-a transakcijama se dodjeljuje vremenska oznaka, te se one po rasporedu izvršavaju jedna za drugom na particijama baze podataka (Yao, Zhang, Lin, Ooi, & Xu, 2017). Transakcijama koje se moraju izvršiti na jednoj particiji upravljaju čvorovi zaduženi za te particije, a za transkacije koje uključuju više particija koristi se centralizirani koordinator. Prilikom izvršavanja, transakcija ima sve ovlasti nad podacima pa se time eliminira potreba za mehanizmima zaključavanja i transakcije se na jednoj particiji izvršavaju vrlo efikasno. Nedostatak je sporo izvršavanje transakcija na više particija zbog čekanja poruka koje se moraju prenositi mrežom, a potrebne su za osiguravanje konzistentnosti.

4.2.5. Sekundarni indeksi

Primarni indeksi su kreirani prema primarnim ključevima tj. vraćaju redove tablica prema primarnim ključevima. Sekundarni indeksi kreiraju se prema drugim podacima, primjerice, prema imenu korisnika, tako da se pomoću njih može pristupiti cijelom redu tablice s tim imenom. Svrha sekundarnih indeksa je ubrzavanje izvršenja upita, a to je bitna činjenica kod obrade internetskih transakcija (Cihan, n.d.). Indeksi koji sadrže podatke i mijenjaju se prilikom izvršavanja transakcija imaju, kao i kod kontrole transakcija, dilemu oko centralizacije ili decentralizacije te načina usklađivanja indeksa. Sustavi za particioniranje baza podataka koriste centralizirani sustav u kojem postoji jedan sekundarni indeks koji je lako održavati, jer se nalazi na jednom mjestu. Nove arhitekture NewSQL-a koriste decentralizirane i particionirane sekundarne indekse gdje svaki čvor ima dio indeksa, s obzirom da se na taj način ubrzava izvršavanje upita (Pavlo & Aslett, 2016). Prilikom kontrole ispravnosti indeksa koriste se dvije različite metode:

- replicirani indeksi, i
- particionirani indeksi.

Replicirani indeksi mogu vratiti podatke upitu za čitanje preko samo jednog čvora jer u svim čvorovima postoje indeksi s podacima iz cijele baze podataka. Nedostatak ovog načina je ažuriranje indeksa koje se, u slučaju izmjene podataka vezanih uz indeks, mora obaviti pomoću transakcije koja obnavlja sve kopije indeksa. Particionirani indeksi vezani su samo uz podatke koji se nalaze na tom čvoru. Prilikom izvršavanja upita sustav mora dobiti odgovor iz

više čvorova za izvršavanje upita, što donekle usporava sustav. Ovaj način ima bržu obradu prilikom izmjene podataka vezanih uz indeks, s obzirom na to se mora obnoviti samo jedan indeks, i to onaj na čvoru gdje je došlo do promjene podataka.

4.2.6. Oporavak od pada sustava

NewSQL SUBP-i orijentirani su na web aplikacije, što znači da moraju imati dobar sustav oporavka od pada sustava (engl. *recovery*) iz više razloga (od web aplikacija se očekuje konstantna dostupnost, vrijeme nedostupnosti aplikacije je skupo za vlasnike aplikacije i sl) (Yao et al., 2017). NewSQL SUBP-ovi dizajnirani su s obzirom na što efikasnije rješavanje tog problema. Distibuirani sustav rješava probleme pada sustava jako dobro iz razloga što postoji više čvorova koji preuzimaju opterećenja čvora koji ne radi. Prilikom povratka pokvarenog čvora u rad koristi se sličan način koji koriste i relacijske baze podataka. Čvor ima svoju točku povratka koja se ažurira u redovnim intervalima. Kad se ona učita u cijelosti, potrebno je od čvora koji je radio dohvatiti popis promjena koje se moraju izvršiti da bi se postigla usklađenost s bazom podataka. Ovaj način funkcionira u slučaju da se promjene iz popisa izvršavaju brže nego se nove promjene događaju. Za pravilan rad ovog načina potrebno je da se popis sastoji od promjena u fizičkom obliku, a ne u obliku SQL upita. Drugi način vraćanja stanja obavlja se na način da se točka povratka uzme od drugog čvora, čime se može poprilično smanjiti broj promjena koje se moraju obaviti za vraćanje čvora u konzistentno stanje.

4.3. Usporedba s relacijskim i NoSQL SUBP-ovima

NewSQL SUBP-ovi nastali su zbog potrebe industrije za unificiranjem dobrih strana relacijskih i NoSQL SUBP-ova. Kao i kod svakog pokušaja spajanja dobrih strana dva sustava u jedan, moraju se napraviti neki kompromisi kako bi krajnji sustav bio realan za izradu i korištenje. U nastavku će biti navedene neke od dobrih i loših strana takvog sustava, te kompromise koji su postignuti, posebno u odnosu na relacijske i NoSQL SUBP-ove.

4.3.1. NewSQL i relacijski SUBP-ovi

Kronološki gledano, nakon NoSQL SUBP-ova koji su se odmaknuli od relacijskih SUBP-ova po odbacivanju relacijskog modela podataka, NewSQL SUBP-ovi predstavljaju povratak na relacijski model podataka. Relacijski model podataka koristi SQL upitni jezik kojemu su prilagođene mnoge aplikacije i koji je generalno pogodan za pisanje upita. NewSQL se vratio tom modelu podataka, jer je to model koji se dugo koristi i industrija ima iskustva s njim te ispunjava zahtjeve atomnosti, konzitencije, izoliranosti i trajnosti transakcija.

Pogodnosti korištenja takvog modela podataka za korisnika su očite: SUBP osigurava da su podaci konzistentni i ispravni u svakom trenutku prilkom izvršavanja upita (Pikeos, n.d.). Za osiguravanje tih zahtjeva postoje mehanizmi kontrole izvođenja transakcija. Kod relacijskih SUBP-ova koji se temelje na arhitekturi jednog poslužitelja, mehanizmi kontrole nisu toliko kompliciranji kao kod distribuirane arhitekture NewSQL SUBP-ova. Ti mehanizmi su kod NewSQL-a znatno kompliciraniji i svaki SUBP nove arhitekture koristi neku verziju MVCC-a i 2PL-a, ili kombinaciju oba. To komplicira dizajn SUBP-a proizvođačima, a krajnji korisnik to ne primjećuje, iako mu donosi brži rad sustava i bolju trajnost podataka. Već je spomenuto da su relacijski SUBP-ovi dizajnirani za arhitekturu jednog poslužitelja, što današnjoj industriji stvara probleme jer takva arhitektura nije pogodna za horizontalno skaliranje. Današnje web aplikacije zahtijevaju mogućnost brze obrade velikog broja malih transakcija, što arhitektura jednog poslužitelja ne može podržati. Rješenje je stvaranje distribuirane arhitekture koju podržava NewSQL gdje se baza podataka nalazi na više poslužitelja koji zajedno imaju dovoljan kapacitet za efektivno posluživanje tih transakcija. Relacijski sustavi mogu se pretvoriti u distribuirane sustave korištenjem posredničkih sustava, ali takva kombinacija nikada neće biti toliko optimizirana da prestigne performanse novih arhitektura NewSQL SUBP-ova, dizajniranih točno s tom namjerom.

Danas velika većina korisnika baze podataka koristi relacijske baze podataka zbog već navedenih pogodnosti vezanih uz način spremanja podataka i stabilnost sustava. Ne smijemo zaboraviti ni godine korištenja, koje pružaju dozu iskustva korisnicima. Samim time, korisnici relacijskih arhitektura odlučuju se koristiti posredničke sustave. Jedan od razloga je i problem prebacivanja cijele korisničke baze podataka u novi SUBP i prilagođavanja korisničkog sustava za rad s novim SUBP-om.

4.3.2. NewSQL i NoSQL SUBP-ovi

NoSQL SUBP-ovi su svoj dizajn temeljili na potrebama web aplikacija koje su zahtijevale brzu obradu malih transakcija. Dvije glavne odrednice tih sustava su odbacivanje relacijskog modela podataka i zamjena s jednim od više različitih modela spremanja podataka i horizontalno skaliranje pomoću stvaranja distribuirane baze podataka. Tim dizajnerskim odlukama povećala se brzina obrade transakcija, pa distribuirani sustav nudi bolju otpornost podataka. Podaci su raspršeni na više čvorova, ali se, s druge strane, najčešće nauštrb konzistencije podataka (Zicari, n.d.). NoSQL SUBP-ovi nude eventualnu konzistenciju, što znači da može doći do krivih čitanja podataka ili čak gubitka podataka.

NewSQL koristi napretke NoSQL sustava u pogledu ubrzavanja rada sustava, a kao novitet dodaje održavanje konzistentnosti podataka. Do tog je razvoja došlo jer se NoSQL-u najviše zamjeralo na eventualnoj konzistentnosti podataka. To je neko vrijeme bilo moguće tolerirati jer je NoSQL s druge strane nudio poboljšanje u obliku ubrzavanja rada sustava. Treba

napomenuti da postoje NoSQL SUBP-ovi koji podržavaju ACID i da kod većine NoSQL SUBP-ova postoji način da se preko posredničkog sloja podrže ACID zahtjevi (Bosworth, n.d.). Prilikom uvođenja ACID modela konzistentnosti u NoSQL baze podataka, one gube na svojem smislu, jer se usporava njihov rad i povećava se rad na strani implementacije, a time se i baza podataka pretvara u relacijsku bazu. Na kraju je onda bolje koristiti relacijsku bazu podataka. Cilj NewSQL-a bi trebao biti da upotpuni opisanu prazninu tako što nudi mogućnost stvaranja baze podataka koja radi brzinom sličnoj NoSQL bazi podataka i dolazi s implementiranim mehanizmima za ostvarivanje ACID zahtjeva.

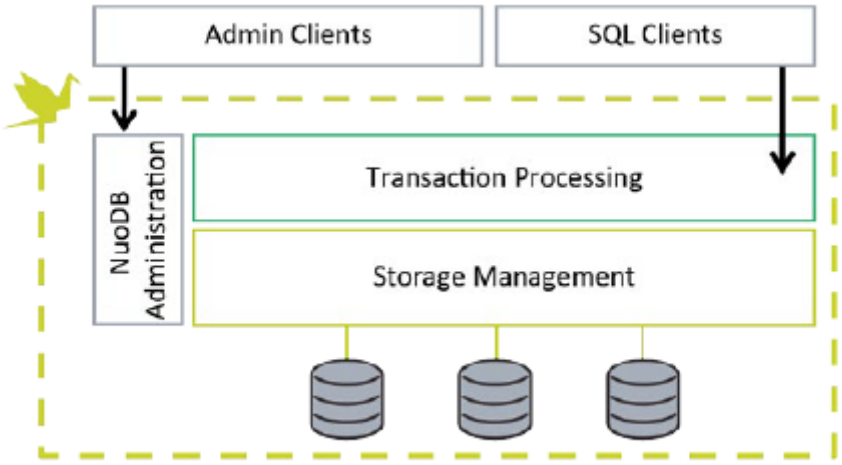
5. NuoDB

U ovom poglavlju bit će objašnjen način funkcioniranja sustava za upravljanje bazama podataka NuoDB prema opisu dizajnera i kreatora NuoDB SUBP-a u dijelu svoje tehničke dokumentacije koja se bavi arhitekturom tog sustava (NuoDB, n.d.-b).

5.1. Arhitektura NuoDB SUBP-a

NuoDB se temelji na dvoslojnoj arhitekturi, gdje jedan sloj obrađuje transakcije i podatke, a drugi sprema podatke. U nastavku će biti objašnjeno kako ti slojevi obavljaju svoje zadaće, kako komuniciraju, te unutarnji ustroj podataka i način kontrole izvođenja transakcija.

5.1.1. Slojevi u arhitekturi NuoDB SUBP-a

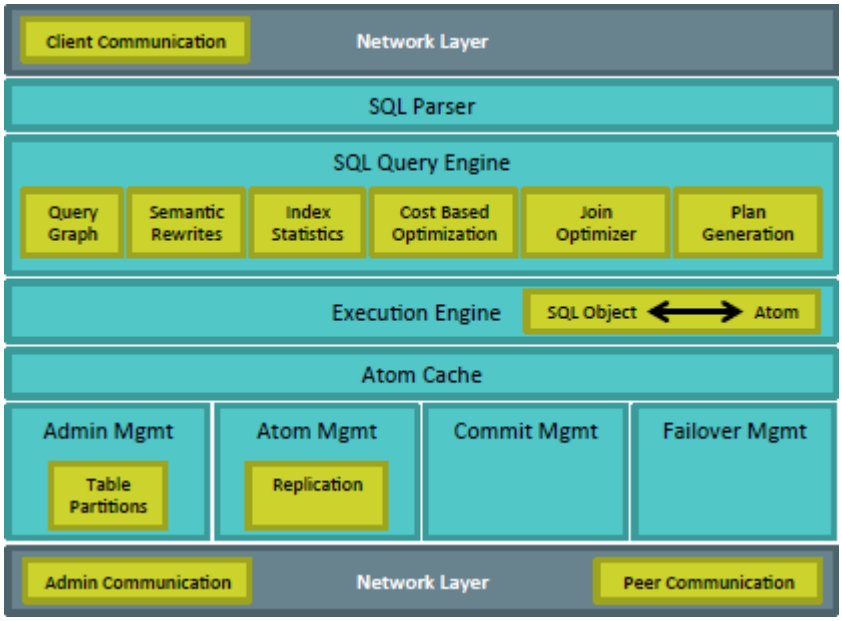


Slika 1: Arhitektura NuoDB SUBP-a (NuoDB, str. 4)

Temelj arhitekture NuoDB SUBP-a čine transakcijski sloj i sloj pohrane (NuoDB, n.d.-b). Transakcijski sloj djeluje kao priručna memorija popunjena podacima za koje se očekuje da će se koristiti. Ona se nalazi u glavnoj memoriji i upravlja transakcijama te osigurava atomnost, izoliranost i konzistenciju transakcija. Sloj pohrane osigurava trajnost podataka i zadužen je za pružanje podataka potrebnih za izvršavanje transakcija u transakcijskom sloju. Ovim razdvajanjem slojeva omogućava se zasebno skaliranje tih slojeva, tj. zasebno se po potrebi može promijeniti moć obrade transakcija ili trajnost podataka.

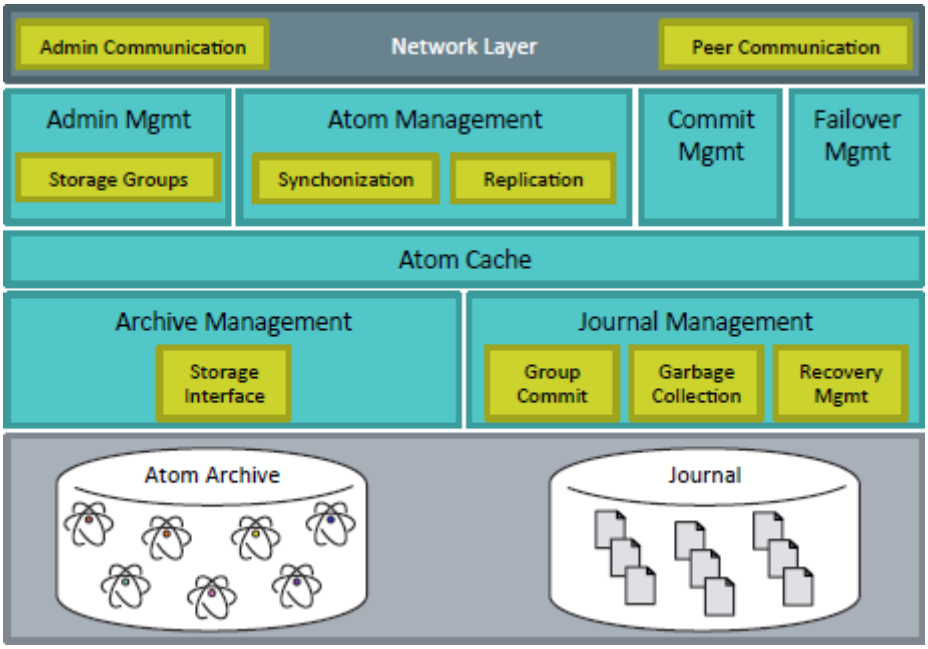
5.1.2. Komunikacija i koordinacija slojeva

Na primjeru jedne baze podataka, transakcijski sloj čine jedan ili više transakcijskih upravitelja (engl. *Transaction Engine*, TE) i jedan ili više upravitelja pohrane (engl. *Storage Manager*, SM) (NuoDB, n.d.-b). Ovi procesi mogu se nalaziti na više poslužitelja i međusobno se tretiraju kao ravnopravni, a komuniciraju kriptiranim kanalima. SQL klijenti spajaju se na TE-ove i oni obrađuju upite nad podacima koje imaju u svojoj pričuvnoj memoriji. Ako potrebni podaci nisu pronađeni u memoriji, procesi komuniciraju preko istorazinskog komunikacijskog protokola i TE pronalazi potrebne podatke kod SM-a ili kod drugog TE-a. Takva komunikacija ne traje dugo jer svaki TE redovito provjerava koji mu je istorazinski proces najdostupniji. Minimalno funkcionalna baza podataka sastoji se od jednog TE-a i od jednog SM-a koji rade na istom poslužitelju. Pokretanjem još jednog TE-a udvostručila bi se moć obrade transakcija i otpornost na prestanak rada sustava u slučaju prestanka rada jednog TE-a. Dodatni SM osigurava otpornost podataka i mogućnost posluživanja TE-a zbog činjenice da se radi potpuna kopija postojećeg SM-a. Potpuno redundantna NuoDB baza podataka postiže se korištenjem dva TE-a i dva SM-a te četiri poslužitelja gdje je na svakog ugrađen samo jedan TE ili SM.



Slika 2: Unutarnja struktura TE-a (NuoDB, str. 6)

5.1.3. Unutarnja struktura upravitelja pohrane



Slika 3: Unutarnja struktura SM-a (NuoDB, str. 9)

NuoDB unutarnja struktura upravitelja podataka se temelji na objektima koji se zovu atomi (NuoDB, n.d.-b). Svaka vrsta podataka u bazi podataka sprema se i manipulira preko korištenja atoma, što atome čini samoodređenim objektima koji mogu predstavljati specifične informacije npr. sheme, podatke ili indekse. Dio transakcijskog sloja koji komunicira s aplikacijom prima SQL zahtjeve, informacije tih zahtjeva pretvara atome i obrađuje ih korištenjem atoma, te prije slanja podataka aplikaciji ponovno prebacuje informacije u SQL oblik. U radu s atomima primjenjuju se principi atomnosti, izolacije i konzistencije bez posebne informacije o sadržaju tih atoma. Atomi su varijabilne veličine, a ta se veličina određuje na način da se poveća učinkovitost komunikacije, smanji broj objekata u priručnoj memoriji i pojednostavni složenost praćenja promjena. Dio unutarnje strukture su kataloški atomi koji služe za razrješavanje drugih atoma i sami čine učinkovitu i uvijek ispravnu tražilicu. Glavni (engl. *Master*) katalog je atom koji za bazu podataka sadrži sve infomacije o lokacijama podataka. TE može brzo nakon pokretanja započeti s radom jer mora učitati samo *master* katalog. Iz njega zna nalazi li se traženi podatak u priručnoj memoriji ili se mora dohvatiti iz drugog TE-a ili SM-a. NuoDB zbog samopodizanja koristi priručnu memoriju na zahtjev. Time se potrebni objekti brzo dohvaćaju u memoriju i brzo se razrješavaju objekti. Kada se objekt više ne koristi, on se izbacuje iz priručne memorije. Sve te promjene prate se u katalogu. Time TE uvijek zna stanje i lokaciju potrebnih podataka. Ovakva manipulacija priručnom memorijom potrebna je zbog ostvarivanja brzog funkcioniranja sustava, ali i zato jer se baza podataka ne nalazi u potpunosti u glavnoj memoriji kao kod nekih drugih NewSQL SUBP-ova. Još jedna

prednost spremanja svih informacija i njihove obrade u obliku atoma je što se prilikom transakcija podaci i kataloške infomacije tretiraju jednako i ne može doći do propusta u kojem se jedan tip informacija ne bi izmijenio.

5.1.4. Viševerzijska kontrola konkurentnosti (MVCC)

Kao SUBP koji nudi konzistentni model podataka i brzinu obrade, NuoDB koristi MVCC način kontrole izvođenja transakcija. Već je spomenuto da se MVCC temelji na stvaranju više verzija podataka prilikom njihove promjene (NuoDB, n.d.-b). TE-ovi mogu u priručnoj memoriji sadržavati više verzija objekta koji mogu biti potvrđeni ili nepotvrđeni zbog transakcije koja se izvršava. MVCC omogućava lako otkazivanje transakcije jer se podaci ne mijenjaju u trenutku izvršavanja svake transakcije, već postoji više verzija koje se samo izbrišu u slučaju otkazivanja. Čvorovi u NuoDB SUBP-u komuniciraju asinkrono, što uz MVCC omogućava da transakcija nastavi s promjenama u slučaju da bi se mogla izvršiti u potpunosti. Ako sustav shvati da to nije moguće, ona se poništava. Naravno da se događa da neke transakcije ne prolaze, ali ovaj pristup generalno znatno ubrzava obradu transakcija. Transakcijama se prilikom njihovog početka izvršavanja daje konzistentna slika podataka, tj. onakva kakva je ona u tom trenutku, i transakcija se izvršava prema tim podacima. Ovisno o tome kako transakcije započinju s izvršavanjem, one imaju i različitu sliku podataka. Ovo je učinkoviti način koordinacije transakcija bez korištenja sata za usklađivanje. Konflikt između transakcija javlja se jedino kod transakcija koje žele izmijeniti isti podatak. Svaki atom ima TE-a koji djeluje kao predsjedatelj (engl. Chairman) za taj atom i u slučaju konflikta određuje koja transakcija dobiva pravo izvršavanja. TE može biti predsjedatelj samo atomima koji su u njegovoj priručnoj memoriji, tako da se razrješavanje konflikta odvija lokalno, a u slučaju gašenja TE-a novi predsjedatelj se određuje brzo, bez puno razmjena poruka.

5.1.5. Trajnost podataka

Spremanje svih podataka u obliku atoma pojednostavljuje održavanje trajnosti (NuoDB, n.d.-b). Arhitektura prati atome prema njihovom jedinstvenom identifikatoru, tj. podaci o atomima se spremaju u obliku ključ-vrijednost. Svaki pristup i premještanje u priručnu memoriju radi se korištenjem cjelovitih atoma. SM-ovi spremaju dnevnik promjena u koji se samo dodaju zapisi o verzijama kako se podaci mijenjaju, što omogućuje brzo pisanje i čitanje dnevnika.

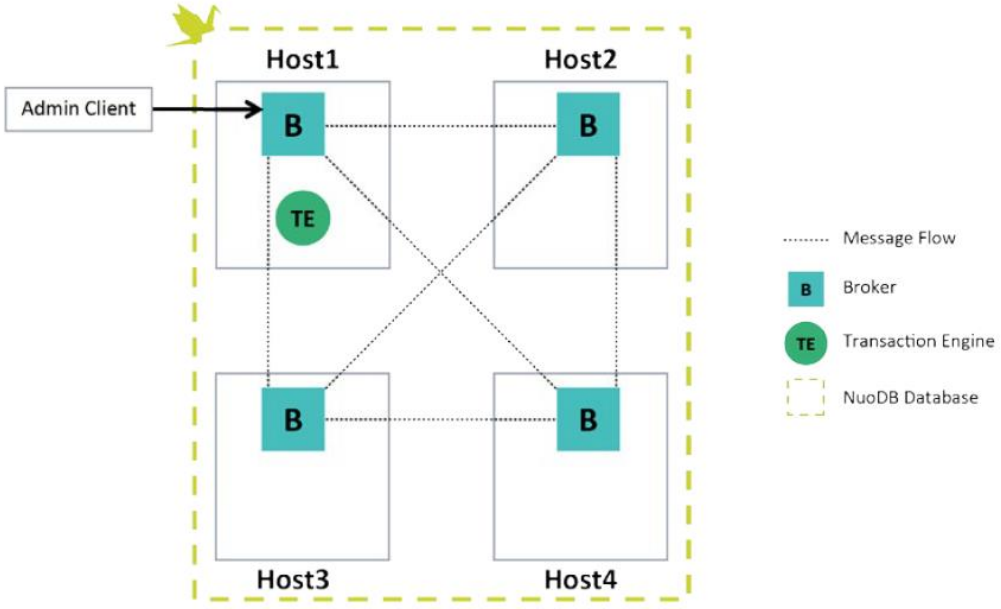
5.1.6. Potvrda transakcija

Za uspješnu potvrdu transakcije SQL klijentu, potrebno je ispuniti sve ACID zahtjeve (NuoDB, n.d.-b). NuoDB šalje sve promjene vezane uz jednu transakciju svim SM-ovima i svim TE-

ovima koji u priručnoj memoriji imaju izmijenjene atome. Najbrži način potvrde transakcije je kad se pouzdane poruke pošalju svim sudionicima i ako kod svih SM-ova istovremeno ne propadne transakcija. Time se osigurava da su promijenjeni podaci ispravni i konzistentni. Sigurnija metoda potvrde je čekanje potvrde od bar jednog ili više SM-a prije nego se SQL klijentu šalje potvrda transakcije. Pouzdanost transakcije povećava se povećavanjem broja SM-ova koji tu transakciju moraju potvrditi. Postoji kontrola kojom se može odrediti koliko SM-ova mora potvrditi transakciju.

5.1.7. Model upravljanja bazom podataka

Upravljački sloj sastoji se od kolekcije ravnopravnih procesa koji se nazivaju brokeri, a nalaze se na svakom poslužitelju gdje je aktivna baza podataka (NuoDB, n.d.-b). Prvo se broker pokreće zato da bi se preko njega pokrenuli drugi procesi baze podataka i da bi poslužitelj postao vidljiv upravljačkoj okolini, tj. bio uključen u upravljačku domenu. Domena postavlja granice i time određuje resurse jedne baze i korisnike kojima su resursi na raspolaganju. Broker je zadužen za upravljanje jednim poslužiteljem, odnosno, on može pokretati i zaustavljati TE i SM procese, pratiti i podešavati te procese zajedno s poslužiteljevim resursima. Svaki broker ima uvid u stanje svih ostalih brokera i njihovih poslužitelja unutar domene. Time se u slučaju prestanka rada jednog brokera omogućava nesmetani nastavak rada sustava. SQL klijent preko brokera zahtijeva komunikaciju, a on ga preusmjerava na određeni TE, čime broker ostvaruje svoj ulogu rasporeda tereta obrade. Dodavanjem brokera povećava se mogućnost boljeg rasporeda tereta obrade. Domena se klijentu predstavlja kao logička baza podataka i tako se klijent prema njoj odnosi, a brokeri služe kao točke na koje se klijent poveže i koje dalje usmjeravaju njegov upit.



Slika 4: Arhitektura NuoDB baze podataka (NuoDB, str. 14)

Model upravljanja uključuje izradu sigurnosne kopije na dva načina. U online načinu upravjanja izdaje se naredba za izradu sigurnosne kopije i time se kopira dnevnik i arhivne datoteke baze podataka. U offline načinu upravljanja može se koristiti jedan SM kao redundantni dio baze ili za izradu sigurnosne kopije. Prvo se SM mora ugasiti za vrijeme kopiranja podataka, a nakon paljenja on se automatski sinkronizira s bazom podataka i prikuplja aktualne podatke. SM se može pokrenuti na temelju bilo koje arhive, čime se baza stavlja u stanje zapisano u toj arhivnoj datoteci. Tako se pojednostavljuje pokretanje novih baza od početka, jer se samo moraju pokrenuti procesi i učitati arhiva.

5.2. Uporedba s drugim NewSQL SUBP-ovima

NewSQL sustavi koriste jedan od dva načina spremanja podataka: klasični, na principu tvrdog diska i noviji, spremanje u glavnu memoriju. NuoDB koristi stariji način pohrane koji, gledajući današnje mogućnosti poslužitelja i cijenu memorije, nije učinkovit u odnosu na novi način. NuoDB ipak koristi taj način zbog podjele na dva sloja, tj. na TE-ove i SM-ove. Budući da SM-ovi samo spremaju podatke, mora se koristiti stari način jer TE-ovi koji poslužuju SQL klijenta moraju imati priručnu memoriju kako bi funkcionirali dovoljno brzo za današnje potrebe obrade internetskih transakcija. NewSQL SUBP-ovi kao što su VoltDB i MemSQL koriste spremanje u glavnu memoriju i time postižu veću brzinu obrade. Prema mojem mišljenju, dizajneri NuoDB-a su koristili ovaku arhitekturu zbog fleksibilnosti koju nudi jer se TE-i i SM-ovi mogu odvojeno dodavati na poslužitelje i time popuniti rupe u nedostatku moći obrade ili sigurnosti podatka.

NuoDB, kao i MemSQL, imaju arhitekturu koja se temelji na heterogenom skupu čvorova, tj. na primjeru NuoDB-a koji se sastoji od TE-a i SM-ova. NuoDB je poseban jer svaki SM sadrži sve podatke koji se nalaze u bazi podataka, za razliku od ostalih SUBP-ova koji te podatke „razbacaju“ po čvorovima. Razlika između MemSQL-a i NuoDB-a je što TE-ovi rade sav posao obrade zahtjeva i dohvaćaju podatke u priručnu memoriju, dok kod MemSQL-a čvorovi sa podacima odrađuju dio upita, a čvorovi za obradu sakupljaju te podatke i šalju odgovor. MemSQL zato sprema podatke u stupčastom obliku ako se moraju raditi složeni upiti ili u obliku redova ako se rade jednostavni i česti upiti nad podacima. NuoDB koji koristi SM-ove samo za pohranu podatka, sprema te podatke u obliku redova. Ostali NewSQL SUBP-ovi koji koriste homogenu strukturu čvorova, organiziraju spremanje podataka u obliku stabla i ta stabla spremaju na jedan čvor. Na taj način, ti SUBP-ovi ubrzavaju rad sustava jer se podaci lako nalaze pa se brže uspiju složiti u odgovor. Brzina odgovora NuoDB-a ovisi ponajviše o tome postoji li traženi podatak u priručnoj memoriji TE-a ili drugih TE-a, a najsporija opcija je da se podatak dohvaća iz SM-a.

Što se tiče kontrole transakcija, NuoDB koristi MVCC kao način koji je zamišljen za distribuirane baze podataka, pa isti mehanizam koristi još i poprilično broj drugih NewSQL SUBP-a, kao što su MemSQL, CockroachDB, HyPer itd. MVCC se koristi u tim sustavima jer klasični relacijski način nije pogodan za usklađivanje transakcija, a alternativa MVCC-u je TO. Navedeni mehanizam koriste H-Store i VoltDB i, kao što je već rečeno, temelji se na vremenskim oznakama kao načinu usklađivanja. MVCC u procesu stvaranja novih verzija omogućava brzo izvođenje transakcija, bez puno potrebe za izmjenom poruka, što još više pogoduje transakcijama koje se izvode na više čvorova. Međutim, nedostatak je što su mogući konflikti te treba razviti način rješavanja konflikta. TO nema problema s konfliktima jer se točno zna koja promjena dolazi po redu, međutim prilikom transkacija na više čvorova mora se razmijeniti određeni broj poruka za usklađivanje, što usporava rad sustava.

SUBP	Spremanje u glavnu memoriju	Arhitektura čvorova	Kontrola transakcija
NuoDB	Ne	Heterogena	MVCC
H-Store	Da	Homogena	TO
MemSQL	Da	Heterogena	MVCC
VoltDB	Da	Homogena	TO
CockroachDB	Ne	Homogena	MVCC
HyPer	Da	Homogena	MVCC

Tablica 1: Karakteristike NewSQL SUBP-a

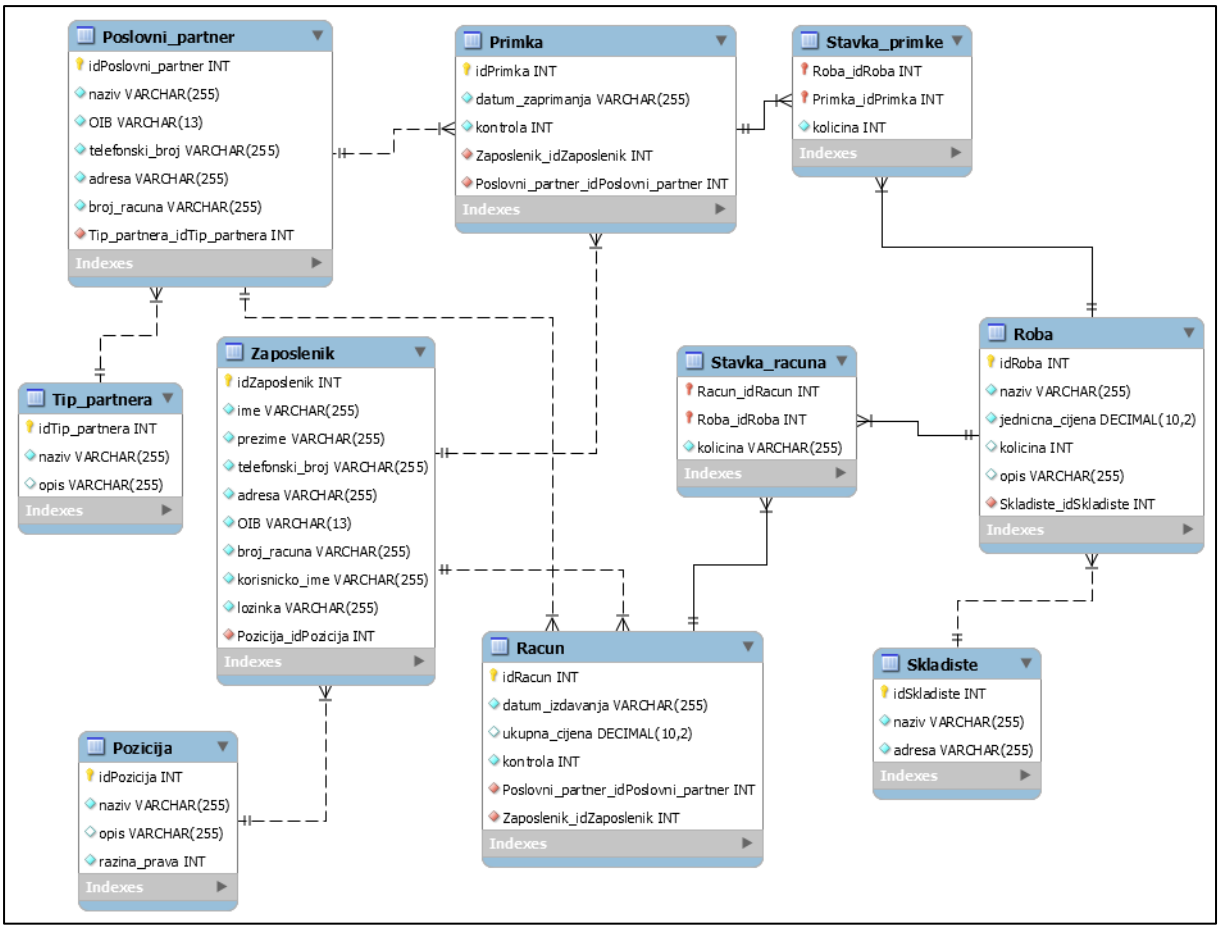
6. Primjena NuoDB SUBP-a za razvoj aplikacije za veleprodajnu trgovinu

U ovom poglavlju prikazat ću proces izrade aplikacije uz primjenu NuoDB SUBP-a. Točnije, to uključuje izradu ERA modela baze podataka u MySQL Workbench alatu i opis izrađenog ERA modela. Baza podataka bit će implementirana u NuoDB SUBP-u, što uključuje izradu tablica i nekoliko okidača, te ću ukratko opisati uspostavljanje i konfiguraciju baze podataka za rad na poslužitelju. Baza podataka će raditi lokalno na mome računalu. Opisat ću izradu sučelja i način realizacije aplikacijske logike u razvojnom okruženju Visual Studio 2017. Na kraju opisa posla napravit ću kratki osvrt na lakoću korištenja alata i složenost posla.

6.1. Aplikacijska domena

Aplikacija je namijenjena za veleprodajnu trgovinu koja određenu robu zaprima i prodaje kupcima. Nova vrsta robe se unosi na skladište koje se prije mora kreirati, a količina robe se mijenja računima i primkama. Primka se kreira prema dobavljaču od kojeg je nabavljena roba i na tu primku dodaju se stavke koje sadrže određenu robu u određenoj količini, te se tim unosima povećava količina robe. Izdavanje računa funkcionira na sličan način, osim što se kod računa zapisuje cijena stavke i ukupna cijena računa, a dodavanjem stavki količina robe se smanjuje. Aplikacija omogućava unos zaposlenika koji se za korištenje prijavljuju u aplikaciju i imaju određena prava, a dok rade unos primki i računa zapisuje se tko je što kreirao. Pozicije prema kojima zaposlenici imaju prava može kreirati i mijenjati administrator. Poslovni partneri unose se prema tipu (dobavljač, kupac) i na račune i primke se zapisuje tko je poslao ili kupio robu. Za račune postoji graf koji pokazuje količinu prodane robe.

6.2. Izrada ERA modela



Slika 5: ERA model baze podataka (vlastita izrada)

Na slici 5 vidi se ERA model baze podataka aplikacije. Ovaj ERA model sadrži 10 tablica koje su napravljene za podršku aplikaciji za veleprodajnu trgovinu. Ova baza podataka omogućuje pohranu podataka o zaposlenicima i poslovnim partnerima te omogućuje vođenje evidencije o zaprimanju robe i izdavanje računa. U nastavku će biti detaljnije pojašnjene tablice u ERA modelu.

Tablica „Pozicija“ sadrži podatke o pozicijama u poduzeću te ima svoj primarni ključ „idPozicija“ tipa podataka *integer* koji se sam uvećava. „Naziv“ označava naziv te pozicije tipa je *varchar* i ne smije biti *null*, „Opis“ je opcionalno polje u koje se mogu upisati dodatne informacije o poziciji tipa *varchar*, a „Razina_prava“ je obavezno polje tipa *integer* koje se koristi prilikom prijave da se utvrdi kojem dijelu aplikacije ima pristup zaposlenik na toj poziciji. Tablica „Pozicija“ povezana je s tablicom „Zaposlenik“ vezom jedna naprema više (1:M), gdje na jednoj poziciji može raditi više zaposlenika, dok svaki zaposlenik radi na jednoj poziciji.

Tablica „Zaposlenik“ sadrži podatke o zaposlenicima kao što su ime, prezime, broj telefona, OIB (*Osobni Identifikacijski Broj*), adresa, broj računa te njihovo korisničko ime i lozinku, što su sve polja tipa *varchar* i moraju biti unesena. Polje „OIB“ ima dodatno ograničenje da mora biti jedinstveno i dužine maksimalno 13 znakova. „idZaposlenik“ je primarni ključ tipa *integer* koji se sam uvećava, a „Pozicija_idPozicija“ je vanjski ključ na tablicu „Pozicija“ tipa *integer*. Tablica „Zaposlenik“ povezana je s tablicama „Racun“ i „Primka“.

Tablica „Primka“ ima svoj primarni ključ „idPrimka“ tipa *integer* koji se samostalno povećava, „Datum_zaprimanja“ sadrži datum, ali je tipa *varchar* jer ću u aplikaciji koristiti isti zapis, radi lakše kasnije implementacije na razini aplikacije. Polje „kontrola“ služi za zaključavanje primke, 0 znači otključano, a 1 znači zaključano, tako da se stavke onda ne mogu dodavati ili brisati. „Primka“ sadrži vanjske ključeve na tablice „Zaposlenik“ i „Poslovni_partner“ tipa *integer* („Zaposleni_idZaposlenik“ i „Poslovni_partner_idPoslovni_partner“). Veze između tablice „Primka“ i tablica „Poslovni_partner“ te „Zaposlenik“ su tipa jedna naprema više (1:M), gdje svaka primka ima jednog partnera za kojeg je kreirana i jednog zaposlenika koji je kreira, dok svaki partner može biti naveden na više primki, a svaki zaposlenik može kreirati više primki.

Tablica „Poslovni_partner“ sadrži podatke o kupcima i dobavljačima, s obzirom na to da su za obje skupine potrebne iste informacije. Primarni ključ je „idPoslovni_partner“ tipa *integer* koji se samostalno povećava, a atributi naziv, OIB, adresa, telefonski broj i broj računa su, kao i kod tablice „Zaposlenik“, tipa *varchar* i OIB ima ograničenje na duljinu i jednistvenost. Tablica „Poslovni_partner“ ima vanjski ključ „Tip_partnera_idTip_partnera“ tipa *integer* i služi za određivanje je li partner dobavljač ili kupac.

Tablica „Tip_partnera“ ima svoj primarni ključ „idTip_partnera“ tipa *integer* koji se samostalno povećava, sadrži obavezno polje „naziv“ tipa *varchar* i neobavezno polje „opis“ tipa *varchar* za dodatne informacije o tipu partnera. Tablica „Tip_partnera“ vezana je sa tablicom „Poslovni_partner“ vezom jedan naprema više (1:M), gdje jednom tipu partnera može pripadati više poslovnih partnera, dok svaki partner pripada jednom tipu partnera (dobavljač ili kupac).

Tablica „Roba“ sadrži podatke o robi koju trgovina prodaje, svaki artikl ima svoj jednistveni identifikator, tj. primarni ključ „idRoba“ tipa *integer* koji se samostalno povećava. Polje „naziv“ je tipa *varchar* i ne smije biti *null*, „jedinicna_cijena“ je isto obavezno polje tipa *decimal*(10,2), što znači da može imati 10 znamenki od kojih su 2 nakon decimalnog zareza. Polje „kolicina“ tipa *integer* može biti nepoznat jer se roba dodaje preko primki, a prilikom stvaranja novog artikla (robe) količina će ostati prazna. Količina određene robe povećavat će se okidačem prilikom dodavanja stavki primke, a smanjivat će se dodavanjem stavki računa. Polje „opis“ tipa *varchar* nije obavezno i može sadržavati dodatne informacije o artiklu. Svaki artikl se nalazi na određenom skladištu, tako da vanjski ključ „Skladiste_idSkladiste“ tipa

integer povezuje tablice „Roba“ i „Skladiste“ vezom jedan naprema više (1:M), gdje jedno skladište može imati više robe, dok se neka roba nalazi na jednom skladištu.

Tablica „Skladiste“ ima primarni ključ „idSkladiste“ tipa *integer* koji se samostalno povećava, obavezno polje „naziv“ tipa *varchar* i obavezno polje „adresa“ tipa *varchar*. Tablica „Skladiste“ povezana je s tablicom „Roba“.

Tablica „Racun“ ima primarni ključ „idRacun“ tipa *integer* koji se samostalno povećava, obavezno polje „datum_izdavanja“ tipa *varchar* za koje vrijedi isto što je napomenuto za datum zaprimanja kod primke. Polje „ukupna_cijena“ tipa je *decimal*(10,2) i može biti *null*,ž jer se prilikom otvaranja računa ne postavlja cijena, već se ona izračunava preko okidača tijekom dodavanja novih stavki računa. Polje „kontrola“ služi za određivanje je li račun izdan, 0 znači da nije izdan, a 1 da je izdan. Tablica ima vanjski ključ prema tablici „Zaposlenik“ „Zaposlenik_idZaposlenik“ koji predstavlja vezu jedan naprama više (1:M), gdje jedan zaposlenik može izdati više računa, a svaki račun izdaje samo jedan zaposlenik. Drugi vanjski ključ „Poslovni_partner_idPoslovni_partner“ povezuje tablicu „Racun“ s tablicom „Poslovni_partner“, veza je jedna naprema više (1:M), kao i veza s zaposlenikom, a označava da poslovni partner može biti naveden na više računa, a svaki račun se izdaje za najviše jednog poslovnog partnera.

Tablica „Stavka_racuna“ slabi je entitet koji povezuje tablice „Racun“ i „Roba“ koji su u vezi više na više (M:N). „Stavka_racuna“ ima kompozitni primarni ključ koji se sastoji od dva vanjska ključa „Racun_idRacun“ i „Roba_idRoba“ koji su tipa *integer*. Polje „kolicina“ je obavezno i tipa *integer* i označava koliko je robe vezano uz tu stavku. Nad ovom tablicom napravljena su dva okidača koji prilikom stvaranja novog zapisa u tablici smanjuju količinu robe i povećavaju ukupnu cijenu na računu. Drugi okidač radi u slučaju brisanja zapisa tako da smanjuje ukupnu cijenu i vraća robu na skladište.

Tablica „Stavka_primke“ slabi je entitet između tablica „Roba“ i „Primka“. Primarni ključ je kompozitni i sastoji se od vanjskog ključa „Roba_idRoba“ i „Primka_idPrimka“. Polje „kolicina“ je obavezno i tipa *integer*. Ova tablica je jako slična stavki račun, ima dva okidača, prvi povećava količinu robe prilikom stvaranja novog zapisa u tablici, a prilikom brisanja smanjuje količinu robe.

Nad tablicama Stavki kreirani su okidači koji se aktiviraju nakon dodavanja (AFTER INSERT) i nakon brisanja (AFTER DELETE), a služe za smanjivanje ili povećavanje količine određene robe koja se navodi u Stavkama. Prilikom INSERT naredbe na tablicom „Stavka_racuna“ smanjuje se količina roba i povećava se ukupna cijena, a prilikom DELETE naredbe količina robe se povećava, a ukupna cijena smanjuje. Kod tablice „Stavka_primke“ ovi okidači rade obrnuto, prilikom INSERT količina se povećava, a prilikom DELETE ona se smanjuje.

Nad svim tablicama (osim „Račun“ i „Primka“) kreirani su okidači koji sprječavaju brisanje tog zapisa ako postoji neki drugi red koji se referencira na taj red preko vanjskog ključa. Nad tablicama „Račun“ i „Primka“ kreiran je okidač koji sprječava brisanje reda ako je taj red referenciran u nekom retku stavki. Svi navedeni okidači aktiviraju se prije brisanja retka u tablicama (BEFORE DELETE).

6.3. Izrada baze podataka

U ovom potpoglavlju obuhvaćeno je uspostavljanje lokalnog poslužitelja i priprema za izradu baze podataka korištenjem NuoDB SUBP-a. U nastavku je objašnjen proces kreiranja tablica i okidača u bazi podataka.

6.3.1. Uspostavljanje poslužitelja

Starije verzije NuoDB SUBP-a imale su mogućnost korištenja grafičkog sučelja za upravljanje poslužiteljima i bazama podataka. NuoDB verzija 3.2.1-1 je zadnja stabilna verzija ovog SUBP-a koja je korištena prilikom izrade završnog rada. Ova verzija prešla je na isključivo korištenje konzole za upravljanje sustavom. U koracima u nastavku bit će opisano korištenje NuoDB konzole za upravljanje. Treba uzeti u obzir da se koristi Windows operacijski sustav te se neki koraci mogu razlikovati prilikom korištenja neke distribucije Linux operacijskog sustava.

Pomoću komandnog sučelja Windowsa pokreće se *nuodbmanager.jar*, program kojim se upravlja poslužiteljem. Nuodbmanager se pokreće prebacivanjem na lokaciju JAR datoteke naredbom „cd <lokacija NuoDB instalacije>\jar“ i izvršavanjem naredbe „java -jar nuodbmanager.jar --broker <ime poslužitelja> --password <lozinka>“. S obzirom na to da pokrećem poslužitelja na svojem računalu, lokalno koristim *localhost* za brokera, a password se određuje tijekom instalacije NuoDB SUBP-a. Za pokretanje poslužitelja potrebno je pokrenuti dva procesa: TE i SM procese (NuoDB, n.d.-a). SM proces se pokreće naredbom *„start process sm database <ime baze podataka> host <ime poslužitelja> archive /<lokacija spremanja podataka>/<ime baze podataka> initialize true“*, pri čemu je naziv baze podataka proizvoljan, naziv poslužitelja koristimo ovisno gdje želimo smjestiti bazu, a lokaciju spremanja podataka određujemo proizvoljno. TE proces se pokreće naredbom „*start process te database <ime baze podataka> host <ime poslužitelja> options '--dba-user <korisnik> --dba-password <password>'“*, gdje se naziv baze podataka i poslužitelja biraju ovisno o kojoj bazi podataka i na kojem poslužitelju želimo pokrenuti TE, a korisnika i lozinku određujemo proizvoljno i te informacije služe za spajanje na bazu podataka bilo preko nuosql programa ili preko klijentske aplikacije. Time je uspostavljen lokalni poslužitelj s mogućnošću upravljanja jednom bazom

podataka. Koristim Community verziju NuoDB SUBP-a koja dopušta korištenje samo jednog TE-a i SM-a, tako da mogu uspostaviti i koristiti samo jednu bazu podataka.

6.3.2. Kreiranje tablica i okidača

NuoDB za upravljanje bazom podataka također koristi komandnu liniju unutar Windowsa. Izvršava se naredba „cd <lokacija NuoDB instalacije>\bin“ i onda se pokreće nuosql program naredbom „nuosql <ime baze podataka>@<ime poslužitelja> --user <korisnik> --password <lozinka>“. Nuosql program koristi se za izvršavanje SQL naredbi i za dobivanje informacija o određenoj bazi podataka. NuoDB kao NewSQL SUBP podržava klasične SQL naredbe koje se koriste i u relacijskim SUBP-ima. Baza podataka dolazi s predefiniranom shemom pod nazivom USER, pa sam za potrebe rada kreirao novu shemu ZAVRSNI korištenjem CREATE SCHEMA naredbe. Tablice su kreirane pomoću CREATE TABLE naredbe, a okidači pomoću CREATE TRIGGER naredbe. Za premještanje između shema koristi se naredba USE <ime sheme>, a za dohvaćanje popisa tablica u shemi koristi se SHOW TABLES. Naredbom SHOW TABLE <ime tablice> dobiju se podaci o stupcima, vrstama podataka tih stupaca i ograničenjima vezanima za tu tablicu.

6.4. Izrada aplikacije

Aplikacija je izrađena u Visual Studiu 2017 korištenjem Windows formi i programskog jezika C#. Veza na bazu podataka ostvarena je korištenjem upravljača *NuoDB Data Client*. Zbog korištenja Data Client-a, nije bilo moguće koristiti Entity Framework i ORM (engl. *Object Relational Mapping*) za upravljanje podacima pa sam morao samostalno implementirati način upravljanja podacima u bazi podataka. Na svakoj formi bilo je potrebno napisati znakovni niz za spajanje (engl. *connection string*) i SQL upite. Upisani SQL upiti su se onda izvršavali nakon što sam ručno otvorio vezu na bazu podataka koju je zatim trebalo i ručno zatvoriti. Nasuprot tome, kod korištenja ORM-a ne bi bilo potrebno sve to samostalno pisati, već bi se samo unosila potrebna naredba dostupna u Entity Framework razvojnom okviru.

Za spajanje na i dohvaćanje podataka iz baze podataka korišten je sljedeći programski kod:

```
string cs =
"Server=localhost;Database=dbukovac;User=dbukovac;Password=63515379;Schema=
NOVA";
NuoDbConnection connection = null;
DbDataReader rdr = null;
connection = new NuoDbConnection(cs);
```

```
connection.Open();
string sqlQuery = "select * from skladiste";
NuoDbCommand command = new NuoDbCommand(sqlQuery, connection);
rdr = command.ExecuteReader();
while (rdr.Read())
{
}
rdr.Close();
connection.Close();
```

6.4.1. Funkcionalnosti aplikacije

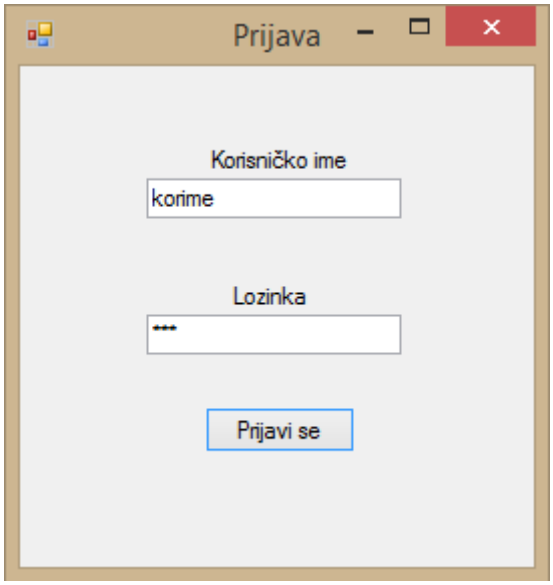
U ovom poglavlju bit će opisane funkcionalnosti aplikacije koje će biti grupirane prema sličnosti zato jer du pojedine funkcionalnosti veoma slične po izgledu forme i namjeni. Funkcionalnost izdavanja računa biti će detaljnije objašnjena jer je najsloženija pa se pomoću nje može detaljnije pojasniti većinu metoda korištenih u aplikaciji.

6.4.1.1. Glavni izbornik

Ova funkcionalnost sastoji se od forme za prijavu i glavne forme. Forma za prijavu je prva koja se pojavljuje kad se pokrene aplikacija i za nastavak njenog korištenja korisnik mora unesti svoje korisničko ime i lozinku. Slijedi provjera ispravnosti podataka i, u slučaju da su podaci ispravni, aplikacija otvara glavnu formu i prosljeđuje joj šifru zaposlenika i njegovu razinu prava. Glavna forma čini bazu za otvaranje ostalih formi jer je konfigurirana kao MDI (engl. *Multiple Document Interface*) spremnik i sadrži *Menustrip* kontrolu za izbor funkcionalnosti. Glavna forma prilikom otvaranja ostalih formi prosljeđuje šifru korisnika ako je to potrebno, a za otvaranje formi koristi drugu dretvu kako bi se smanjilo opterećenje na glavnoj dretvi. To se obavlja korištenjem *ShowFormDelegate* komponente, a metoda za otvaranje forme sadrži sljedeći programski kod:

```
private void ShowForm(Form form, Form parentForm)
        {
            if (form.InvokeRequired)
            {
                ShowFormDelegate showFormDelegate = new
ShowFormDelegate(ShowForm);
                form.BeginInvoke(showFormDelegate, form, parentForm);
            }
            else
            {
                form.MdiParent = parentForm;
                parentForm.Size = new Size(form.Width + 15, form.Height +
25);
                form.WindowState = FormWindowState.Maximized;
```

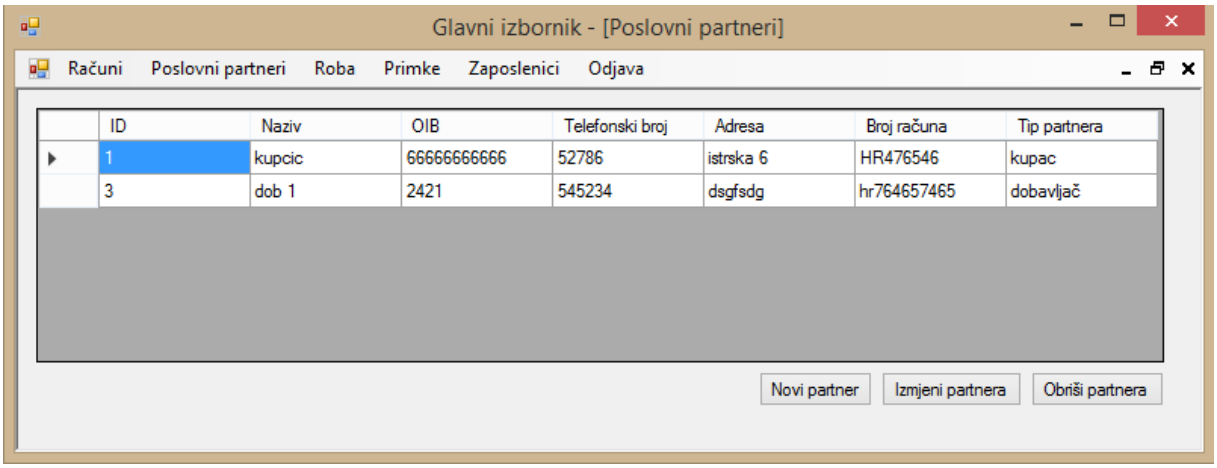
```
            form.Show();
        }
    }
```



Slika 6: Forma za prijavu (vlastita izrada)

6.4.1.2. Forme za unos, izmjenu i brisanje podataka

Funkcionalnosti koje spadaju u ovu kategoriju su: Poslovni partneri, Roba, Zaposlenici. Ove funkcionalnosti dozvoljavaju unos, izmjenu i brisanje poslovnih partnera, robe i zaposlenika. Poslovni partneri se unose prema dvije vrste, kupci i dobavljači, dok kod robe postoji mogućnost unosa, izmjene i brisanja skladišta te se upravlja pozicijama koje se dodjeljuju zaposlenicima. Svim ovim funkcionalnostima je zajedničko da se sastoje od dvije forme: forme za pregled i forme za unos.



Slika 7: Forma za pregled poslovnih partnera (vlastita izrada)

Na slici 7 prikazana je tipična forma za pregled koja nudi mogućnosti pregleda i brisanja podataka o poslovnim partnerima. Unos i izmjena rade se preko forme za unos, a tipke sa ove

forme za pregled otvaraju formu za unos. Formi za unos prilikom izmjene prosljeđuje se šifra poslovnog partnera, a prije toga provjerava se je li označen red koji se želi mijenjati. Forma za pregled ima dvije metode: *Get* i *obriši*. *Get* metoda popunjava *Datagridview* kontrolu svim zapisima iz određene tablice, a *obriši* u sklopu klika na tipku *obriši* briše označeni redak iz tablice i *Datagridview* kontrole.

Get metoda sastoji se od sljedećeg programskog koda:

```
private void GetPartneri()
        {
            try
            {
                connection.Open();
                string sqlQuery = "select * from poslovni_partner join
tip_partnera on poslovni_partner.tip_partnera_idtip_partnera =
tip_partnera.idtip_partnera";
                NuoDbCommand command = new NuoDbCommand(sqlQuery,
connection);
                partneriDGV.Rows.Clear();
                rdr = command.ExecuteReader();
                while (rdr.Read())
                {
                    string[] row = new string[] {
rdr["idposlovni_partner"].ToString(), rdr["naziv"].ToString(),
rdr["oib"].ToString(), rdr["telefonski_broj"].ToString(),
rdr["adresa"].ToString(), rdr["broj_racuna"].ToString(), rdr[8].ToString()
};
                    partneriDGV.Rows.Add(row);
                }
                rdr.Close();
            }
            catch (NuoDbSqlException ex)
            {
                Console.WriteLine("Error: {0}", ex.ToString());
            }
            finally
            {
                if (connection != null)
                    connection.Close();
            }
        }
```

U ovom isječku koda koristi se *try-catch* blok za hvatanje iznimaka koje se mogu dogoditi prilikom rada s bazom podataka kako bi se spriječilo da iznimka prekine rad aplikacije. U *try* dijelu otvara se prije definirana konekcija na bazu podataka, znakovni niz koji sadrži SQL kod pretvara se u objekt *NuoDBCommand* kako bi se nad njime koristio *DataReader* iz biblioteke *System.Data.Common*. *While* petlja služi za povlačenje svih redaka iz baze podataka preko *Readera* koji čita redak po redak pomoću *Read*() funkcije. Unutar *while* petlje

kreira se novi redak u *string* obliku s određenim informacijama iz *Readera* i taj red se dodaje u *Datagridview* kontrolu. *Catch* blok koji se izvršava samo ako se javi iznimka hvata iznimku i ne dopušta joj da sruši aplikaciju, dok se *finally* blok uvijek izvršava te zatvara konekciju ako je otvorena.

Kod *briši* metode je sljedeći:

```
private void obrisiPartneraButton_Click(object sender, EventArgs e)
        {
            if (partneriDGV.SelectedRows.Count > 0)
            {
                try
                {
                    connection.Open();
                    string sqlQuery;
                    sqlQuery = "delete from poslovni_partner where
idposlovni_partner = " +
int.Parse(partneriDGV.CurrentRow.Cells[0].Value.ToString()) + "";
                    NuoDbCommand command = new NuoDbCommand(sqlQuery,
connection);
                    command.ExecuteNonQuery();
                    MessageBox.Show("Uspješno izvršeno");
                }
                catch (NuoDbSqlException ex)
                {
                    Console.WriteLine("Error: {0}", ex.ToString());
                }
                finally
                {
                    if (connection != null)
                        connection.Close();
                    GetPartneri();

                }
            }
            else
            {
                MessageBox.Show("Nije odabran partner");
            }
        }
```

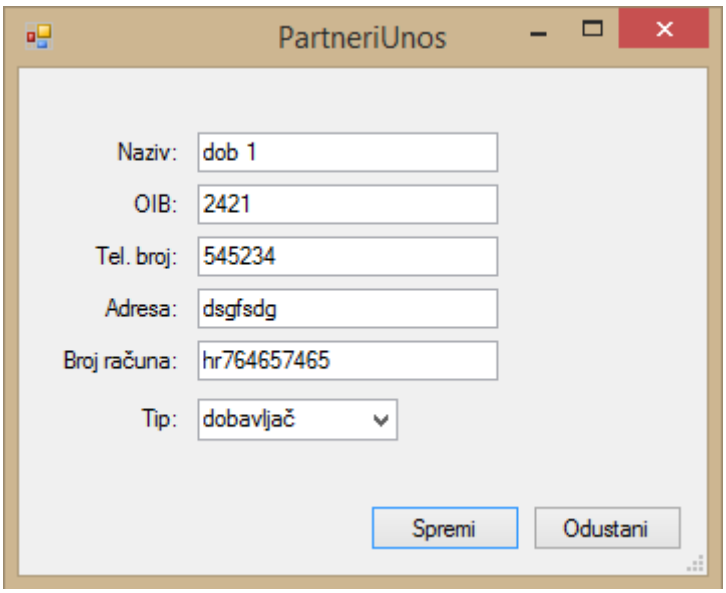
Prilikom svakog ostvarivanja konekcije na bazu podataka koristi se isti format sa *try-catch-finally* blokom. Tako i prilikom brisanja, gdje se na isti način kao i gore otvara veza, formira SQL *string* i objekt klase *NuoDBCommand*, jedina razlika je što se prilikom brisanja (*delete*), izmjene (*update*) i unosa (*insert*) ne koristi *Reader* nego *ExecuteNonQuery*() funkcija. Prije nego se uopće krene u *try-catch* bloku, provjerava se je li označen red koji se briše.

Za ostvarivanje konekcije na bazu podataka potreban je dio koda u imenskom prostoru (engl. *namespace*) forme i konstruktoru:

```
string cs =
"Server=localhost;Database=dbukovac;User=dbukovac;Password=63515379;Schema=
NOVA";
        NuoDbConnection connection = null;
        DbDataReader rdr = null;

        public PartneriPregled()
        {
            InitializeComponent();
            connection = new NuoDbConnection(cs);
        }
```

Ovim dijelom koda kreira se znakovni niz za spajanje na bazu podataka te se inicijaliziraju objekti klasa *NuoDBConnection* i *DBDataReader*. *NuoDBConnection* objektu daje se vrijednost u konstruktoru forme, a *Reader*-u netom prije korištenja u funkciji čitanja.



Slika 8: Forma za unos i izmjenu poslovnih partnera (vlastita izrada)

Slika iznad prikazuje tipičnu formu za unos, u ovom slučaju, poslovnih partnera. Ova forma ima dva konstruktora, ovisno o tome prosljeđuje li joj se šifra objekta prilikom izmjene ili se, pak, poziva forma bez argumenta prilikom kreiranja novog zapisa. Prilikom pokretanja forme poziva se metoda *FillComboBox*() kojoj je cilj dohvatiti poziciju za zaposlenika ili tip za poslovnog partnera i omogućiti biranje preko padajućeg izbornika. *FillBoxes*() metoda se poziva u slučaju pozivanja forme s ciljem izmjene objekta, gdje ta metoda, s obzirom na proslijeđenu šifru, popunjuje tekstualne blokove (engl. *textbox*) odgovarajućim podacima za lakšu izmjenu. Klikom na tipku *spremi* provjerava se sadržavaju li OIB i telefonski broj samo brojke i, u slučaju da je ispravan unos, provjerava se radi li se o izmjeni ili novom unosu. U slučaju izmjene koristi se SQL naredba UPDATE i navode se svi podaci za ažuriranje neovisno o tome jesu li promijenjeni ili ne, a u slučaju unosa koristi se SQL naredba INSERT.

```
private void FillComboBox()
        {
            try
            {
                connection.Open();
                string sqlQuery = "select * from tip_partnera";
                NuoDbCommand command = new NuoDbCommand(sqlQuery,
connection);
                tipCbox.Items.Clear();
                rdr = command.ExecuteReader();
                tipCbox.DisplayMember = "Text";
                tipCbox.ValueMember = "Value";
                idList = new List<int>();
                while (rdr.Read())
                {
                    tipCbox.Items.Add(new { Text = rdr["naziv"].ToString(),
Value = rdr["idtip_partnera"].ToString() });

idList.Add(int.Parse(rdr["idtip_partnera"].ToString()));
                }
                rdr.Close();
            }
            catch (NuoDbSqlException ex)
            {
                Console.WriteLine("Error: {0}", ex.ToString());
            }
            finally
            {
                if (connection != null)
                    connection.Close();
            }
        }
```

Metoda *FillComboBox(),* čiji je programski kod naveden gore, koristi prethodno opisanu strukturu upita čitanja korištenjem *try-catch* bloka, SQL naredbe i *Reader* objekta. Od pročitanih informacija stvara se *ComboBox* objekt koji se dodaje u listu objekata padajućeg izbornika. U listu *idList* zapisuje se šifra pozicije ili tipa partnera kako bi se kasnije ta šifra mogla koristiti tijekom zapisa u bazu.

Programski kod *FillBoxes*() metode je sljedeći:

```
private void FillBoxes()
        {
            try
            {
                connection.Open();
```

```
                string sqlQuery = "select * from poslovni_partner join
tip_partnera on poslovni_partner.tip_partnera_idtip_partnera =
tip_partnera.idtip_partnera where idposlovni_partner = "+idPartner+";";
                NuoDbCommand command = new NuoDbCommand(sqlQuery,
connection);
                rdr = command.ExecuteReader();
                while (rdr.Read())
                {
                    nazivTbox.Text = rdr["naziv"].ToString();
                    oibTbox.Text = rdr["oib"].ToString();
                    telbrojTbox.Text = rdr["telefonski_broj"].ToString();
                    adresaTbox.Text = rdr["adresa"].ToString();
                    brojracunaTbox.Text = rdr["broj_racuna"].ToString();
                    tipCbox.Text = rdr[8].ToString();
                }
                rdr.Close();
            }
            catch (NuoDbSqlException ex)
            {
                Console.WriteLine("Error: {0}", ex.ToString());
            }
            finally
            {
                if (connection != null)
                    connection.Close();
            }
        }
```

Ovo je tipična funkcija čitanja koja iz baze podataka dohvaća red iz tablice prema proslijeđenoj šifri. Tijekom čitanja *Textbox* objektima se dodjeljuju vrijednosti kako bi ih se moglo lako izmijeniti. Prilikom čitanja iz baze podataka, korištenjem *Reader* objekta mogu se izvući određeni podaci prema nazivu stupca korištenjem naredbe `rdr["telefonski_broj"].ToString();`

Kod koji se aktivira nakon pritiska na tipku *Spremi* je sljedeći:

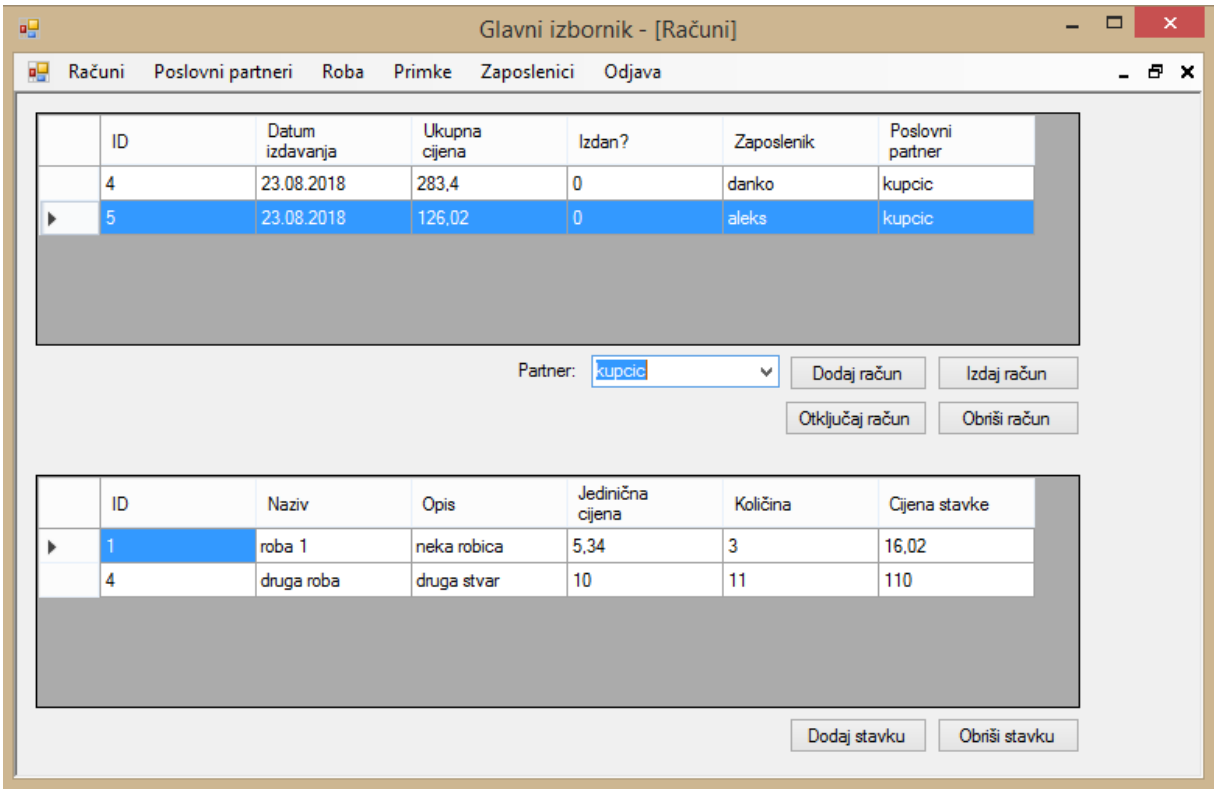
```
private void spremiButton_Click(object sender, EventArgs e)
        {
            double OIB;
            int telBroj;
            if (double.TryParse(oibTbox.Text, out OIB) &&
oibTbox.Text.Length == 11 && int.TryParse(telbrojTbox.Text, out telBroj) &&
tipCbox.SelectedIndex >= 0)
            {
                try
                {
                    connection.Open();
                    string sqlQuery;
                    if (idPartner == 0)
                    {
                        sqlQuery = "insert into poslovni_partner values
(default, '" + nazivTbox.Text + "', '" + OIB + "' " + ", '" + telBroj + "'
```

```
" + ", '" + adresaTbox.Text + "' " + ", '" + brojracunaTbox.Text + "', " +
idList[tipCbox.SelectedIndex] + " ); ";
                    }
                    else
                    {
                        sqlQuery = "update poslovni_partner set naziv = '"
+ nazivTbox.Text + "', oib = '" + oibTbox.Text + "', telefonski_broj = '" +
telBroj + "', adresa = '" + adresaTbox.Text + "', broj_racuna = '" +
brojracunaTbox.Text + "', tip_partnera_idtip_partnera = " +
idList[tipCbox.SelectedIndex] + " where idposlovni_partner = " + idPartner
+ ";";
                    }
                    NuoDbCommand command = new NuoDbCommand(sqlQuery,
connection);
                    command.ExecuteNonQuery();
                    MessageBox.Show("Uspješno izvršeno");
                }
                catch (NuoDbSqlException ex)
                {
                    Console.WriteLine("Error: {0}", ex.ToString());
                }
                finally
                {
                    if (connection != null)
                        connection.Close();
                }
            }
            else
            {
                MessageBox.Show("Krivo upisani OIB");
            }
        }
```

Ovaj kod prvo provjerava jesu li OIB i telefonski broj uneseni u numeričkom obliku korištenjem *try.Parse()* funkcije i provjerava se je li odabrana jedna vrijednost iz padajućeg izbornika. Nakon toga, provjerava se radi li se o unosu ili izmjeni ovisno o tome pokreće se SQL naredba INSERT za unos ili UPDATE naredba za izmjenu. Vidi se da je *sqlQuery* string objekt doslovno ručno napisani kod u SQL-u kakav bi se mogao izvesti u NuoDB konzoli ili drugom SUBP-u koji koristi SQL jezik.

6.4.1.3. Forme s unosom stavki

U ovu kategoriju spadaju funkcionalnosti Računi i Primke. Ova kategorija podrazumijeva unos novih računa i primki, njihovo brisanje te unos i brisanje stavki na njima. Ove funkcionalnosti sastoje se od dvije forme: forma za pregled, unos i brisanje računa/primki te forma za unos stavki.



Slika 9: Forma za pregled, unos i brisanje računa (vlastita izrada)

Ova forma omogućuje pregled računa i njegovih stavki. Za prikaz računa koristi se *GetRacuni*() metoda koja je ista kao i *Get* metoda iz prethodnog potpoglavlja. Za prikaz stavki računa aktivira se funkcija *GetStavke*() prilikom označavanja nekog reda u *Datagridview* kontroli. Unos novog računa izvršava se odabirom poslovnog partnera iz padajućeg izbornika i pritiskom na tipku *Dodaj račun*, a nude se samo kupci kao vrsta poslovnog partnera. Datum izdavanja automatski se postavlja na trenutni datum, cijena na 0, *izdan* na 0, a identifikator zaposlenika prosljeđuje se formi prilikom njezinog otvaranja. Izdavanjem računa on se zaključava i ne može se brisati ni mijenjati stavke. Otključavanje računa za promjenu dostupno je samo administratoru. Brisanje računa može se izvršiti samo kad ne postoje stavke na tom računu. Brisanje i dodavanje stavki moguće je samo kad je račun otključan i mora biti označen red s računom za koji se želi dodati stavka.

Kod dodavanja novog računa izvršava se sljedeći programski kod:

```
private void dodajRacunButton_Click(object sender, EventArgs e)
        {
            if (partneriCbox.SelectedIndex >= 0)
            {
                try
                {
                    connection.Open();
                    string sqlQuery;
```

```
                    sqlQuery = "insert into racun values (default, '" +
DateTime.Today.ToString("dd/MM/yyyy") + "', 0 , 0, " +
idList[partneriCbox.SelectedIndex] + ", " + idzaposlenik + " ); ";
                    NuoDbCommand command = new NuoDbCommand(sqlQuery,
connection);
                    command.ExecuteNonQuery();
                    MessageBox.Show("Uspješno izvršeno");
                }
                catch (NuoDbSqlException ex)
                {
                    Console.WriteLine("Error: {0}", ex.ToString());
                }
                finally
                {
                    if (connection != null)
                        connection.Close();
                    GetRacuni();
                }
            }
        }
```

Na početku se provjerava je li izabran kupac preko padajućeg izbornika, ako jest, onda se izvršava INSERT naredba koja stvara račun na koji se mogu dodavati stavke. *DateTime.Today* koristi se za dohvaćanje trenutnog datuma, a nakon zatvaranja konekcije pokreće se metoda *GetRacuni*() za obnavljanje *Datagridview* kontrole s novim računom.

Programski kod za izdavanje računa je sljedeći:

```
private void izdajRacunButton_Click(object sender, EventArgs e)
        {
            if (racuniDGV.SelectedRows.Count > 0 &&
int.Parse(racuniDGV.CurrentRow.Cells[3].Value.ToString()) == 0)
            {
                try
                {
                    connection.Open();
                    string sqlQuery;
                    sqlQuery = "update racun set kontrola = 1 where idracun
= " + int.Parse(racuniDGV.CurrentRow.Cells[0].Value.ToString()) + "";
                    NuoDbCommand command = new NuoDbCommand(sqlQuery,
connection);
                    command.ExecuteNonQuery();
                    MessageBox.Show("Uspješno izvršeno");
                }
                catch (NuoDbSqlException ex)
                {
                    Console.WriteLine("Error: {0}", ex.ToString());
                }
                finally
                {
                    if (connection != null)
                        connection.Close();
                    GetRacuni();

                }
            }
```

```
            else if
(int.Parse(racuniDGV.CurrentRow.Cells[3].Value.ToString()) == 1)
            {
                MessageBox.Show("Račun je već zaključan");
            }
            else
            {
                MessageBox.Show("Nije odabran račun");
            }
        }
```

U metodi za izdavanje računa provjerava se je li označen red u *Datagridview*-u i je li račun otključan. Ako su oba uvjeta zadovoljena, koristi se UPDATE naredba koja mijenja vrijednost u stupcu *kontrola* u 1 i time zaključava račun. Šifra računa iz *Datagridview*-a dobiva se korištenjem `racuniDGV.CurrentRow.Cells[0].Value`. Dodatno se provjerava je li račun zaključan kako bi se ispisala ispravna poruka upozorenja. Kod za otključavanje računa identičan je ovome, uz razliku da se provjerava je li račun zaključan i vrijednost stupca *kontrola* mijenja se u 0.

Kod brisanja računa izvršava se sljedeći programski kod:

```
private void obrisiRacunButton_Click(object sender, EventArgs e)
        {
            if (racuniDGV.SelectedRows.Count > 0 &&
int.Parse(racuniDGV.CurrentRow.Cells[3].Value.ToString()) == 0)
            {
                try
                {
                    connection.Open();
                    string sqlQuery;
                    sqlQuery = "delete from racun where idracun = " +
int.Parse(racuniDGV.CurrentRow.Cells[0].Value.ToString()) + "";
                    NuoDbCommand command = new NuoDbCommand(sqlQuery,
connection);
                    command.ExecuteNonQuery();
                    MessageBox.Show("Uspješno izvršeno");
                }
                catch (NuoDbSqlException ex)
                {
                    Console.WriteLine("Error: {0}", ex.ToString());
                    MessageBox.Show("Račun se ne može brisati dok ima
stavke");
                }
                finally
                {
                    if (connection != null)
                        connection.Close();
                    GetRacuni();

                }
            }
```

```
            else
if(int.Parse(racuniDGV.CurrentRow.Cells[3].Value.ToString()) == 1)
            {
                MessageBox.Show("Račun je izdan i ne može se mijenjati");
            }
            else
            {
                MessageBox.Show("Nije odabran račun");
            }
        }
```

Kod je jako sličan onome za brisanje u prethodnom potpoglavlju, uz dodatnu provjeru je li račun zaključan i poruku koja se javlja unutar *catch* bloka zbog korištenja okidača za uzrokovanje iznimke u slučaju da račun ima stavke na sebi.

Kod dohvaćanja stavki računa izvršava se sljedeći programski kod:

```
private void GetStavke()
        {
            try
            {
                if (racuniDGV.SelectedRows.Count > 0)
                {
                    connection.Open();
                    string sqlQuery = "select * from stavka_racuna join
roba on stavka_racuna.roba_idroba = roba.idroba where
stavka_racuna.racun_idracun = " +
int.Parse(racuniDGV.CurrentRow.Cells[0].Value.ToString());
                    NuoDbCommand command = new NuoDbCommand(sqlQuery,
connection);
                    stavkaDGV.Rows.Clear();
                    rdr = command.ExecuteReader();
                    while (rdr.Read())
                    {
                        decimal cijena =
decimal.Parse(rdr["jedinicna_cijena"].ToString());
                        int kolicina = int.Parse(rdr[2].ToString());
                        decimal uk_cijena = cijena * kolicina;
                        string[] row = new string[] {
rdr["idroba"].ToString(), rdr["naziv"].ToString(), rdr["opis"].ToString(),
rdr["jedinicna_cijena"].ToString(), rdr[2].ToString(), uk_cijena.ToString()
};
                        stavkaDGV.Rows.Add(row);
                    }
                    rdr.Close();
                }
            }
            catch (NuoDbSqlException ex)
            {
                Console.WriteLine("Error: {0}", ex.ToString());
            }
```

```
            finally
            {
                if (connection != null)
                    connection.Close();
            }
        }
```

Ova funkcija koristi SELECT naredbu za dohvaćanje svih stavki računa i informacija o pripadajućoj robi preko šifre računa. Unutar petlje čitača izračunava se cijena svake stavke i ta informacija uz ostale čine redak u Datagridview-u.

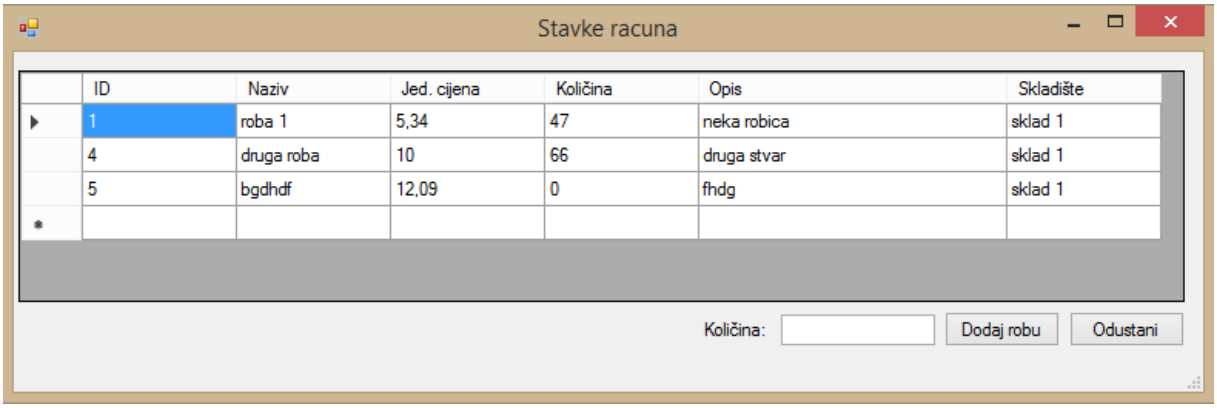
Kod za brisanje stavke:

```
private void obrisiStavkubutton_Click(object sender, EventArgs e)
        {
            if (racuniDGV.SelectedRows.Count > 0 &&
int.Parse(racuniDGV.CurrentRow.Cells[3].Value.ToString()) == 0 &&
stavkaDGV.SelectedRows.Count > 0)
            {
                try
                {
                    connection.Open();
                    string sqlQuery;
                    sqlQuery = "delete from stavka_racuna where
racun_idracun = " +
int.Parse(racuniDGV.CurrentRow.Cells[0].Value.ToString()) + " and
roba_idroba = "+ int.Parse(stavkaDGV.CurrentRow.Cells[0].Value.ToString())
+ "";
                    NuoDbCommand command = new NuoDbCommand(sqlQuery,
connection);
                    command.ExecuteNonQuery();
                    MessageBox.Show("Uspješno izvršeno");
                }
                catch (NuoDbSqlException ex)
                {
                    Console.WriteLine("Error: {0}", ex.ToString());
                }
                finally
                {
                    if (connection != null)
                        connection.Close();
                    GetStavke();

                }
            }
            else if
(int.Parse(racuniDGV.CurrentRow.Cells[3].Value.ToString()) == 1)
            {
                MessageBox.Show("Račun je izdan i ne može se mijenjati");
            }
            else
```

```
            {
                MessageBox.Show("Nije odabran račun");
            }
        }
```

Brisanje stavke radi se kao i brisanje računa, uz činjenicu da je potrebno znati za koji račun se briše koja roba, tj. radi se provjera je li označen redak u svakoj tablici. Nakon zatvaranja veze pokreće se metoda *GetStavke*() da bi se ažurirala promjena u *Datagridview*-u.



Slika 10: Forma za unos stavki (vlastita izrada)

Pomoću forme prikazane na slici 10 unose se stavke na račun ili primku. U *Datagridview* kontroli ispiše se sva roba i odabirom reda i određivanjem količine kreira se stavka. Podaci u *Datagridview* kontroli dohvaćaju se metodom *GetRoba*() čija je logika ista kao kod drugih metoda za dohvaćanje podataka.

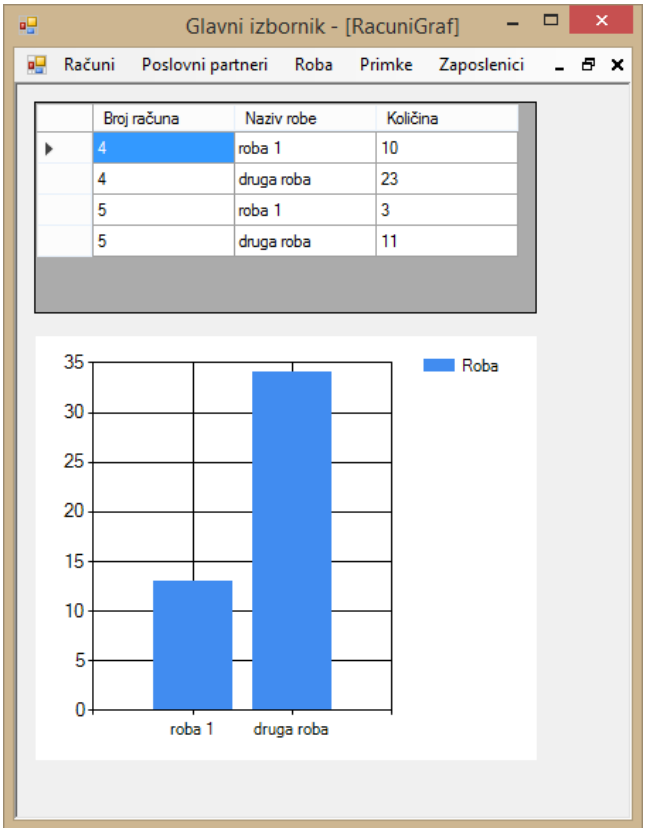
Pritiskom na tipku *Dodaj robu* izvršava se sljedeći programski kod:

```
private void dodajRobuButton_Click(object sender, EventArgs e)
        {
            if(int.Parse(kolicinaTbox.Text) <=
int.Parse(robaDGV.CurrentRow.Cells[3].Value.ToString()) &&
robaDGV.SelectedRows.Count > 0 && int.Parse(kolicinaTbox.Text)> 0)
            {
                try
                {
                    connection.Open();
                    string sqlQuery;
                    sqlQuery = "insert into stavka_racuna values (" +
idracun + "," + int.Parse(robaDGV.CurrentRow.Cells[0].Value.ToString()) +
"," + int.Parse(kolicinaTbox.Text) + ")";
                    NuoDbCommand command = new NuoDbCommand(sqlQuery,
connection);
                    command.ExecuteNonQuery();
                    MessageBox.Show("Uspješno izvršeno");
                }
                catch (NuoDbSqlException ex)
```

```
                {
                    Console.WriteLine("Error: {0}", ex.ToString());
                    MessageBox.Show("Ta roba već postoji na računu");
                }
                finally
                {
                    if (connection != null)
                        connection.Close();
                }
            }
            else if(int.Parse(kolicinaTbox.Text) >
int.Parse(robaDGV.CurrentRow.Cells[3].Value.ToString()))
            {
                MessageBox.Show("Nema dovoljno tražene robe");
            }
            else
            {
                MessageBox.Show("Nije odabrana roba");
            }
        }
```

Uz provjeru je li označena roba i je li određena količina, provjerava se je li količina veća od raspoložive količine za tu robu. Kod stavke primke ne postoji ograničenje na količinu jer se radi o povećanju količine robe. Preko konstruktora forme prosljeđuje se šifra računa koja u INSERT naredbi, uz šifru robe koja se dobije preko označivanja reda u tablici, služi za jedinstveno identificiranje stavke. U *catch* dijelu koda hvata se iznimka koju baza podataka generira u slučaju da već postoji označena roba na označenom računu. U tom slučaju, ako se želi promijeniti količina, stavka se mora obrisati i kreirati nova.

6.4.1.4. Graf prodane robe



Slika 11: Forma sa grafom (vlastita izrada)

Unutar posebne forme se prikazuje statistika prodane robe prikazana pomoću tablice i grafa. Prilikom učitavanja forme prikazuje se tablica sa svim stavkama računa i pripadajući graf koji pokazuje koliko je koje robe prodano. Koriste se metode *GetPrimke()*, *FillList()* i *FillChart()*.

Kod funkcije *GetPrimke*() je sljedeći:

```
private void GetPrimke()
        {
            try
            {
                connection.Open();
                string sqlQuery = "select s.racun_idracun, r.naziv,
s.kolicina from roba r join stavka_racuna s on s.roba_idroba = r.idroba
order by 1;";
                NuoDbCommand command = new NuoDbCommand(sqlQuery,
connection);
                grafDGV.Rows.Clear();
                rdr = command.ExecuteReader();
                while (rdr.Read())
                {
```

```
                    string[] row = new string[] {
rdr["racun_idracun"].ToString(), rdr["naziv"].ToString(),
rdr["kolicina"].ToString() };
                    grafDGV.Rows.Add(row);
                }
                rdr.Close();
            }
            catch (NuoDbSqlException ex)
            {
                Console.WriteLine("Error: {0}", ex.ToString());
            }
            finally
            {
                if (connection != null)
                    connection.Close();
            }
        }
```

Ovaj dio koda koristi upit koji dohvaća broj računa, količinu sa stavke i naziv robe iz tablice Roba spajajući tablice Roba i Stavka računa te sortira rezultat prema broju računa. U svemu ostalome funkcija je ista onima koje dohvaćaju podatke za *Datagridview* kontrole.

Kod metode *FillList*() je sljedeći:

```
private void FillList()
        {
            foreach (DataGridViewRow row in grafDGV.Rows)
            {

if(dict.ContainsKey(row.Cells["Naziv_robe"].Value.ToString()))
                {
                    dict[row.Cells["Naziv_robe"].Value.ToString()] +=
int.Parse(row.Cells["Kolicina"].Value.ToString());
                }
                else
                {
                    dict.Add(row.Cells["Naziv_robe"].Value.ToString(),
int.Parse(row.Cells["Kolicina"].Value.ToString()));
                }
            }
        }
```

Ova metoda prema svim redovima iz tablice provjerava da li se u riječniku nalazi roba s određenim nazivom. U slučaju da se ne nalazi, ubacuje taj naziv robe i količinu kao par ključ-vrijednost, a ako se ta roba već nalazi u rječniku, onda se samo tom paru uvećava količina za tu trenutnu vrijednost. Rječnik se koristi zbog idealnog načina spremanja podataka u obliku ključ-vrijednost jer graf na taj način prima podatke za prikaz.

Kod metode *FillChart*() je sljedeći:

```
private void FillChart()
        {
            foreach (KeyValuePair<string, int> entry in dict)
            {
                chart1.Series["Roba"].Points.AddXY(entry.Key, entry.Value);
            }
        }
```

Kao što se može vidjeti iz gore navedenog programskog koda, radi se o jednostavnoj metodi koja iterira kroz rječnik koji već sadrži agregirane i obrađene podatke te ih samo unosi u graf. *Foreach* petlja iterira, a korištenjem naredbe *AddXY*() za vrstu „Roba“ dodaju se vrijednosti u obliku ključ-vrijednost, gdje je ključ na X osi i sadrži ime robe, a vrijednost je količina na Y osi.

7. Zaključak

Tema ovog rada je NuoDB sustav za upravljanje bazama podataka, uz dodatan prikaz karakteristika NewSQL tehnologija. U radu su predstavljene NewSQL tehnologije i detaljnije NuoDB SUBP na način da se pomoću njega kreirala baza podataka i aplikacija koja je komunicirala s tom bazom podataka.

Najprije bih se osvrnuo na NewSQL tehnologije općenito i njihov položaj na današnjem tržištu. Spomenute tehnologije su proizašle iz potrebe da se spoji pouzdanost relacijskih tehnologija i njihovo korištenje SQL jezika s distribuiranima bazama podataka NoSQL sustava. NewSQL unatoč naizgled pogodnim karakteristikama još uvijek nisu osjetno zaživile i zauzele značajan dio tržišta iz par razloga. Prvi razlog, koji je moguće i najvjerojatniji, je trošak i sveukupni problem prelaska na korištenje nove tehnologije za neku tvrtku. NewSQL sustavi trude se omogućiti lakši prelazak na njihovo korištenje, međutim iskustva u korištenju novih sustava su još uvijek skromna i nedostatna pa se takav potez smatra rizičnim. Drugi razlog je što se relacijski sustavi mogu unaprijediti na način da omogućuju kreiranje distribuiranih baza podataka i što su neki NoSQL sustavi počeli podržavati neke od zahtjeva atomnosti, trajnosti, izolacije i konzistencije.

U ovom radu, baza podataka za aplikaciju izrađena je koristeći NuoDB SUBP, a kod je pisan u jeziku C# u Visual Studiu 2017. Prilikom izrade baze podataka jedini nedostatak koji bih naveo je što NuoDB nema vizualno sučelje za kreiranje i administraciju baze podataka. U prošlim verzijama NuoDB-a to sučelje je postojalo, međutim u novijim verzijama ono je ukinuto. Osim toga, ne postoje neke dodatne prepreke u izgradnji baze putem konzole. Izgradnju i interakciju baze podataka s više TE-a i SM-ova nisam mogao isprobati jer besplatna verzija NuoDB-a dopušta uspostavljanje samo jednog TE-a i SM-a. Pisanje koda u razvojnom okruženju Visual Studio 2017 je vrlo lako i nudi mnogo pogodnosti, prvenstveno njihov *Intellisense* koji jako ubrzava pisanje koda. NuoDB je prije imao bolju podršku za C# u pogledu korištenja Entity Frameworka. Starije verzije Visual Studia omogućavale su interakciju s bazom podataka preko *Data Source*-a i ADO .Net Entity Modela, međutim za Visual Studio 2017 trenutno funkcionira samo *NuoDBDataClient*. Rad preko tog *DataClienta* zahtijeva pisanje vlastitog koda za spajanje na bazu podataka, otvaranje i zatvaranje veze te izvršavanje upita. U tom pogledu se pisanje koda zakompliciralo više nego što bi to bio slučaj da sam koristio ADO .Net Entity Model.

Popis literature

13 NewSQL Databases - Compare Reviews, Features, Pricing in 2018 -PAT RESEARCH: B2B Reviews, Buying Guides & Best Practices. (n.d.). Retrieved August 2, 2018, from https://www.predictiveanalyticstoday.com/newsql-databases/

Amitai, B. (n.d.). Tech Blog. Retrieved August 2, 2018, from https://www.spectory.com/blog/NewSQL databases

Bosworth, B. (n.d.). DataStax CEO: Let's clear the air about NoSQL and ACID | InfoWorld. Retrieved August 2, 2018, from https://www.infoworld.com/article/2611540/nosql/datastax-ceo--let-s-clear-the-air-about-nosql-and-acid.html

Carvelli, R. (2015). SQL & NOSQL: A Brief History | Grio Blog. Retrieved July 21, 2018, from http://blog.grio.com/2015/11/sql-nosql-a-brief-history.html

Cihan, B. (n.d.). Indexing Big Data: Global vs. Local Indexes in Distributed Databases - DZone Database. Retrieved August 2, 2018, from https://dzone.com/articles/faster-indexing-and-query-global-vs-local-indexing

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, *13*(6), 377–387. https://doi.org/10.1145/362384.362685

Database Guide. (n.d.). What is a Column Store Database? | Database.Guide. Retrieved August 7, 2018, from https://database.guide/what-is-a-column-store-database/

Date, C. J. (1990). *An introduction to database systems*. Addison-Wesley.

Fortune, S. (2014). A Brief History of Databases - Avant.org. Retrieved July 21, 2018, from http://avant.org/project/history-of-databases/

Gobla, R. (n.d.). Row Store vs. Column Store Databases - DZone Database. Retrieved August 7, 2018, from https://dzone.com/articles/row-store-and-column-store-databases

Kerstiens, C. (n.d.). Database sharding explained in plain English. Retrieved August 2, 2018, from https://www.citusdata.com/blog/2018/01/10/sharding-in-plain-english/

Kumar, R., Gupta, N., Maharwal, H., Charu, S., & Yadav, K. (2014). Critical Analysis of Database Management Using NewSQL. *International Journal of Computer Science and Mobile Computing*, *35*(5), 434–438. Retrieved from http://s3.amazonaws.com/academia.edu.documents/33752078/V3I5201499a2.pdf?AWSAccessKeyId=AKIAIWOWYYGZ2Y53UL3A&Expires=1495314606&Signature=VhHyc%2BR0An%2FF8Oa6W5EAkCgxO9c%3D&response-content-disposition=inline%3B

filename%3DCritical_Analysis_of_Database_Ma

Mendis, W. (n.d.). From RDBMS to Key-Value Store: Data Modeling Techniques. Retrieved August 9, 2018, from https://medium.com/@wishmithasmendis/from-rdbms-to-key-value-store-data-modeling-techniques-a2874906bc46

Neo4j. (n.d.-a). Relational Databases vs. Graph Databases: A Comparison. Retrieved August 9, 2018, from https://neo4j.com/developer/graph-db-vs-rdbms/

Neo4j. (n.d.-b). What Is a Graph Database and Property Graph | Neo4j. Retrieved August 9, 2018, from https://neo4j.com/developer/graph-database/

NuoDB. (n.d.-a). Creating Your First Database. Retrieved September 4, 2018, from http://doc.nuodb.com/Latest/Content/Creating-Your-First-Database.htm

NuoDB. (n.d.-b). NuoDB Architecture, *1*, 1–24. Retrieved from http://go.nuodb.com/rs/nuodb/images/Technical-Whitepaper.pdf?aliId=43188293

Pavlo, A., & Aslett, M. (2016). What's Really New with NewSQL? *SIGMOD Record*, *45*(2), 45–55. https://doi.org/10.1145/3003665.3003674

Pikeos, J. (n.d.). SQL vs. NoSQL vs. NewSQL: Finding the Right Solution - Dataconomy. Retrieved August 2, 2018, from http://dataconomy.com/2015/08/sql-vs-nosql-vs-newsql-finding-the-right-solution/

Rouse, M. (n.d.). What is database replication? - Definition from WhatIs.com. Retrieved August 2, 2018, from https://searchdatamanagement.techtarget.com/definition/database-replication

Venkatesh, P. (2012). *NewSQL - The New Way to Handle Big Data*. Retrieved from http://www.linuxforu.com/2012/01/newsql-handle-big-data/

Yao, C., Zhang, M., Lin, Q., Ooi, B. C., & Xu, J. (2017). Scaling Distributed Transaction Processing and Recovery based on Dependency Logging. Retrieved from http://arxiv.org/abs/1703.02722

Zicari, R. (n.d.). On RDBMS, NoSQL and NewSQL databases. Interview with John Ryan | ODBMS Industry Watch. Retrieved August 2, 2018, from http://www.odbms.org/blog/2018/03/on-rdbms-nosql-and-newsql-databases-interview-with-john-ryan/

Popis slika

Popis tablica

Popis kratica

SUBP	Sustav za upravljanje bazama podataka
SQL	Standard Query Language
ACID	Atomicity, Consistency, Isolation, Durability
ERA	Entity-Relationship-Attributes
API	Application Programming Interface
SSD	Solid State Disk
MVCC	Multi-version Concurrency Control
2PL	Two-phase Locking
TO	Time Ordering
TE	Transaction Engine
SM	Storage Manager
OIB	Osobni Identifikacijski Broj
ORM	Object Relational Mapping
MDI	Multiple Document Interface