OPTIMALNO KORIŠTENJE OPEN-SOURCE SUSTAVA ZA UPRAVLJANJE BAZAMA PODATAKA

dr.sc. Alen Lovrenčić
Sveučilište u Zagrebu
Fakultet organizacije i informatike
alovrenc@foi.hr

1. Normalizacija - logika baze podataka

Normalizacija je postupak koji je poznat već tridesetak godina i kojim se osigurava da se u bazi podataka ne događaju neželjene pojave ili tzv. *anomalije* tijekom dodavanja, brisanja ili promjene podataka. Normalizacija osigurava da su entiteti baze podataka definirani tako da u potpunosti zadovoljavaju uvjete relacijskog modela.

Sam se postupak klasične normalizacije provodi nad pojedinim relacijama u bazi podataka njihovim razbijanjem na dvije ili više komponenti, koje se međusobno djelomično preklapaju. Time se povećava redundantnost baze podataka, tj. ponavljanje istih podataka unutar baze podataka na više mjesta, no smanjuju se već ranije napomenute anomalije.

Ako se u modelu baze podataka jave anomalije i odstupanja od normalnih formi, to u svakom slučaju pokazuje na pogrešku kod modeliranja baze podataka i svakako se mora ukloniti. Stoga je pravilna normalizacija baze podataka nužna za dobar dizajn baze podataka i za njen brz rad bez anomalija.

1. normalna forma – atomarnost podataka

Prva normalna forma (1NF) osigurava atomarnost podataka u relacijama. Drugim riječima, podaci u jednoj koloni tablice ne smiju biti višestruki. Treba upozoriti da je prvobitno 1NF bila definirana tako da je zahtijevala da podaci budu atomarni, jednostavni, nedjeljivi. No, to je postalo problem pojavom tipova podataka koji nisu atomarni. Tako, npr. PostgreSQL, ima detaljno razrađenu podršku za spacijalne (prostorne) podatke, koji se ne razmatraju kao elementarne vrijednosti. Također, u sustavima za upravljanje bazama podataka danas se često javlja složeni temporalni (vremenski) tip podataka, koji isto tako nije atomaran. Na kraju, u PostgreSQL-u se, kao i u mnogim drugim sustavima za upravljanje bazama podataka, danas kao podaci u relaciji mogu definirati polja.

2. normalna forma – svi podaci moraju zavisiti o cijelom ključu

Druga normalna forma (2NF) bavi se problematikom složenih ključeva i zavisnosti drugih vrijednosti o dijelu ključa.

Prije no što možemo definirati 2NF, treba definirati značenje pojma funkcijske zavisnosti. Neki atribut A funkcijski zavisi o skupu atributa S, što pišemo $S \rightarrow A$, ako svi slogovi u tablici koji imaju jednake vrijednosti svih atributa skupa S moraju imati nužno i jednake vrijednosti atributa A.

Relacija je u 2NF ako nijedan neključni atribut ne zavisi funkcijski o dijelu ključa.

U ovoj definiciji treba posebno obratiti pažnju na to da se ovdje radi o zavisnosti neključnih atributa. Neključni su oni atributi koji ne sudjeluju ni u jednom ključu. Naime, relacija može imati više ključeva, tj. skupova atributa koji jedinstveno određuju svaki zapis u relaciji. Tako, npr. relacija AUTOMOBIL može imati za ključ broj registarske tablice, ali isto tako može za ključ imati i broj šasije. Bez obzira na to koji je od ključeva izabran za primarni, svi se atributi koji sudjeluju u bilo kojem ključu smatraju ključnima. Shema po kojoj se rješava problem nepoštovanja 2NF je sljedeća: Neka je zadana tablica *r(A, B, C, D)*, te neka atribut *C* zavisi samo o atributu *B*, ali ne i o atributu *A*. Onda se tablica *r* treba razbiti u *r1(A, B, D)* i *r2(A, C)*. Drugim riječima, atributi koji zavise samo o dijelu ključa sele se u drugu tablicu, kojoj ključ postaje onaj dio polazne tablice o kojem su ti atributi zavisili.

U sljedećem se primjeru daje relacija koja nije u 2NF i navode se problemi koji zbog toga mogu nastati.

Radi se o bazi podataka knjižnice. Tablica koja predstavlja problem jest KNJIGA(CIP_BR, UDK_NASLOV, UDK_AUTOR, BR_PRIMJERKA, NASLOV, ISBN, BR_STRANA, GOD_IZDANJA, TEMATIKA, IME_AUTORA, PREZIME_AUTORA, GODINA_ROĐENJA, ZEMLJA, POSUĐENO).

Prije nego što krenemo u daljnje razmatranje problema koji se javljaju pri korištenju ove tablice, primijetimo da skup ISBN, PRIMJERAK također predstavlja kandidata za ključ, pa su atributi iz ovog skupa ključni.

U ovoj tablici ima čitav niz problema koji su nastali zbog toga što tablic nije u 2NF. Jedan je u tome što, iako knjižnice imaju unaprijed definiran šifrarnik tema po kojima katalogiziraju knjige, koji se zove CIP katalogizacija, ako je ovako definirana baza podataka, neka se tema neće moći upisati u bazu podataka ako knjižnica ne posjeduje bar jednu knjigu koja pripada toj tematici. Također, obrišemo li iz baze podataka sve knjige koje pripadaju nekoj tematici, iz baze podataka će nestati i sam naziv tematike. Na kraju, ako u knjižnici postoji više knjiga koje pripadaju istoj tematici, može se dogoditi da je naziv teme kojoj knjige pripadaju različito zapisan u slogovima za te knjige, što predstavlja problem kod pretraživanja. Isto tako, želimo li promijeniti naziv teme kojoj te knjige pripadaju, morat ćemo mijenjati zapise svih knjiga koje pripadaju danoj tematici.

Svi ovi problemi nastali su zbog toga što postoji parcijalna zavisnost neključnog atributa TEMATIKA o atributu CIP_BR, tj. CIP_BR$\rightarrow$TEMATIKA, pa kako je TEMATIKA neključni atribut, to narušava 2NF. Sam problem koji je nastao može e objasniti time što je dizajner baze podataka pogrešno smjestio dva različita entiteta – KNJIGA i CIP, u istu tablicu. Stoga se problemi rješavaju tako da se ova tablica podijeli na dvije tablice:

KNJIGA(CIP_BR, UDK_NASLOV, UDK_AUTOR, BR_PRIMJERKA, NASLOV, ISBN, BR_STRANA, GOD_IZDANJA, IME_AUTORA, PREZIME_AUTORA, GODINA_ROĐENJA, ZEMLJA, POSUĐENO)

CIP(ŠIFRA, NAZIV)

Slični će se problemi javljati i s imenom i prezimenom autora, koje se briše brisanjem svih primjeraka njegovih knjiga, koje može biti različito upisano u bazu podataka za različite primjerke njegove knjige itd. Problemi ovdje opisani nastaju, naravnom jer imamo zavisnost

ISBN→ NASLOV, BR_STRANA, GOD_IZDANJA, IME_AUTORA, PREZIME_AUTORA, GODINA_ROĐENJA, odnosno, CIP_BR, UDK_NASLOV, UDK_AUTOR→ NASLOV , BR_STRANA, GOD_IZDANJA, IME_AUTORA, PREZIME_AUTORA, GODINA_ROĐENJA. Drugim riječima, opet su u istoj tablici pomiješana dva entiteta – KNJIGA i PRIMJERAK. Stoga će naša baza nakon razbijanja tablice KNJIGA izgledati ovako:

KNJIGA(CIP_BR, UDK_NASLOV, UDK_AUTOR, NASLOV, ISBN, BR_STRANA, GOD_IZDANJA, IME_AUTORA, PREZIME_AUTORA, GODINA_ROĐENJA, ZEMLJA)

KNJIGA(CIP_BR, UDK_NASLOV, UDK_AUTOR, BR_PRIMJERKA, POSUĐENO)

CIP(ŠIFRA, NAZIV)

3. Normalna forma – atributi ne smiju tranzitivno zavisiti o ključu

Pretpostavimo da je skup *S* kandidat za ključ, *R* skup koji nije kandidat za ključ, a atribut *A* takav da vrijedi $R \rightarrow A$. Time smo dobili sljedeću shemu zavisnosti:

Ako atributi zadovoljavaju ovakvu shemu, onda kažemo da atribut *A* tranzitivno zavisi o ključu.

Problem koji nastaje zbog toga u našoj bazi podataka jest da podaci koji se odnose na autora, kao što su GODINA_ROĐENJA i ZEMLJA ne mogu biti unešeni nevezano uz knjigu. Zbog toga se javljaju slični problemi koji se javljaju kod problema s 2NF. Naime, godina rođenja autora ili domicilna zemlja mogu biti različito unešene za različite knjige istog autora.

Sada definiramo 3NF.

Relacija je u 3. normalnoj formi (3NF) ako nijedan neključni atribut ne zavisi tranzitivno o ključu.

Neka u relaciji *r(A, B, C, D)* atributi *B* i *C* nisu ključni, te neka postoji zavisnost $A \rightarrow B \rightarrow C$. Tada atribut *C* tranzitivno zavisi o ključu. tada se tablica *r* treba razbiti n a dvije: *r1(A, B, D)* i *r2(B,C)*.

Dakle, skup atributa koji tranzitivno zavisi o ključu seli se u zasebnu tablicu, kojoj ključ postaje onaj skup atributa preko kojeg se ta tranzitivna zavisnost ostvaruje.

U našem primjeru javlja se sljedeća tranzitivna zavisnost CIP_BR, UDK_NASLOV, UDK_AUTOR→ IME_AUTORA, PREZIME_AUTORA→GODINA_ROĐENJA, ZEMLJA. Stoga se naša baza podataka transformira u sljedeći oblik:

KNJIGA(CIP_BR, UDK_NASLOV, UDK_AUTOR, NASLOV, ISBN, BR_STRANA, GOD_IZDANJA, IME_AUTORA, PREZIME_AUTORA)

KNJIGA(CIP_BR, UDK_NASLOV, UDK_AUTOR, BR_PRIMJERKA, POSUĐENO)

CIP(ŠIFRA, NAZIV)

AUTOR(IME_AUTORA, PREZIME_AUTORA, GODINA_ROĐENJA, ZEMLJA)

Boyce-Coddova normalna forma

Posljednja normalna forma koju ćemo ovdje spomenuti jest Boyce-Coddova normalna forma (BCNF). Ona je nešto jača nego 3NF, ali nije toliko jača da bi se nazivala četvrtom normalnom formom. Kod nje se također radi o tranzitivnim zavisnostima, no za razliku od 3NF, ona zabranjuje tranzitivnu zavisnost i o ključnim atributima.

Problemi koji nastaju zbog nepoštivanja BCNF slični su onima koji se događaju zbog nepoštivanja prethodnih normalnih formi, pa je stoga nužno poštovati i ovu normalnu formu. Razbijanje tablice koja nije u BCNF na dvije ili više tablica koje su u BCNF radi se na isti način kao i kod 3NF.

U našem primjeru nema narušavanja BCNF, pa ćemo konstruirati poseban primjer u kojem će se narušiti BCNF.

Neka je dana tablica *r(A,B,C)* koja ima ključeve *AB* i *BC*. Neka u njoj vrijedi zavisnost $A \rightarrow C$. Tada ta relacija nije u BCNF zato jer postoji transizivna zavisnost $BC \rightarrow A \rightarrow C$. Stoga ovu relaciju treba rastaviti na dvije tako da se u jednu stave svi atributi koji zavise tranzitivno o ključu zajedno s atributima preko kojih se ostvaruje tranzitivna zavisnost, a u drugu se stavi svi atributi, osim onih koji su bili tranzitivno zavisni o ključu. U našem primjeru to relacija *r* bi se rastavila na *r1(A,C)*, *r2(A,B)*.

***

Normalizacija se može provoditi i dalje. Postoje četvrta, peta i inkluzijska normalna forma, no one su već vrlo specijalne i manja je vjerojatnost da će biti narušene. Osim toga, one se odnose na složenije zavisnosti, kakve često nije jednostavno uočiti ako se vrlo detaljno ne poznaje domena koja se aplicira u bazi podataka. Stoga ih ovdje nećemo izlagati.

2. Indeksi – kako i koliko

Druga važna stvar koja utječe na brz i kvalitetan rad baze podataka jest pravilna upotreba indeksa. Indeksi sami po sebi nisu dio relacijskog modela, koji je teorijski. Oni su nastali zbog praktične potrebe ubrzanja pretraživanja baze podataka. Stoga su i pravila korištenja indeksa praktičnije naravi nego normalizacija i više su vezana uz sustav koji se koristi. Ovdje ćemo govoriti o korištenju indeksa u PostrgeSQL-u, iako se mnoge stvari koje će ovdje biti iznešene mogu primijeniti i na druge postojeće sustave za upravljanje bazama podataka.

Tablice u PostgreSQL bazi podataka su smještene u jednu ili više datoteka. Datoteke su, kao što je to uobičajeno, podijeljene na blokove i kada se vrši čitanje podataka uvijek se u memoriju učitava cijeli blok. U PostgreSQL-u je veličina bloka 8 kB, no to je moguće mijenjati tako da se u datoteci `src/include/config.h` promijeni vrijednost u retku `BLCKSZ`. Sekvencijalno pretraživanje tablice zahtijeva uzastopno čitanje blokova datoteke u

kojima su smješteni podaci tablice koju pretražujemo. Indeksi će ubrzati ovo pretraživanje. Naime, indeks sadrži samo dio podataka iz svakog sloga tablice. Zbog toga se u jednom bloku indeksa nalaze podaci iz znatno više slogova tablice negoli u samoj tablici, pa će pretraživanje indeksa zahtijevati čitanje znatno manjeg broja blokova datoteke negoli pretraživanje same tablice. Stoga je jasno da indekse valja kreirati tako da sadrže onaj dio podataka iz sloga po kojem se očekuju česta pretraživanja.

Treba napomenuti da indeksi igraju značajnu ulogu i u definiranju ograničenja nad podacima koji se nalaze u tablici, no o toj upotrebi indeksa bit će više riječi u sljedećem poglavlju.

Prije nego kažemo kako treba kreirati indekse opisat ćemo kako izgledaju indeksi u PostgreSQL-u.

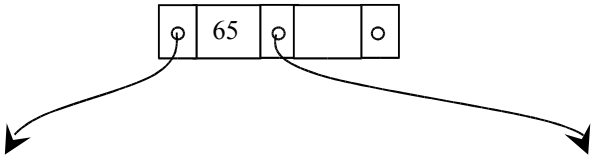
B-tree indeksi

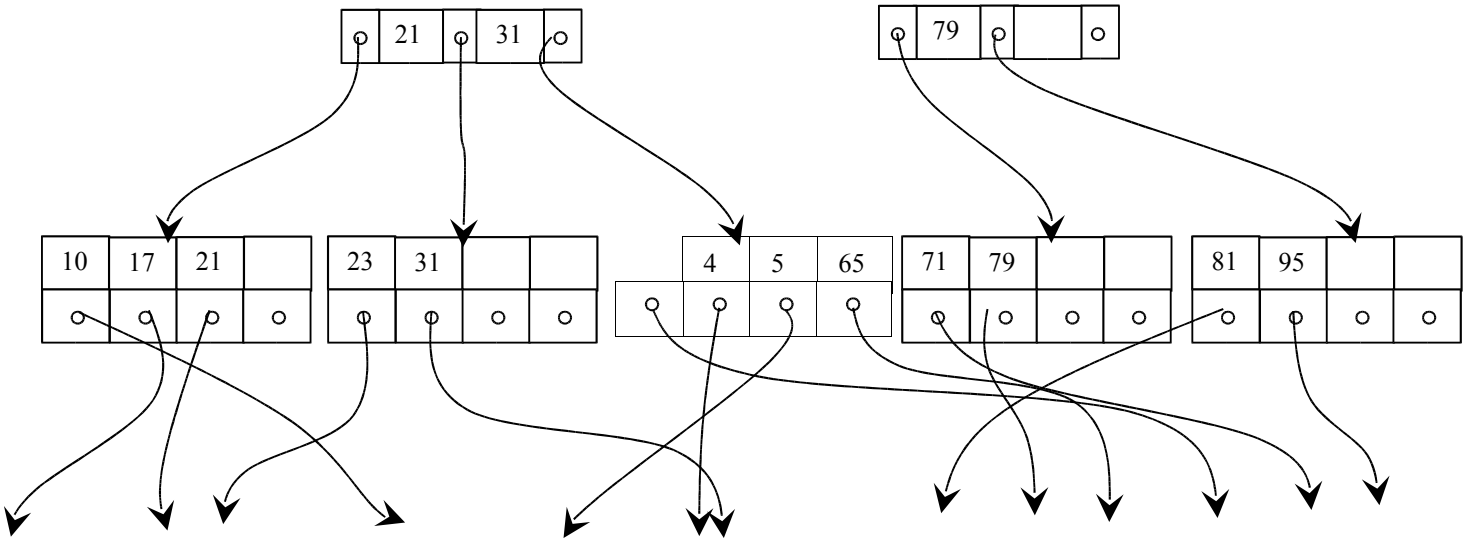
Osnovna vrsta indeksa koji se koriste u PostgreSQL-u jesu B-tree (*balanced tree*) indeksi. Kao što i samo ime kaže, ovdje se radi o stablastim indeksima. To su indeksi koji se danas najčešće koriste u bazama podataka.

B-tree indeks ima strukturu stabla. Čvorovi tog stabla su blokovi datoteke. Listovi u tom stablu, tj. čvorovi koji nemaju djece, sadrže pokazivače na blokove u bazi podataka u kojima se nalaze traženi podaci. U B-stablu svi se listovi stabla moraju nalaziti na istoj razini stabla. Nadalje, listovi B-stabla su generirani na taj način da nikada (osim u slučaju da se indeks sastoji od jednog jedinog bloka) nisu popunjeni manje od polovice.

Unutarnji čvorovi B-stabla imaju drugačiju strukturu od listova. Oni se sastoje od graničnih vrijednosti indeksa i od pokazivača na djecu čvora. Ispred i iza svake granične vrijednosti nalazi se pokazivač. Slijedimo li pokazivač ispred granične vrijednosti, ulazimo u podstablo koje se u potpunosti sastoji od vrijednosti koje su manje od promatrane granične vrijednosti, a ako slijedimo indeks iza granične vrijednosti, ući ćemo u podstablo koje se sastoji od vrijednosti koje su veće od promatrane granične vrijednosti. Na taj način moguće je pronaći list u stablu koji sadrži traženu vrijednost indeksa prolaskom kroz samo jednu rutu od korijena do lista.

Radi lakšeg prikaza pretpostavit ćemo da su blokovi datoteke znatno manji nego što su u stvarnosti. Pretpostavimo da unutarnji čvorovi mogu sadržavati 2 granične vrijednosti (pa time 3 pokazivača), a da listovi mogu sadržavati 4 vrijednosti indeksa s pokazivačima na blokove tablice na kojoj je kreiran indeks. Pretpostavimo da je indeks kreiran nad jednim jedinim atributom tablice, i da su vrijednosti tog atributa brojčane. Pretpostavimo da indeks sadrži vrijednosti 10, 17, 21, 23, 31, 40, 48, 56, 65, 71, 79, 81, i 95. Sljedeća slika prikazuje moguće B-stablo ovog indeksa:





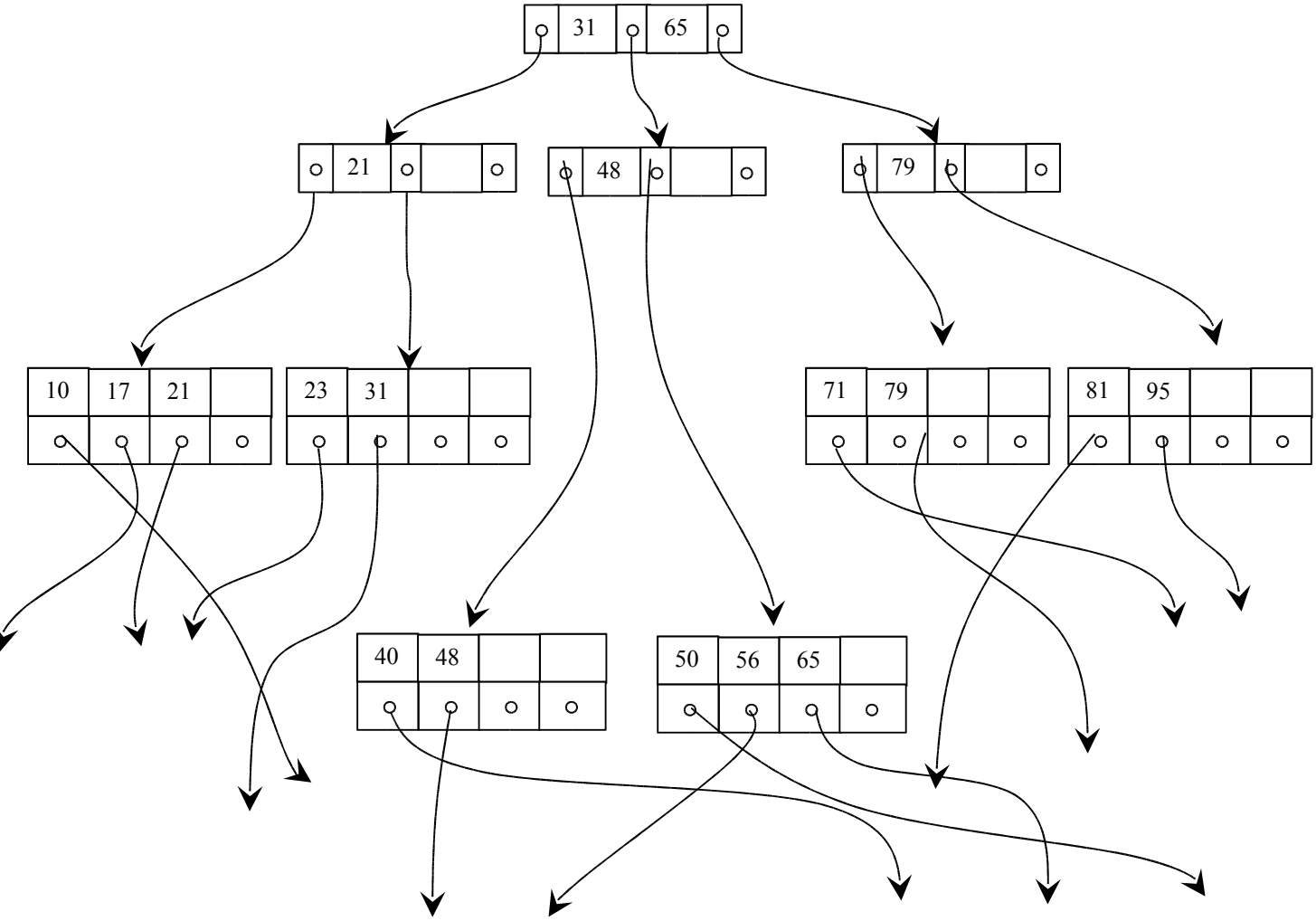
Pretraživanje ovakvog indeksa se vrši tako da se krene od korijena i redom se promatraju granične vrijednosti koje su u njemu navedene. Ako se naiđe na graničnu vrijednost koja je veća od tražene vrijednosti, odmah se prelazi na blok koji je naveden prije te vrijednosti. Ako se pročitaju sve granične vrijednosti zapisane u korijenu, i sve budu manje od tražene vrijednosti, onda se prelazi na blok na koji pokazuje pokazivač iza posljednje granične vrijednosti. Ovaj se postupak ponavlja na bloku na koji smo došli, sve dok ne dođemo na blok koji je list stabla. Kada smo došli na list stabla, njega pretražujemo sekvencijalno. Ako slog s traženom vrijednošću postoji u tablici, on mora biti baš u tom bloku. Npr. tražimo li vrijednost 48 krenut ćemo od korijena. Pročitat ćemo prvu (i jedinu) graničnu vrijednost zapisanu u njemu te kako je tražena vrijednost manja od te granične, krećemo na blok koji pokazuje prvi pokazivač korijena. Čitamo redom pokazivače u bloku u koji smo došli. Oni su oba manja od tražene vrijednosti, pa ćemo slijediti treći pokazivač u tom bloku. I time smo došli u list stabla, koji zaista sadrži vrijednost 48.

Ovakav indeks, osim što sadrži više podataka po bloku, ima stablastu strukturu, što omogućuje da vrijeme potrebno za traženje sloga u indeksu ne raste linearno s povećanjem broja slogova u tablici, već raste logaritamski, što je od presudne važnosti za brzinu rada velikih baza podataka.

Dodavanje novog čvora u indeks se vrši tako da se pretražuje stablo na gore opisani način, dok se ne dođe do lista stabla. Kada se dođe do lista, u njega se umetne nova vrijednost. Može se, međutim, dogoditi da je blok u koji treba staviti novu vrijednost popunjen. U tom slučaju se taj blok dijeli na dva bloka, tako da u prvi blok ulazi prva polovica vrijednosti iz bloka, a u drugu druga. Kako dodajemo u jedan od blokova i novu vrijednost koja u blok nije stala prije dijeljenja, svaki od novih blokova bit će popunjen više od polovice. Naravno, ovu je promjenu potrebno zabilježiti u bloku roditelju podijeljenog bloka, tako da se u njemu doda još jedna granična vrijednost. No, i roditelj bloka može biti popunjen. Tada se i on dijeli na dva bloka, pri čemu u prvi blok ide prva polovica graničnih vrijednosti s pripadajućim pokazivačima, a u drugi blok druga polovica. Ovo dijeljenje se može propagirati prema vrhu stabla sve do korijena. Ako se to dogodi te ako je i korijen popunjen, onda će se korijen stabla podijeliti na dva bloka, a iznad korijena će se kreirati još jedan blok koji će postati novi korijen stabla i koji će pokazivati na blokove na koje je podijeljen stari korijen.

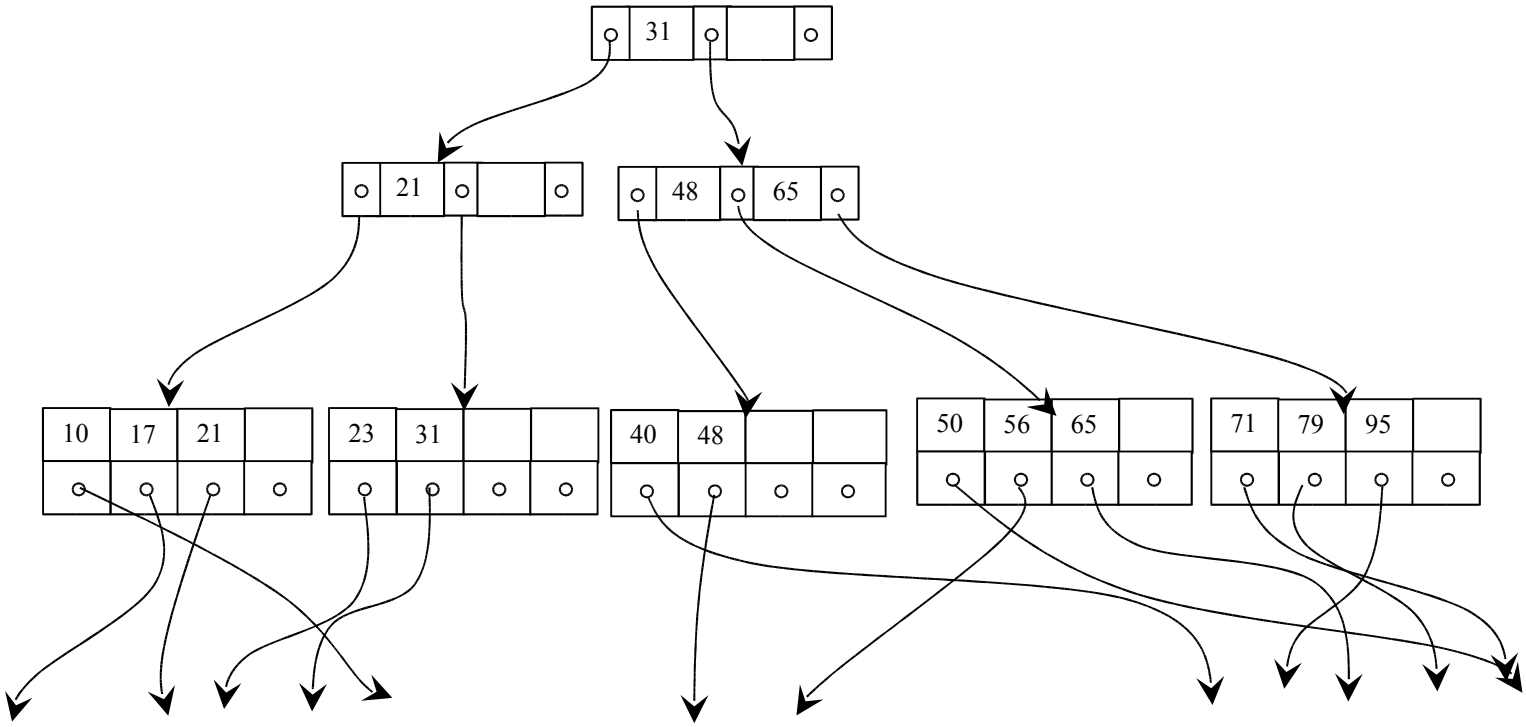
Želimo li tako u naš indeks iz primjera dodati vrijednost 50, krenut ćemo od korijena i slijediti prvi pokazivač. U bloku u koji smo došli slijedit ćemo treći pokazivač i tako doći u treći list

stabla s lijeva. U njega ne možemo staviti novu vrijednost jer je pun, pa ga dijelimo na dva bloka. Točnije, alociramo još jedan blok i posljednju polovicu vrijednosti, zajedno s novom vrijednošću, na početku zapisujemo u taj novi blok. U blok roditelj ovog bloka trebalo bi dodati još jednu graničnu vrijednost – 48, no to nije moguće jer je i taj blok popunjen. Stoga i njega dijelimo na dva bloka, u prvi upisujemo prvu graničnu vrijednost s pripadajućim pokazivačima, a u drugi novu upisujemo graničnu vrijednost 48. Također, potrebno je promijeniti i korijen stabla. tako da mu se doda još jedna granična vrijednost – 31.



Brisanje vrijednosti iz tablice vrši se na sličan način – pretraži se stablo do lista te se iz lista izbriše zadana vrijednost. Može se, međutim, dogoditi da je time blok pao ispod polovice popunjenosti. Tada ćemo taj blok obrisati, a njegove vrijednosti preseliti u prethodni blok. Pri tome se može desiti da se taj blok prepuni, pa da ga je opet potrebno dijeliti na dva. Jasno je da je potrebno ažurirati i roditelja obrisanog bloka. Može se desiti da je brisanjem tog bloka njegov roditelj ostao samo s jednim djetetom, što je neodrživo. Tada se i njegov roditelj briše, a njegovi se pokazivači sele u prethodni čvor na istom nivou stabla. Time se može dogoditi da se prethodni čvor prepuni, pa da ga je potrebno opet dijeliti. S druge strane, ovo se brisanje može propagirati sve do korijena. U tom slučaju može se dogoditi da korijen stabla ostane samo s jednim djetetom. U tom se slučaju korijen stabla briše, a njegovo jedino dijete postaje novi korijen stabla.

Obrišimo iz našeg indeksa iz primjera vrijednost 81. Blok u kojem se ta vrijednost nalazi time će pasti ispod polovice popunjenosti, te ga je potrebno obrisati, a njegove vrijednosti prepisati u prethodni blok. Time je, međutim, roditelj opisanog bloka ostao samo s jednim djetetom, pa i njega treba obrisati, a njegove pokazivače premjestiti u prethodni blok na istom nivou.



Na kraju, promjena indeksiranih vrijednosti u nekom slogu tablice uzrokuje i promjenu indeksa, jer vrijednosti u indeksu moraju uvijek biti sortirane. Ne postoji poseban algoritam za promjenu vrijednosti indeksa, već se vrši brisanje stare vrijednosti i dodavanje nove.

Hash indeksi

Hash indekse ćemo opisati vrlo kratko, jer oni nisu toliko interesantni. Hash indeksi se koriste u vrlo specijalnim situacijama i u većini slučajeva se ne preporuča njihovo korištenje. Naime, hash indeksi ne podržavaju isključivo jednakosno pretraživanje. To znači da se pri korištenju operatora <, >, <=, >=, <> te BETWEEN hash indeks ne može iskoristiti. S druge strane, hash indeks je tek neznatno brži kod jednakosnog pretraživanja od stablastog indeksa. Jedina značajno poboljšanje koje hash indeksi imaju pred stablastim jest kada se koriste pri spajanju dvije ili više tablica.

PostgreSQL koristi takozvano linearno hashiranje pri kojem se povećanjem broja slogova u tablici povećava i broj pretinaca u hash indeksu. Međutim, iako je tome tako, može se dogoditi da ključevi i pripadni pokazivači prijeđu veličinu jednog bloka te da pretinac ima više od jednog bloka. Hash funkcija koja se u PostgreSQL-u koristi je Jenkinsova hash funkcija, opisana na web stranici `http://burtleburtle.net/bob/hash/doobs.html`. Ovdje nećemo detaljnije opisivati linearno hashiranje, jer ovi indeksi nemaju bitne prednosti pred B-tree indeksima i svugdje u PostgreSQL literaturi se sugerira da se izbjegava njihovo korištenje.

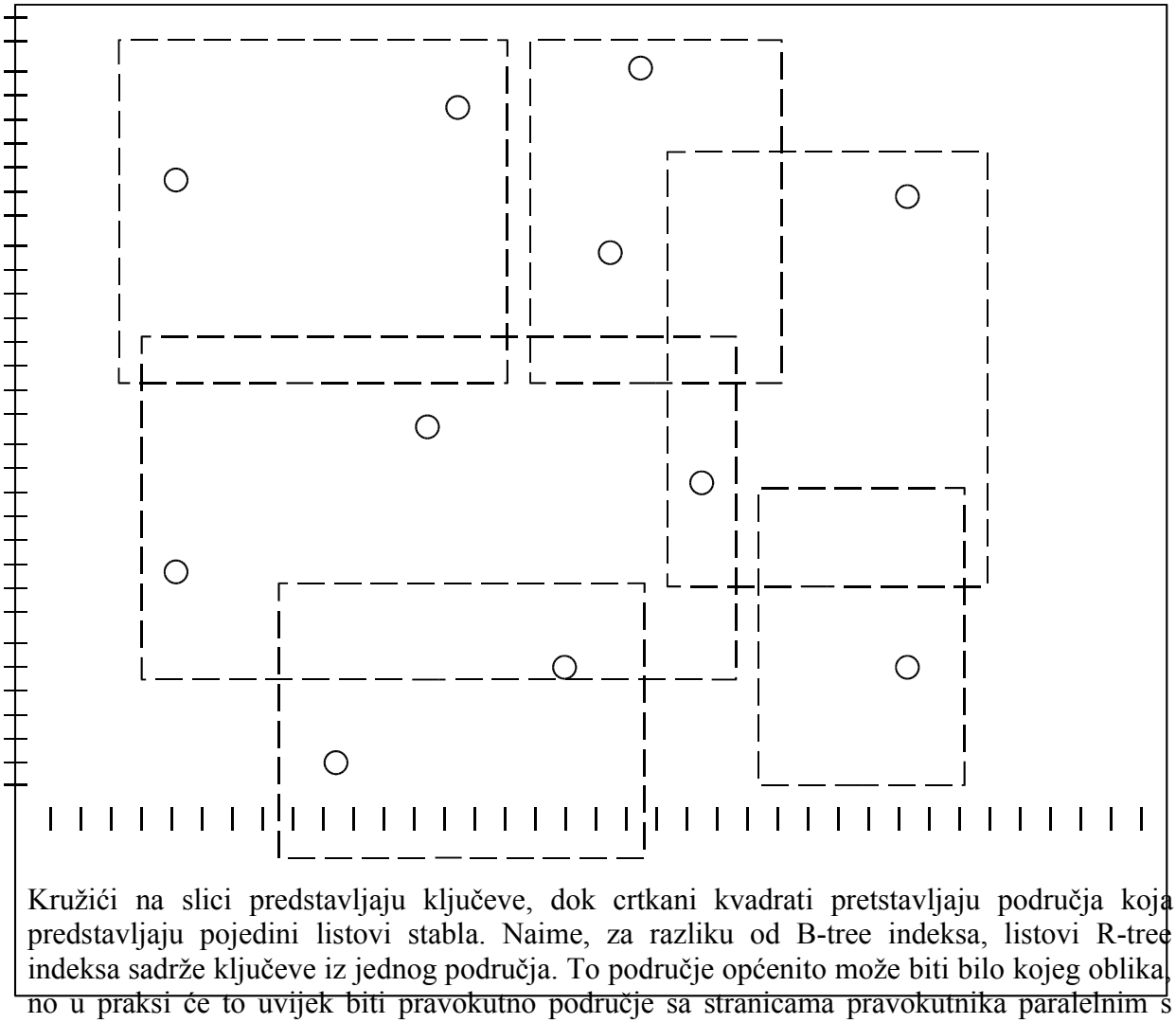
Na kraju napomenimo da je dodatni nedostatak koji ovi indeksi u PostgreSQL-u imaju što se ne mogu definirati preko više od jedne kolone tablice te što ne mogu biti definirani kao UNIQUE indeksi.

R-tree indeksi

U PostgreSQL-u postoji čitav niz prostornih, točnije, ravninskih tipova podataka. Tu spadaju podaci tipa POINT, LINE, LSEG, BOX, PATH, CIRCLE i POLYGON. Uz te su tipove podataka definirani posebni prostorni operatori: <<, <&, &>, >>, @, ~= i &&. Za ove operatore B-tree indeksi neće biti efikasni.

B-tree indeksi djeluju tako da granične vrijednosti indeksa definiraju intervale vrijednosti ključeva. Dakle, može se reći da se kod B-tree indeksa vrijednosti ključeva nalaze na pravcu, u jednoj dimenziji. To ne omogućuje efikasno pretraživanje dvodimenzionalnih podataka. Zbog toga su uvedeni R-tree indeksi (region tree).

Sljedeća slika prikazuje jedan R-tree indeks sa samo dvije razine stabla.



Kružići na slici predstavljaju ključeve, dok crtkani kvadrati pretstavljaju područja koja predstavljaju pojedini listovi stabla. Naime, za razliku od B-tree indeksa, listovi R-tree indeksa sadrže ključeve iz jednog područja. To područje općenito može biti bilo kojeg oblika, no u praksi će to uvijek biti pravokutno područje sa stranicama pravokutnika paralelnim s koordinatnim osima.

Općenito, u R-tree indeksu korijen stabla se definira tako da pokriva cijelo područje ključeva indeksa (pravokutnik pune linije na slici). Djeca korijena će biti definirana tako da pokrivaju manja područja, koja se, općenito mogu preklapati. Tako se isti ključ u R-tree indeksu, za

razliku od B-tree indeksa, može naći u više od jednog lista stabla. Djeca korijena mogu imati svoju djecu, koja područje još više usitnjuju, i tako u dubinu koliko je potrebno.

Ovakvi su indeksi zaista mnogo efikasniji nad prostornim podacima. Treba reći da R-tree indeksi dozvoljavaju i indervalna pretraživanja, pa i intervalna prostorna pretraživanja.

R-tree indeksi u PostgreSQL-u ne mogu biti definirani kao UNIQUE indeksi, a isto tako, mogu biti definirani samo nad jednom kolonom tablice.

GiST indeksi

GiST (Generalized Search Tree) indeksi su u PostgreSQL literaturi najslabije dokumentirani. Oni su najopćenitiji stablasti indeksi koji su danas u teoriji poznati. R-tree indeksi i B-tree indeksi su samo specijalni slučajevi GiST indeksa. Time su B-tree i R-tree indeksi efikasniji od GiST indeksa. No oni ne omogućuju sve što omogućuju GiST indeksi. GiST indeksi omogućuju indeksiranje po kolonama koje imaju korisnički definirane tipove podataka.

GiST indeksi, slično hast indeksima i R-tree indeksima ne mogu biti definirani kao UNIQUE indeksi, ali za razliku od hash indeksa i R-tree indeksa, mogu biti definirani preko više kolona tablice.

***

Sada, kad smo opisali rad svih vrsta indeksa koji se u PostgreSQL-u javljaju, dajmo naputke kako i koliko koristiti indekse.

Prvo pitanje koje se postavlja jest kada koristiti indekse, odnosno kada će oni ubrzati stvar. Ako je u tablici malo slogova, onda će korištenje indeksa uzrokovati usporenje pretraživanja. Na pitanje što je mala tablica nije tako jednostavno odgovoriti. To ovisi o mnogim parametrima. Prvenstveno to ovisi o veličini sloga u tablici, o veličini bloka koji se u bazi podataka koristi, o veličini indeksiranih podataka, o vrsti indeksa koji se koristi i o mnogim drugim parametrima. No, generalno gledano, što je tablica veća, pretraživanje po indeksima donosit će više koristi.

Drugo je pitanje što se korištenjem indeksa gubi. Jasno je da oni odnose dodatni prostor na disku. No, to nije najveći problem. Veći problem, kao što se vidi iz algoritama za B-tree indekse, a slično je i s ostalim vrstama indeksa, jest u znatnom usporenju operacija ubacivanja novih slogova u tablicu, brisanja slogova iz tablice te promjene vrijednosti u ključu.

Zato, iako se preporučuje korištenje indeksa, ne preporučuje se definiranje nepotrebnih indeksa. Neki su naputci kojih bi se trebalo držati sljedeći:

- Broj indeksa po tablici ne bi trebao prelaziti 5. Povećanje broja indeksa usporuje ažuriranje tablice, dodavanje novih slogova te brisanje slogova iz tablice. Naime, te će operacije uzrokovati ažuriranje indeksa definiranih nad tablicom. Povećanjem broja indeksa povećava se broj operacija koje treba izvesti pri dodavanju i brisanju slogova iz tablice te pri promjeni indeksiranih vrijednosti. Broj indeksa neće utjecati na usporenje pretraživanja u tablicama, no može znatno usporiti dodavanje novih slogova u tablicu, brisanje slogova iz tablice te neke promjene podataka u tablici. To treba imati na umu, posebno ako se radi o dinamičnoj tablici kojoj se često mijenja sadržaj.

- Veličina podataka koji su indeksirani ne bi trebala preći desetinu veličine sloga u tablici, a indeks u kojem veličina indeksiranih podataka prelazi 20% veličine sloga za pretraživanje neće dati znatnije ubrzanje, osim ako je tablica vrlo velika. Indeksi, naime, ubrzavaju pretraživanje tako što smanjuju broj blokova koje treba pročitati. Pri tome se čita jedan jedini blok tablice, ali se čita nekoliko blokova indeksa. Druga je stvar koju treba napomenuti da će se razlika u brzini pretraživanja preko indeksa i bez njega povećavati kako raste tablica. S druge strane, povećanjem indeksiranih podataka indeks će postati manje efikasan zbog toga što će se u jednom bloku indeksa moći smjestiti manje podataka, pa će za indeksiranje biti potrebno više blokova.
- Kod složenih indeksa redoslijed kolona u indeksu mora biti onako naveden kako je zgodan za pretraživanje. Naime, kada se radi o složenom indeksu, nikako nije svejedno kojim redoslijedom su upisana polja u nj. Naime, relacijski sustav za upravljanje bazama podatka (RSUBP) znade pretraživati samo po početnom dijelu indeksa. Stoga se dobro mora promisliti kakvi se upiti očekuju na tablici i prema tome treba definirati indekse.
- Treba napomenuti da indeksi u PostgreSQL-u, kao i u mnogim drugim RSUBP-ovima imaju i ulogu implementiranja ograničenja. Naime, organičenja UNIQUE i PRIMARY KEY se ostvaruju preko indeksa koji ne dozvoljavaju višestruke vrijednosti u indeksiranim kolonama. Kod primjene indeksa u ovu svrhu, neki od prethodnih naputaka ne vrijede, ali neki ostaju. Tako, npr. ne vrijede drugi i treći naputak, osim ako se po takvom indeksu, osim što se njime osiguravaju ograničenja, ne želi i pretraživati. S druge strane, definira li se previše indeksa koji osiguravaju ograničenja, usporit će se upisivanje i brisanje slogova jednako kao i kod indeksa namjenjenih pretraživanju.

Na kraju ovog poglavlja napomenimo da je moguće da se tablica vizički, na disku sortira prema određenom indeksu, što će još više ubrzati pristup po tom indeksu. U tu svrhu služi naredba CULSTER. No, prilikom dodavanja novih slogova, te ažuriranja slogaova, neće se automatski vršiti dodavanje sloga prema danom klasteru. Stoga, želimo li održavati sortiranost tablice po indeksu potrebno je s vremena na vrijeme ponavljati naredbu CLUSTER.

3. Ograničenja - bolje definiranje podataka

U PostgreSQL-u postoji nekoliko vrsti ograničenja, koja su različito implementirana, no svima njima je zajednička jedna funkcija – detaljnije definiranje dozvoljenih vrijednosti pojedine kolone ili tablice.

Ograničenje koje je prvo implementirano u RSUBP-ovima jest ograničenje NOT NULL. Ovo ograničenje je implementirano kao dodatni podatak pri definiciji tablice i kao takvo predstavlja neznatno usporenje rada. Naime, ovo se ograničenje provjerava samo kod dodavanja, brisanja i promjene sloga, a sve što se provjerava radi se u glavnoj memoriji, što tek neznatno usporava ove operacije. Stoga se preporuča da se ovo ograničenje koristi uvijek kada se želi izbjeći da kolona sadrži neodređene vrijednosti. Inače, null vrijednosti imaju značajnu ulogu u bazama podataka i ne smije se izbjegavati njihova primjena. One imaju značeje nepostojeće vrijednosti ili nepoznate vrijednosti i ne mogu se nadomjestiti nijednom drugom vrijednošću.

Sljedeće ograničenje, koje omogućuje da se vrijednosti koje se unose u bazu podataka računski provjere jest CHECK ograničenje. Ovo ograničenje je implementirano slično kao i NOT NULL ograničenje – kao dodatni podatak u podacima definicije tablice, no provjera je

izraza znatno zahtjevnija od provjere postojanja podatka. Bez obzira na to, ovo se ograničenje također provjerava u glavnoj memoriji i ne bi trebalo predstavljati problem. Treba napomenuti da velik broj CHECK ograničenja ipak može vidljivo usporiti masovna dodavanja ili promjene slogova, pa kod tablica u koje se podaci dodaju ili mijenjaju u većim grupama treba o tome voditi računa.

Sljedeće je UNIQUE ograničenje. Ono osigurava da se u istoj koloni ne može javiti isti podatak u više slogova. Pri tome je jedna vrijednost izuzeta od pravila, tj. može se javljati više puta, a to je NULL vrijednost. Ovo je ograničenje znatno složenije nego prethodna dva i nije implementirano samo kao dodatni podatak u definiciji tablice. Osim kao podatak u definiciji tablice, ovo je ograničenje definirano kao UNIQUE indeks koji se dodaje tablici. Kao što je već rečeno kada smo govorili o indeksima, to usporava unošenje i brisanje slogova iz tablice, a isto tako usporava promjenu vrijednosti koja je ograničena ograničenjem UNIQUE. Iako nema ograničenja kod broja UNIQUE ograničenja po tablici, s njima treba biti oprezan.

Jače ograničenje od ograničenja UNIQUE je ograničenje PRIMARY KEY. Iako se ponaša slično, ovo ograničenje ima potpuno drugačiju funkciju. Kao što smo govorili u prvom poglavlju, svaka relacija, odnosno tablica ima jedan ili više kandidata za ljuč, od kojih se jedan izdvaja kao primarni ključ. Primarni ključ ima posebnu funkciju, pa se u posljednje vrijeme u SUBP-ovima želi osigurati da će on uvijek imati sve osobine ključa. Da bi se to osiguralo, koristi se PRIMARY KEY ograničenje. Za razliku od UNIQUE ograničenja, na jednoj se tablici može definirati samo jedno PRIMARY KEY ograničenje. Implementacija PRIMARY KEY ograničenja je slična implemenntaciji UNIQUE ograničenja. Ono se definira kao podatak u definiciji tablice, i kao UNIQUE indeks na tablici. Dodatno, ovo ograničenje dodaje i NOT NULL ograničenja na sve kolone koje obuhvaća. Iako PRIMARY KEY ograničenje predstavlja dodavanje jednog indeksa, preporuča se da se na svakoj tablici definira PRIMARY KEY ograničenje.

Na kraju, navodimo posljednje ograničenje, koje je u PostgreSQL-u implementirano posljednje, a to je FOREIGN KEY ograničenje. Ovim se ograničenjem definira referencijalni integritet baze podataka. Ovo je jedino ograničenje koje se odnosi na dvije tablice baze podataka. Sva su ostala ograničenja definirana unutar jedne same tablice. U relacijskom modelu tablice nisu međusobno nepovezane, već se međusobno povezuju preko zajedničkih podataka. Zajednički su podaci definirani na takav način da uvijek u jednoj od te dvije tablice oni predstavljaju *primarni ključ*. U drugoj tablci oni služe isključivo povezivanju s prvom tablicom, pa se zovu *vanjski ključ*. Vanjski se ključ sloga u sekundarnoj tablici mora podudarati s nekim primarnim ključem primarne tablice u vezi ili se mora sastojati od samih NULL vrijednosti. U primarnoj se tablici primarni ključ definira, kao što smo već vidjeli ograničenjem PRIMARY KEY. U sekundarnoj će se tablici on definirati kao FOREIGN KEY. U jednoj se tablici može definirati više FOREIGN KEY ograničenja. Ovo se ograničenje implementira isključivo kao podatak u definiciji tablice, ali znatno više utječe na brzinu rada s tablicom od ostalih ograničenja koja se tako definiraju. Uvođenjem ovog ograničenja može se znatno usporiti operacija unošenja novih slogova u tablicu, kao i operacija promjene vrijednosti vanjskog ključa. Naime, kako bi se provjerilo zadovoljavaju li unešeni podaci ili promjenjeni podaci FOREIGN KEY ograničenje, potrebno je pretražiti primarnu tablicu veze prema primarnom ključu. Jasno je da će se to pretraživanje vršiti po UNIQUE indeksu, kojeg je definiralo PRIMARY KEY oograničenje, no ako je primarna tablica velika, ova provjera može potrajati. Posebno to može biti značajno kod masovnog unosa novih slogova, kada se mora provjeriti svaki unešeni slog. Na kraju, veći broj FOREIGN KEY ograničenja u tablci uzrokovat će kod unosa novih slogova pretraživanje više različitih tablica. Stoga nekoliko ovakvih ograničenja može značajno usporiti unos novih slogova u tablicu čak i ako primarne

tablice s kojima je tablica povezana nisu jako velike. Kod baza podataka u kojima se podaci mijenjaju većom brzinom može se razmišljati da se odustane od automatske provjere referencijalnog integriteta, odnosno od definiranja FOREIGN KEY ograničenja, čime se može znatno ubrzati rad takve baze podataka.

4. Transakcije - kad nismo sami s RDBMS-om

Transakcije su jedan od osnovnih koncepata koje RSUBP mora zadovoljavati. Na početku ovog poglavlja treba reći da rad s transakcijama znatno usporava rad SUBP-a. Upravo zbog toga MySQL postiže u nekim testovima toliko dobre rezultate. Međutim, svaki bi inženjer baze podataka morao u svakom slučaju odbiti raditi s alatom koji ne podržava transakcijski rad ili koji čak i omogućuje da se transakcijski rad izbjegne. Transakcije su previše važan aspekta rada RSUBP da bi se mogle mijenjati za brzinu rada. Usudio bih se reći da alat koji ne koristi transakcije ili u kojem ih je moguće izbjeći nije pravi RSUBP. Upravo zato PostgreSQL poznaje samo jedan režim rada – transakcijski.

Zašto su transakcije toliko važne? One osiguravaju da u svakom trenutku, bez obzira na padove sustava, nedovršene operacije, konkurentni rad, baza podataka bude konzistentna s obzirom na sve definicije i ograničenja koja su u njoj definirana. Odustajanjem od transakcija prihvaća se opasnost da baza podataka uđe u nekonzistentno stanje i počinje vraćati neispravne, pa i besmislene odgovore.

Same transakcije imaju nekoliko uloga. Prva i najvažnija je osiguranje cjelovitosti operacija. Operacije moraju biti u bazi podataka definirane na takav način da prevode bazu podataka iz konzistentnog stanja u konzistentno stanje. Treba napomenuti da vrlo često to nije moguće izvesti jedno SQL naredbom, već je za to potrebno više sukcesivnih SQL naredbi.

Druga je važna uloga transakcija osiguranja konzistentnosti baze podataka pri konkurentnom radu. Ovdje se ogleda uska veza transakcija i zaključavanja podataka u bazi podataka. Kada s bazom podataka istovremeno radi više korisnika, treba osigurati da baza podataka u svakom trenutku svakom korisniku vraća konzistentne odgovore. Ovo se postiže upravo pomoću transakcija i zaključavanja, te pomoću snimaka (snapshot) baze podataka. Posljednji se koriste pri propitivanju baze podataka. Naime, bez obzira na promjene koje drugi korisnici mogu napraviti na bazi podataka tijekom rada upita (koji može raditi i duže vrijeme), podaci koje upit koristi moraju biti konzistentni – onakvi kakvi su bili na početku rada upita. To se osigurava tako da se napravi snimak dijela baze podataka koji se u upitu koristi i taj se snimak baze podataka koristi pri radu upita. Na taj se način izbjegava zaključavanje podataka u bazi podataka samo radi njihovog čitanja. Zaključavanje služi osiguranju od pristupa nekom podatku tijekom njegove promjene. Transakcija u kojoj se promijeni neki podatak u bazi podataka taj podatak zaključa tako da nitko drugi ne može mijenjati taj podatak, ali ni pristupiti njegovoj novoj vrijednosti sve dok transakcija koja ga je zaključala ne završi. Zbog toga je važno da se transakcija ne definira kao jedna SQL naredba, već kao niz SQL naredbi nakon kojih će baza podataka biti u konzistentnom stanju.

Zaključavanje međutim može dovesti do drugih problema – do čekanja na resurse, pa i do deadlockova i do livelockova. Naime ako jedan proces započne transakciju i promijeni neki podatak, on će biti zaključan do kraja transakcije i nedostupan svim ostalima koji ga žele promijeniti. Sve će transakcije koje žele mijenjati taj podatak morati čekati da se taj podatak otključa i tek mu onda pristupiti. Sljedeća slika prikazuje dvije transakcije kako rade paraleleno. Prva zahtjeva slogove A i B, a druga slogove B i C.

TRANSAKCIJA 1	TRANSAKCIJA 2
ZAKLJUČAJ SLOG A	ZAKLJUČAJ SLOG B
ZAKLJUČAJ SLOG B	ZAKLJUČAJ SLOG C
.	

Transakcija 1 neće moći zaključati slog B, jer je taj slog već zaključan od transakcije 2. Stoga će transakcija 1 morati stati i sačekati dok transakcija 2 završi i oslobodi slog 2.

Tu se, međutim javlja problem deadlockova. Sljedeći primjer prikazuje jednostavni slučaj deadlocka.

TRANSAKCIJA 1	TRANSAKCIJA 2
ZAKLJUČAJ SLOG A	ZAKLJUČAJ SLOG B
ZAKLJUČAJ SLOG B	ZAKLJUČAJ SLOG A

U ovom će primjeru u drugom koraku transakcija 1 stati jer neće moći zaključati slog B. Transakcija 1 će čekati da završi transakcija 2 te oslobodi slog B. Međutim, transakcija 2 će također stati, čekajući da završi transakcija 1 i oslobodi slog A. Dakle, ove će se dvije transakcije međusobno čekati i nikada neće završiti. Prekine li se bilo koja od ove dvije transakcije, druga će moći normalno nastaviti s radom. Tako nastaje deadlock. Naravno, ovo je najjednostavniji primjer deadlocka. U deadlocku može sudjelovati i više od dvije transakcije. PostgreSQL može, bez obzira na njegovu složenost, detektirati deadlock i razriješiti ga. Razrješuje ga tako da poništi transakciju koja sudjeluje u deadlocku, a kojoj je prošlo najmanje vremena od njenog početka, te tu transakciju pokrene iz početka. Tako se oslobađaju svi slogovi koje je ona zaključala, te se na taj način razrješava deadlock. Razrješavanje deadlockova će, međutim, usporiti rad pojedinih transakcija. Stoga je, generalno, najbolje izbjegavanje deadlockova. U tu se svrhu treba pridržavati sljedećih pravila:

- Ne praviti transakcije koje traju više nego je potrebno.
- Podatke uvijek koristiti u istom redoslijedu.
- Zaključavati minimalni potrebni broj podataka.
- U transakcijama koje koriste veći broj tablica koje se ažuriraju u više navrata može se koristiti SELECT FOR UPDATE naredba kako bi se na početku transakcije zaključavli svi potrebni podaci.

Treba napomenuti da PostgreSQL pretpostavlja da mu je svaka SQL naredba jedna transakcija, no to se može promijeniti. Ako se navede naredba BEGIN, onda će transakcija trajati sve od te naredbe, pa do sljedeće naredbe COMMIT. Naravno, ovo se prepostavljeno ponašanje može promijeniti tako da se promijeni parametar AUTOCOMMIT na OFF. To se može napraviti u datoteci `postgresql.conf` ili pak u samoj aplikaciji postavljanjem parametra

SET AUTOCOMMIT OFF

Ako se napravi ovo, PostgreSQL neće smatrati da je transakcija dovršena jednom SQL naredbom, već će transakcija trajati do prve naredbe COMMIT ili pak do kraja rada procesa u kojem se naredbe izvršavaju. Jednostavnije rečeno, ako se autocommit parametar isključi, neće biti potrebno navoditi BEGIN naredbu da bi se ušlo u transakciju s više naredbi. Ako se ovo napravi, PostgreSQL će raditi na ačin na koji radi Oracle.

U imalo važnim programima, preporuča se da se ne koristi standardni PostgreSQL mod, već da se transakcije definiraju tako da prevode bazu podataka u konzistentno stanje. Dakle, kao što je tako često slučaj, i predugačke i prekratke transakcije mogu prouzročiti probleme. Prekratke transakcije su one koje završavaju prije negoli je baza podataka dovedena u konzistentno stanje. Pri tome se može dogoditi da baza podataka ostane u nekonzistentnom stanju. Predugačke pak transakcije povećavaju mogućnost deadlockova, koji onda zahtijevaju ponavljanje pojedinih transakcija, pa smanjuju brzinu rada sustava.

Kako bi se maksimalno smanjilo čekanje, u SUBP-ovima se smanjuje granuitet zaključavanja. Radi se o tome da se pokušava zaključati što manji komad podataka. PostgreSQL omogućuje zaključavanje na razini sloga i na razini tablice. Nažalost, on nema, kao što danas već polako postaje standard, zaključavanje na razini kolone u tablici. Iako se u većini slučajeva preporučuje standardno zaključavanje na razini sloga, ima slučajeva kada će zaključavanje na razini tablice biti bolje. U svakom slučaju, kritične operacije koje zahtijevaju ekskluzivnost pristupa, kao što su backup baze podatka, replikacija, VACUUM naredba, REINDEX naredba i sl., bit će sigurnije izvedene ako se koristi zaključavanje na razini tablice. Također, operacija ažuriranja slogova (UPDATE) koja odjednom obuhvaća velik dio slogova tablice (recimo, preko 70%) radit će bolje ako se umjesto zaključavanja na razini sloga koristi zaključavanje na razini tablice.

PostgreSQL ima više vrsta zaključavanja. Najslabiji je ACCESS SHARE. To zapravo i nije pravo zaključavanje. Zaključani podaci ostaju dostupni ostalim transakcijama u potpunosti. Ovo je zapravo naziv za pristup podacima bez zaključavanja. Tako podacima pristupa naredba SELECT, tj. upit koji ne mijenja podatke kojima pristupa, već ih samo čita. Kao što je već rečeno, konzistentnost pri čitanju ostvaruje se preko snimaka (snapshootova) baze podataka, a ne preko zaključavanja.

Naredba SELECT FOR UPDATE koristi sljedeću razinu zaključavanja. Ona zapravo ima ulogu najave zaključavanja, a ne pravog zaključavanja. Ovo će zaključavanje isključiti samo mogućnost najzahtjevnijih naredaba koje zaključavaju tablicu s ROW SHARE zaključavanjem, a koje su spomenute ranije.

Naredbe koje mijenjaju podatke, kao što su INSERT, UPDATE i DELETE koriste ROW EXCLUSIVE zaključavanje. Ovo zaključavanje onemogućuje da ove operacije nad istim podacima napravi bilo koja druga transakcija sve dok transakcija koja ih je zaključala ne završi.

Naredba VACUUM zahtjeva sljedeću razinu zaključavanja, koja se već odnosi na cijelu tablicu, a ne na pojedine slogove. To je tzv. SHARE UPDATE EXCLUSIVE. Ova vrsta zaključavanja će onemogućiti bilo kakvo mijenjanje bilo kojeg podatka u tablici. Uz ovo je zaključavanje još uvijek moguće čitanje podataka iz tablice, ali i naznačivanje zaključavanja naredbom SELECT FOR UPDATE.

Sljedeća razina zaključavanja koja ne dozvoljava niti ROW SHARE zaključavanje od strane druge transakcije. Dakle, neće biti moguć SELECT FOR UPDATE, ali običan će SELECT još uvijek biti moguć. Ova se razina zaključavanja zove EXCLUSIVE.

Najjače zaključavanje vrši se pomoću ACCESS EXCLUSIVE zaključavanja. Ovim se zaključavanjem onemogućuje bilo koja operacija na tablici, pa čak i čitanje podataka.

Još je jedan vrlo važan aspekt zaključavanja – *izolacija transakcija*. Izolacija transakcija definira kako će se pomoću transakcija razrješavati neke neželjene situacije. Sljedeće se neželjene situacije razrješavaju pomoću zaključavanja:

- **Izgubljene promjene –** To se događa kada jedna transakcija prebriše promjene druge transakcije
- **Nepotvrđene zavisnosti** – Ova se situacija još naziva i **prljavo čitanje** (*dirty read*), a događa se kada jedna transakcija čita nepotvrđene podatke neke druge transakcije.
- **Inkonzistentna analiza** – Situacija kada jedna transakcija čita neki podatak dva ili više puta, ali između dva čitanja neka druga transakcija promijeni taj podatak. Ova se situacija još naziva i **neponovljivo čitanje** (*nonrepeatable read*).
- **Fantomsko čitanje** *(Phantom read*) – Ova se situacija događa kada jedna transakcija promijeni sve podatke u nekom opsegu u tablici, a nakon toga druga transakcija doda novi slog u tom opsegu u tablicu. Nakon toga, prva transakcija dobija fantomski slog koji nije obradila u prvoj promjeni, a trebala je.

PostgreSQL ima dvije razine izolacije transakcija i prema tome koja se od njih koristi izbjegavaju se neke od gore navedenih neželjenih situacija. READ COMMITED i SERIALIZABLE. PostgreSQL pretpostavlja mod READ COMMITED. Oba ova moda će rješavati situaciju nepotvrđenih zavisnosti, odnosno prljavog čitanja. Glavna razlika je u tome što drugi mod ne dozvoljava izgubljene promjene dok prvi dozvoljava. Pretpostavimo da neka transakcija želi promijeniti neki slog tablice, no taj je slog već zaključan za promjenu od druge transakcije. Tada će u READ COMMITED modu prva transakcija stati i čekati da se slog oslobodi. Kad se slog oslobodi, provjerit će da on nije u međuvremenu obrisan ili promijenjen tako da više ne zadovoljava uvjet promjene prve transakcije. Ako to nije slučaj, prva će transakcija promijeniti dani slog i time prebrisati promjene koje je napravila transakcija koja je prva zaključala slog. U slučaju da je slog obrisan ili da više ne zadovoljava uvjet, prva transakcija neće napraviti promjenu. Na taj način moguće su sve neželjene situacije osim prljavog čitanja. S druge strane, SERIALIZABLE mod će u gornjem slučaju javiti pogrešku i transakcija će se zaustaviti i poništiti. Dakle, oba moda imaju svoje prednosti i mane, kojih korisnik mora biti svjestan. SERIALIZABLE mod osigurava potpunu konzistentnost baze podataka, no treba napomenuti da se kod tog moda može desiti da neke transakcije budu automatski poništene. PostgreSQL ne pokreće te transakcije automatski ponovo, pa aplikacija koja koristi ovaj mod treba imati ugrađenu provjeru uspješnosti operacije i mogućnost njenog ponavljanja.

Mod izolacije transakcija se može mijenjati u `postgresql.conf` datoteci navođenjem parametra

DEFAULT_TRANSACTION_ISOLATION ='serializable'

ili u samoj aplikaciji s

SET DEFAULT_TRANSACTION_ISOLATION TO 'serializable'

Transakcije se odnose samo na naredbe QL-a (SELECT) i DML-a (UPDATE, INSERT, DELETE). Natredbe DDL-a (DROP i CREATE) nisu transakcijske naredbe i one odmah utječu na bazu podataka kada se izvrše. To također znači da one ne mogu biti poništene naredbom ROLLBACK.

Na kraju, treba napomenuti da PostgreSQL ne podržava ugnježđenje transakcije, tj. ne dozvoljava transakciju unutar transakcije. Autor ovog teksta misli da to nije neki veći nedostatak i nije u praksi naišao na situaciju kada bi ugnježđene transakcije bile nužne za obavljanje nekog posla unutar baze podataka.

5. Funkcije i pogledi – kako pojednostavniti pokretanje niza naredbi

Pogledi (*views*) su vrlo star način da se u SQL-u materijaliziraju upiti. To znači da se definiranjem pogleda SQL upit može u SQL-u promatrati kao materijalni objekt (tablica) i s njim se može tako i raditi. Pogledi, međutim, nikada ne mogu postati ekvivalentni tablicama. Naime, poznato je da svi SQL upiti ne moraju biti takvi da je moguće jednoznačno ažuriranje pogleda kojeg oni definiraju. Razni proizvođači SUBP-ova ulažu znatne napore da bi u svojim alatima što je više moguće proširili klasu pogleda koji se mogu ažurirati.

U PostgreSQL-u se, prema SQL standardu, pogledi definiraju pomoću CREATE VIEW naredbe. Nažalost, mora se napomenuti da je PostgreSQL još uvijek nema direktno ažuriranje nikakvih pogleda. U PostgreSQL-u se umjesto ažuriranja pogleda nude pravila (RULES) preko kojih se to može isprogramirati. Ovo nikako nije zadovoljavajuće niti praktično za korisnike PostgreSQL-a, no to je trenutno sve što PostgreSQL nudi.

Funkcije su objekti baze podataka koji omogućuju da se više SQL naredbi poveže u jednu cjelinu, te da se pokreću zajedno jednom naredbom. Funkcije su u RSUBP-ovima znatno novijeg datuma nego pogledi. U postgreSQL-u se funkcije definiraju naredbom CREATE FUNCTION. Unutar te se naredbe može definirati niz naredbi koje se izvršavaju. Taj niz naredbi može biti definiran u programskim jezicima C, SQL i plpqsql, kao i u nekom drugom, korisnički definiranom proceduralnom jeziku.

Funkcije su implementirane tako da se njihova definicija zapisuje uz ostale definicije objekata baze podataka, kao što su tablice, indeksi i sl., te se iz baze podataka čita kada je to potrebno.

6. Okidači i pravila - baza podataka postaje aktivna

Okidači (*triggeri*) su mehanizmi koji su novijeg datuma u RSUBP-ovima. Oni omogućuju da baza podataka neke poslove vrši sama, bez potrebe njihovog ručnog pokretanja. Okidači su objekti u bazi podataka koji omogućuju da se neke operacije same pokreću kada nastupi određeno stanje u bazi podataka. Dakle, određeno stanje baze podataka se detektira i onda okidač koji je vezan uz to stanje pokreće unaprijed određenu funkciju. Funkcije koje izvode okidači mogu se pisati u programskom jeziku C ili u nekom drugom programskom jeziku, dok, trenutno, pisanje funkcija okidača u SQL-u u PostgreSQL-u nije moguće. Da bi se okidač uopće mogao definirati, najprije je potrebno definirati funkciju koja ima izlazni tip *trigger*, koju će okidač okidati.

Okidači se u PostgreSQL-u definiraju nad tablicama baze podataka. Vežu se uz naredbe DML-a, tj. uz INSERT, UPDATE i DELETE naredbe. Nad istom naredbom iste tablice može biti definirano više od jednog okidača. U tom će se slučaju oni okidati abecednim redoslijedom. Treba primijetiti da pisanje okidača u PostgreSQL-u nije jednostavno. Preporučuje se da se pišu u programskom jeziku C. Funkcije okidača imaju posebnu strukturu podataka kroz koju im se upućuju parametri i ne prihvaćaju parametre na uobičajen način kao ostale PostgreSQL funkcije.

Drugi mehanizam, sličan okidačima, jesu pravila (RULES). Pravila u PostgreSQL-u pokrivaju daleko širu upotrebu i daleko su jednostavnija za definiranje. Slično kao pokazivači, pravila se definiraju nad tablicom. Mogu biti definirani nad INSERT, UPDATE i DELETE naredbom. Međutim, pravilo može biti definirano i nad SELECT naredbom. Isto tako, za razliku od okidača, koji mora biti definiran nad tablicom, pravilo može biti definirano i nad pogledom. Još je jedna razlika između pravila i okidača što okidač nadopunjuje naredbu nad kojom je definira, a pravilo je nadopunjuje ili zamjenjuje. Dakle, kod okidača će se naredba nad kojom je okidač definiran izvesti i još će se dodatno izvesti i sam okidač. Pravilo može raditi na isti način kao i okidač, ali isto tako može i u potpunosti zamijeniti naredbu, tako da se ona ne izvodi već da se izvodi samo pravilo umjesto nje. Tijelo pravila, odnosno naredbe koje definiraju što će pravilo raditi, jesu SQL naredbe SELECT, INSERT, DELETE, UPDATE ili pak naredba NOTIFY. Jedini je nedostatak pravila što nije moguće definirati da se pravilo izvodi prije ili poslije naredbe na koju se odnosi i nije moguće definirati da se izvodi na razini promijenjenog sloga, već samo na razini cijele naredbe. Na kraju treba napomenuti da se pravila ne definiraju tako da se izvode prije ili poslije originalne naredbe, već se ona na određeni način konkateniraju s originalnom naredbom i izvode se zajedno kao jedna jedina naredba.

7. Autentifikacija i enkripcija - nije sve za svakoga

Autentifikacija je vrlo važan, moglo bi se reći osnovni oblik kojim se osiguravaju podaci od neželjenog pristupa. Autentifikacijom se definira koji korisnici i korisničke grupe imaju ovlasti izvršavati neke operacije nad određenim objektima baze podataka. Prvi korak je definicija načina prepoznavanja korisnika baze podataka. U PostgreSQL-u način na koji se prepoznaju korisnici baze podataka zapisuje se u datoteku `pq_hba.conf`. U toj se datoteci definira kako će se prepoznavati korisnici koji pristupaju bazi podataka s lokalnog računala, a kako oni koji pristupaju preko mreže (ODBC-om) ili nekako drugačije. Sljedeće su mogućnosti definiranja načina autentifikacije korisnika:

- *trust* – PostgreSQL neće pitati za korisničko ime ili lozinku, već će pretpostaviti da je operacijski sustav izvršio autentifikaciju korisnika, te će preuzeti korisničko ime koje korisnik ima pri pristupu operacijskom sustavu kao korisničko ime u bazi podataka. Ovo je pretpostavljeni način rada PostgreSQL-a.
- *reject* – Omogućuje da se nekom korisniku u potpunosti zabrarani pristup bazama podataka.
- *password* – Ovo je opcija suprotna opciji *trust*. Ako je ova opcija navedena, PostgreSQL neće preuzimati korisnike od operacijskog sustava već će kod pristupa korisnika sam tražiti navođenje korisničkog imena i lozinke.

- *crypt* - Ova je opcija jednaka opciji *password*, ali je posebno prilagođena autentifikaciji preko mreže. Naime, pri ovoj opciji lozinka neće mrežom putovati u čitljivom obliku, već će biti kriptirana, pa tako i nečitljiva.
- *krb4, krb5* – Autentifikacija preko sustava Kerberos 4 i Kerberos 5.
- *ident* - Ova opcija definira da se prilikom udaljenog pristupa sa stroja s ispravnim IP-om koristi tzv. ident mapa.

Sada kad je definiran način prepoznanja korisnika, treba provesti definiranje korisnika i korisničkih grupa. U PostgreSQL-u se mogu definirati korisnici i korisničke grupe, pomoću naredbi CREATE USER i CREATE GROUP te korisnici učlanjivati u grupe pomoću naredbe ALTER GROUP.

Dobra je praksa da se korisnicima ne pridodaju nikakve zasebne ovlasti, već da se korisnici učlanjuju u korisničke grupe, a da se korisničkim grupama određuju ovlasti nad objektima bazep podataka. Tako se osigurava da dodavanje novog korisnika neće uzrokovati potrebu ponovnog definiranja ovlasti novog korisnika nad svim objektima baze podataka, nego samo uključivanje tog korisnika u njemu adekvatne korisničke grupe te implicitnu definiciju njegovih prava na taj način.

Naredbe kojima se grupi ili korisniku dozvoljavaju, odnosno onemogućuju, određene operacije nad objektima baze podataka jesu GRANT i REVOKE. Ovdje treba napomenuti da osim definiranih korisničkih grupa postoji jedna predefinirana korisnička grupa, koja se zove PUBLIC i kojoj pripadaju svi korisnici koji su definirani u bazi podataka. Pomoću ove grupe se mogu definirati općenite ovlasti koje želimo pridružiti svim korisnicima baze podataka. Objekti kojima se mogu definirati ovlasi jesu tablica (TABLE), baza podataka (DATABASE), funkcija (FUNCTION), brojni red (SEQUENCE), jezik (LANGUAGE) i shema (SCHEME). Tim se objektima mogu dodijeliti ili onemogućiti različite ovlasti. Najčešće se o ovlastima govori u terminima tablice, pa ćemo i mi ovdje tako govoriti. Ovlasti su:

- SELECT – Mogućnost čitanja podataka iz tablice.
- INSERT – Mogućnost umetanja novih slogova u tablicu.
- UPDATE – Mogućnost promjene postojećih podataka u tablici.
- DELETE – Mogućnost brisanja podataka iz tablice.
- RULE – Mogućnost definiranja pravila na tablici.
- REFERENCES – Mogućnost definiranja stranih ključeva u tablici.
- TRIGGER – Mogućnost definiranja okidača nad tablicom.

Sljedeće se ovlasti dodijeljuju na razini baze podataka i definiraju koje koristi korisnička grupa ili korisnik imaju pri definiranju objekata baze podataka.

- CREATE – Mogućnost kreiranja novih objekata baze podataka.
- TEMPORARY – Mogućnost kreiranja privremenih objekata u bazi podatka.

Sljedeća se ovlast odnosi samo na funkcije u bazi podataka:

- EXECUTE – Mogućnost izvršavanja funckije
- USAGE – Ovime se definira koje proceduralne jezike korisnička grupa ili korisnik može koristiti pri definirnjanju funkcija.

Umjesto imena pojedinačne ovlasti može se navesti ALL PRIVILEGES i tako jednom naredbom korisničkoj grupi ili korisniku omogućiti ili zabraniti sve ovlasti nad objektom.

U svakoj bazi podataka i u svakom objektu baze podataka posebno se ističe jedan korisnik – vlasnik baze podataka ili objekta u njoj. Vlasnik ima povlašten status i mogućnost neograničenog pristupa podacima. Ako se drugačije ne definira, vlasnik će objekta imati sve ovlasti nad tim objektom, dok drugi korisnici neće imati nikakve ovlasti nad njim.

Jasno je da su dodijeljene ovlasti pojedinog korisnika kumulativne, tj. korisnik ima sve ovlasti koje su dodjeljene njemu osobno, kao i sve ovlasti koje su dodijeljene svakoj grupi kojoj on pripada. Treba napomenuti da REVOKE naredba ne radi kao naredba DENY za pravila u operacijskom sustavu. REVOKE je samo naredba koja poništava neku GRANT naredbu. Ovlast koja je korisniku oduzeta naredbom REVOKE imat će potpuno isti status kao ovlast koja tom korisniku nije bila ni dodijeljena.

Autentifikacija je osnova zaštite podataka, no ona ne može zaštititi podatke kada oni putuju mrežom, niti ih može zaštititi od neovlaštenog pristupa podacima izravno, bez posredstva RSUBP-a. To se može napraviti jedino tako da se podaci učine nečitljivima neovlaštenim korisnicima. Tome služi ekripcija.

Tri su mogućnosti koje Linux nudi za siguran pristup bazi podataka s udaljenog računala, odnosno preko mreže:

- SSL – Ako se PostgreSQL kompilira s opcijom *–with-ssl* onda će u nj biti ugrađena vlastita SSL podrška i onda će psql klijent uvijek pristupati bazi podataka preko SSL protokola.
- SSH –Koristi se kada se podaci šalju preko mreže na udaljeno računalo.
- Stunnel – Mogućnost enkriptiranog prijenosa podataka od poslužioca do klijenta preko definiranog tunela.

Inače sam PostgreSQL nema mogućnosti da SUBP podatke sam enkriptira i enkriptirane ih stavlja u bazu podataka.

Treba napomenuti da će enkripcija donekle usporiti rad, odnosno prijenos podataka. Zavisno od metode, to može biti do 10% usporenja. No uz današnju brzinu rada računala, smatra se da je usporenje koje enkripcija uzrokuje zanemarivo, pa se svakako preporuča da se, posebno kod prijenosa podataka po mreži, koristi prijenos osiguran enkripcijom.

Literatura

[1] *: PostgreSQL 7.4 *Static Documentation*, www.postgresql.org
[2] *: PostgreSQL 7.4 *Interactive Documentation*, www.postgresql.org
[3] *: *Programming a Microsoft SQL Server 2000 Database Workbook*, Microsoft Corporation, 2000
[4] Garcia-Molina, H; Ullman, J.D.; Widom, J: *Database System Implementation*, Prentice-Hall, 2000
[5] Ramakrishnan, R.: *Database Management Systems*, 1998
[6] Worsley, J.; Drake, J.: *Practical PostgreSQL*, Command Prompt, 2001