NoSQL baze podataka

Gačić, Jaka

Master's thesis / Diplomski rad

2017

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:074378

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-29**



Repository / Repozitorij:

Repository of the Faculty of Science - University of Zagreb







SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Jaka Gačić

NOSQL BAZE PODATAKA

Diplomski rad

Voditelj rada:
izv. prof. dr. sc. Saša Singer

Zagreb, rujan, 2017.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Sadržaj **iii**

Uvod **2**

1 Baze podataka **3**
1.1 Pojava baza podataka 3
1.2 Relacijski model 7
1.3 SQL jezik 12

2 NoSQL pokret **17**
2.1 Skalabilnost 17
2.2 Vanjski servisi 19
2.3 Opuštena konzistentnost 22
2.4 Fleksibilnost 24

3 Modeli podataka **27**
3.1 Ključ–vrijednost baze 27
3.2 Graf baze 33
3.3 Stupčane baze 40
3.4 Dokument baze 47

Bibliografija **55**

Uvod

Dokaz bogate povijesti baza podataka, prije nego što smo ih zvali tim imenom, seže u stari Egipat, gdje su pronađenji zapisi pohranjeni na papirusu i kamenim pločama, koji datiraju još iz 2800. godine prije Krista. Navedeni spisi pokazuju da su se već tada popisivali stanovnici, podaci o njihovim posjedima, poreznim obavezama i ispunjavanju istih. Brigu o popisu faraon je prepuštao svojim najvišim časnicima koji bi zatim obilazili zemlju te dopunjavali popis.

Sačuvani su i mnogi zapisi britanskih redovnika porijeklom iz 15. stoljaća koji bilježe prihode i troškove njihovih crkvenih zajednica. Jedan redovnik unutar zajednice bio bi odabran za blagajnika. Njegovo zaduženje bilo je prikupljanje, bilježenje i kategoriziranje podataka o crkvenim posjednima, donacijama koje su dobivali od monarhije i stanovništva te svim troškovima zajednice.

Veliki korak u obradi podataka u 17. stoljeću napravila je tzv. *časna istočnoindijska kompanija*, dioničko društvo sa sjedištem u Londonu koje se bavilo trgovinom s Indijom, Kinom i Amerikom. Svi podaci o trgovini morali su se pohranjivati u sjedištu, ali i u svim lokalnim podružnicama. Usklađivanje ovog rastućeg broja podataka, iz dana u dan, postajao je sve veći izazov. Lokalni podaci su do Londona pristizali brodovima, zajedno s pošiljkama, a središnju pohranu su zatim ažurirali brojni zaposlenici. Takva putovanja su trajala dugo, a brodovi su znali i ne stići do posljednjeg odredišta. Zbog toga su se odlučili na udvostručavanje podataka tako da je svaki brod nosio podatke o svim poznatim teretima, uključujući i one na drugim brodovima. Zbog ogromne količine podataka, 1800. godine *Kompanija* je izgradila zgradu koju su koristili isključivo za njihovu pohranu. Dakle, već tu vidimo sustav koji ima obilježja modernih *distribuiranih* baza podataka sa *sigurnosnim kopijama*.

Knjižnice su, također, odigrale važnu ulogu u razvoju baza podataka kakve danas poznajemo, uvođenjem indeksnih kartica i njihovim grupiranjem u kartoteke. Radi lakšeg pretraživanja, bilježili su podatke o knjigama na manje listiće koje bi zatim grupirali prema različitim kriterijima. Kada bi im stigao upit o pojedinoj knjizi, bilo je puno lakše pretražiti određenu grupu indeksnih kartica nego nepreglednu količinu polica punih knjiga. Belgijski vizionar Paul Otlet još je krajem 19. stoljeća razradio sistem indeksnih kartica, koje podsjećaju na današnji hipertekst, koji mu je omogućio relativno brzu pretragu njegove bogate

arhive znanstvenih knjiga i članaka. Upravo na sličan način danas funkcionira *indeksiranje* baza i pretraživanje Interneta. No, što ako nas, osim naslova ili autora, zanimaju drugi podaci, a ne toliko česti upiti? Kako bismo saznali koje su sve knjige u knjižnici tiskane 1977. godine? Ukoliko ne želimo posvetiti nekoliko dana vremena ručnom pretraživanju svih podataka, jasno je da ovakvi zadaci zahtijevaju drugi način obrade.

Amerikanac Herman Hollerith 1884. godine izrađuje prvi stroj za automatsku obradu podataka. Takozvani *sortirni stroj* koristio se 1890. godine za obradu podataka prilikom popisa stanovništva u SAD-u. Kao ulaz je primao *bušene kartice* – papirne kartice na kojima se podaci zapisuju s pomoću rupica tako da postojanje ili nedostatak rupice određuje binarni podatak 0 ili 1. U ovom slučaju, te kartice sadržavale su informacije o stanovništvu. Stroj je zbrajao podatke s danih bušenih kartica pomoću elektromagnetskih brojila te time obradu podataka sveo sa nekoliko godina na samo nekoliko mjeseci. Za svoj rad stroj je koristio električnu energiju dobivenu iz baterija. Uskoro je započela komercijalna proizvodnja ovakvih strojeva, a gospodin Hollerith je osnovao kompaniju koja je kasnije izrasla u današnji *IBM*, jednog od najvećih proizvođača računala na svijetu.

U sljedećem stoljeću krenuo je užurbani razvoj računala i njihove memorije – bušene kartice su zaboravljene u korist jeftinijih, bržih i manjih jedinica s više mjesta za pohranu. Računala su krenula u komercijalnu proizvodnju. Počela je izrada raznih aplikacija za obradu podataka. Jedini dostupan sustav memorije bio je *datotečni sustav*, a mogućnosti koje je pružao su kreiranje i brisanje datoteke, te pisanje i čitanje. Programer aplikacije je morao voditi računa o položaju svakog znaka na disku i pristupati podacima preko njihove apsolutne adrese. Nemoguće je bilo podatak premjestiti na drugu fizičku lokaciju bez da se izmijeni sav kod aplikacije koji je radio s tim podatkom. Dodavanje novog polja zahtijevalo je pomicanje svih do tada spremljenih podataka. Svaka instanca aplikacije stvarala je zasebne datoteke, što je često uzrokovalo udvostručavanje podataka. Tu su nastajali i problemi konzistentnosti kod modificiranja bilo kojeg podatka. Zbog toga su ovakvi sustavi bili jako teški za održavanje. Također, zbog veličine datoteka i nedostatka meta podataka, pretraživanje je bilo krajnje neefikasno. Arhitekti tadašnjih aplikacija većinu su vremena ulagali na procedure za rukovanje podacima. Svaka je aplikacija, čak i unutar iste tvrtke, najčešće bila nekompatibilna sa svima drugima. Iako su funkcije datotečnih sustava i danas dio jezgre operacijskih sustava, više se ne koriste izravno kao aplikacijska memorija. Razvojem sve složenijih sustava, postalo je jasno da je vrijeme za promjene.

Poglavlje 1

Baze podataka

U ovom poglavlju objasnit ćemo što su uopće baze podataka i kako su se razvijale. Predstavit ćemo nekoliko različitih modela podataka koji su se pojavili kroz godine. Posebnu pažnju posvetit ćemo relacijskom modelu koji je već dugi niz godina najrašireniji u primjeni. Osvrnut ćemo se i na osnove SQL jezika.

1.1 Pojava baza podataka

Najranija poznata upotreba termina *baza podataka* potječe iz lipnja 1963. kada je Društvo za razvoj sustava uzelo pod pokroviteljstvo simpozij pod naslovom *Razvoj i upravljanje računalno centriranom bazom podataka*. Osnovne operacije koje su zahtijevane od baza podatake (popularno nazvane *CRUD*[1]) su stvaranje, čitanje, ažuriranje i brisanje, a nastoje se ostvariti sljedeći ciljevi:

Fizička nezavisnost podataka – Razdvaja se logička definicija baze od njezine stvarne fizičke građe. Promjene u fizičkoj građi baze, ne smiju zahtijevati promjene u postojećim aplikacijama koje rade s njom.

Logička nezavisnost podataka – Razdvaja se globalna logička definicija cijele baze podataka od lokalne logičke definicije za jednu aplikaciju. Znači, ako se logička definicija promijeni (na primjer, uvede se novi zapis ili veza), to neće zahtijevati promjene u postojećim aplikacijama. Lokalna logička definicija obično se svodi na izdvajanje samo nekih elemenata iz globalne definicije, uz neke jednostavne transformacije tih elemenata.

Fleksibilnost pristupa podacima – U starijim mrežnim i hijerarhijskim bazama, staze pristupanja podacima bile su unaprijed definirane, dakle korisnik je mogao pretraživati podatke jedino onim redoslijedom koji je bio predviđen u vrijeme projektiranja i implementiranja baze. Danas se zahtijeva da korisnik može slobodno prebirati po podacima, te po svom nahođenju uspostavljati veze među podacima.

[1] od engl. *Create*, *Read*, *Update*, *Delete*

Istovremeni pristup do podataka – Baza mora omogućiti istovremeno korištenje istih podataka većem broju korisnika. Pritom ti korisnici ne smiju ometati jedan drugoga, te svaki od njih treba imati dojam samostalnog rada s bazom.

Čuvanje integriteta – Nastoji se automatski sačuvati korektnost i konzistencija podataka, i to u situaciji kad postoje greške u aplikacijama, te konfliktne istrovremene aktivnosti korisnika.

Mogućnost oporavka nakon kvara – Mora postojati pouzdana zaštita baze u slučaju kvara hardvera ili grešaka u radu sistemskog softvera.

Zaštita od neovlaštenog korištenja – Mora postojati mogućnost ograničavanja prava korištenja baze, dakle za svakog se korisnika može definirati što smije, a što ne smije raditi s podacima.

Zadovoljavajuća brzina pristupa – Operacije s podacima moraju se odvijati dovoljno brzo, u skladu s potrebama određene aplikacije. Na brzinu pristupa može se utjecati odabirom pogodnih fizičkih struktura podataka, te izborom pogodnih algoritama za pretraživanje.

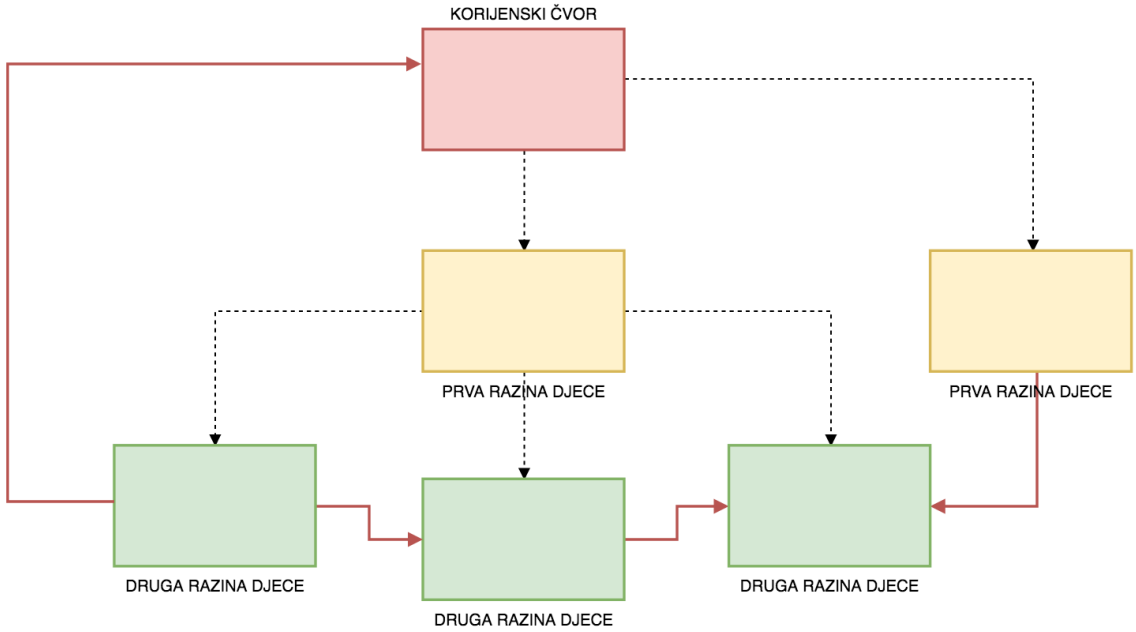
Mogućnost podešavanja i kontrole – Velika baza zahtijeva stalnu brigu: praćenje performansi, mijenjanje parametara u fizičkoj građi, rutinsko pohranjivanje rezervnih kopija podataka, reguliranje ovlaštenja korisnika. Također, svrha baze se vremenom mijenja, pa povremeno treba podesiti i logičku strukturu. Ovakvi poslovi moraju se obavljati centralizirano. Odgovorna osoba zove se **administrator** baze podataka.

Danas **baze podatake** najčešće definiramo kao skup međusobno povezanih podataka, pohranjenih u vanjskoj memoriji računala. Podaci su istovremeno dostupni mnogim korisnicima i aplikacijskim programima. Svakog od njih nazivamo **klijentom**. Osnova svake baze je njezin **model podataka** – skup pravila koja određuju kako mora izgledati logička struktura baze. On čini osnovu za koncipiranje, projektiranje i implementaciju baze. Ubacivanje, izmjene, brisanje i čitanje podataka obavljaju se posredstvom zajedničkog softvera – **sustava za upravljanje bazom podataka**, u nastavku SUBP. Njegova uloga je oblikovanje fizičkog prikaza baze, obavljanje svih operacija s podacima u ime klijenta, briga za sigurnost podataka, te automatizacija administrativnih poslova. Sustav, također, omogućava svojim klijentima da ne moraju poznavati detalje fizičkog prikaza podataka, već samo logičku strukturu baze.

Hijerarhijski i mrežni model

Prva dva SUBP-a razvijeni su u 1960-im godinama. Iako su imali mnogo sličnosti, razvijani su neovisno jedan o drugome, od različitih kompanija, na temelju različitih modela podataka. Glavni izazov, kod oba modela, bio je predviđanje svih mogućih veza između čvorova, te alokacija memorije za njihovo spremanje. Iako su brzo su izašli iz široke upotrebe, neki njihovi prinicipi su korišteni i u kasnije razvijenim modelima.

Na slici 1.1 prikazana je shema mrežnog i hijerarhijskog modela. Iscrtkane veze su ispravne u oba modela, dok su s crvenom bojom naznačene veze koje u hijerarhijskom modelu ne bi bile dopuštene.



Slika 1.1: Mrežni i hijerarhijski model

Prvi model, razvijen u to vrijeme, je od strane kompanije IBM, a nazivamo ga **hijerarhijski model**. Ovdje je baza predočena jednim stablom ili skupom stabala. Čvorovi su tipovi zapisa, a hijerarhijski odnos "roditelj–nasljednik" izražava veze među tipovima zapisa. Dozvoljeni su višestruki nasljednici, no roditelj svakog čvora mora biti jedinstven. Zbog toga se isti podaci ponekad moraju zapisati u više čvorova u stablu. Roditelj ima listu koja sadrži pokazivače na njegove nasljednike. Dostupne su samo operacije lociranje korijena stabla, pomicanje na sljedeći zapis, te pomicanje na sljedeće stablo. Za pristup određenom čvoru, potrebno je znati fizičke veze između zapisa, od korijena do njega, kako bismo ga pronašli u stablu. Niti jedan čvor, osim korijenskog, ne može postojati samostalno, tj. za svaki čvor osim korijena mora postojati roditeljski čvor. Zbog tako strogo definiranih veza i redundantnih podataka, otežano je ažuriranje i brisanje čvora. Prednost ovakve strukture je brzo dodavanje i dohvaćanje podataka, budući da su podaci hijerarhijski posloženi te je unaprijed poznat put do pojedinog podatka. Najpoznatije implementacije su IBM-ov Information Managment System s programskim jezikom DL 1 i Microsoft-ov

Windows Registry.

Začetnik drugog modela bio je Charles Bachman, na čijim je idejama CODASYL[2] razvio **mrežni model**. Bio je to prvi standard na području baza podataka. Kod takvog modela baza je predočena jednim usmjerenim grafom. Čvorovi su tipovi zapisa, a lukovi definiraju veze među tipovima zapisa. Ovaj model je dopustio i višestruke veze prema roditeljskim čvorovima te pristup čvoru direktno, a ne samo preko puta od korijenskog čvora. Moglo se, po potrebi, spremati i veze prema prethodnim ili sljedećim čvorovima. Iako je time postignuto poboljšanje u odnosu na hijerarhijski model podataka, mnogi problemi i dalje nisu bili riješeni. Mogle su se prikazivati samo relacije jedan-prema-mnogo i mnogo-prema-mnogo, a odnos jedan-prema-jedan se ostvarivao preko njih. Prednosti mrežnog modela su brzi pristup podacima, dobro upravljanje integritetom i podatkovna neovisnost. Nedostaci su složenost sustava, alokacija velike količine memorije i zahtjevna implementacija modela. Najpoznatije implementacije su Computer Associates-ov proizvod Integrated Database Management System te proizvodi tvrtke Digital Equipment Corporation.

Objektni model

Ideja o **objektno orijentiranom** modelu podataka pojavila se početkom 70-ih godina, približno u isto vrijeme kada i relacijski model, no prvi prototipovi sustava za upravljanje takvim bazama pojavili su se tek desetak godina kasnije. Glavni cilj ovog modela bio je približiti strukturu baze objektima (klasama, sučeljima, ...) kakvi se koriste u objektno orijentiranom programiranju. Također, pokušao se riješiti problem spremanja multimedijalnog sadržaja u baze. Navedimo osnovna svojstva bez kojih se model ne može proglasiti objektno orijentiranim.

Apstrakcija – Pojednostavljivanje složenih objekata iz realnog svijeta tako da se izdvoje bitne karakteristike tog objekta i njegovo ponašanje. Svaka vrsta objekata ima vlastite metode i atribute koji ga opisuju.

Enkapsulacija – Sučelje preko kojeg se pristupa objektu, odnosno, njegovim atributima i metodama, mora biti odvojeno od same implementacije objekta. Drugi objekti mogu koristiti sučelje bez poznavanja implementacije ponašanja. Implementacija se mora moći izmijeniti bez mijenjanja sučelja.

Modularnost – Formiranje smislenih cijelina, takozvanih *modula*, koji se mogu koristiti neovisno, te mogu komunicirati s drugim modulima.

Nasljeđivanje – Definiranje novih objekata na temelju postojećih. Novi objekt nasljeđuje sve metode i atribute prethodno definiranog objekta. Novi objekt može izmijeniti ponašanje nasljeđenih metoda i dodati nove atribute.

[2]od engl. *Committee on Data Systems Languages* – odbor osnovan 1959. godine u svrhu pokušaja definiranja univerzalnog programskog jezika

Polimorfizam – Svojstvo promjenjivosti oblika. Objekt može predstavljati više od jednog tipa podatka. Isti izraz može se koristiti za izvođenje različitih operacija, jer stvoritelj klase zna ispravan način izvršavanja izraza.

Iako se ovaj model pokazao puno pogodnijim od relacijskog za aplikacije koje zahtijevaju spremanje složenijih podataka s mnogo relacija, popularnost mu je počela rasti tek zadnjih godina. Očekivalo se bolje prihvaćanje ovog modela (koji se temelji na poznatoj semantici) od strane programera, kao i preuzimanje glavne uloge objektnog modela u daljnjem razvoju baza podataka, no to se nikada nije ostvarilo. Pretpostavlja se da su glavni uzroci toga nemogućnost integracije s relacijskim modelom i nedostatak standardiziranog jezika upita. Također, promjene objekata unutar aplikacije, zahtijevale su složenu sinkronizaciju baze sa stanjem u aplikaciji. Budući da svaka aplikacija ima definirane vlastite strukture, onemogućeno je korištenje iste baze od strane različitih aplikacija.

Unatoč tome, ovaj model doprinio je razvoju relacijskog, potaknuvši proširenje određenih relacijskih SUBP-ova kako bi podržali neke od gore navedenih objektnih značajki. Microsoft-ov SQL Server i PostgreSQL implementiraju stvaranje klasa, podržavaju nasljeđivanje ("*is a kind of*"), pointere i agregaciju ("*is a part of*"). Bez obzira na ova proširenja, ne radi se o objektnim modelima jer značajke nisu implementirane do kraja – svi podaci se i dalje spremaju u tablice, spremati se mogu samo osnovni tipovi podataka, kod nasljeđivanja se ne nasljeđuju ograničenja i slično. Detaljan pregled objektnih i objektno-relacijskih modela može se naći u [4].

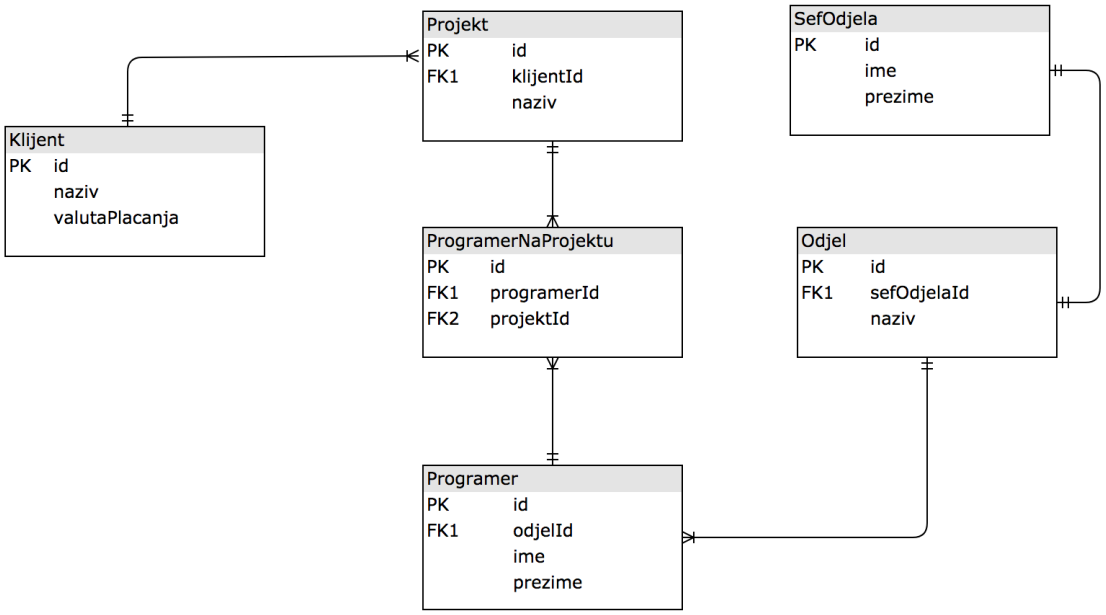
1.2 Relacijski model

Britanski računarni znanstvenik, Edgar Frank Codd 1970. godine u časopisu *Communications of the Association for Computing Machinery* objavljuje članak pod nazivom *A Relational Model of Data for Large Shared Data Banks*. U tom radu Codd iznosi mišljenje da bi se određene informacije morale moći pretraživati po njihovom sadržaju. Upravo se ta Coddova ideja najčešće smatra začetkom razvoja sustava za pretraživanje baza podataka. Relacijski model se temelji na čvrstoj teorijskoj pozadini – **relacijskoj algebri**, koju je Codd iznio u istom radu. Relacijska algebra definira skup formaliziranih operacija nad relacijama. Rezultat bilo koje operacije je relacija. Većina tih operacija su otprije dobro poznate u teoriji skupova – unija, razlika, Kartezijev produkt, projekcija, restrikcija, prirodno i theta spajanje, presjek, te dijeljenje.

Unatoč činjenici da je Codd bio u to vrijeme zaposlenik IBM-a, prva kompanija koja je implementriala njegov model podataka bila je Oracle. Iako su prve realizacije na računalu bile suviše spore, intenzivnim istraživanjem i razvojem računala, efikasnost relacijskih baza se poboljšala. Samo neki od najpoznatijih sustava za upravljanje relacijskim bazama osim Oracle-a, su SQL Server, Microsoft Access, PostgreSQL i MySQL.

Temelj ovog modela su **relacije** – dvodimenzionalne tablice gdje redci predstavljaju jedinke unosa, formalno – **entitete**, a stupci **atribute** koji opisuju te entitete. Svaka jedinka sprema se zajedno na disk. **Shema** je logička definicija baze, a predstavlja se tekstom ili dijagramom. Sastoji se od naziva relacija, konačnog skupa atributa i definicija njihovih tipova te popisa veza između podataka. Atributi identificiraju, kvalificiraju, klasificiraju, prebrojavaju ili izražavaju stanje entiteta.

Prilikom stvaranja tablice, relacijski model zahtijeva definiranje domene za svaki atibut. **Domena** atributa predstavlja skup vrijednosti koje određeni atribut može poprimiti i poželjno je da se ne mijenja. Svi entiteti u tablici moraju poštivati zadane domene svakog atributa. U nekim slučajevima za vrijednost atributa dozvoljavamo i specijalnu vrijednost ***NULL*** koja označava "nepoznat podatak". Obično postoji jedan istaknuti atribut koji jedinstveno opisuje svaki od postojećih entiteta i naziva se **ključ**. Ključ može biti i **složen**, tj. sastavljen od više atributa, pod uvjetom da se izbacivanjem bilo kojeg od tih atributa gubi svojstvo jedinstvenosti ključa. Ako imamo više mogućih kandidata za ključ, odabiremo jedan i njega proglašavamo **primarnim ključem**. On služi za neosporivu identifikaciju pojedinog entiteta i povezivanje s drugim tablicama.



Slika 1.2: Primjer sheme tablica i veza među njima

Entiteti se međusobno mogu povezivati raznim vezama, od kojih su najjednostavnije i najčešće: jedan–prema–jedan, jedan–prema–mnogo i mnogo–prema–mnogo. Veze **jedan–prema–jedan** i **jedan–prema–mnogo** najčešće se ostvaruju dodavanjem atributa koji sa-

drži primarni ključ povezanog entiteta. Tako nadodani primarni ključ drugog entiteta nazivamo **strani ključ**. Budući da veze **mnogo–prema–mnogo** nisu kvantitativno ograničene, njih nije moguće implementirati pomoću stranog ključa. Praksa kod takvih veza je stvaranje dodatne relacije, čiji su atributi, najčešće, samo primarni ključevi povezanih entiteta. Na ovaj način se jednostavno može ostvariti odnos "roditelj–nasljednik" među zapisima, kao kod već opisanih modela koji se temelje na stablima. Ako svaki primjerak entiteta nekog tipa mora sudjelovati u zadanoj vezi, tada kažemo da tip entiteta ima **obavezno članstvo** u toj vezi. Obveznost članstva je, najčešće, stvar odluke projektanta baze. Ukoliko je članstvo neobavezno, entitetima koji ne sudjeluju u vezi, u atribut stranog ključa postavljamo *NULL*-vrijednost.

Začetnik relacijskog modela, Codd, se dugi niz godina borio s brojnim proizvodačima SUBP-ova smatrajući da su krivo ili samo djelomično implementirali njegov podatkovni model, a nazivali su ga relacijskim. Njegova borba isplatila se 1985. godine kad su objavljena pravila koja SUBP mora poštivati kako bi se mogao nazivati relacijskim. U današnje vrijeme se, zbog praktičnih razloga, prilikom razvoja aplikacija s relacijskom bazom ponekad ne poštuju sva od ovih pravila. SUBP priznajemo kao relacijski ako poštuje barem šest od sljedećih trinaest pravila[3].

Nulto pravilo – Bilo koji sustav za upravljanje bazama podataka koji se smatra relacijskim, mora upravljati bazom podataka na potpuno relacijski način i isključivo relacijskom metodom.

Predstavljanje informacija – Podaci se reprezentiraju na jedinstven način kao vrijednosti u relacijama, tj. tablicama – jednostavno i dosljedno.

Obvezna dostupnost – Svaki podatak (vrijednost atributa) u tablici mora uvijek biti dostupan preko kombinacije imena tablica, vrijednosti primarnog ključa i imena atributa.

Tretiranje nepoznatih vrijednosti – Vrijednost *NULL* se tretira kao nepoznata vrijednost, neovisno o tipu podatka. Nepoznata vrijednost nije isto što i prazan znak ili broj nula ili varijabla.

Dinamički online katalog – Rječnik baze podataka, u kojem se nalaze informacije o samoj relacijskoj shemi tablica, mora biti pohranjen kao i svi ostali podaci u bazi. Nad tim podacima autorizirani korisnici mogu postavljati upite koristeći upitni jezik.

Pravilo sveobuhvatnog jezika – Relacijski sustav mora podržavati najmanje jedan jezik za komunikaciju s bazom podataka, čiji se izrazi, pomoću dobro definirane sintakse, mogu prezentirati kao nizovi znakova i koji mora podržavati modifikaciju i definiciju podataka te administriranje.

Ažuriranje pogleda – Sve poglede, koji se po relacijskoj teoriji mogu ažurirati, sustav mora moći ažurirati i u implementiranome modelu.

Ažuriranje skupova – Podaci iz relacijske baze podataka mogu biti preuzeti u skupovima podataka iz jedne ili više tablica. Ovo pravilo, također, zahtijeva da operacije

[3]Preuzeto iz [2].

umetanja, ažuriranja i brisanja moraju biti podržane za skupove podataka, a ne samo za jedan redak jedne tablice.

Fizička nezavisnost podataka – Aplikacije koje pristupaju podacima u relacijskoj bazi podataka ne smiju biti ovisne o metodi pristupa niti o promjenama u strukturi spremanja podataka.

Logička nezavisnost podataka – Logička nezavisnost znači da se odnosi između tablica mogu mijenjati, pri čemu se istovremeno ne utječe na funkcije aplikacija koje se spajaju na tablice. Promjena strukture baze podataka ne smije uzrokovati ponovnu izradu baze podataka ili aplikacije.

Nezavisnost integriteta – Ograničenja na integritet podataka ne smiju biti dio aplikacije, već moraju biti sadržana u katalozima baze.

Distribuirana nezavisnost – Bez obzira je li sustav trenutno distribuiran ili ne, jezik mora podržavati distribuciju i centralizaciju. Sve aplikacije moraju moći normalno nastaviti s radom kada se uvede distribuirana verzija centraliziranog sustava ili kada se distribuirana verzija centralizira.

Nenarušavanje integriteta – Ako sustav podržava jezik niskog nivoa, taj jezik ne smije biti korišten za zaobilaženje pravila o integritetu i ograničenjima podataka.

Integritet podataka

Jedan od velikih izazova višekorisničkih baza podataka je čuvanje **integriteta** podataka u bazi. Integritetom podataka osigurava se njihova suvislost i dosljednost kod uporabe, u svakom trenutku, za svakog korisnika. Ključni faktori čuvanja integriteta su ispravno uočena i provedena ograničenja.

Kako bi podaci u bazi bili korektni, potrebno zadati već spomenutu domenu za svaki atribut. Uobičajena podrška SUBP-a uključuje dvadesetak raznih tipova koji postavljaju ograničenja na tip vrijednosti atributa. Moguće je definirati i nove tipove na temelju postojećih. Domena se može definirati i pojedinačnim nabrajanjem svih vrijednosti koje atribut smije poprimiti. Budući da ta podrška nije velika, čuvanje integriteta domene može zahtijevati i kontrolu na aplikacijskoj razini. Ovakva kontrola se protivi načelu neovisnosti integriteta, no u praksi je ponekad neophodna.

Dodatno, u tablici ne smije postojati vrijednost stranog ključa za koju ne postoji ista vrijednost primarnog ključa u povezanoj tablici. Ovo pravilo naziva se **referencijalni integritet**, a čuva se definiranjem željenog ponašanja prilikom unosa, brisanja ili ažuriranja primarnog ključa. Nakon toga, SUBP će sam osigurati da se očekivano ponašanje zaista ispuni. Iako su ovakva ograničenja praktična jer smanjuju posao na aplikacijskoj razini, valja spomenuti i da svako ograničenje usporava daljnja ažuriranja i brisanja. Razlikujemo tri dopuštena ponašanja.

Ograničeno – Operacija se odbacuje ako primarni ključ postoji kao vrijednost stranog ključa u bilo kojoj tablici.

Kaskadno – Ažuriranje (ili brisanje) se primjenjuje i na sve entitete koji za vrijednost stranog ključa imaju taj primaran ključ.

Nuliranje – Primarni ključ se svugdje zamijeni s *NULL* vrijednosti, a tek onda se provede operacija nad primarnim ključem. Ukoliko je ovo željeno ponašanje, tada članstvo u vezi (koju primarni ključ opisuje) ne smije biti obavezno.

Možemo primijetiti da se kod kaskadnog ponašanja i nuliranja brisanje ili ažuriranje provodi u dva koraka. Operacija se prvo primjenjuje na svu djecu, a zatim i na roditelja. Prirodno je zapitati se što bi se dogodilo da se nakon primjene na djecu izgubi napajanje. Pretpostavimo da baza ostaje u stanju kakvom jest u tom trenutku. Tada bismo imali djecu koja naizgled nemaju roditelja, iako on još uvijek "živi" u nekoj drugoj tablici. Ukoliko ponovo pokušamo provesti istu operaciju, više ne bismo imali vezu prema toj djeci i ona bi ostala u tablici i nakon trajnog brisanja roditelja. Ovaj primjer ilustrira **nekonzistentnost** baze koju želimo izbjeći.

Srećom, relacijski SUBP-ovi imaju ugrađenu podršku koja omogućava eliminaciju takvih slučajeva, kao i konflikata kod višekorisničke upotrebe. Ta podrška implementirana je u obliku transakcija. Ugrubo, **transakcija** predstavlja niz operacija nad podacima koje se moraju ili izvršiti sve, ili nijedna operacija ne smije biti izvršena. U relacijskim bazama, zahtijeva se da transakcije budu oblikovane prema takozvanom *ACID* modelu. Taj model postavlja zahtjev za sljedeća 4 svojstva transakcija.

Atomiziranost (engl. *Atomicity*) – Transakcija se mora obaviti ili u potpunosti, ili nikako. Zato kažemo da je transakcija nedjeljiva logička cjelina operacija nad podacima. Sustavi koji osiguravaju atomarnost moraju biti spremni na sve moguće probleme, poput softverskih greški, hardverskih problema, kvarova na mreži, zakazivanja diskova ili rušenja cijelog sustava.

Konzistentnost (engl. *Consistency*) – Dok se transakcija izvršava, dopuštamo da je baza privremeno u nekonzistentnom stanju, no ono nije vidljivo korisnicima. Jednom kad su izvršene sve naredbe u nizu i transakcija je potvrđena ili je transakcija odlustala od izmjena, možemo biti sigurni da je baza ponovo u konzistentnom stanju. Dakle, transakcije prevode bazu iz jednog konzistentnog stanja u drugo.

Izoliranost (engl. *Isolation*) – Paralelno pokrenute transakcije ne utječu jedna na drugu tijekom izvršavanja. Izolacija omogućava da transakcija, nakon što je pokrenuta, ne dopušta čitanje međustanja podataka koje ona mijenja, dok se ona ili ne potvrdi, ili odustane od izmjena. To omogućava da svi korisnici vide točne i ažurirane podatke. Izoliranost se ostvaruje raznim metodama zaključavanja podataka, poput lokota, dvofaznog protokola ili vremenskim žigovima.

Trajnost (engl. *Durability*) – Promjene koje transakcija uzrokuje ostaju trajno zapisane u bazi podataka nakon potvrde transakcije. U slučaju pada sustava, nestanka električne

energije i sl., podatak će ostati nepromijenjen u bazi jer je transakcija već potvrđena. U slučaju da se dogodi ispad električne mreže za vrijeme trajanja transakcije, kod oporavka sustava poništiti će se dotadašnji rad transakcije i baza će ostati u konzistentnom stanju.

1.3 SQL jezik

U relacijskom modelu veze su samo implicitno naznačene time što se isti atribut pojavljuje u više relacija. Veza nije fizički pohranjena kao podatak u memoriji, već se dinamički uspostavlja za vrijeme rada s podacima, usporedbom vrijednosti atributa entiteta unutar raznih relacija. Zato jezik za rad s relacijskim bazama, osim jednostavnih operacija s jednom relacijom, mora omogućiti slobodno povezivanje podataka iz raznih relacija. Zahtijevamo da se, kao kriterij povezivanja, osim jednakosti vrijednosti atributa, mogu koristiti i složeniji usporedni kriteriji. Prednost ovakvog pristupa je da korisnik može, prilikom rada s bazom, uspostaviti nove veze, koje nisu bile predviđene u fazi modeliranja. To znači da se veza između dva entiteta svaki put mora iznova uspostavljati, što troši vrijeme, iako ona često ostaje nepromijenjena. Velikim je dijelom zbog toga, unatoč brojnim napretcima kroz godine, relacijski model ostao najsporiji od svih dosada navedenih.

Usprkos činjenici da dinamičke veze usporavaju pretraživanje i dohvaćanje podataka, upravo je njihova fleksibilnost privukla brojne projektante aplikacija na korištenje relacijskog modela. Zbog opisane dinamike, upotrebljivost relacijskog SUBP-a vrlo je zavisna o mogućnostima jezika kojim će se podaci analizirati. Poželjno je da taj jezik bude lako razumljiv krajnjim korisnicima i da se do rezultata može doći bez poznavanja fizičke strukture baze. Jedan od najpoznatijih takvih jezika danas je SQL[4].

Godine 1974. IBM Research Labaratories, u sklopu njihovog projekta "System R", objavljuje članak dvojice autora – Donalda D. Chamberlina i Raymonda F. Boycea. Cilj objave članka bio je približiti relacijske baze korisnicima i predstaviti jezik za manipuliranje podacima koji je IBM u to vrijeme razvijao, tada pod nazivom **SEQUEL**[5]. Kako bi korisnici prihvatili njihov jezik, trudili su se da on bude što sličniji svakodnevnom jeziku i što jednostavniji za korištenje. Pod time se podrazumijeva da je jezik **neproceduralan** – nije potrebno definirati korake izvođenja programa, nego samo što želimo dobiti kao rezultat.

Već krajem 70-ih godina IBM i Oracle su, gotovo istovremeno, predstavili prve komercijalne sustave koji su radili s bazom pomoću SEQUEL-a. Ostale kompanije, koje su do tada razvijale svoje vlastite jezike, nisu imale izbora nego prilagoditi se. Rezultat tih prilagodbi bila je pojava raznih verzija SEQUEL-a. Kako bi se omogućila prenosivost definicija baza podataka i aplikacijskih programa među različitim implementacijama, 1986.

[4] od engl. *Structured Query Language*

[5] od engl. *Structured English Query Language*

godine objavljen je prvi standard tog jezika, kojeg su morale poštivati sve softverske kuće. Zbog globalne popularnosti, danas na novijim verzijama tog standarda radi velik broj organizacija za standardizaciju. Standardizacija pruža mogućnost lake integracije ili prijelaz između različitih implementacija, što je jedna od prednosti relacijskih baza nad NoSQL sustavima.

Zanimljivo je da je prvotna kratica SEQUEL bila zaštitni znak zrakoplovne tvrtke *Hawker Siddeley* iz Ujedinjenog Kraljevstva, koja je zahtijevala da se to ime ne koristi. Tako je kratica promijenjena u SQL, bez kojeg je danas gotovo nemoguće zamisliti relacijske baze podataka.

SQL je primjer *integriranog jezika*, što otprilike znači da sve što poželimo učiniti s bazom ili podacima, možemo pomoću njega ostvariti. To uključuje sve radnje, od sigurnosnih postavki do pretraživanja. S obzirom na njihovu namjenu, naredbe možemo podijeliti u nekoliko skupina.

1. Naredbe za vršenje upita nad podacima. Uključuju zadavanje kriterija i pretragu podataka, grupiranje podataka i formatiranje ispisa.
   Primjeri: SELECT, JOIN, WHERE.
2. Naredbe za definiranje strukture tablica i stupaca te veza među entitetima. Važno je napomenuti da se strukture i veze definiraju na logičkoj razini, a ne fizičkoj.
   Primjeri: CREATE, ALTER, DROP.
3. Naredbe za manipulaciju podacima. Obuhvaćaju ubacivanje, ažuriranje i brisanje entiteta.
   Primjeri: UPDATE, INSERT, DELETE.
4. Naredbe za kreiranje korisnika, korisničkih uloga i dodjelu prava korisnicima.
   Primjeri: GRANT, REVOKE.
5. Naredbe za upravljanje tokom transakcija.
   Primjeri: ROLLBACK, COMMIT.

Osim gore navedenih osnovnih primjera, SQL ima ugrađene i vrlo korisne **agregatne funkcije**. Takve funkcije najčešće se primjenjuju na grupe podataka. Prilikom upita, one mogu izračunati projsek (AVG), zbroj (SUM), minimum (MIN) ili maksimum (MAX) i mnoge druge vrijednosti na danom skupu.

Upiti

Neka je baza definirana kao na slici 1.2. Pokazat ćemo nekoliko upita na tom primjeru, koristeći MySQL dijalekt jezika. Svaki osnovni upit mora sadržavati barem jednu SELECT naredbu, u kojoj imenujemo atribute (stupce tablice) koje želimo vidjeti u rezultatu. Dozvoljeno je koristiti ”*” kao naznaku da želimo vidjeti sve atribute. Nakon

nje, dolazi naredba FROM koja služi za navođenje imena jedne ili više tablica iz kojih želimo dobiti podatke. Moguće je da se isti atribut pojavljuje u više tablica, kao što je slučaj s atributom "naziv" u našem primjeru. Ako želimo izvršiti upit nad više tablica koje imaju atribut istog imena, potrebno je u SELECT naredbi specificirati i naziv tablice ispred imena atributa, npr. "klijent.naziv", inače sustav javlja grešku jer ne može jedinstveno identificirati atribut. Opcionalno, pomoću ključne riječi WHERE, možemo ograničiti prikaz rezultata tako da zahtijevamo zadovoljavanje određenih uvjeta. Uvjeti se međusobno povezuju logičkim operatorima AND ili OR, s tim da se naredba WHERE ne ponavlja. Uvjetom smatramo bilo koji izraz koji se može evaluirati u *boolean* vrijednost (istina ili laž). Unutar izraza možemo koristiti matematičke operacije nad skalarima i atributima. Nakon uvjeta mogu slijediti naredbe za određivanje željenog poredka – ORDER BY, ili LIMIT za ograničavanje konačnog broja rezultata.

Primjer 1.3.1. *Na ovom primjeru, pokazat ćemo da je SQL jezik doista vrlo blizak engleskom govornom jeziku, što ga čini vrlo intuitivnim. U hrvatskom prijevodu, bazi govorimo sljedeće: "Odaberi atribute klijentId i naziv iz tablice Projekt, gdje je atribut klijentId jednak 5, posloži ih abecedno uzlazno prema nazivu i pokaži mi prvih 5 rezultata".*

```
SELECT klijentId, naziv
FROM Projekt
WHERE klijentId = 5
ORDER BY Projekt.naziv ASC
LIMIT 5
```

Neproceduralnost u primjeru je jasna – nismo morali navesti kako će se podaci abecedno sortitrati ili ograničiti na 5, samo da to želimo dobiti kao rezultat. Kod ovakvih upita, baza interno radi nekoliko koraka. Stvara se nova, privremena tablica čiji su stupci atributi navedeni u SELECT dijelu, s kopiranim definicijama iz originalnih tablica. Kreće skeniranje tablice – što, bez dodatnih optimizacija, znači prolazak od prvog do posljednjeg retka. Ovakvo direktno pretraživanje cijele tablice nazivamo **full table scan**. Ukoliko redak zadovoljava uvjet, traženi atributi iz tog retka se kopiraju u privremenu tablicu. Ako je zadan kriterij sortiranja, kada su skenirani svi retci, baza sortira privremenu tablicu prema tom kriteriju. Zatim se rezultat ili dio rezultata, ovisno o tome jesmo li definirali LIMIT, ispisuje u konzolu. Rezultat primjera gore, bit će nova tablica s 5 redova, čiji su stupci naziv i klijentId. KlijentId će za sve retke biti 5, jer je to bio kriterij pretraživanja.

Primjer 1.3.2. *Zamislimo sada da nas zanima koji sve programeri rade na projektu "Diplomski rad". Iz slike 1.2 jasno je da moramo povezati tablicu ProgramerNaProjektu s odgovarajućim programerima i projektima. Srećom, definirali smo strane ključeve pa ćemo JOIN naredbom lako doći do traženog rezultata.*

```
SELECT Programer.*
FROM ProgramerNaProjektu
JOIN Projekt
    ON Projekt.id = ProgramerNaProjektu.projektId
JOIN Programer
    ON Programer.id = ProgramerNaProjektu.programerId
WHERE Projekt.naziv = "Diplomski rad"
ORDER BY Programer.ime
```

Optimizacija

Već i najjednostavniji upiti, nad velikim tablicama, mogu biti jako spori. Srećom, proces pretraživanja može se ubrzati dodavanjem **indeksa**. Indeks možemo definirati nad bilo kojim atributom ili skupom atributa iz tablice. Naredba za stvaranje indeksa ima jednostavnu sintaksu. Na primjer, ako želimo stvoriti indeks nad tablicom Programer sa slike 1.2 i atributom prezime, to postižemo naredbom

```
CREATE INDEX programer_prezime ON Programer(prezime)
```

Na taj način ustvari kažemo bazi da u dodatnu tablicu spremi vrijednosti prezimena za svaki redak iz tablice Programer i fizičku adresu tog retka. Ta dodatna tablica omogućuje bazi da zamijeni prolazak cijele originalne tablice s pretraživanjem manjeg postotka redaka. Iako je odluka o (ne)korištenju indeksa pri upitu na samoj bazi, najčešće će on biti korišten u situacijama kad zahtijevamo čitanje manje od 15 posto redaka, u suprotnom je jednostavnije napraviti *full table scan*.

Relacijske baze poznaju nekoliko vrsta indeksa, no mi ćemo se fokusirati na *uravnotežena stabla*, koja su najraširenija i najdulje u upotrebi. Karakteristike uravnoteženog stabla su potpuna balansiranost, sortiranje podataka po vrijednosti ključa i čuvanje maksimalno zadanog broja elemenata u jednom čvoru stabla. Operacije nad podacima takvih stabla obavljaju se u logaritamskom vremenu.

Vrijednosti atributa iz tablica se raspoređuju na približno jednake segmente, koje nazivamo **stranice**, i spremaju tako da je svaki **list** – čvor na zadnjoj razini stabla, jedna stranica. U čvorove na višim razinama spremaju se granične vrijednosti koje pripadaju između čvorova na sljedećoj razini. Kod pretrage, krećemo od korijenskog čvora i uspređujemo traženu vrijednost s vrijednosti u čvoru. Na temelju rezultata biramo za sljedeći čvor lijevo ili desno dijete. Isti postupak ponavljamo sve dok ne dođemo do listova. Segment spremljen u tom listu se tada skenira od prvog retka, dok ne nađemo traženu vrijednost. Tada SUBP prosljeđuje adresu spremljenu u tom listu čitaču diska koji dohvaća cijeli redak spremljen na toj adresi.

Iako indeksi mogu poboljšati performanse upita nad bazom, nepromišljena upotreba može dovesti do sasvim suprotnog efekta. Očito, raspoređivanje u stablo zahtijeva dodatne

operacije kod ažuriranja, brisanja i ubacivanja. Razlog tomu je što se kod svake takve operacije struktura stabla mora prilagoditi novom stanju. Također, spremanje dodatnih tablica zahtijeva dodatnu memoriju. Zato treba pažljivo birati atribute koji će se indeksirati i pritom ne treba pretjerivati. Ako je domena atributa malobrojna, ne možemo dobiti strukturu stabla koja bi ubrzala pretraživanje, jer bi svi entiteti s istom vrijednosti tog atributa završili u istom segmentu i opet bismo na zadnjoj razini morali pretraživati veliku količinu redaka. Indeksiranje se preporuča za one atribute koje su dobro distribuirani i koji se često koriste u upitima. Prirodni kandidati su primarni ključevi, čak većina SUBP-ova automatski dodaje indeks na taj atribut, i strani ključevi koji se često koriste kod spajanja. Važno je naglasiti da je kod malih tablica, ovaj proces sporiji nego *full table scan* pa indeksiranje samo troši memoriju, a ne pomaže u pretraživanju. Detalji o fizičkoj građi indeksa i njihovom korištenju mogu se pronaći u [5].

Sve moderne baze imaju ugrađen *optimizator* koji odlučuje o redoslijedu izvršavanja naredbi prema skupu pravila. Redoslijed izvršavanja ovisi o raspoloživim indeksima i količini redaka koje obuhvaća pojedini uvjet. Danas se koristi takozvani *cost based* model. Baza za svaku tablicu sprema statistike o veličini, kardinalnosti i distribuciji. Iako prikupljanje statistika troši vrijeme i memoriju, jako je bitno da se one spremaju redovito, kako bi odražavale stvarnu sliku podataka. Kasnije baza, upravo na temelju tih statistika, procjenjuje potrebnu količinu procesorske snage za izvršavanje svakog mogućeg redoslijeda upita te bira onaj koji će najmanje opteretiti sustav. Na primjer, kod spomenutih malih tablica, baza neće niti pokušavati koristiti indeks, jer iz statistika može zaključiti da je obično pretraživanje brže. Također, ako imamo dva izraza unutar WHERE uvjeta, prvo će se izvršiti onaj koji eliminira više redaka.

Poglavlje 2

NoSQL pokret

Vidjeli smo da relacijske baze, uz SQL jezik, doista pružaju širok spektar mogućnosti rukovanja podacima. SUBP se sam brine o bazi na fizičkoj razini i čuvanju integriteta u skladu s našom definicijom. ACID model transakcija osigurava najvišu razinu pouzdanosti. Veze se mogu uspostavljati dinamički. Podaci su oblikovani u pregledne tablice. Normalno je zapitati se treba li nam uopće ikakva druga tehnologija. U sljedećem poglavlju objasnit ćemo zašto su i kako, nakon duge ere vladavine relacijskih baza podataka, NoSQL modeli postali popularni.

2.1 Skalabilnost

Nevjerojatan rast popularnosti interneta potpuno je promijenio zahtjeve na baze podatka. Sve više korisnika web aplikacija i sve veća količina podataka drastično su srozali njihove performanse. Neko vrijeme se problem pokušavao riješiti dodavanjem dodatnih indeksa. No, kao što smo već spomenuli, prekomjerno indeksiranje često može biti kontraproduktivno. Indeksi rasterećuju procesor kod čitanja podataka, no usporavaju poslove ažuriranja i ubacivanja. Zbog toga se intenzivno radilo na **vertikalnom skaliranju** – koristili su se sve snažniji strojevi, ubrzavali procesori i dodavala memorija za pohranu podataka. Paralelno s time, količina podataka u bazama je iz dana u dan rasla i uvijek bila korak ispred tehnologije. Procesori jesu bili brži, no kod obrade tako velikih količina podataka, često su se pregrijavali. To je natjeralo inžinjere da se usmjere na **horizontalno skaliranje** – prebacivanje obrade s jednog na više procesora. Više procesora zajedno ima više mjesta za pohranu i može obraditi više podataka, ako rade paralelno.

Sustave koji se sastoje od više procesora, koji komuniciraju razmjenom poruka preko komunikacijske mreže, nazivamo **distribuiranim sustavima**. Svaka pojedina procesorska jednica, u kontekstu mreže, zove se **čvor**. Skupinu čvorova, povezanu fizičkim vezama, nazivamo ***rack***. Pojedini rack možemo zamišljati kao jedno, vrlo moćno, računalo s više

procesora. Više rack-ova, koji međusobno blisko surađuju te su povezani vrlo brzim vezama, nazivamo **klaster**. Jedna lokacija koja sadrži nekoliko povezanih klastera naziva se ***data center***.

Razlikujemo dva osnovna modela distribucije – replikaciju i fragmentaciju. Ovi modeli se međusobno ne isključuju i u nastavku ćemo podrazumijevati njihovo simultano korištenje. **Fragmentacija** znači razbijanje podataka na manje dijelove i zatim se ti dijelovi rasporede na različite čvorove, a **replikacija** znači kopiranje istih podataka na više čvorova. Postoje dvije vrste replikacije: gospodar–rob i peer–to–peer.

Kod **gospodar–rob** replikacije sve zahtjeve za ubacivanje i ažuriranje odrađuje jedan istaknuti čvor – gospodar. Ostali čvorovi, takozvani robovi, sinkroniziraju podatke s gospodarom i odrađuju zahtjeve čitanja. Ovakva raspodjela posla znatno ubrzava odgovaranje na zahtjeve čitanja. Čitanje se još više može ubrzati dodavanjem dodatnih robova. No, čitanjem s robova možemo dobiti i zastarjele podatke, jer treba vremena da se ažuriranje od gospodara propagira na sve robove. U sustav se obično ugrađuje neke metoda rješavanja ovakve situacije.

Prva poznata metoda naziva se **ispravak kod čitanja**. Ako se kod čitanja ustvrdi da je vrijednost zastarjela, ona se ažurira i vraća se nova vrijednost. Zastarjelost možemo utvrditi na temelju vremenske oznake zadnjeg ažuriranja. Proces ažuriranja usporava dohvaćanje rezultata, no zauzvrat klijent uvijek dobiva ispravnu vrijednost.

Drugu metodu zovemo metoda **odgođenog ispravka** ili slaba konzistentnost. Ako se kod čitanja ustvrdi da je vrijednost zastarjela, ona se svejedno vraća klijentu kao rezultat, no zatim se označava kao zastarjela. Čvor-gospodar će u nekom trenutku ažurirati sve vrijednosti koje su označene kao zastarijele. Ovakav pristup pruža brz odgovor, na uštrb moguće neažurnosti podataka. Očito, greška na nekom od robova neće ugroziti funkcioniranje sustava, a ukoliko dođe do greške u radu trenutnog gospodara, bilo koja njegova replika može preuzeti ulogu gospodara i sustav nastavlja funkcionirati.

U **peer–to–peer** replikaciji svi su čvorovi ravnopravni, što znači da svaki od njih može obraditi bilo koju vrstu zahtjeva. Time se rješava problem uskog grla kod ažuriranja i dodavanja vrijednosti. Ovakav sustav nastavlja raditi i nakon kvara nekog čvora. Njegova zaduženja se raspodjeljuju na preostale čvorove i funkcionalnost ostaje neugrožena. Ukoliko je potrebno poboljšati performanse, lako se mogu dodati novi, ravnopravni, čvorovi. Problem konflikta se javlja kada se zapis istovremeno ažurira na više čvorova. Postoji nekoliko provjerenih metoda za rješavanje ovakvog konflikta, no svaka od njih je dodatno opterećenje za sustav.

Jedna od spomenutih metoda je **testiranje**. Prije novog ažuriranja, provjerava se je li u međuvremenu taj podatak već ažuriran. Zatim se, redom, primjenjuju oba ažuriranja.

Poznata je i metoda **oznaka konflikta**. Kod ove metode spremaju se obje verzije ažuriranog zapisa i označava se da su u konfliktu. Zapisi u konfliktu će u nekom trenutku biti razriješeni, ili ručno ili automatski.

Posljedna metoda je metoda uspostavljanja **kvoruma**. Ova metoda zahtijeva da se ažuriranje potvrdi od većine čvorova. Što više čvorova potvrdi zapis kod ažuriranja, baza je konzistentnija pa je potrebno manje čvorova za potvrdu čitanja. Dakle, za bazu s puno čitanja, u interesu je definirati što veći broj čvorova za kvorum ažuriranja. Općenito, za konzistentnu bazu vrijedi nejednakost koja kaže da zbroj broja čvorova koji moraju potvrditi čitanje i broja čvorova koji moraju potvrditi zapisivanje, mora biti strogo veći od broja replikacija čvora.

Iako distribucija povećava složenost i otežava upravljanje bazom, te zahtijeva dodatnu brigu o sigurnosti, njezine mnogobrojne prednosti nadoknađuju te propuste. Kao prvo, distribucija je financijski puno isplatljivija od vertikalnog skaliranja sustava. Nadalje, distribuirani sustavi pružaju veću razinu pouzdanosti, jer i prilikom kvara nekog čvora, ostatak sustava može nastaviti funkcionirati. S obzirom da umrežena računala mogu raditi potpuno paralelno, moguće je obaviti više posla nego u jednakom vremenu na jednom računalu. Još jedna prednost kod globalnih aplikacija je činjenica da kopiranje na razne lokacije omogućuje posluživanje korisnika s bližih servera. Time se dodatno smanjuje vrijeme odgovora.

Relacijski sustavi teoretski podržavaju distribuciju, ali je teško izvediva. Jedan od ključnih razloga za to su operacije spajanja. Iako je dinamičnost veza između entiteta velika prednost tog modela, ukoliko su podaci fizički razdvojeni, spajanja postaju vrlo neefikasna. Sav kod koji se bavi distribucijom, mora biti napisan u aplikacijskom sloju, dok se NoSQL sustavi sami bave time. Također, profinjene razine dodjele prava korisnicima je jako teško kontrolirati i testirati kada su podaci, pa time i pogledi, spremljeni na različite servere.

2.2 Vanjski servisi

Za razliku od relacijskih sustava, koji implementiraju vlastite mehanizme keširanja, pretrage, zaključavanja i slično, gotovo sve NoSQL baze oslanjaju se na vanjske servise za izvršavanje ovih složenih procesa. Servisi su integrirani na za to opredijeljenim čvorovima i mogu raditi paralelno, što rasterećuje glavni dio baze. Budući da svi veći servisi pružaju jednostavno sučelje za korištenje, integracija je daleko jednostavnija od implementacije tih značajki. Također, servisi su dobro istestirani od strane svojih inženjera i ne moramo brinuti o njihovom razvoju. Velika prednost je i što je lako dodati novu kompomentu, tj. servis i proširiti funkcionalnost baze. Isto tako, ukoliko nam neka komponenta nije potrebna, ne moramo ju uključiti. Napomenimo i da je većina njih otvorenog koda, dakle besplatna za uporabu svima i dostupna za prilagodnu. Važno je da krajnji korisnik ne zamjećuje da radi sa zasebnim servisom, nego uvijek ima osjećaj da komunicira direktno s bazom. Više o razvoju ovih servisa može se pronaći u [6] ili na njihovim službenim stranicama.

MapReduce

Jedan od ključnih trenutaka NoSQL pokreta svakako je bila Googleova objava rada *MapReduce: Simplified Data Processing on Large Clusters* 2004. godine. U njemu su predstavili svoj model programiranja procesa koji se bave paralelnom obradom velikih količina podataka. U njihovom slučaju, izazov je bio indeksiranje sadržaja milijardu web stranica kako bi se moglo što efikasnije pretraživati, pokušavajući pritom koristiti postojeće procesorske jedinice. Koncept koji je rad predstavio je podjela obrade podataka na dvije faze – mapiranje i redukciju.

Prva faza, **mapiranje**, odgovorna je za izvlačenje, formatiranje i filtriranje podataka. Za efikasno izvršavanje ove faze, bitno je da su podaci izravno dostupni procesoru koji odrađuje map funkciju. Ona čita podatke iz baze, razlaže zahtjev u niz neovisnih funkcija koje se mogu izvršiti zasebno i raspoređuje te funkcije na različite procesore. Rezultat rada mapiranja svih procesora je serija ključ–vrijednost parova. Svi procesori koji mapiraju podatke moraju koristiti iste ključeve.

Ovako obrađeni podaci šalju se na drugu fazu – **redukciju**. Na temelju vrijednosti ključa, za svaki par posebno, bira se procesor za redukciju. Redukcija prima parove ključ–vrijednost kao ulaz i odrađuje tražene operacije. Tako dobiveni podaci mogu biti sortirani, grupirani ili sažeti. Obrađeni podaci se vraćaju procesoru koji ih je podijelio te ih on spaja, kako bi se dobio završni rezultat koji odgovara klijentovom zahtjevu. Primijetimo da je komunikacija s bazom još u prvoj fazi, kad se podaci dohvaćaju. Sve daljnje manipulacije obavljaju se na razini MapReduce servisa.

Iako Google nije prvi koji je koristio ovaj koncept, oni su ga proširili na rad tisuća i tisuća procesorskih jedinica. Izolacija podataka i nezavisan rad jednostavnih funkcija, na kojima se temelji funkcionalno programiranje, ključni su za efikasnu implementaciju ovakvog modela, koja nije jednostavna. Potrebno je osigurati da rezultat mapiranja ovisi samo o danom ulaznom nizu podataka. Također, mapiranje ne smije mijenjati postojeće podatke. Jedina dopuštena komunikacija između dvije faze je prosljeđivanje ključ–vrijednost parova, koji su rezultat mapiranja. Nažalost, Google nikada nije otvorio svoju implementaciju za javnost, no objavom rada potaknuli su i druge tvrtke da se okrenu istraživanju funkcionalnog programiranja i MapReduce modela pa danas postoji mnogo servisa koji se bave isključivo njime. Jedan od najpoznatijih i najčešće korištenih algoritama za MapReduce model je Apache-ov Hadoop.

Hadoop je radni okvir (engl. *framework*) otvorenog koda, dakle, besplatno dostupan svima za uporabu. Njegovi zadaci su biranje procesora za mapiranje, raspoređivanje ključ–vrijednost parova na procesore za redukciju i osiguravanje uspješnog izvršavanja zadatka, čak i kod greške u radu sustava. To su ujedno i najveći općeniti problemi map i reduce operacija pa na programeru aplikacije ostaju samo najjednostavniji zadaci. Hadoop se koristi u većini poznatih NoSQL sustava, izuzev MongoDB-a, koji ima vlastitu implementaciju ovog modela.

Memcache

Unatoč tome što su distribuirani sustavi daleko ubrzali i pojeftinili rad računala s velikim količinama podataka, problem veličine RAM-a nije bio posve riješen. Iako je zbroj memorije RAM-a bio i do nekoliko puta veći nego kod jednog superračunala, on je na ovakvim sustavima bio puno manje iskoristiv. Naime, velika prednost RAM-a je mogućnost keširanja rezultata čestih upita i držanje najkorištenijih podataka "pri ruci". Budući da u distribuiranom sustavu svaki procesor ima svoj vlastiti RAM, često se dešavalo da su podaci spremljeni u RAM-ovima različitih procesora jednaki. Također, moglo se dogoditi da čvor izvršava ispočetka složen upit ili dohvaća veliku količinu podataka iz baze, ne znajući da njegov susjedni čvor već ima rezultate spremne u svom RAM-u. Zbog svega navedenog, globalno popularne web stanice su gotovo na dnevnoj bazi morale dodavati nove čvorove u mrežu, kako bi osigurale najbolji korisnički doživljaj brzorastućem broju posjetitelja.

Inženjeri blog sustava LiveJournal nisu željeli tratiti mogućnosti svojih sustava i krenuli su tražiti rješenje. Prvo su uveli specifičnu notaciju upita kako bi mogli cijeli upit komprimirati u kratki hash string. Zatim su razvili sustav komunikacije između čvorova u mreži, koji su nazvali **Memcache**. Kada čvor dobije upit koji treba izvršiti, on prvo putem protokola pošalje spomenuti hash upita svim drugim čvorovma. Ako neki od čvorova koji primi hash, ima spremljen rezultat u RAM-u, on taj rezultat šalje nazad čvoru koji je poslao upit. Originalni čvor sada ima rezultate svog upita, bez da je uopće komunicirao s bazom. Na taj način je Memcache omogućio dijeljenje RAM-a između procesora u mreži i dodatno poboljšao mogućnosti distribuiranih sustava. Memcache sustav je, također, otvorenog koda.

Lucene i Solr

Lucene je moćan paket otvorenog koda koji se bavi indeksiranjem i potpunom pretragom tekstualnih dokumenata. Njegov autor je Doug Cutting, no od 2001. godine razvija ga *Apache Software Foundation*. Originalan paket je napisan u programskom jeziku Java, no danas postoje integracije za mnoge druge programske jezike. Kao posrednik za komunikaciju s Lucene-om, često se koristi Apache-ov web servis Solr. Prednost korištenja posrednika je korištenje standardnog HTTP protokola, kojeg Lucene sam po sebi ne podržava. Također, Solr obavlja poslove replikacije, keširanja, prevođenja dokument markup-a (ako postoji) i slično.

Zbog uske povezanosti, ova dva proizvoda se nerijetko zamijenjuju, no njihov odnos može se dobro opisati metaforom. Solr možemo zamišljati kao jako dobar auto, a Lucene kao snažan motor koji ga pokreće.

Lucene i Solr kao unos primaju tekstualne dokumente. Ti dokumenti mogu biti u gotovo bilo kakvom formatu – JSON, PDF, Word, HTML, itd. U našem kontekstu, baza će kao dokumente Solr-u slati vrijednosti koje želi indeksirati. Solr ih zatim prosljeđuje

Lucene-u, koji razbija dani dokument na riječi i gradi indekse za njih. Taj proces sličan je onome opisanom u odjeljku 1.3. Svaki indeks za riječ sadrži jedinstven ID riječi, broj dokumenata u kojima se riječ nalazi i pozicije riječi u tim dokumentima. Svi upiti koje baza dobiva, opet se prosljeđuju na Solr, koji zatim koristi Lucene za pretragu i vraća rezultate.

Stvaranje ovakvih indeksa veliki je teret kod dodavanja novih dokumenta i ažuriranja, no budući da taj teret nije direktno na bazi, nije problem prihvatiti ga. Solr omogućuje vrlo jednostavno i brzo pronalaženje tražene riječi u velikoj količini dokumenata. Ukoliko pretraga sadrži više riječi, traži se presjek skupa dokumenta koji sadrže tražene riječi. Lucene koristi različite algoritme kako bi rangirao rezultate. Na primjer, kod pretrage jedne riječi, najrelevantnijim rezultatom možemo smatrati onaj dokument koji sadrži najviše ponavljanja te riječi, pa će on biti vraćen na prvom mjestu.

2.3 Opuštena konzistentnost

Već smo spomenuli da distribucija podataka ozbiljno utječe na performanse relacijskih baza. Između ostaloga, razlog tomu je i što poštivanje strogih zahtjeva ACID modela predstavlja veliko opterećenje na bazu i kada su podaci na jednom mjestu. Iako je prednost transakcija najviša razina konzistetntnosti koje osiguravaju, one mogu zaključati veliku količinu resursa kod složenijih upita i time ih učiniti privremeno nedostupnim svim drugim korisnicima. To je danas nedopustivo, jer korisnici očekuju obradu podataka u "stvarnom vremenu". Odgovor baze na mnogobrojne upite mora biti trenutan. Zamislimo samo da na web tražilici, poput Google-a, koju istovremeno koriste milijuni korisnika, moramo čekati svoj red za pristup bazi. Bilo bi brže potražiti enciklopediju i iz nje pročitati odgovor. Dakle, takav sustav ne bi imao nikakvog smisla, niti bi ga korisnici imali volje koristiti.

BASE model

Na primjeru web tražilice jasno je da, na tako velikim sustavima s puno korisnika, ne smijemo dozvoliti zaključavanje resursa. Ipak, zaključavanje je jako korisno za čuvanje konzistentnosti, jer se njime, već na razini baze, izbjegava svaka mogućnost konflikta kod istovremene upotrebe. Za takav pristup kažemo da je *pesimističan*. Napredni sustavi, koji zahtijevaju visoku razinu dostupnosti, moraju koristiti *optimističniji* pristup. Umjesto izbjegavanja konflikata, takav pristup je orijentiran na njihovo otkrivanje i rješavanje. Neke primjere smo već naveli kada smo govorili o distribuciji – ispravak kod čitanja, oznake konflikta, itd. Iako se realizacija u tim primjerima razlikuje, svi se oni temelje na istom modelu, koji je jedan od osnova NoSQL sustava, a nazivamo ga **BASE model**. Objasnimo odakle potječe i kratica BASE.

Uglavnom dostupan, od engl. *Basic availability* – Baza mora biti dostupna što je više moguće vremena, pa čak i na privremenu štetu konzistentnosti.

Mekog stanja, od engl. *Soft state* – Sustavu su dozvoljene privremene netočnosti i mijenjanje podataka koji su trenutno u upotrebi.

U konačnici konzistentan, od engl. *Eventual consistency* – Nakon što se obrade svi zahtjevi, sustav će biti u konzistentnom stanju.

Sustavi koji se temelje na BASE modelu, u pravilu su puno brži i jednostavniji od onih na ACID-u. Razlog tomu je što se kod ovog modela, sustav ne mora brinuti o otključavanju i zaključavanju resursa, što je inače složen postupak. Ukoliko želimo podržati ACID model na ovakvom sustavu, taj postupak moramo implementirati na aplikacijskoj razini. Sljedeći primjer pokazuje koliko je dostupnost bitna za velike online servise.

Primjer 2.3.1. *Svi servisi tvrtke Amazon su 2013. godine, zbog tehničkih poteškoća, bili nedostupni 59 minuta. Forbes je trošak nedostupnosti tih servisa, prema podacima prihoda za 2012. godinu, procijenio na oko 66.000 dolara po minuti. Dakle, u nešto manje od sat vremena, tvrtka je izgubila oko 2 milijuna dolara profita. Sljedeći veliki pad sistema, dogodio se 2 godine kasnije. Iako je ovoga puta trajao samo 13 minuta, s obzirom da se Amazonov profit do tada popeo na vrtoglavih 203.577 dolara u minuti, tvrtka je opet izgubila oko 2 milijuna dolara.*

CAP teorem

Iz svega navedenog, možemo primijetiti da je jedan od glavnih izazova kod dizajniranja baze pronalazak zadovoljavajuće razine konzistentnosti podataka koju možemo ostvariti, bez srozavanja performansi baze. Izazov je još veći kad govorimo o distribuiranim sustavima, jer ne samo da je teže osigurati konzistentnost, nego zbog nepouzdanosti mreže, moramo računati da komunikacija s čvorovima može biti prekinuta. Upravo o tome govori takozvani **CAP teorem**, kojeg je Eric Allen Brewer predstavio krajem 90-ih godina. Teorem kaže da svaki distribuirani sustav može imati **najviše dva** od sljedeća tri poželjna svojstva.

Visoka razina konzistentnosti (engl. *High Consistency*) – Postoji samo jedna, ažurna verzija, koja je dostupna za čitanje svim klijentima. Za razliku od definicije konzistentnosti kod relacijskog modela, ovdje se ona odnosi na podudaranje kopija podataka koje su raspoređene po različitim čvorovima. Ne dopuštamo da klijenti koji čitaju podatke s *kopije 1*, dobiju različite rezultate od klijenata koji čitaju s *kopije 2*.

Visoka razina dostupnosti (engl. *High Availability*) – Baza mora dopustiti klijentima ažuriranje podataka u svakom trenutku, bez vremenske odgode. Greške u internoj komunikaciji između repliciranih podataka ne smiju sprječavati ažuriranje.

Tolerancija na podjelu (engl. *Partition tolerance*) – Sustav nastavlja odgovarati na zahtjeve klijenata i kada je onemogućena komunikacija između njegovih particija[1]. To-

[1] Particija je podjela skupa na disjunktne podskupove kojima je unija taj skup.

lerancija se očituje u mogućnosti neovisnog funkcioniranja svake particije. Formalnije, riječima Setha Gilberta i Nancy Lynch u [3], tolerancija na podjelu znači da je mreži dozvoljeno izgubiti proizvoljno mnogo poruka u komunikaciji između particija.

Primijetimo da nedistribuirani sustav nema toleranciju na podjelu. Budući da se on sastoji od samo jedne particije, kvar na toj particiji uzrokuje prestanak rada sustava. S druge strane, kod distribuiranih sustava, u praksi se najčešće upravo to svojstvo osigurava uvijek, a zatim se odabire još jedno od preostala dva. Odabir između konzistentnosti i dostupnosti je na arhitektu aplikacije i ovisi o tipu aplikacije koju razvija. Za neke je aplikacije najbitnije osigurati što veću dostupnost, dok je za druge konzistentnost važnija. Važno je napomenuti da se teorem odnosi na situacije kada nismo u mogućnosti komuniciranja sa svim čvorovima. U slučaju normalnog rada, od baze očekujemo da zadovoljava sva tri svojstva. U sljedećem primjeru objasnit ćemo što znači odabrati konzistentnost ispred dostupnosti i obrnuto.

Primjer 2.3.2. *Neka je baza distribuirana prema modelu gospodar–rob, gdje su robovi kopije gospodara. Pisanje je dozvoljeno samo preko gospodara, a čitati možemo i s kopija. U toku normalnog rada, kad klijent pošalje zahtjev za pisanje, on se prvo izvrši na gospodaru i zatim prosljeđuje svim kopijama. Pretpostavimo da smo, zbog greške u mreži, izgubili mogućnost komunikacije s nekom kopijom, ne nužno sa svima. Klijent šalje gospodaru zahtjev za ažuriranjem.*

Ako smo, u smislu CAP teorema, odabrali konzistentnost, zahtjev za pisanje mora biti odbijen. U suprotnom, kopija s kojom gospodar nije u mogućnosti komunicirati, imala bi spremljenu staru vrijednost, a kopije koje su funkcionalne bi imale novu. Dakle, baza ne bi bila konzistentna, jer bi neki čvorovi–kopije imali različite podatke.

S druge strane, ako smo odabrali dostupnost, ažuriranje će se obaviti na glavnom čvoru, iako se ne može propagirati na sve kopije. Baza će privremeno biti nekonzistentna, ali klijentov zahtjev za ažuriranjem će biti izvršen. Ovakvo stanje bit će ispravljeno kada se uspostavi ponovna komunikacija s privremeno nedostupnim čvorom. Do tada će neki klijenti potencijalno dobivati zastarjele podatke, no to moramo prihvatiti kako bismo osigurali visoku dostupnost.

2.4 Fleksibilnost

Veliki problem relacijskih baza je uniformnost podataka – svi entiteti unutar jedne relacije moraju imati iste atribute. Ovo može biti prednost, ukoliko su podaci s kojima radimo doista uvijek jednako strukturirani pa, već na razini baze, možemo osigurati poštivanje pravila. No, u stvarnom svijetu, iznimke su česte. Zamislimo da želimo za jednog programera zapisati neku vrstu opaske. Iako nam opaska treba samo za taj jedan entitet, prisiljeni smo dodati stupac ”opaska” nad cijelom tablicom. Osim što je spora, naredba ALTER TABLE

zaključat će sve podatke u toj i svim povezanim tablicama, dok se ne izvrši u cijelosti. Za to vrijeme, podaci neće biti dostupni niti za čitanje. Veliki servisi s milijunima korisnika si takvu sitauciju ne smiju dopustiti, jer nedostupnost znači financijski gubitak, kao što smo vidjeli u primjeru 2.3.1.

Također, relacijske baze podržavaju spremanje tekstualnih polja, datuma, binarnih nizova i brojeva, a sve je veća potreba za spremanjem cijelih dokumenata, multimedijalnog sadržaja, geografskih podataka i sličnih. Poznat je i takozvani *"fenomen otpora nepodudaranju"*. Aplikacije su proceduralne, stvaraju složene podatkovne strukture i obrađuju podatak po podatak. S druge stane, baze su deskriptivne, podatke čuvaju u tablicama i rade sa skupovima podataka. Premošćivanje ovih razlika zahtijeva veliko iskustvo i trud programera. Ponekad je nužan "debeli" aplikacijski sloj transformacija, kako bi se objekti iz aplikacije spremili kao redovi u bazi ili redovi iz baze prikazali u aplikaciji.

Prednost NoSQL baza je što one ne zahtijevaju shemu podataka pa se brzo i lako prilagođavaju novim poslovnim zahtjevima. To je ključno na današnjem, promjenjivom tržištu. Dodatno, one nude dosta širok izbor modela podataka, koji su pogodni za spremanje raznolikih struktura. Upravo razlike u strukturi onemogućavaju standardizaciju NoSQL sustava, što se često navodi kao njihov najveći problem. Ipak, smatramo da se, zbog prednosti koje daje odabir modela najbližeg skupu podataka aplikacije, isplati proći proces učenja i prilagodbe. U nastavku ćemo detaljno predstaviti neke modele i objasniti kada je njihova primjena poželjna, a kada bi bilo bolje okrenuti se nekom drugom modelu. Fokus će biti stavljen na korištenje baza, dok se detalji o fizičkoj implementaciji mogu proučiti u [10].

Poglavlje 3

Modeli podataka

3.1 Ključ–vrijednost baze

Dobro je poznato da je jedini ”jezik” koji računala razumiju strojni jezik, čija je abeceda binarna. To znači da računalo svaki podatak koji spremamo u njega pretvara u niz nula i jedinica. Ključ–vrijednost baze funkcioniraju na sličan način. Ovakve strukture najlakše je zamisliti kao tablice s dva atributa – ključ i vrijednost. Za razliku od relacijskog modela, ovdje vrijednosti koje spremamo nemaju definiranu domenu. One mogu biti gotovo bilo što – objekt, tekst dokumenta, slika, video, zvučnih zapis, itd. Prije spremanja, potrebno je danu vrijednost prevesti u skupinu binarnih podataka i dodijeliti joj zadani ključ. Ovako spremljene binarne podatke nazivamo **BLOB**, od engl. Binary Large Objects, u prijevodu *veliki binarni objekti*. Bazi nije poznat tip vrijednosti niti sadržaj koji ona predstavlja. Klijent sam mora brinuti o tom dijelu koda. Ključ je proizvoljan string – hash generiran iz vrijednosti, adresa web stranice, SQL upit ili slično, koji ne premašuje maksimalnu duljinu. Dopuštena duljina ovisi o implementaciji. Veličina BLOB-a je gotovo proizvoljna i ograničena jedino fizičkim prostorom za pohranu, koji baza ima na raspolaganju.

Operacije koje ove baze pružaju su ubacivanje novog para ključa i vrijednosti te ažuriranje, dohvaćanje i brisanje vrijednosti preko ključa. Dakle, pristup vrijednosti se obavlja isključivo preko njezinog ključa i nikakvi drugi upiti nisu podržani. Iz ovoga je očito da ključ svake vrijednosti mora biti jedinstven na razini cijele baze. Baza automatski indeksira sve vrijednosti prema ključevima, tako da ključ pokazuje direktno na vrijednost, što čini dohvaćanja jako brzima, bez obzira koliko je velika baza. Moguće definirati i *”rok trajanja”* ključa, nakon čijeg isteka se on i pripadajuća vrijednost brišu iz baze.

Ovaj model je najprikladniji za spremanje korisničkih podataka, odnosno *session*-a, jer je, u originalu, napravljen upravo za tu svrhu. Čest primjer uporabe je i implementacija rječnika. Razlog tomu su očite sličnosti tih struktura podataka. Kod rječnika su riječi ključevi, a njihov izgovor, definicije i sinonimi su vrijednosti. Riječi su posložene po

abecedi pa ih lako pretražujemo, na sličan način kao što baza pretražuje stablo indeksa. Zbog fleksibilnosti vrijednosti, ovaj model se često koristi i za spremanje multimedijalnog sadržaja, koji predstavljaja problem za standardni relacijski model. Također, zbog brzine čitanja, čak i neki relacijski sustavi koriste ovaj model kao pomoć pri ubzanju pretraživanja, tako da spremaju rezultate prošlih upita (engl. *query cache*).

Jedna od najpoznatijih implementacija ovog modela je sustav Riak. Riak je baza otvorenog koda napisana u programskom jeziku Erlang. Kao i mnoge druge implementacije, on zahtijeva grupiranja parova u **kante** (od engl. *bucket*). Kante omogućuju da se logički povezani podaci drže i fizički zajedno. Za stvaranje nove kante, potrebno je (na razini sučelja) samo zadati jedinstveno ime. Prednost ovakvog pristupa je što više ne mora postojati jedinstven ključ na razini cijele baze, nego samo na razini svake kante.

Kante možemo logički grupirati po **tipovima**. Ako bismo htjeli sačuvati podatke korisnika neke internet trgovine, mogli bismo stvoriti novu kantu, čije ime bi bilo *session ID* tog korisnika. U tu kantu bismo onda mogli spremati podatke, poput njegovog profila, trenutnih proizvoda u košarici i slično. Bilo bi korisno i ograničiti rok trajanja ključeva unutar košarice te kante na neki razuman rok od dodavanja. Na taj način, korisnikova košarica će se sama isprazniti nakon nekog vremena neaktivnosti. Još jedna novost je dodatno polje *links*, koje je Riak uveo radi lakšeg snalaženja u gomili podataka. Links predstavlja niz kategorija kojima trenutno pripada vrijednost, npr. ako spremamo podatke o nekom modelu tipkovnice, polje links bi moglo sadržavati oznake ”računalna oprema”, ”tipkovnica”.

Manipulacija podacima se odvija preko *REST API*-ja, korištenjem standardne HTTP metode. Ruta na koju se šalju zahtjevi kreira se prema kanti i ključu pod kojem se podatak nalazi ili dodaje. Moguće je dodati i ”Content-Type” header, kako bismo naznačili tip podatka koji šaljemo pa ga nismo dužni sami kodirati u traženi format.

Ukoliko šaljemo **PUT** zahtjev na nepostojeći ključ, ključ će biti kreiran i vrijednost spremljena. Ako ključ već postoji, ažurirat će se vrijednost spremljena pod njime ili dodati, tzv. *rođak* pod istim ključem, ako kanta to dozvoljava.

Metodu **GET** koristimo za dohvaćanje vrijednosti spremljenje pod danim ključem, a **DELETE** za njezino brisanje. Iako je ovo najizravniji način korištenja Riak sustava, direktni HTTP zahtjevi u praksi se, zapravo, rijetko koriste. Razlog tomu je postojanje provjerenih implementacija Riak klijenata za mnoge programske jezike, koji olakšavaju sve operacije.

Iako to nije uobičajeno za ključ–vrijednost baze, Riak omogućuje i pretragu vrijednosti korištenjem *Riak search* komponente. Riak search integriran je sa Solr sistemom za indeksiranje i pretragu. Budući da se radi o vanjskom sistemu, indeksiranje nije jednostavno kao u relacijskim sustavima. Kako bi se nestrukturirani podaci mogli indeksirati, Riak ima ugrađene *ekstraktore* za nekoliko tipova podataka – JSON, XML, čisti tekst, te njegove vlastite tipove podataka. Ukoliko je potrebno, moguće je definirati i vlastite ekstraktore. Nakon dodanog indeksa, pretraga se može vršiti prosljeđivanjem parametra *q* u HTTP zah-

tjevu. Za razliku od direktnog dohvaćanja po ključu, rezultat pretrage je jedan ili više Solr objekata, koje zatim, u aplikacijskom sloju, moramo pretvoriti u željeni izlaz.

Riak distribuira bazu fragmentiranjem i repliciranjem na klastere. Kod dodavanja novog para, na temelju vrijednosti ključa se određuje na koji čvor će se dodati. Konzistentnost se ostvaruje kvorumima. Broj čvorova koji moraju potvrditi pisanje (W) ili čitanje (R) moguće je ručno odabrati. Ako želimo osigurati konzistentnost (u smislu CAP teorema), moramo zahtijevati da barem više od pola čvorova potvrdi novu vrijednost. Također, možemo odabrati faktor repliciranja – broj kopija koji se stvara za čvor (N). Mana ovakvog pristupa je što, za operaciju pisanja u bazu, moramo imati barem W funkcionalnih čvorova, u suprotnom, baza nije u mogućnosti izvršiti zahtjev. Ove konstante možemo podesiti na razini tipa kanti, razini pojedinačne kante, te na razini svakog zahtjeva zasebno.

Jasno je da ključ–vrijednost baze imaju i neke mane. Model nije prikladan za podatke kod kojih nemamo na temelju čega odabrati jedinstveni ključ. Ne odgovaraju mu ni aplikacije koje zahtijevaju veze između podatka ili pretragu po vrijednostima. Sustavi nemaju implementirane transakcije niti operacije nad skupovima podataka, pa, ukoliko su nam potrebne ove značajke, moramo ih sami implementirati na aplikacijskoj razini.

Iako ga danas svrstavamo pod NoSQL modele, koji se smatraju modernim "izumom", ovaj model podataka implementiran je u sustavu upravljanja *Berkeley DB* još 1992. godine od strane istraživačke grupe sveučilišta Berkeley. Proizvod je tada ugrađen u njihov UNIX operativni sustav – *Berkeley Software Distribution*. Kasnije je sva prava na Berkeley DB otkupila tvrtka Oracle. On je i danas u širokoj uporabi, a Google i Amazon su samo neki od klijenata. Zanimljivo je da je Amazon svojevremeno za jedan zahtjev za dohvat početne stranice ulogiranog korisnika slao otprilike 800 zahtjeva prema bazi. Ipak, danas Amazon koristi vlastitu implementaciju ovog modela – DynamoDB. Upravo je DynamoDB prva baza podataka koja je uspjela značajno skrenuti pažnju s relacijskog modela. Njegovi glavni aduti su rad s ogromnom količinom podataka preko jednostavnog sučelja i pouzdana dostupnost 24 sata na dan, 7 dana u tjednu.

Implementacija

Na operativnom sustavu MacOS, najlakše je instalaciju Riak sustava provesti preko paketa *homebrew*, a detalji o ručnoj instalaciji te instalaciji na ostale operativne sustave mogu se pronaći na [9]. Zatim je nužno pokrenuti Riak servis.

```
# instalacija putem homebrew-a
$ brew install riak

# pokretanje Riak servisa
$ riak start
```

Već pri instalaciji dobivamo tip kante ”default”, a možemo dodati i proizvoljno mnogo vlastitih tipova. Ukoliko koristimo ”default”, to nije potrebno eksplicitno navoditi, dok vlastiti tipovi to zahtijevaju. Svaki korisnički definiran tip, nakon dodavanja, potrebno je i aktivirati. Ovaj tip administracije nije dostupan preko API sučelja, nego zahtijeva rad direktno na servisu.

```
# stvaranje tipa kanti osoba
$ riak-admin bucket-type create osobe
osobe created

# aktivacije tipa kanti osoba
$ riak-admin bucket-type activate osobe
osobe has been activated

# definiramo putanju do tipa osobe
$ export OSOBE="http://localhost:8098/types/osobe"
```

Kante, same po sebi, nije potrebno definirati, jer se one automatski stvaraju pri dodavanju prvog para ključa i vrijednosti u nepostojeću kantu. Sljedećom naredbom ćemo, putem API-ja, stvoriti kantu studenti tipa osobe i dodati u nju jedan par – ključ i vrijednost koja će biti JSON objekt. Nakon dodavanja, objekt možemo dohvatiti preko definiranog ključa.

```
$ curl -XPUT $OSOBE/buckets/studenti/keys/jaka \
     -H 'Content-Type: application/json' \
     -d '{
     "jmbag": "1191221523"
     "ime":"Jaka Gacic",
     "godine":26,
     "zavrsila_preddiplomski":true
    }'

$ curl $OSOBE/buckets/studenti/keys/jaka
{
    "jmbag": "1191221523"
     "ime":"Jaka Gacic",
     "godine":26,
     "zavrsila_preddiplomski":true
}
```

Ključ nije nužno zadati i najčešće se njegovo generiranje prepušta bazi. Razlog tomu je, između ostaloga, činjenica da je ugrađeno ponašanje kanti da dozvoljava stvaranje rođaka.

To znači da, ukoliko pošaljemo više PUT zahtjeva u istu kantu, na isti ključ, baza pod tim ključem svaki put stvara novi zapis, kojem interno dodaje i automatski generirani ključ. Posljedica toga je da niti jedan od tih zapisa više ne možemo dohvatiti direktno preko ključa kojeg smo odabrali, nego dobivamo odgovor sličan ovome:

```
$ curl $OSOBE/buckets/studenti/keys/jaka
Siblings:
3PcbDvtoez7iuKfaiX86qn
5GF84kcCvp7nGLFUXghEC2
```

Ovom problemu možemo doskočiti prosljeđivanjem parametra ”vtag” s vrijednošću izgeneriranog ključa za željenog rođaka ili dodavanjem headera, kojim dopuštamo vraćanje višestrukih objekata. Ipak, u praksi se najčešće ovo ponašanje isključuje, kako bismo mogli ažurirati vrijednosti putem PUT zahtjeva, umjesto dodavanja rođaka.

```
# izmjena svojstava kante
$ curl -XPUT $OSOBE/buckets/studenti/props \
     -H'content-type:application/json' \
     -d'{"props":{"allow_mult":false}}'

# preko PUT zahtjeva možemo ažurirati vrijednost pod ključem "jaka"
$ curl -XPUT $OSOBE/buckets/studenti/keys/jaka \
     -H 'Content-Type: application/json' \
     -d '{
     "jmbag": "1191221523"
     "ime":"Jaka Gacic",
     "godine":26,
     "zavrsila_preddiplomski":true,
     "zavrsila_diplomski":true
    }'

# dohvaćanje ažuriranog objekta
# napomena: posljednji zahtjev je izbrisao sve prije dodane rođake
$ curl $OSOBE/buckets/studenti/keys/jaka
{
    "jmbag": "1191221523"
     "ime_s":"Jaka Gacic",
     "godine":26,
     "zavrsila_preddiplomski":true,
     "zavrsila_diplomski":true
}
```

U sljedećem primjeru šaljemo POST zahtjev bez eksplicitno zadanog ključa i u odgovoru dobivamo njegovu točnu lokaciju.

```
# dodavanje bez eksplicitno zadanog ključa
$ curl -v -XPOST $OSOBE/buckets/studenti/keys/ \
     -H 'Content-Type: application/json' \
     -d '{
     "jmbag": "1191221000"
     "ime":"Pero Peric",
     "godine":21,
     "zavrsio_preddiplomski":false
    }'

HTTP/1.1 201 Created
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.10.9 (cafe not found)
Location: /types/osobe/buckets/studenti/keys/BlzRr5wFe6kAT3lnQf5jFVld04K
```

Nadalje, pokazujemo izvršavanje još nekih korisnih operacija.

```
# brisanje elementa pod ključem
$ curl -XDELETE $OSOBE/buckets/studenti/keys/BlzRr5wFe6kAT3lnQf5jFVld04K

# izlistavanje svih ključeva u kanti studenti
$ curl $OSOBE/buckets/studenti/keys?keys=true
{
"keys": [
    "3PcbDvtoez7iuKfaiX86qn",
    "5GF84kcCvp7nGLFUXghEC2",
    ]
}

# izlistavanje svih kanti tipa osoba
$ curl $OSOBE/buckets?buckets=true
{
"buckets": [
    "studenti"
    ]
}
```

3.2 Graf baze

Graf baze pohranjuju entitete i njihove međusobne poveznice. Entiteti su čvorovi koji imaju svojstva i možemo ih mapirati na instance objekta u aplikaciji. Čvorovi grafa najčešće predstavljaju uobičajene objekte, poput ljudi, organizacija, web stranica i slično. Poveznice među objektima predstavljaju se lukovima koji povezuju čvorove koji sudjeluju u vezi, slično kao kod mrežnog i hijerarhijskog modela. Bitna razlika u odnosu na te modele je što u graf bazama, poveznice također mogu imati vlastita svojstva. Osim toga, svaka poveznica ima određen smjer, koji je bitan za pronalazak obrazaca u podacima i njhovu interpretaciju.

Graf baze su optimizirane za efikasno spremanje čvorova i veza te dozovljavaju upite nad grafom. Budući da graf baze nemaju shemu, dodavanje novih čvorova ili veza ne zahtijeva promjene na modelu podataka. Poveznice se ne uspostavljaju dinamički, kao kod relacijskog modela, nego su fizički pohranjene kao i sami objekti. Ovdje opet uočavamo sličnost s najranijim modelima podataka. Svojstva čvorova i poveznica mogu se indeksirati. Ova vrsta baza posebno je važna za aplikacije koje zahtijevaju uspostavu i analizu složenih veza među objektima. Očiti kandidati za klijente ovakvih baza su društvene mreže, jer njihova struktura odgovara strukturi grafa. Njihove implementacije moraju brzo obrađivati složene poveznice između korisnika i pronalaziti obrasce unutar mreže.

Upiti nad graf bazama su, u stvari, obilasci čvorova grafa. Samo neki od primjera upita koje graf baze efikasno rješavaju su: pronalazak najkraćeg puta između dva čvora, otkrivanje svih čvorova koji u svojoj blizini imaju čvorove koji zadovoljavaju određeni uvjet, koliko su slične okoline dvaju čvorova, kolika je prosječna povezanost između nekoliko danih čvorova, itd. Odgovor na ovakve upite je opet graf struktura. Vidimo da se oni ne odnose toliko na same objekte koji su spremljeni u čvorovima, nego više na veze koje ih spajaju. Ono što tim bazama omogućava efikasno izvršavanje ovako složenih upita je činjenica da su čvorovi, u memorijskom smislu, mali pa računalo može držati cijeli graf u RAM-u. Kad se graf jednom dohvati s diska, više nije potrebno komunicirati s tim sporim dijelom memorije. Ipak, ovo zahtijeva nešto moćnija računala nego što je uobičajno kod NoSQL sustava.

Osim kod društvenih mreža, graf baze su korisne za pronalaženje uzoraka i u mnogim drugim sustavima. Tako telefonska centrala može graditi mrežu načinjenu od odlaznih i dolaznih poziva, mailing servis pratiti komunikaciju koja se odvija njegovim protokolom, banke i kartične kuće analizom uzoraka mogu otkriti prijevare i pranje novca. Ovaj model jako je popularan i među obavještajnim agencijama, jer im omogućuje brzo otkrivanje poveznica među ljudima koji su im od interesa. Također, često se koristi za servise za preporuku sličnih proizvoda ili prijedloge temeljene na prethodnim odabirima.

Za razliku od većine ostalih NoSQL sustava, graf baze poštuju principe ACID modela transakcija. Kod tih transakcija, najvažnije je ne dopustiti poveznice koje imaju samo

jedan čvor. Početni i završni čvor uvijek moraju postojati, a čvor se može obrisati tek kada su obrisane sve njegove poveznice. Iako se operacije čitanja ne moraju izvršavati kroz transakcije, one su obavezne kod dodavanja ili ažuriranja čvora. Ipak, za razliku od relacijskih SUBP-ova, koji automatski započinju i završavaju ovakve transakcije, ovdje se to mora napraviti na razini aplikacijskog sloja te će sustav javiti grešku ako programer na to zaboravi.

Zbog važnosti poveznica između čvorova, graf baze moraju držati blisko povezane čvorove zajedno. Prednost ovakvog pristupa je gore spomenuta efikasnost upita, no ako ju želimo održati, moramo se odreći fragmentacije. Naime, ukoliko su fragmenti raspoređeni na različite klastere, koji su potencijalno na različitim lokacijama, morali bismo osigurati da veze ne prelaze iz jednog klastera u drugi, kako bi se brzo mogao dohvatiti cijeli graf. Ako bismo i uspjeli u tom naumu u nekom trenutku, takva situacija bila bi brzo narušena dodavanjem novih veza. Zato se kod ovakvih baza najčešće koristi samo jedno računalo.

Ukoliko je distribucija nužna, jer jedno računalo ne može efikasno rješavati probleme, koristi se gospodar–rob replikacija koja je opisana u poglavlju 2. Razlika u odnosu na klasičan model gospodar–rob je što je ovdje čvorovima robovima dopušteno i pisanje, no oni su ga dužni odmah prijaviti čvoru gospodaru. Čvor gospodar zatim odlučuje hoće li ga prihvatiti i, ako je prihvatio, prosljeđuje novu vrijednost svim drugim robovima. Rob ne smije prihvatiti novu vrijednost od nikoga drugoga osim od gospodara. Budući da se ovaj proces odvija u transakciji, osigurana je visoka konzistentnost. No, budući da su dozvoljena čitanja s čvorova i za vrijeme trajanja transakcije, također imamo i visoku dostupnost.

Drugi način za postizanje efikasne distribucije je distribuirati sustav na aplikacijskoj razini. Ovakva distribucija ostvariva je ukoliko bazu možemo logički podijeliti na cjeline koje nemaju potrebu međusobno komunicirati. Takav slučaj može se javiti ako, na primjer, imamo aplikaciju distribuiranu u različitim državama i podatke možemo odvojiti po državama. Samo neke od prednosti distribucije jedne baze, u odnosu na izradu posebne baze za svaku državu, su lakše održavanje i generiranje izvještaja na globalnoj razini. Ipak, ovakva distribucija je složenija opcija od gospodar–rob, jer u ovom slučaju arhitekt aplikacije mora sam osigurati da se svaki zahtjev šalje na odgovarajući klaster.

Obrada prirodnog jezika i danas je veliki izazov za računala. Srećom, graf baze su korisne za povezivanje i pretraživanje velike količine tekstualnih dokumenata. Korištenjem procesa obrade prirodnog jezika, takozvanog prepoznavanja entiteta, servis može skeniranjem danih dokumenata relativno brzo identificirati ključne pojmove u dokumentu. Zatim se, na temelju rezultata, mogu izgraditi poveznice među dokumentima koji spominju iste pojmove. Nad ovakvom strukturom spremljenom u graf bazi lako ćemo izvesti i komplicirane pretrage. Prepoznavanje entiteta, uz pomoć dodatnih alata, možemo iskoristiti i za izvlačenje cijelih tvrdnji iz danih dokumenata. Tako detektirane tvrdnje, pogodno je spremiti u graf baze, koje, zbog dopuštenih svojstava poveznica, mogu pružiti i kontekst tvrdnjama te tako olakšati aplikaciji razumijevanje novih dokumenata.

Sljedeća važna primjena graf baza je *linked open data*, odnosno, LOD – skup alata koji omogućuju integraciju podataka iz više javno dostupnih, posve odvojenih graf baza, u svrhu stvaranja novih aplikacija i izvještaja. Te baze najčešće su skupovi podataka različitih organizacija i nisu svjesne međusobnog postojanja. LOD integracije iznimno su korisni alati za istraživanje tržišta, analizu trendova, razvoj analize sentimenta te stvaranje novih informacijskih servisa. Važno je da ti novi servisi opet poštuju pravila LOD struktura. Na taj način stvara se velika mreža javno dostupnih, na neki način strukturiranih, općenitih podataka, čija analiza može dati smjernice poslovnog razvoja, marketing kampanje i slično.

Istaknimo još da su graf baze ipak neprikladne za uporabu u aplikacijama koje zahtijevaju puno ažuriranja većeg broja čvorova odjednom. Budući da se, čak i kod fragmentacije, sva ažuriranja obavljaju preko jednog procesora, nije ga teško preopteretiti.

Implementacija

Jedna od najpopularnijih implementacija graf baze je **Neo4j**. Uz brojne druge, najpoznatiji klijent ove baze je online trgovina Ebay. Neo4j opet instaliramo pomoću *homebrew*-a, dok se detaljnije upute za instalaciju mogu naći na [8]. Moramo napomenuti da Neo4j zahtijeva instalaciju programskog jezika Java. Kako bismo mogli rukovati bazom, potrebno je prvo pokrenuti instancu Neo4j servisa. Za razliku od dosadašnjih modela, jedna instanca servisa podržava samo jednu bazu, odnosno, graf. U primjerima ćemo koristiti ugrađeni graf imena *graph.db*.

```
# instalacija
$ brew cask install java
$ brew install Neo4j

# pokretanje servisa
$ neo4j start
```

Odmah po pokretanju servisa, na portu 7474 dostupno je pregledno korisničko sučelje koje omogućuje rukovanje bazom i postavkama sustava. Nakon otvaranja adrese (npr. http://localhost:7474/browser/) u pregledniku, prijava se prvi puta mora napraviti koristeći "neo4j" za korisničko ime i lozinku. Korisnik tada mora definirati novu lozinku koju će koristiti ubuduće.

Neo4j pruža vlastiti jezik upita – **Cypher**. Višestruke naredbe u Cypher-u odvajaju se prelaskom u novi red. U korisničkom sučelju ćemo, kroz prozor za upite, najprije dodati nekoliko čvorova i osnovnih veza. Oznaka čvora predstavlja logički tip čvora i zadaje se iza znaka ":", kojemu može prethoditi privremeno ime čvora. Privremeno ime čvora je, u stvari, referenca na taj čvor, koju definiramo ako imamo potrebu koristiti taj čvor na više

mjesta unutar upita, kao što je u primjeru slučaj s čvorovima Fakultetima. Ona postoji samo unutar trenutnog upita i nakon njegovog izvršavanja više nema nikakvog značaja. Čvorovima Studentima pridružujemo svojstva ”ime” i ”godina”, Fakultetima ”naziv”, a vezama STUDIRA ”godina”.

```
CREATE (pmf:Fakultet {naziv:"PMF-MO"})
CREATE (vern:Fakultet {naziv:"VERN"})

CREATE (:Student {ime:"Jaka Gacic", godina: 5})
    -[:STUDIRA]->(pmf)
CREATE (:Student {ime:"Pero Peric"})
    -[:STUDIRA {godina: 5}]->(vern)
CREATE (:Student {ime:"Ivo Ivic"})
    -[:STUDIRA {godina: 5}]->(pmf)
CREATE (:Student {ime:"Ana Anic"})
    -[:STUDIRA {godina: 2}]->(vern)
CREATE (:Student {ime:"Marko Markovic"})
    -[:STUDIRA {godina: 3}]->(vern)
CREATE (:Student {ime:"Josip Josipic"})
    -[:STUDIRA {godina: 2}]->(pmf)
CREATE (:Student {ime:"Jelena Jelenic"})
    -[:STUDIRA {godina: 4}]->(pmf)

> Added 8 labels, created 8 nodes, set 15 properties,
> created 7 relationships, completed after 5 ms.

# ažuriranje svojstva
MATCH (u:Student {ime:"Jaka Gacic"})
SET u.ime = "Jaka Gacic Pojatina"
RETURN u

# pokušaj dohvaćanja sa starim imenom
MATCH (u:Student {ime:"Jaka Gacic"})
RETURN u

# što je ekvivalentno s
MATCH (u:Student)
WHERE u.ime="Jaka Gacic Pojatina"
RETURN u
```

```
# rezultat
> (no changes, no records)
> Completed after 1 ms.

# dodavanje novog svojstva
MATCH (u:Student {ime:"Jaka Gacic Pojatina"})
SET u.tema_diplomskog = "NoSQL baze podataka"
RETURN u
```

Ključnom riječi MATCH ističemo čvorove ili veze koje želimo koristiti. Moguće je odrediti i smjer poveznica koje želimo slijediti – ulazne, izlazne ili oboje. U samom MATCH dijelu ili u zasebnoj naredbi WHERE možemo zadati uvjete na svojstva poveznica ili čvorova. Naredbom RETURN definiramo što želimo vidjeti kao izlaz. Također, možemo zadati i parametre ORDER BY i LIMIT, koji imaju isto značenje kao i u SQL upitima. Rezultat svakog upita koji sadrži ključnu riječ RETURN je opet graf struktura.

Čest primjer efikasnosti obilaska grafa je traženje "prijatelja prijatelja". U relacijskom modelu, tip veze "prijateljstvo" bi vjerojatno bio realiziran preko pomoćne tablice s glavnim atributima prviPrijateljID i drugiPrijateljID, u kojima bi bili pohranjeni strani ključevi. Vrlo vjerojatno bi arhitekt indeksirao te atribute pa bismo lako mogli saznati tko su sve prijatelji nekoj osobi. Međutim, već pregled sljedeće razine prijateljstva – tko su prijatelji svih prijatelja neke osobe, predstavlja ogroman problem. Ako osoba koju tražimo ima 100 prijatelja i svaki njezin prijatelj isto toliko, trebamo pregledati 10.000 redaka. U relacijskom modelu, ovakav zahtjev bi se mogao riješiti samo uz složen upit i mnogo truda oko optimizacije, a i tada bi njegova efikasnost bila upitna. S druge strane, za graf bazu, ovo bi bio jednostavan obilazak čvorova. Velika prednost graf baza je što one mogu eliminirati nepotrebne čvorove pri samom obilasku. U nastavku donosimo primjer upita koji rješava problem traženja Jakinih prijatelja preko prijatelja, isključujući izravne prijatelje. Prvo ćemo povezati nekoliko studenata vezom prijateljstva i zatim obilaziti graf preko novododanih poveznica. Primijetimo da svaki smjer prijateljstva dodajemo zasebno, jer u CREATE naredbi nije dopušteno koristiti obostrane veze, kao kod pretrage.

```
# dodajemo veze prijateljstva između postojećih čvorova
MATCH (u:Student {ime:"Marko Markovic"}), (r:Student {ime:"Ana Anic"})
CREATE (u)-[:PRIJATELJ]->(r)
CREATE (r)-[:PRIJATELJ]->(u)

MATCH (u:Student {ime:"Marko Markovic"}), (r:Student {ime:"Jaka Gacic"})
CREATE (u)-[:PRIJATELJ]->(r)
CREATE (r)-[:PRIJATELJ]->(u)
```

```
# tražimo Jakine prijatelje preko prijatelja
MATCH
  (pocetni:Student)
       -[:PRIJATELJ]-
  (prijatelj:Student)
       -[:PRIJATELJ]-
  (prijatelj_preko_prijatelja:Student)
WHERE
       pocetni.ime = "Jaka Gacic Pojatina"
  AND NOT
       (pocetni)-[:PRIJATELJ]-(prijatelj_preko_prijatelja)
  AND NOT
   pocetni = prijatelj_preko_prijatelja
RETURN
   prijatelj_preko_prijatelja

> vraća čvor s "Ana Anic"
```

Kao veliku prednost graf baza istaknuli smo mogućnost dodavanja svojstava poveznicama. Rezultate možemo, pomoću tih svojstava, filtrirati na isti način kao što smo to radili sa svojstvima čvorova. U sljedećem primjeru tražimo studente svih fakulteta koji još nisu na 5. godini studija, tako što dodajemo uvjet na svojstvo "godine" veze STUDIRA. Zatim pokazujemo kako možemo grupno ažurirati svojstva veza. Pokazat ćemo i kako možemo dodati ili maknuti indeks.

```
# prva 3 studenta koji nisu apsolventi, poredani abecedno
MATCH
  (student:Student)
       -[studira:STUDIRA]-
  (fakultet:Fakultet)
WHERE
       studira.godina < 5
RETURN
   student, fakultet
ORDER BY student.ime
LIMIT 3

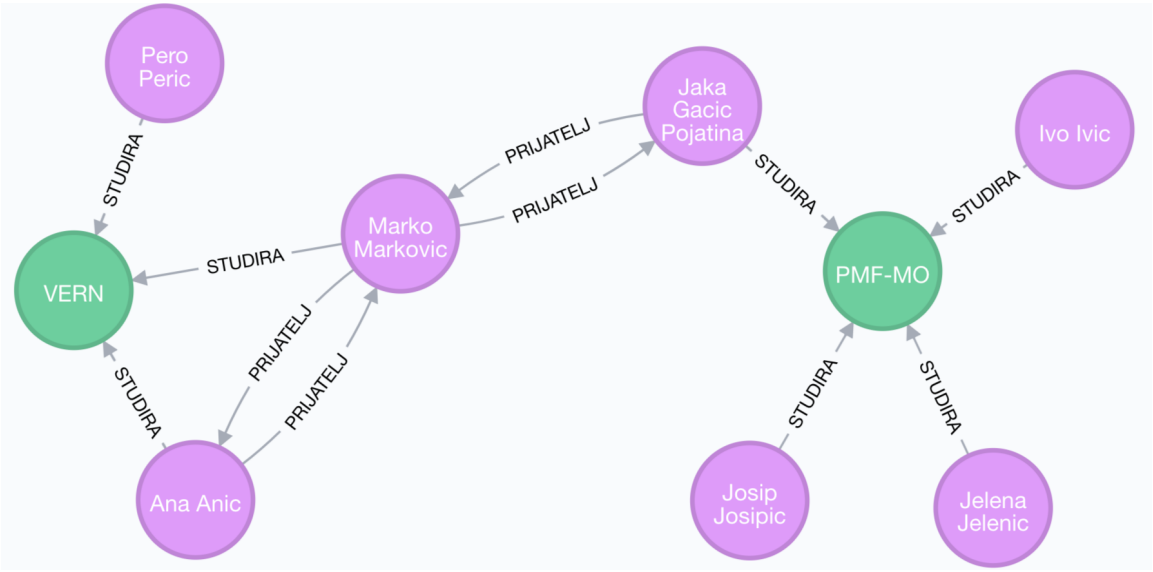
> vraća čvorove Ana, Jelena, Josip i fakultete
```

```
# možemo i promovirati sve studente na sljedeću godinu
MATCH (u:Student)-[studira:STUDIRA]->(:Fakultet)
SET studira.godina = studira.godina + 1
RETURN u, studira

# dodavanje indeksa na stupac godina
CREATE INDEX ON :STUDIRA(godina)
> Added 1 index, completed after 52 ms.

# brisanje indeksa
DROP INDEX ON :STUDIRA(godina)
> Removed 1 index, completed after 10 ms.

# za prikaz svih čvorova u grafu možemo koristiti
MATCH (n) RETURN n
```



Slika 3.1: Svi čvorovi i veze u grafu

Za kraj, pokažimo kako možemo obrisati suvišne elemente. Za brisanje čvorova i veza koristimo naredbu DELETE. Kao što smo već spomenuli, čvor ne može biti obrisan dok postoje veze između njega i bilo kojeg drugog čvora. Rješenje je eksplicitno obrisati sve veze tog čvora prije pokušaja brisanja ili, još bolje, koristiti DETACH DELETE naredbu, koja u transakciji prije brisanja čvora obriše sve njegove veze.

```
# pokušaj brisanja rezultira greškom
MATCH (n)
WHERE n.ime='Jaka Gacic Pojatina'
DELETE n

> Cannot delete node<614>, because it still has relationships.

# brisanje svih Jakinih veza
MATCH (n)-[veza]-(r)
WHERE n.ime='Jaka Gacic Pojatina'
DELETE veza

# naredba DETACH DELETE je uvijek dopuštena
# brisanje svih Aninih čvorova prijatelja
MATCH (n)-[veza:PRIJATELJ]-(prijatelj)
WHERE n.ime='Ana Anic'
DETACH DELETE prijatelj

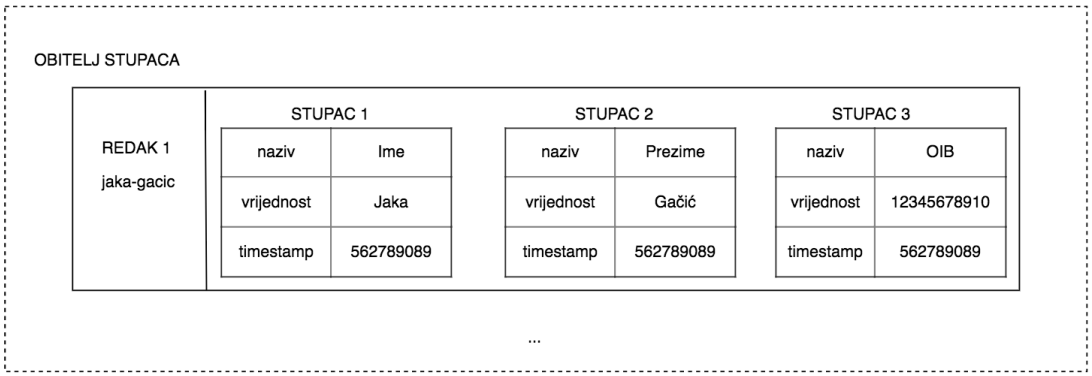
# brisanje svih veza s fakultetom PMF gdje je godina veća od 5
MATCH (n)-[veza:STUDIRA]-(fakultet:Fakultet)
WHERE
    veza.godina > 5 AND fakultet.naziv = "PMF-MO"
DELETE veza

# za brisanje svih podataka iz grafa možemo koristiti
MATCH (n) DETACH DELETE n
```

3.3 Stupčane baze

Budući da se mnogobrojne implementacije značajno razlikuju, radi konkretnosti, promatrat ćemo popularan SUBP Cassandra-u. Cassandra je još jedan moćan proizvod otvorenog koda, napisan u jeziku Java. Razvili su ga Facebookovi inženjeri za potrebe pretrage Inbox-a. Zbog fleksibilnosti i brze obrade podataka, ovaj sustav danas koriste Apple, CERN, eBay, GitHub, GoDaddy, Hulu, Instagram, Intuit, Netflix, Reddit i još više od 1.500 drugih klijenata koji su prepoznali njegove kvalitete. Samo tvrtka Apple koristi oko 75.000 računalnih čvorova za pohranu i obradu preko 10.000 TB podataka. Prema službenoj stranici, Cassandra dnevno uspješno izvrši i oko trilijun zahtjeva s Netflix-a. Njihovih 288 računalnih čvorova sposobno je odraditi više od milijun upisa u sekundi. Iz ovoga je jasno zašto Cassandra-u često ističu kao **najskalabilniju** implementaciju SUBP-a.

Podatke u stupčanim bazama najlakše je zamišljati kao obrnute tablice. Osnovna jedinica pohrane je stupac. Svaki stupac sastoji se od **naziva**, pripadajuće **vrijednosti** i **vremenske oznake**. Te vremenske oznake koriste se za rješavanje konflikata, određivanje isteka valjanosti podataka, za automatsko brisanje i slično. Dopušteno je i ugnježđivanje stupaca – vrijednost stupca može sadržavati listu drugih stupaca. Za stupce s takvim vrijednostima kažemo da su **super–stupci**. Super–stupci nemaju vremenske oznake. Kolekciju stupaca povezanu istim ključem, koji može biti proizvoljan string, nazivamo **redak**. Stupci koji se često upotrebljavaju zajedno, grupiraju se u **obitelj stupaca**, odnosno, obitelj super–stupaca, ako su stupci u njoj super–stupci. Slično kao i u relacijskom modelu, kod stvaranja obitelji potrebno je definirati koje stupce ćemo spremati unutra i zadati tip podataka za svaki od njih. Dodatno, vrijednosti u stupcima u Cassandra-i, osim teksta i brojeva, mogu biti liste, mape, skupovi i, kao što smo već spomenuli, drugi stupci. Svaka obitelj sprema se, dok god je to moguće, u zasebnu datoteku, sortirano po vrijednosti imena stupca. Sve obitelji stupaca moraju biti definirane unutar nekog ***keyspace***-a (prostora ključeva). Keyspace se koristi za grupiranje obitelji unutar sustava prema aplikacijama koje ih koriste. Napomenimo da novije verzije Cassandra-e preporučuju izbjegavanje korištenja super–stupaca i preporučuju korištenje mapa ili listi, kao zamjene za ovu funkcionalnost.



Slika 3.2: Obitelj stupaca

Jednu instancu baze unutar relacijskog SUBP-a, možemo poistovijetiti s jednim *keyspace*-om u Cassandra-i. Prije svakog upita potrebno je odabrati keyspace na koji se upiti odnose, jednako kao u relacijskoj bazi. Lako je uočiti i paralelu između obitelji stupaca i tablica u relacijskom modelu. Pojmovi retka se, također, podudaraju – skupina atributa koji opisuju jedan entitet. Ipak, važna razlika je što redci unutar obitelji stupaca ne moraju imati istu strukturu. Svaki redak može sadržavati proizvoljno mnogo stupaca, a oni se mogu, u

bilo kojem trenutku, brisati i dodavati, bez utjecaja na ostale retke unutar obitelji. Razlog tomu je što se, u relacijskom modelu, svi atributi jednog retka spremaju zajedno u memoriji, kao jedna cjelina. Kada nas zanima vrijednost jednog stupca neke tablice, potrebno je locirati sve njezine retke, odnosno, dohvatiti cijelu tablicu, i iz svakog iščitati vrijednost traženog stupca. Kod stupčanih baza, na disk se zajedno spremaju obitelji stupaca, tj. vrijednosti pojedinog stupca za sve entitete. Zato je za vrijednost stupca dovoljno iščitati datoteku njegove obitelji. Razlika u brzini je značajna već za iole veći skup podataka.

Primjer 3.3.1. *Pretpostavimo da u tablici jedan cijeli stupac (pod tim mislimo na skup vrijednosti tog atributa za sve entitete) ima veličinu* 1 MB. *Tablica Osobe ima* 6 *takvih stupaca, što znači da će za čitanje vrijednosti Ime, relacijska baza morati pročitati* 6 MB *podataka s diska. Stupčana baza bi vrijednosti mogla pročitati iz datoteke obitelji na slici 3.2, koristeći nešto malo više od* 3 MB *(razlika dolazi od spremanja vremenske oznake). Dakle, dobili bismo isti rezultat s upola manje čitanja. Ukoliko bi u obitelji bilo više stupaca, morali bismo ih pročitati sve, što bi moglo malo pokvariti rezultate. Zato je važno paziti da su grupirani podaci doista vrlo bliski, u smislu zajedničkog korištenja, kako bi se izbjeglo čitanje nepotrebnih stupaca. Zbog ovakvog pristupa, u aplikaciji se sve operacije nad stupcima mogu odraditi jako brzo – sumiranje, računanje prosjeka, traženje maksimalne ili minimalne vrijednosti i slično.*

Ovaj tip podataka je vrlo pogodan za implementaciju sustava za upravljanje sadržajem, koji za svaki blog mora pamtiti kategorije, oznake, poveznice i slično. Budući da se keyspace mora definirati prije svakog zahtjeva, stupčane baze su izvrsne i za čuvanje zajedničkog dnevnika događaja aplikacija, koje su dio neke veće grupe. Ipak, mana ovog pristupa je izvlačenje podataka po retcima. Razlog tomu je upravo gore spomenuta fizička organizacija podataka. Za spajanje vrijednosti iz različitih stupaca, potrebno je pročitati nekoliko datoteka i u svakoj pronaći stupac čiji ključ odgovara ključu traženog retka. S druge strane, relacijske baze ovo rade iznimno brzo, jer su svi atributi jednog retka pohranjeni zajedno.

Jasno je da se u stupčanim bazama lako može implementirati relacijski model podataka, no to nam, naravno, nije cilj. Implementacija u obrnutom smjeru je, također, izvediva, dok god koristimo tipove podataka koje relacijski SUBP podržava, no nepotrebna. Ako bismo baš željeli, mogli bismo tablice razlomiti na manje dijelove po stupcima i, po potrebi, spajati JOIN naredbama. Jasno, to ne znači da bismo dobili iste performanse, jer je baza ipak najefikasnija kada radi s modelom podataka za koji je namijenjena.

Cassandra je namijenjena radu na klasteru prema peer–to–peer modelu, s ugrađenim faktorom replikacije 3, kojeg klijent može, po potrebi, promijeniti. To znači da su svi čvorovi ravnopravni, odnosno, mogu izvršavati i operacije pisanja. Konzistentnost se ostvaruje kvorumima, što omogućuje odabir količine čvorova koji moraju potvrditi čitanje i pisanje. Ova vrijednost može biti bilo koji broj između 1 i faktora replikacije, uključujući

granice. Veći broj znači veću konzistentnost, a manju dostupnost, i obrnuto. Kao i u ranije navedenim primjerima, važno je pronaći zadovoljavajući omjer konzistentnosti i dostupnosti.

Standardne transakcije ne postoje – dopušteno je istovremeno pisati po istom retku, no ta pisanja su ipak **atomarna**. To znači da se ažuriranje ili dodavanje više stupaca u istom redu tretira kao jedno pisanje, koje može uspjeti ili ne. Transakcije mogu biti implementirane korištenjem vanjskih paketa, poput *ZooKeeper*-a. Ipak, ukoliko su transakcije bitan dio sustava, ovaj model se ne preporuča za upotrebu. Sva pisanja prvo se bilježe u **memtable** i **dnevnik promjena**. Te zabilježbe dovoljne su da se operacija smatra uspješno izvršenom. Razlog tomu je što, ako u kasnijim koracima i dođe do greške, nakon oporavka čvor sam sebe može ažurirati koristeći zabilježene promjene. Oporavak čvorova može zatražiti i klijent, naredbom *repair*, na razini cijelog keyspace-a ili određenih obitelji stupaca. Ta naredba će, za svaki ključ spremljen u danoj strukturi, usporediti njegovu vrijednost na svim drugim replikama i, po potrebi, ažurirati spremljene vrijednosti.

Implementacija

Opet ćemo koristiti *homebrew* za instalaciju, a direktna poveznica na paket može se pronaći na [1]. Naredbe se u Cassandra-i zadaju korištenjem vlastitog jezika upita – **Cassandra Query Language**, koji je, kao što ćemo vidjeti, sintaksom jako sličan SQL-u. Ipak, mnogo mogućnosti koje SQL i relacijske baze pružaju su ispuštene. Neke nedostaju tek privremeno i njihova implementacija je planirana u budućnosti, dok su neke ciljano izbačene, jer bi mogle ugroziti dostupnost i efikasnost baze, što su njezine najveće prednosti.

```
# instalacija paketa
$ brew install cassandra

# pokretanje servisa
$ cassandra -f

# dok je servis u radu taj prozor terminala je blokiran
# pa u novom terminal prozoru pokrećemo CQL
$ cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.10 | CQL spec 3.4.4 | Native protocol v4]
```

Korištenjem naredbe HELP možemo dobiti popis svih raspoloživih naredbi. Primijetimo da radimo na Test Cluster-u koji se automatski stvara pri instalaciji. Trenutno, u našoj instanci Cassandra-e postoje samo sistemski keyspace-ovi pa ćemo morati dodati novi.

Kod stvaranja keyspace-a nužno je zadati osnovna svojstva replikacije, poput strategije i faktora replikacije. Mi ćemo koristiti najučestalije postavke.

```
# ispis svih postojećih keyspace-a
cqlsh > DESCRIBE keyspaces;
system_traces system_schema system_auth system system_distributed

# stvaramo keyspace "osobe"
cqlsh > CREATE KEYSPACE osobe WITH
replication = {'class':'SimpleStrategy', 'replication_factor' : 3};

# prebacujemo se u novostvoreni keyspace
cqlsh > USE osobe;

# stvaranje nove obitelji stupaca
# definiramo jmbag kao primarni ključ
cqlsh:osobe > CREATE TABLE studenti (
    jmbag text,
    upisao int,
    trenutni_stupanj text,
    zavrsio boolean,
    PRIMARY KEY (jmbag)
);

# ispis svih obitelji stupaca u trenutnom keyspace-u
cqlsh:osobe > DESCRIBE TABLES;
studenti

# dodavanje vrijednosti
cqlsh:osobe > INSERT INTO studenti
    (jmbag, upisao, trenutni_stupanj, zavrsio)
VALUES
    ('1191221523', 2010, 'univ. bacc. math', false);

cqlsh:osobe > INSERT INTO studenti
    (jmbag, upisao, trenutni_stupanj, zavrsio)
VALUES
    ('1192202020', 2010, 'univ. bacc. oec.', false);
```

```
cqlsh:osobe > INSERT INTO studenti
    (jmbag, upisao, trenutni_stupanj, zavrsio)
VALUES
    ('1192202222', 2013, 'univ. bacc. oec.', false);

cqlsh:osobe > INSERT INTO studenti
    (jmbag, upisao, trenutni_stupanj, zavrsio)
VALUES
    ('1192203452', 2013, 'univ. bacc. educ. math', false);

cqlsh:osobe > INSERT INTO studenti
    (jmbag, upisao, trenutni_stupanj, zavrsio)
VALUES
    ('1191222523', 2012, 'univ. bacc. math', false);
```

Napomenimo da, iako je stupac jmbag definiran kao primarni ključ, pokušaj dodavanja novog zapisa s istim ključem neće rezultirati greškom, nego će se ažurirati vrijednost pod tim ključem. Međutim, ovo ne bi trebalo koristiti kao metodu ažuriranja, jer može dovesti do neočekivanih rezultata. Zato koristimo metodu UPDATE. Prvo moramo imenovati obitelj stupaca koju želimo ažurirati, a zatim ključnom riječi SET imenujemo atribute i zadajemo vrijednosti koje želimo pridružiti. Na kraju možemo dodati WHERE uvjet ako ne želimo ažurirati sve retke. Više o definiranju uvjeta pokazat ćemo u kasnijim primjerima, a za sada identificiramo redak prema primarnom ključu.

```
# ažuriranje vrijednosti
# student s jmbagom 1191222523 je završio fakultet
cqlsh:osobe > UPDATE studenti
   SET zavrsio = true
WHERE jmbag = '1191222523';
```

Velika mana ovog jezika upita, u odnosu na SQL, je što je AND jedini podržani logički operator za spajanje uvjeta. U nekim slučajevima to možemo nadoknaditi korištenjem operatora IN koji dozvoljava pobrojavanje vrijednosti, no njegova uporaba je, zasad, dozvoljena samo na primarnom ključu. Također, korištenje operatora, osim jednakosti, nije dozvoljeno na stupcima koji nisu indeksirani, osim ako eksplicitno ne naglasimo da to želimo dopustiti prilikom upita. Na taj način Cassandra se pokušava zaštititi od izvršavanja neefikasnih upita. Upiti koji se pak izvršavaju jako efikasno su agregatne funkcije – SUM, AVG, COUNT i druge. Razlog tomu je što su vrijednosti stupaca spremljene uzastopno u memoriji. U nastavku ćemo pokazati jednostavne slučajeve dohvaćanja i brisanja vrijednosti iz obitelji.

```
# ispis svih vrijednosti u obitelji
cqlsh:osobe > SELECT * FROM studenti;
```

```
 jmbag      | trenutni_stupanj        | upisao | zavrsio
------------+-------------------------+--------+---------
 1191221523 |       univ. bacc. math |   2010 |   False
 1192202222 |       univ. bacc. oec. |   2013 |   False
 1192202020 |       univ. bacc. oec. |   2010 |   False
 1192203452 | univ. bacc. educ. math |   2013 |   False
 1191222523 |       univ. bacc. math |   2012 |    True

(5 rows)
```

Slika 3.3: Obitelj stupaca studenti

```
# spajanje više uvjeta
cqlsh:osobe > SELECT jmbag FROM studenti
WHERE trenutni_stupanj = 'univ. bacc. oec.'
AND zavrsio = false;

# operator < nad upisao zahtijeva odobravanje filtriranja
cqlsh:osobe > SELECT jmbag, trenutni_stupanj FROM studenti
WHERE upisao < 2011 ALLOW FILTERING;

# dodavanje indeksa
cqlsh:osobe > CREATE INDEX indeks_upisao ON studenti (upisao);

# sad više ne moramo dozvoljavati filtriranje
cqlsh:osobe > SELECT jmbag, trenutni_stupanj FROM studenti
WHERE upisao < 2011

# korištenje operatora IN za vrijednost primarnog ključa
cqlsh:osobe > SELECT jmbag, trenutni_stupanj FROM studenti
WHERE jmbag IN ('1191222523', '1192202020');
```

```
# korištenje agregatnih fukcija

cqlsh:osobe > SELECT AVG(upisao) AS prosjek FROM studenti;
prosjek -> 2011.5

cqlsh:osobe > SELECT MIN(upisao) AS najstariji FROM studenti;
najstariji -> 2010

cqlsh:osobe > SELECT COUNT(*) AS brojac FROM studenti;
brojac -> 5

# brisanje retka
cqlsh:osobe > DELETE FROM studenti
WHERE zavrsio = true;

# brisanje indeksa
cqlsh:osobe > DROP INDEX indeks_upisao;

# brisanje tablice
cqlsh:osobe > DROP TABLE studenti;

# brisanje keyspace-a
cqlsh:osobe > DROP KEYSPACE osobe;
```

3.4 Dokument baze

2001. godine, grupa inženjera iz San Francisca, koji su imali iskustva u pretraživanju dokumenata, osnovala je tvrtku MarkLogic, koja se fokusirala na upravljanje velikim kolekcijama XML dokumenata. Definirali su dva tipa čvorova u klasteru – dokument čvorove i čvorove za upite. Čvorovi za upite primaju dolazne upite i koordiniraju sve aktivnosti povezane s njihovim izvođenjem. Dokument čvorovi sadrže XML dokumente i odgovorni su za izvođenje upita nad dokumentima koje imaju u lokalnom sustavu pohrane. Kada čvor primi upit, on ga distribuira na sve dokument čvorove. Svaki takav čvor interno izvršava upit nad svojim dokumentima i, one koji odgovaraju upitu, vraća čvoru koji mu je proslijedio upit. Kada svi čvorovi odgovore, čvor koji je upit primio spaja rezultate i vraća ih klijentu.

Ovaj neobičan pristup – dovođenje upita dokumentima, umjesto dotadašnjeg dovođenja dokumenata upitima, omogućio im je rad s tisućama terabyte-a dokumenata. MarkLogic je brzo pronašao tržište za svoje proizvode. Američke obavještajne agencije i izdavačke

kuće prepoznale su potencijal u ideji i vidjele priliku da organiziraju svoje ogromne zalihe podataka. Proizvode inspirirane ovim pristupom danas nazivamo **dokument baze**. Jedna od najpoznatijih implementacija dokument baza je sustav MongoDB, čije ćemo detalje razmotriti u nastavku.

Osnovni element ovog modela je **dokument**. Pod dokumentom smatramo uređeni skup ključeva s pridruženim vrijednostima. Ključevi moraju biti jedinstvene vrijednosti, no nisu od velike važnosti kao kod ranije opisanog ključ–vrijednost modela. Razlog je tome što ovaj model sav fokus prebacuje na sadržaj vrijednosti dokumenata. Svaki dokument mora sadržavati i primarni ključ, čija je vrijednost specijalnog tipa *ObjectId*. Ukoliko on nije eksplicitno naveden, baza će ga automatski dodijeliti na temelju trenutnog vremena, procesa, računala s kojeg zahtjev dolazi i broja spremljenih podataka u kolekciji.

Dokumenti se pohranjuju u **BSON**[1] formatu. BSON objekt je po notaciji vrlo sličan JSON-u, što omogućuje da ga ljudi lako razumiju. Međutim, BSON proširuje ovaj format uvođenjem dodatnih tipova vrijednosti, poput datuma, uvođenjem mogućnosti definiranja redoslijeda i dohvaćanja vrijednosti pod određenim ključem, bez dohvaćanja cijelog objekta. Format je fleksibilan, a osim raznih tekstualnih i numeričkih polja, omogućuje spremanje boolean vrijednosti, datuma, identifikatora, pokazivača, objekata, nizova i drugih dokumenata. BSON sprema i neke dodatne informacije, poput duljine vrijednosti i vremena spremanja. U mnogo slučajeva, zauzima i manje fizičkog prostora za pohranu od ekvivalentnog JSON formata. Dodatno, omogućeno je spremanje cijelih brojeva (integera), bez pretvaranja u tekst, pa se dobiva na brzini prilikom (de)kodiranja objekata.

Da bi klijent znao što je spremljeno u kojem dokumentu, uvode se **kolekcije**. Za svaku kolekciju definiramo tip dokumenata koje čuva, ali samo na logičkoj razini, nevezano za njihovu strukturu. Na primjer, jedna kolekcija može sadržavati knjige, a druga studente. Aplikacija sama mora voditi računa o tome koji podatak se sprema u koju kolekciju. Kolekcije donekle možemo poistovijetiti s tablicama u relacijskom modelu, a dokumente s pojedinim entitetima. Velika prednost kolekcija je što nemaju definiranu shemu, što znači da svaki dokument, čak i unutar iste kolekcije, može imati proizvoljnu strukturu. Daljnja razlika je što je u dokumentima dopušteno sadržavati druge dokumente. To omogućuje da se svi povezani dokumenti sažmu u jedan veći dokument. Prednost ovakvog pristupa je što se može brzo i lako pristupiti svim povezanim dokumentima, jer su spremljeni zajedno u memoriju. Time se eliminira potreba za operacijama spajanja i upitima kroz više kolekcija. Kolekcije namijenjene zajedničkom korištenju definiraju se unutar jedne instance baze.

Dokument baze općenito su napravljene s ciljem distribucije podataka na više servera. Sustavi sami vode brigu o raspodjeli podataka i povezivanju s drugim serverima. Distribucija se najčešće ostvaruje replikacijom i fragmentacijom prema gospodar–rob principu. MongoDB se, također, oslanja na ovaj princip. Ukoliko dođe do kvara gospodara, čvorovi među sobom odabiru novog vođu. Kako bismo osigurali da će u tom slučaju biti odabran

[1] od engl. *Binary JSON*

najprikladniji sljedeći čvor, možemo im dodijeliti prioritete. Ova odluka može se temeljiti na lokaciji čvora ili količini RAM-a koju posjeduje. Iako distribuirani model pruža mnoge prednosti, kako bi se osigurala visoka brzina odaziva, bilo je nužno izbaciti podršku ACID transakcijama. Transakcije nisu podržane na razini baze, niti kolekcija, no sve operacije unutar pojedinog dokumenta izvršavaju se u cjelini. Ovakav princip rada nazivamo **atomarne transakcije**.

Konzistentnost ažuriranja se osigurava tako da sva ažuriranja idu preko gospodara. Svi pokušaji pisanja se prihvaćaju i smatraju uspješnim, osim ako klijent ne definira postavke drugačije. To znači da se, bez modifikacija, pisanje smatra uspješnim čim čvor gospodar primi zahtjev. Ipak, možemo zahtijevati svojevrstan kvorum – ako se pisanje ne uspije provesti na zadanom broju čvorova, smatra se neuspješnim. Broj čvorova možemo zadati na razini aplikacije, kolekcije ili pojedinog zahtjeva, dodavanjem parametra *w*. Veći broj osigurava veću konzistentnost, no narušava performanse. Dodatno, podržan je i parametar *slaveOk*, kojim možemo reći bazi da želimo dopustiti čitanje s robova i dok traje propagacija ažuriranja. Također ga je moguće podesiti na nekoliko razina. Odabir ovih parametara je na arhitektu aplikacije i ovisi o osjetljivosti podataka unutar aplikacije, odnosno, pojedinih kolekcija ili dokumenata.

Implementacija

Instalacija MongoDB sustava na MacOS je vrlo jednostavna preko *homebrew*-a. Na službenim stranicama [7] mogu se naći upute za instalaciju na druge operativne sustave. Na istom mjestu, moguće je i kroz par koraka stvoriti javno dostupan, besplatan klaster. Iako je ovaj kluster veličine samo 512 MB, potpuno je funkcionalan i idealan je za istraživanje mogućnosti sustava. No, vratimo se na lokalni server. Nakon instalacije, potrebno je pokrenuti instancu MongoDB-a i odabrati bazu s kojom ćemo raditi. Kolekcije i baze nije potrebno eksplicitno definirati, jer se one automatski stvaraju pri prvom pokušaju korištenja. Kod dodavanja nećemo zadavati identifikator, nego ćemo odgovornost kreiranja prepustiti samoj bazi. U odgovoru, nakon uspješnog dodavanja, baza vraća novododani identifikator.

```
# instalacija
$ brew install mongodb

# pokretanje mongo instance
$ mongo

# implicitno stvaramo bazu
> use osobe;
```

```
# implicitno stvaramo kolekciju studenti i dodajemo studente
> db.studenti.insertMany( [ {
        "jmbag" : "1191221523",
        "godina" : 5,
        "upisao" : 2010,
        "ocjene" : [ {
             "predmet" : "Matematička analiza 1",
             "ocjena" : 2
         } , {
             "predmet" : "Linearna algebra 1",
             "ocjena" : 2
         } , {
             "predmet" : "Elementarna matematika 1",
             "ocjena" : 2
         } ]
      }, {
        "jmbag" : "2435678998",
        "godina" : 3,
        "upisao" : 2013,
        "ocjene" : [ { ... } ]
      },  {
        "jmbag" : "35467896789",
        "godina" : 2,
        "upisao" : 2016,
        "ocjene" : [ { ... } ]
      },  {
        "jmbag" : "5467896578",
        "godina" : 5,
        "upisao" : 2011,
        "ocjene" : [ { ... }]
      } ] );

# odgovor sadrži dodijeljene identifikatore
    "insertedIds" : [
        ObjectId("598b418c907392218eaf3dda"),
        ObjectId("598b418c907392218eaf3ddb"),
        ObjectId("598b418c907392218eaf3ddc"),
        ObjectId("598b418c907392218eaf3ddd")
    ]
```

```
# prebrojavamo sve objekte u kolekciji studenti
> db.studenti.count()
4
```

Iako su dosad ubačeni objekti bili jednake strukture, kolekcije predstavljaju samo logičku vezu među njima pa nas, tehnički, ništa ne sprječava da u istu kolekciju ubacimo nepovezani objekt, potpuno različite strukture.

```
# ovo je također valjano ubacivanje
$ db.studenti.insertOne( {
      "ime" : "Jaka"
   } );
```

Dokument baze omogućuju bogatu pretragu sadržaja dokumenata jezikom upita. Kako bi pretrage bile efikasne, kod dodavanja novog dokumenta, cijeli se njegov sadržaj automatski indeksira. U nastavku ćemo pokazati neke jednostavne upite koje je moguće izvršavati. Osnovna naredba za pretragu je *db.imeKolekcije.find(uvjet)*. Uvjet ne mora biti zadan i, u tom slučaju, baza vraća sve objekte u kolekciji. Moguće je da neki dokumenti nemaju definirana sva svojstva zadana u uvjetu, no to bazi ne stvara problem. Ukoliko se zadaju, uvjeti moraju biti JSON objekti, a, osim standardnih ključeva i vrijednosti, mogu sadržavati i ključne riječi. U nastavku navodimo nekoliko najčešćih.

```
$ne - različito
$lt(e) - manje (ili jednako)
$gt(e) - veće (ili jednako)
$and, $or - logički operatori
$elemMatch - uvjet na objekte unutar niza
```

Ako nije naznačen niti jedan operator, podrazumijeva se jednakost. Napomenimo da se uvjeti mogu zadavati i na svojstvima ugnježđenih objekata i svojstvima objekata u nizu. Rezultate možemo i sortirati prema jednom ili više svojstava. Dodatak *pretty()* u primjerima je funkcija koja formatira ispis objekata, tako da svaki par ključa i vrijednosti vraća u zasebnom retku konzole.

```
# vraća formatiran ispis svih objekata u kolekciji studenti
> db.studenti.find().pretty();

# sortiranje po "godina" uzlazno i "upisao" silazno
> db.studenti.find().sort({ "godina" : 1, "upisao" : -1 });
```

```
# vraća formatiran ispis studenta s jmbagom 1191221523
> db.studenti.find({ jmbag : "1191221523"}).pretty();

# sintaksa za logičke operatore
> db.studenti.find( {
      $or: [
         { "upisao" : 2013 }, { "godina" : 2 }
      ]
} ).pretty();

# operator AND se ne mora eksplicitno navesti
> db.studenti.find(
     { "upisao" : 2013 }, { "godina" : 2 }
).pretty();

# sintaksa za relacijske operatore
> db.studenti.find( {
    "upisao": { $gte: 2013 }
} ).pretty();

> db.studenti.find( {
    "upisao": { $ne: 2013 }
} ).pretty();

# pretraga po svojstvima objekata u nizu ocjena
# vraća sve studente koji imaju upisan predmet MA1
> db.studenti.find( {
    "ocjene.predmet" : "Matematička analiza 1",
} ).pretty();

# pomoću $elemMatch-a možemo zadati i složenije uvjete
> db.studenti.find( {
    "ocjene" : {
        $elemMatch:
          { "predmet": "Matematička analiza 1", "ocjena": { $gte: 2 } }
     }
} ).pretty();
```

Osim dodavanja i pretraživanja, dokumente, naravno, možemo i ažurirati. MongoDB ima dvije osnovne metode ažuriranja. Prva je *update()* kojoj prosljeđujemo 3 objekta. U prvom zadajemo uvjet na dokumente koje želimo ažurirati. U drugom dokumentu navode se svi atributi i nove vrijednosti koje želimo ažurirati. Ako ne naglasimo drugačije, baza će ažurirati samo prvi dokument koji odgovara zadanom kriteriju. Ukoliko želimo promijeniti to ponašanje, u trećem dokumentu možemo proslijediti ključ *multi* s vrijednošću *true*. Također, ova metoda dozvoljava i neke dodatne opcije, osim postavljanja vrijednosti. Možemo povećati ili pomnožiti postojeću vrijednost za neki dani skalar, preimenovati ili obrisati cijelo svojstvo i slično. Ova metoda je osobito korisna za rad s nizovima objekata. Postoje posebne ključne riječi za modifikaciju nizova – dodavanje i izbacivanje jednog ili više elemenata, primjenjivanje operacija na svaki element niza, dohvaćanje pozicije, itd. Druga metoda je *save()*. Ona je puno manje fleksibilna od *update* te prima samo jedan objekt. Taj objekt mora sadržavati objekt identifikator i novi dokument, koji će se spremiti pod tim identifikatorom. Stari dokument bit će u cjelosti zamijenjen novim. Ova metoda djeluje na samo jedan dokument, jer je identifikator jedinstven. U nastavku navodimo neke primjere uporabe za obje metode.

```
# pomicanje svih studenata koji su upisali fakultet
# prije 2013. godine na 5. godinu
> db.studenti.update(
    { "upisao": { $lt: 2013 } },
    { $set : { "godina" : 5 } },
    { multi : true }
);

# svi studenti dobivaju ocjenu 5 iz MA1
# obavezno je koristiti operator $ za pozicioniranje
> db.studenti.update(
    { "ocjene.predmet" : "Matematička analiza 1" },
    { $set : { "ocjene.$.ocjena": 5 } },
    { multi : true }
);

# povećanje godine za 1 svim studentima
> db.studenti.update(
    {},
    { $inc : { "godina" : 1 } },
    { multi : true }
);
```

```
# brisanje svojstva godina studentima
# koji su upisali fakultet prije 2010. godine
> db.studenti.update(
    { "upisao": { $lt: 2010 } } ,
    { $unset : { "godina" : "" } },
    { multi : true }
);

# svim studentima prve godine dodajemo ocjenu 5 iz Izbornog predmeta
> db.studenti.update(
    { "godina" : 1 },
    { $push : {
        "ocjene" :
            {  "predmet": "Izborni predmet", "ocjena": 5 }
    } },
    { multi : true }
);

# stari objekt se zamjenjuje novim
# više ne postoji student s jmbagom 1191221523
> db.studenti.save( {
        "_id" : ObjectId("598b418c907392218eaf3dda"),
        "jmbag" : "32443433423",
        "godina" : 2,
        "upisao" : 2017,
        "ocjene" : [ { ... }]
} );
```

I za kraj, pogledajmo kako možemo obrisati dokumente iz kolekcije.

```
# brišu se svi studenti upisani prije 2010. godine
> db.studenti.remove( { "upisao": { $lt: 2010 } } );

# briše sve dokumente u kolekciji
> db.studenti.remove();
```

Bibliografija

[1] *Apache Cassandra web stranica*, https://cassandra.apache.org/.

[2] Tonči Carić i Mario Buntić, *Uvod u relacijske baze podataka*, 2015.

[3] Seth Gilbert i Nancy A. Lynch, *Perspectives on the CAP Theorem*, Computer **45** (2012), br. 2, 30–36, http://dx.doi.org/10.1109/MC.2011.389.

[4] Peter Lake i Paul Crowther, *Concise Guide To Databases*, Springer, 2013.

[5] Robert Manger, *Baze Podataka*, rujan 2013.

[6] Dan McCreary i Ann Kelly, *Making Sense of NoSQL*, Manning Publications, 2014.

[7] *MongoDB Atlas web stranica*, https://www.mongodb.com/.

[8] *Neo4j web stranica*, https://neo4j.com/.

[9] *Riak KV web stranica*, https://docs.basho.com/riak/kv/2.2.3/.

[10] Parmod J. Sadalge i Martin Fowel, *NoSQL Destilled*, Addison-Wesley, 2013.

Sažetak

Potreba za spremanjem, kategorizacijom i pretraživanjem podataka stara je gotovo koliko i samo čovječanstvo. Iako su se promijenili načini pohrane i obrade podataka, glavni problemi rukovanja podacima ostali su isti. Smanjenje financijskog troška čuvanja podataka, brzo i pregledno čitanje podataka, lako održavanje i konzistentnost, samo su neki od konstantnih izazova s kojima se susreću arhitekti baza podataka.

Razvojem računala i pojavom relacijskih baza podataka u 70-im godinama, činilo se da su svi problemi riješeni, no popularizacijom Interneta javili su se i novi tehnološki izazovi. Količina podataka povećala se eksponencijalnom brzinom, a uniformna struktura informacija se počela gubiti. Postalo je jasno da globalno popularan relacijski model u određenom broju slučajeva više nije dovoljno dobar. Velike online trgovine, društvene mreže i web tražilice bile su suočene s velikim problemom, koji je rastao iz dana u dan. Te su kompanije odlučile uzeti stvari u svoje ruke i započele su *NoSQL* pokret.

Općeprihvaćena formalna definicija NoSQL sustava još uvijek ne postoji, no možemo je shvatiti kao skup principa arhitekture baza podataka koji odmiču od centraliziranog, strogo definiranog relacijskog modela i SQL upita. Osnovna svojstva takvih baza su brzina, koju omogućuje horizontalno skaliranje operacija, visoka tolerancija na veličinu baze, zbog fragmentacije podataka na klastere, te prilagodljivost, zahvaljujući izostanku definicije sheme podataka. Budući da je svaka kompanija razvijala model podataka koji bi savršeno odgovarao njezinim potrebama, danas poznajemo mnogo implementacija NoSQL sustava. Neki od najpopularnijih modela podataka implementiranih u ovim sustavima su: ključ–vrijednost, graf, dokument i stupčane baze podataka.

Cilj rada je detaljno objasniti spomenute principe i predstaviti sve gore navedene tipove NoSQL sustava. Rad će pokazati primjere implementacije i primjene za svakoga od njih, te izložiti njihove prednosti i nedostatke u odnosu na relacijske baze. Rad ne želi predložiti napuštanje relacijskih baza podataka, nego osvjestiti čitatelja da jedan model baze ne može odgovarati svakoj aplikaciji. Zato želimo pomoći u donošenju informirane odluke pri odabiru sustava na kojem će se temeljiti aplikacija.

Summary

The need for storing and manipulating data is about as old as the humanity itself. Although the way the data is stored and manipulated has changed a lot through the years, the main problems stayed the same. Lowering the cost of storing data, ease and speed of searching through big amounts of data, maintainability and consistency, are just a few constant challenges in the work of database architects.

As computers evolved and relational databases appeared in the 70's, it seemed all problems are solved, but after the Internet bloomed, new challenges emerged. The amount of data on the Internet grew exponentially fast and the uniformity of data began to fade out. It became clear that the globally popular relational model is just not good enough in some cases. Big online shops, social networks and web search companies found themselves facing a big problem that was growing day by day. Those companies took the matter into their own hands and started the NoSQL movement.

Although a formal definition of NoSQL systems is still missing, we can think of it as a set of principles that look away from the centralized, strictly defined relational model and SQL queries. Main characteristics of such systems are speed, which is increased by using horizontal scaling of operations, high tolerance to database size, due to fragmentation of data on many clusters, and agility, thanks to the leftout definition of database schema. Since each company developed a model to fit their needs perfectly, today we know a lot of different implementations of NoSQL systems. Some of the most popular data models implemented are: key–value, graph, document and column databases.

This thesis aims to explain into detail the principles of NoSQL systems and to introduce popular data models mentioned above. We will show examples of implementation and usage for each of them. The goal is not to suggest that we have to discard relational databases, but rather to accept that one database model cannot fit all software. This is why we aim to help in making an informed decision on which system will be used in development of a particular application.

Životopis

Jaka Gačić rođena je 23. svibnja 1991. godine u Zagrebu. U rodnom gradu završila je Osnovnu školu Ante Kovačića i X. gimnaziju "Ivan Supek", opći smjer. 2010. godine upisala je preddiplomski studij matematike na Matematičkom odsjeku Prirodoslovno–matematičkog fakulteta Sveučilišta u Zagrebu. Prvostupničku diplomu stekla je 2014. godine i potom upisala diplomski studij Računarstvo i matematika pri istom fakultetu. Istovremeno se zapošljava u digitalnoj agenciji Degordian, gdje i danas radi na poziciji *Software Development Intern*.