**Univerzitet u Nišu, Elektronski fakultet**

logo_1960_4**Katedra za računarstvo**

**Obrada transakcija, planovi izvršavanja transakcija, izolacija i zaključavanje kod PostgreSQL-a**

**Stefan Živić 698**

**Predmet: Sistemi za upravljanje bazama podataka**

Sadržaj

[Uvod 3](#_Toc72245644)

[Transkacije 4](#_Toc72245645)

[Transakcije i ACID svojstva 5](#_Toc72245646)

[Transakcije kod PostgreSQL-a 6](#_Toc72245647)

[Transakcije i konkurentost 12](#_Toc72245648)

[MVCC u PostgreSQL-u 12](#_Toc72245649)

[Nivoi izolacije transakcija u PostgreSQL-u 16](#_Toc72245650)

[Mehanizmi eksplicitnog zaključavanja 25](#_Toc72245651)

[Zaključavanje na nivou tabela 25](#_Toc72245652)

[Modeli zaključavanja tabela 25](#_Toc72245653)

[Zaključavanje na nivou reda 29](#_Toc72245654)

[Modeli zaključavanja reda 29](#_Toc72245655)

[Deadlock-ovi 32](#_Toc72245656)

[Zaključavanja koja postavlja aplikacija (Advisory lock) 33](#_Toc72245657)

[Zaključak 35](#_Toc72245658)

[Literatura 36](#_Toc72245659)

# Uvod

Da bi sistem za upravljanje bazom podataka funkcionisao na prihvatljiv i ispravan način potrebno je da zna kako treba da radi u višekorisničkom okruženju, u kom se upiti izvršavaju istovremeno, odnosno u kome je prisutna konkurentnost. Tu se pre svega mora voditi računa o tome u kakvom je stanju baza u svakom trenutku, tj stanje mora biti konzistentno. Ne sme se ugroziti ispravnost podataka ni u kom trenutku, što znači da se moraju zabraniti polovično nepotpuno uspešne operacije, kao i one koje bi ugrozile tačnost ostalih operacija koje su u toku. Sa druge strane, tu su vremeneska ograničenja koja, zbog zahteva o konzistentosti, nije lako ispuniti. Zbog svega ovoga koncept transakcija je jako bitan u bilo kom sistemu za upravljanje bazama podataka. One obezbeđuju da se skup operacija izvršava kao celina, po principu sve ili ništa, čime direktno utiče na to da stanje u bazi bude konzistentno i da se tamo nađu samo ispravni podaci. Pomoću njih programer ima veću kontrolu nad onim što se dešava u bazi. Pored toga, konkurentnost bi bila nezamisliva da nike transakcija.

Rad je organizovan tako da je u prvo poglavlju objašnjen koncept transakcija kao takav,  uopšteno, na bilo kom sistemu. Nabrojane su neke od osnovnih komandi za rad sa transakcijama, čemu služe i kakva je njihova sintaksa. Takođe, u tom delu je prvi put ukratko objašnjeno šta podrazumevaju ACID svojstva transakcije. Drugo poglavlje se bavi transakcijama kod PostgreSQL-a. U njemu ćemo kroz primere iz realnog sistema videti kako se radi sa transakcijama u ovom sistemu.

Nakon toga sledi deo koji se bavi problemom konkurentnosti u vezi sa transakcijama. Objašnjen je koncept MVCC-a i kako on funkcioniše na primeru. Sledeće poglavlje vezano je za izolovanost, tačnije nivoe izolovanosti kod transakcija. Dat je pregled svih nivoa izolovanosti i šta podrazumeva svaki od njih. Glavni deo tog poglavlja zapravo su efekti koje izazivaju zaključavanja nekom od opcija jer se tu na primerima vidi kako ceo taj sistem radi. Za svaki efekat dat je primer, objašnjeno na kom nivou izolovanosti se može javiti i zašto se javlja baš na tom nivou.

Na kraju, tu je deo o mehanizmima ekspicitnog zaključavanja. Videćemo šta je zaključavanje na nivou tabele i reda, zaključavanje koje se radi od strane aplikacije, kao i problem deadlock-ova. Kao i kod prethodnih poglavlja, sve to će biti objašnjeno na primerima kako bismo u potpunosti razumeli kako taj deo funkcioniše.

Za demonstraciju primera i rad sa bazom korišćen je pgAdmin tool.

# Transkacije

Transakcije su ključni koncept u sistemima za upravljanje bazama podataka. Relacioni modeli predstavljaju logičku jedinicu obrade podataka kao *transakciju.* Transakcije predstavljaju skup operacija izvedenih u nizu, kao jedna sekvenca. Relacione baze podataka obezbeđuju mehanizme za zaključavanje (*locking mechanism*) kako bi osigurale integritet transakcije. Zbog toga transkacije garantuju konzistentnost i trajnost podataka u bazi. Gotovo je nemoguće implementirati sistem za upravljanje bazama podataka bez odgovarajućeg dela koji bi se bavio upravljanjem transakcijama.

Transakcije se mogu sastojati od pojedinačnih operacija čitanja, pisanja, brisanja ili ažuriranja ili njihove kombinacije. Glavna ideja transakcija je da se, kada bilo koja od operacija koje su deo transkacije vrati grešku, celokupne modifikacije ponište (*rollback-uju)*  kako bi se obezbedio integritet podataka. S druge strane, ako se sve operacije uspešno završe, promene podataka će se primeniti i postaće trajne u bazi podataka.

Sada ćemo videti neke od osnovnih komandi pri radu sa transakcijam, sa napomenom da se one mogu razlikovati od sistema do sistema. Za eksplicitno zadavanje početka transakcije kod najvećeg broja sistema koristi se komanda ***BEGIN / BEGIN TRANSACTION***, dok se kraj transakcije, ukoliko pre toga ne dođe do greške, koriste ***COMMIT****,* i u nekim sistemima, ***END****.* Primer za to je:

***BEGIN;***

***CREATE TABLE user (id serial primary key, username text, email text, password text);***

***COMMIT;***

U nastavku *BEGIN* naredbe može se naći jedinstveni identifikator, odnosno string, čime bismo transakciji dodelili ime. Ako je sve u redu sa svim naredbama u okviru jedne transakcije, sve promene se zajedno snimaju u bazu podataka (budu *komitovane)*. Naredba COMMIT snima sve transakcije u bazu podataka od poslednje naredbe *COMMIT*, *ROLLBACK*, ili *BEGIN*. Naredba ***ROLLBACK TRANSACTION*** izvršava poništavanja svih modifikacija podataka koje su urađene transakcijom. ***ROLLBACK*** naredba može stajati sama, čime bi poništila sve od poslednje naredbe COMMIT, ili od početka trenutne transakcije ako nema COMMIT-a, ili se može dodati identifikator koji označava transakciju ili savepoint.

***SAVEPOINT*** naredba kreira tačke unutar grupa transakcija na koje se može vratiti pomoću *ROLLBACK*. Ovim je obezbeđeno da možemo poništit promene samo iz dela transkacije, umesto iz celokupne transkacije, čime se poboljšava kontrola nad podacima. Sintaksa za ovu naredbu je:

***SAVEPOINT SAVEPOINT\_NAME;***

,nakon čega bi sledilo:

***ROLLBACK TO SAVEPOINT\_NAME;***

## Transakcije i ACID svojstva

Sistem za upravljanje bazama podataka mora da obezbedi da operacije budu atomične, kao i da obezbedi konzistentost, izolaciju i trajnost podataka. Ovo se naziva ***ACID*** svojstvom baze podataka zbog reči: ***A****tomicity,* ***C****onsistency,* ***I****solation,* ***D****urability.*

Transakcija je logička izvršna jedinica i nedeljiva je, što znači sve ili ništa. Ovo svojstvo poznato je kao ***atomičnost***. To važi bez obzira na razlog zbog kog operacije nije uspela. Na primer, transakcija može da bude nespešna zbog matematičke greške, pogrešno napisano ime relacije ili čak pad operativnog sistema.

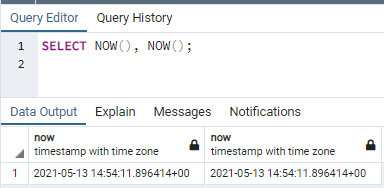
Nakon što se transakcija uspešno izvrši, efekat transakcije mora se održati čak i u slučaju hardverskih grešaka sistema (pada sistema). Ovo svojstvo se naziva ***trajnost***. Dalje, u višekorisničkom okruženju nekoliko korisnika može izvršiti nekoliko transakcija, a svaka transakcija sadrži skup radnji ili operacija. Svaka transakcija treba da se izvrši bez ometanja istovremeno izvršenih transakcija. Ovo svojstvo se naziva ***izolacija***.

Konačno, svojstvo ***konzistentnosti*** nije svojstvo same transakcije, već je poželjan efekat izolacije i atomičnosti transakcije. Konzistentnost baze podataka bavi se poslovnim zahtevima, koji su definisani pravilima, uključujući okidače, ograničenja i bilo koju njihovu kombinaciju. Ako je stanje baze podataka konzistentno neposredno pre izvršavanja transakcije, tada stanje baze podataka mora biti konzistentno i nakon izvršavanja transakcije. Za konzistentnost baze podataka odgovoran je programer koji kreira transakciju.

ACID svojstva nije lako ispuniti. Na primer, trajnost je jako zahtevna jer postoji mnogo faktora koji mogu prouzrokovati gubitak podataka, poput pada operativnog sistema, nestanka struje ili otkaza hard diska. Takođe, računarska arhitektura je prilično složena. Na primer, podaci mogu da prolaze kroz nekoliko slojeva memorije, kao što su glavna memorija, IO baferi i keš memorija diska, sve dok se ne zadrže na disku.

# Transakcije kod PostgreSQL-a

Ono što prvo treba znati je da je u Postgres-u sve transakcja. To se može videti i na primeru



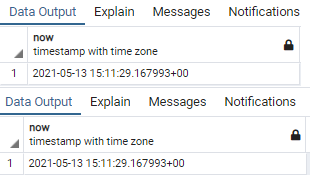
Slika 1 - SELECT naredba u okviru transakcije

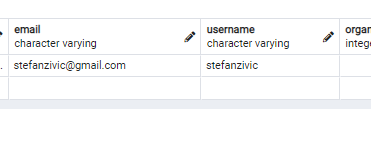
U ovom sučaju *SELECT* naredba biće izvršena u okviru posebne transakcije. Ako bi isti upit bio izvršen još jednom dobili bismo različite vrednosti. Obzirom da funckija *NOW()* vraća vreme transakcije, oba poziva vraćaju istu vrednost, što potvrpđuje da su deo iste transakcije.

Kao i kod drugih sistema, ako u okviru transakcije treba da budu izvršene više od jedne naredbe, mora da se koristi ***BEGIN***. *BEGIN* naredba će osigurati da sve komande koje slede budu “upakovane” u jednu transakciju.



Slika 2 - Primer transakcije



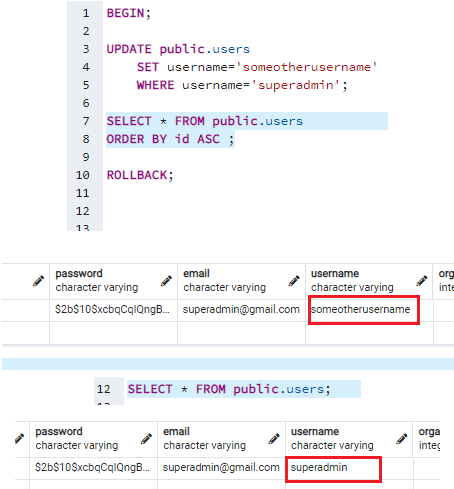


Slika 3 - Rezultat transakcije sa slike br 2

Iz retultata možemo videti da je transakcija uspešno izvršena (promenjeno username polje), kao i to da obe komande *SELECT now()* pripadaju istoj transakciji jer smo dobili istu vrednost.

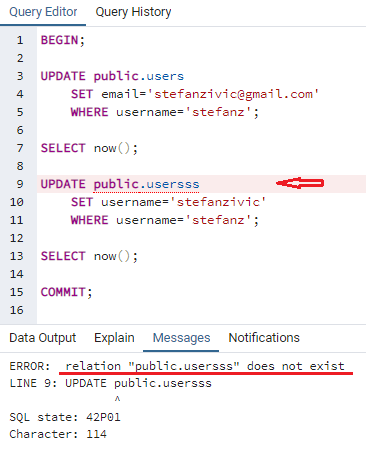
Da bismo transakciju okončali možemo koristiti COMMIT naredbu, kao što je to učinjeno na primeru iznad. Osim toga, postoje još dve naredbe koje se razlikuju samo semantički, a imaju isto značanje i to su ***COMMIT WORK*** i ***COMMIT TRANSACTION***. Pored njih, tu je i ***END*** komanda koja se koristi u iste svrhe, za okončanje transakcije.

Suprotna naredbi COMMIT je naredba ***ROLLBACK***. Kao što je to već napomenuto, ROLLBACK će umesto da uspešno okonča transakciju i promene u njoj primeni na bazu, transakciju jednostavno prekinutu bez toga da promene u njoj budu vidljive ostalim transakcijama, odnosno neće ih upisati u bazu. Neke aplikacije u ovu svrhu koriste komandu ***ABORT***. Značenje je potpuno isto.

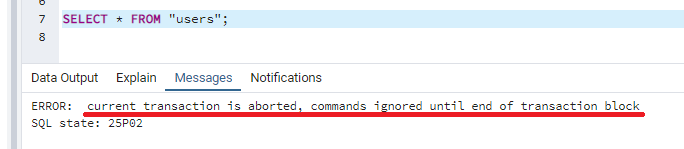


Slika 4 - Primer transakcije poništene sa ROLLBACK

Nije uvek slučaj da su transakcije tačne od početka do kraja. Stvari mogu jednostavno krenuti po zlu iz bilo kog razloga. Međutim, u PostgreSQL-u mogu se izvršiti samo transakcije bez grešaka. Sledeći primer prikazuje neuspešnu transakciju, u kojoj pokušava da se uradi UPDATE nad tabelom koja ne postoji.



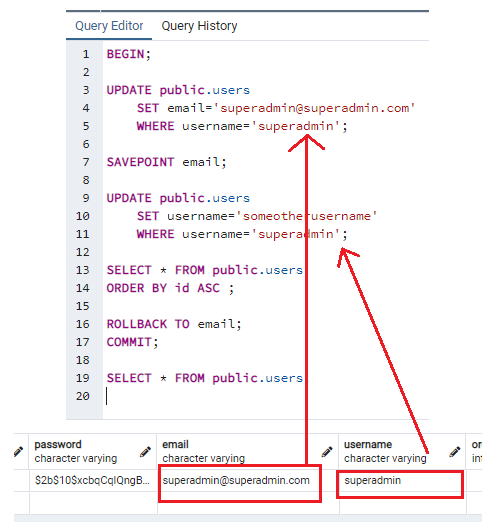
Slika - Primer transakcije koja u sebi ima grešku



Slika - Primer transakcije sa greškom koja ne prihvata naredbe

Važno je naglasiti da će PostgreSQL u ovom slučaju prijaviti grešku i poništiti transakciju, za razliku od, recimo,  MySQL-a,  koji je daleko manje striktan po tom pitanju. Nakon što se dogodi greška, neće biti prihvaćena više ni jedna naredba, čak i ako su te naredbe semantički i sintaksno ispravne. Još uvek je moguće izdati *COMMIT*, međutim, PostgreSQL će uraditi rollback transakcije, jer je to jedina ispravna stvar koja se u ovom trenutku može učiniti.

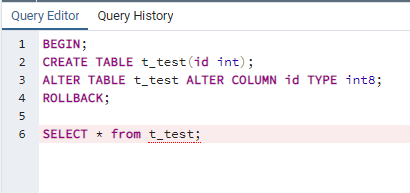
To može biti veliki problem, jer u profesionalnim aplikacijama može biti prilično teško napisati dugačke transakcije, a da nikada ne naiđete ni na jednu grešku. Da bi rešili problem, korisnici mogu da koriste nešto što se zove *SAVEPOINT*. Evo primera:

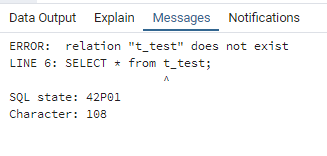


Slika - Primer transakcije sa SAVEPOINT-om

Ovde možemo videti da je posle prve update naredbe, koja postavlja *“email”* na novu vrednost,  kreiran savepoint. Nakon toga urađena je još jedna update naredba, ovog puta menja *“username”* polje, posle čega sledi rollback na prethodno kreiran savepoint. To znači da se poništava efekat samo druge update naredbe jer se nalazi nakon savepoint-a. Iz rezultata select naredbe koja je izvršena nakon transakcije možemo videti da je to upravo i slučaj - promenjen je samo email.

PostgreSQL ima vrlo korisnu osobinu, koja nažalost nije prisutna u mnogim komercijalnim sistemima baza podataka. Naime, u PostgreSQL-u je moguće pokrenuti DDL-ove (komande koje menjaju strukturu baze) unutar transakcija. U tipičnom sistemu baza podataka, DDL će implicitno izvršiti *commit* trenutne transakcije. To se ne dešava u PostgreSQL-u. Osim nekih manjih izuzetaka (DROP DATABASE, CREATE TABLESPACE, DROP TABLESPACE, i tako dalje), svi DDL-ovi u PostgreSQL-u su transakcioni, što je ogroman plus za krajnje korisnike. To se može videti na sledećem primeru:





Slika - Primer transakcije koja sadrži DDL

U ovom primeru je kreirana i izmenjena tabela i cela transakcija je prekinuta. Kao što vidite, nema implicitnog COMMIT-a ili bilo kog drugog čudnog ponašanja. PostgreSQL se jednostavno ponaša onako kako se i očekivalo. Transakcioni DDL-ovi su posebno važni kada se radi postavljanje nove verzije nekog softvera. Ako izađe nova verzija, želećete da uradite ažuriranje. Pokretanje stare verzije bi i dalje bilo u redu; pokretanje nove verzije je takođe u redu, ali ono što definitivno ne želite je mešavina starog i novog. Stoga je postavljanje ažuriranja, gde bi se prvo obrisali podaci, pa onda kreirali novi, u jednoj transakciji definitivno vrlo korisno.

# Transakcije i konkurentost

Osnovni problem u vezi sa konkurentnošću je istovremeni pristup zajedničkom globalnom resursu. Do sukoba neće doći ako programeri ne pišu kod u konkurentnom maniru, već u serijskom, gde se naredbe izvršavaju jedna za drugom. Međutim, u stvarnosti to nije praktično i uzrokuje mnogo gubitaka i dovodi do znatno lošijih performansi.

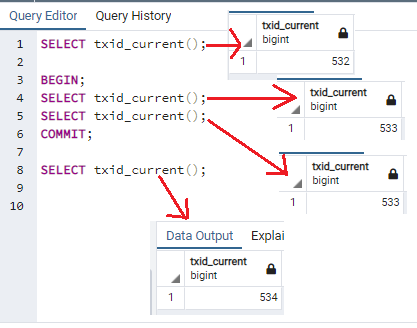
Kontrola paralelnosti se sastoji u koordinisanju izvršavanja istovremenih transakcija u višekorisničkom okruženju i rešavanju potencijalnih problema koji nastaju kao posledica takvog načina izvršavanja, kao što su nekonzistentni podaci, izgubljena ažuriranja itd. PostgreSQL ima podršku za konkurentnost, čak i kod aktivnosti čitanja i pisanja u kobminaciji. U PostgreSQL-u, aktivnost pisanja ne blokira aktivnost čitanja, ili obrnuto. PostgreSQL koristi tehniku koja se naziva ***Multi-Version Concurrenci Control (MVCC)***. Ovu tehniku takođe koristi nekoliko komercijalnih i nekomercijalnih baza podataka, kao što su Oracle, MySQL sa InnoDB, Informic, Firebird i CouchDB. MVCC nije jedini način za rukovanje paralelnog izvršenja u RDBMS-u; SQL Server koristi drugu tehniku koja se naziva *strong strict two-phase locking (SS2PL).*

## MVCC u PostgreSQL-u

Pristup određenim podacima istovremeno za čitanje i pisanje može prouzrokovati nedosledno stanje podataka ili pogrešno čitanje. Na primer, operacija čitanja može dobiti delimično prepisane podatke, odnosno one koje sadrže istovremeno i staru i novu vrednost. Najjednostavnija tehnika za rešavanje ovog problema je sprečavanje operacije čitanja da čita podatke dok se operacije čitanja ne završi sa pisanjem podataka. Postoji mnogo problema u vezi sa ovim rešenjem a najveći su performanse, zbog toga što operacija čitanja čeka dok se upis ne završi.

U *MVCC*, kada se podaci ažuriraju, kreira se novija verzija podataka i originalni podaci se ne prepisuju. To znači da postoji nekoliko verzija podataka, pa otuda i naziv *Multi Version (više verzija)*. Svaka transakcija može pristupiti verziji podataka na osnovu nivoa izolacije transakcije, trenutnog statusa transakcije i stanja relacije. Postgres rešava problem izolacije transakcija, a samim tim i problem konkurentosti transakcija koristeći MVCC za stvaranje koncepta nazvanog “***snapshots***“. Poenta ovog koncepta je da se sačuva stanje baze u nekom trenutku, recimo pre početka određene transakcije, i da se sa sa tim stanjem radi. Kad god upit započne, napravi se novi snimak trenutnog stanja baze podataka. Upitu su vidljivi samo efekti transakcija izvršenih pre kreiranja tog snimka. Redovi umetnuti transakcijama koje nisu izvršene pre započinjanja upita nevidljivi su za upit, a redovi izbrisani transakcijom koja nije izvršena pre započinjanja upita i dalje su vidljivi.

Svakoj transakciji u PostgreSQL-u dodeljuje se ID-u transakcije koji se zove ***XID/TXID***. *XID* se povećava kad god se započne nova transakcija. ID-ovi transakcija su 4-bajtni celi brojevi. Na sledećem primeru demonstrirano je kako se ID transakcija povećava:

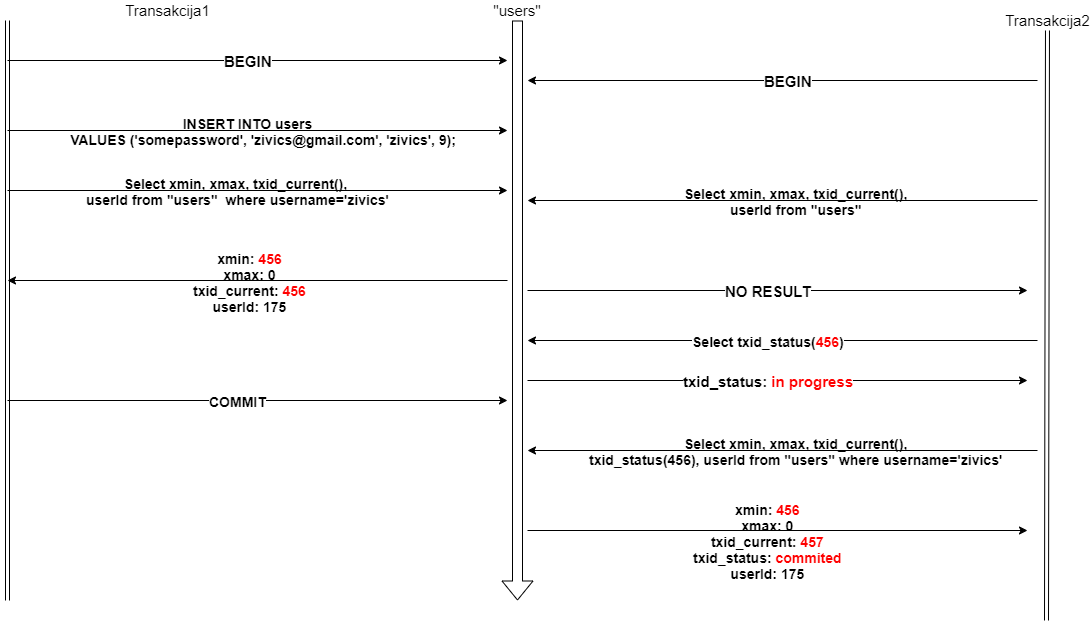


Slika - Dodeljivanje ID-a transakcijama

Ono što je interesantno u vezi sa ovim primerom je da se XID povećava bez toga da je eksplicitno zadata naredba za početak transakcije, što potvrđuje priču da je u PostgreSQL-u svaka naredba transakcija.

Kao deo MVCC, svaki red beleži i *TXID* transakcije koja ga je ubacila, a ako je red izbrisan, i *TXID* transakcije koja ga je izbrisala. Uz ove informacije i mogućnost utvrđivanja da li je transakcija završena pre početka upita, Postgres ima dovoljno informacija za primenu koncepta snapshot-a. Red treba da bude uključen u snimak ako je dodat transakcijom koja je završena pre početka upita, a da pri tome nije izbrisan transakcijom koja je završena pre upita.

Interno, PostgreSQL upisuje ID transakcije koja je kreirala, tj dodala red, u *xmin* (negde se sreće pod imenom *insert id*), a ID transakcije koja obriše red u *xmax (delete id)*. Operacija ažuriranja, tj *update*  je samo brisanje praćeno dodavanjem novog reda, i u ovom slučaju kreira se zapravo novi red kome se dodeljuju vrednosti, za početak xmin, na već objašnjen način. Budući da se red fizički ne briše kada ga upit briše, prostor na fizičkom disku moraće da se oslobodi nešto kasnije. Ovo se obično obavlja u pozadini koja se naziva *vacuum*. Postgres koristi informacije o kreiranju i brisanju određenog reda (xmin i xmax), statusu transakcije ( da li je ona u statusu *committed, in progress, rolled back),* nivou izolacije transakcije i njenjoj vidljivosti za rešavanje problema konkurentnosti. Na sledećem primeru videćemo kako se dodeljuju *xmin* i *xmax* u slučaju INSERT naredbe, i kako konurentnost funkcioniše na primeru sa 2 transakcije koje rade sa istom tabelom:



Slika - MVCC

Pre nego što počnemo sa objašnjavanjem primera treba napomenuti da je u svrhu demonstracije kako funkcioniše dodeljivanje xmin i xmax vrednosti i konkurentnost, tabela “users” prvo ispražnjena. Što se primera tiče, prvo se započinje jedna, a zatim i druga transakcija. Nakon toga se u okviru prve transakcije dodaje red u tabelu *“users”* i odmah nakon toga se čita xmin, xmax, xid\_current (id trenutne transakcije), kao i id reda koji je dodat. Za rezultat dobijamo da su xmin i txid\_current **456**, što je i očekivano obzirom da je trenutna transakcija upravo ona koje je i dodala taj red. Za xmax se dobija **0** jer još uvek ni jedna transakcija nike uradila brisanje tog reda. Iste ove podatke probala je da pročita i druga transakcija, ali nije dobila ništa kao rezultat. Razlog je taj što u trenutku kada je ona započeta u tabeli “users” nije bilo redova, odnosno snapshot koji je ona “dobila” nema redove, jer prva transakcija još uvek nije završena. To se potvrđuje i sledećom naredbom u okviru druge transakcije *select txid\_status(456),* kojom se dobija status prve transakcije. Kao što se na primeru vidi taj status je još uvek ***in progress***. Ono što se dešava sledeće je da se poziva COMMIT u okviru prve transakcije, čime se ona potvrđuje i okončava, a red se upisuje u tabelu. Sada transakcija broj 2 pokušava po drugi put da dobije već pomenute podatke, ali ovog puta čitanje je uspešno, jer je transakcija 1 u stanju *commited*  i snapshot koji ima ovog puta sadrži podatke. Xmin ostaje nepromenjen, odnosno ima vrednost id-a transakcije koje je dodala red, dok je *txid\_current* sada **457**, odnosno povećan je za 1.

Na ovom primeru ukratko je demonstrirano kako radi MVCC pomoću snapshot-ova i informacijama o transakcijama i redovima. Kao što je rečeno, određena transakcije će videti samo one podatke koji su upisani u transakcijama koju se potvrđene pre njenog čitanja istih.

Na kraju treba još jednom napomenuti da je MVCC odličan iz više razloga, a dva najvažnija su:

* Prvo, nudi jednostavan model za upravljanje upitima u okviru transaskcija. Svi upiti kreiraju snapshot baze podataka kada se pokrenu i vide samo te podatke
* Drugo, nudi pristojne performanse. Sa MVCC, gotovo da nema potrebe za eksplicitnim zaključavanjem. MVCC onemogućava blokiranje upita za čitanje zbog upita koji rade pisanje ili obrnuto

# Nivoi izolacije transakcija u PostgreSQL-u

Nivoi izolovanosti transakcija definišu na koji način će objekti baze podataka, kao što su tabele, biti zaključani. Takođe, definiše stepen do kojeg transakcija treba da bude izolovana od modifikacija podataka izvršenih bilo kojom drugom transakcijom. Nivo izolacije transakcije može se postaviti pozivanjem sledeće SQL naredbe:

***SET TRANSACTION ISOLATION LEVEL***

***{ SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }***

Izraz *SET TRANSACTION ISOLATION LEVEL* treba pozvati unutar bloka transakcije pre bilo kog upita, jer u suprotnom neće imati efekta. Alternativa je upotreba na sledeći način:

***BEGIN TRANSACTION ISOLATION LEVEL***

***{ SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }***

Poslednja opcija je promeniti podrazumevani nivo izolovanosti za čitavu bazu:

***ALTER DATABASE <DATABASE NAME>***

***SET DEFAULT\_TRANSACTION\_ISOLATION TO SERIALIZABLE ;***

PostgreSQL podržava 3 nivoa izolovanosti, iako ćemo ovde pomenuti i četvrti. Kao što se to vidi na primerima, nivo izolovanosti može biti:

* ***SERIALIZABLE***: najjači nivo što se konzistentnosti podataka tiče, najstriktniji po pitanju zaključavanja. U njemu se programer oslobađa toga da rešava probleme vezane sa konkurentnost. Loša strana upotrebe ovog modela često su performanse. Po SQL standardu, *SERIALIZABLE* je podrazumevani nivo izolacije, dok je u PostgreSQL-u podrazumevani nivo izolacije je *READ COMMITTED*.

* ***REPEATABLE READ***: drugi po snazi nivo izolacije transakcije. Sličan je *READ COMMITTED* po tome što omogućava samo čitanje podataka koji su commit-ovani, tj potvrđeni i definitivno upisani. Takođe garantuje da se bilo koji pročitani podatak ne može promeniti.

* ***READ COMMITTED***: Ovo je podrazumevani PostgreSQL model. Omogućava transakciji čitanje commit-ovanih podataka. Na ovom nivou performanse se favorizuju u odnosu na konzistentost.

Pomenućemo i ***READ UNCOMMITTED*** iako on nije podržan u Postgres-u i tretira se kao READ COMMITED.  Kod ovog modela moguće je da pročitate i podatke koji još uvek nisu potvrđeni, pa je zbog toga model sa najslabijim nivoom izolacije, odnosno najmanje je restriktivan po pitanju zaključavanja.

Kako svaki od ovih nivoa funckioniše najlakše je objasniti ako vidimo koji sve efekte, odnosno posledice, može imati svaki od njih. Pomenućemo nekoliko i objasniti ih na primeru:

* ***Dirty read:*** “*Dirty read*” se događa kada transakcija čita podatke iz reda koji je modifikovan drugom aktivnom transakcijom koja još uvek nije commit-ovana. U PostgreSQL-u se to **ne može** dogoditi jer READ UNCOMMITTED nije podržan.

* ***Nonrepeatable read***: “*Nonrepeatable read*” se dešava kada se tokom transakcije red dva puta preuzme i vrednosti unutar reda se razlikuju između čitanja. Javlja se na nivou izolacije *READ COMMITTED* i često je rezultat ažuriranja reda nekoliko puta drugim transakcijama.

* ***Phantom read***: “*Phantom read*” se dešava kada novi red (ili redovi) budu izbrisani od trenutka početka transakcije. Ovo je često rezultat potvrđenih dodavanja novih redova praćenih brisanjem. *Phantom read* se dešava kada nekoliko puta pokušate da dobijete skup redova i u okviru iste transakcije vrati se drugačiji rezultat. Po standardnoj SQL definiciji phantom read se dešava na nivoima *READ COMMITTED* i *REPEATABLE READ*. U PostgreSQL-u se *phantom read* javlja samo u *READ COMMITTED*.

* ***Serialization anomaly:*** Dešava se kada postoji grupa transakcija gde je krajnji rezulatat nekonzistentan, odnosno različit u zavisnosti od redosleda kojim se transakcije izvršavaju. Može se desiti u slučaju *REPEATABLE READ*-a.

Prema SQL standardu moguće je da se *phantom read*  desi ako je nivo izolacije postavljen na *REPEATABLE READ*. U PostgreSQL-u to nije moguće, tako da je na tom nivou moguće samo da dođe do serialization anomaly.

Da bismo što bolje razumeli kako funkcioniše svaki od nivou izolacije transakcija, na primeru ćemo videti kako dolazi do ***nonrepeatable read-a****.* Nivo izolovanosti je podrazumevan - *READ COMMITED*. Oznaka *T (T1, T2, T3 …. Tn)* koristi se za vreme izvršenja gde veći indeks označava kasnije vreme izvršenja.

|  |  |  |
| --- | --- | --- |
|  | Transakcija1 | Transakcija2 |
| T1 |  |  |
| T2 |  |  |
| T3 |  |  |

Slika - Nonrepeatable read

Vidimo da se u okviru prve transakcije polje *username* za user-a sa id-em 2 promenilo sa “stefanzivic” na “szivic”. Upravo to se naziva *nonrepeatable read.* Treba primetiti da smo u oba slučaja koristili samo *BEGIN* bez specificiranje nivoa izolacije, što znači da je korišćen podrazumevani nivo izolacije.

Sledeći efekat koji ćemo videti na primeru je ***phantom read***. Nivo izolovanosti je opet podrazumevani.

|  |  |  |
| --- | --- | --- |
|  | Transakcija1 | Transakcija2 |
| T1 |  |  |
| T2 |  |  |
| T3 |  |  |

Slika - Phantom read

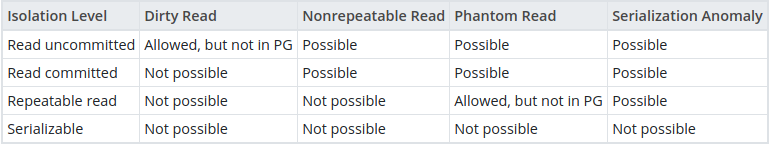
*Phantom read* i *nonrepeatable read* mogu se javiti, kao što je prikazano u prethodnim primerima, na nivou izolovanosti READ COMMITTED. Opseg *nonrepeatable read-a* često je određeni red ili skup redova, dok je opseg *phantom read-a* cela tabela. *Nonrepeatable read* se dešava kada transakcija čita podatke ažurirane sa *update* iz druge transakcije, dok se *phantom read* dešava kao rezultat potvrđenih operacija *insert* i *delete.*

Ako sada iste primere pokrenemo opet, a nivo izolovanosti postavimo na SERIALIZABLE ili REPEATABLE READ, videćemo da se promene iz transakcije 2 ne vide u transakciji 1. To je prikazano i na primeru:

|  |  |  |
| --- | --- | --- |
|  | Transakcija1 | Transakcija2 |
| T1 |  |  |
| T2 |  |  |
| T3 |  |  |

Slika - Seriazable nivo izolacije

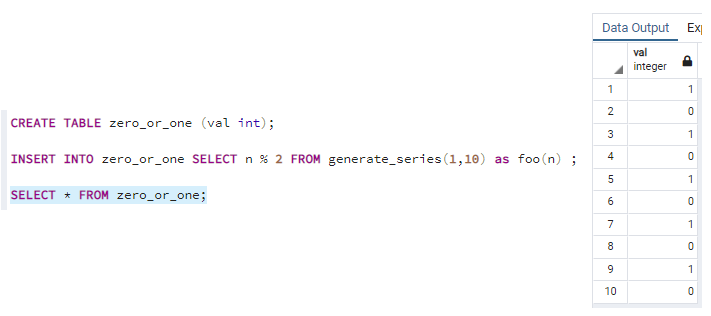
U sledećoj tabeli dat je prikaz na kom nivou izolacije se može naći svaki od efekata o kojima smo pričali:



Slika - Efekti koji se javljaju na svakom od nivoa

U verzijama PostgreSQL-a pre verzije 9.1, PostgreSQL imao je samo dva nivoa izolacije transakcija. Ovo se promenilo uvođenjem PostgreSQL 9.1, gde je dodat *SERIALIZABLE* model. *SERIALIZABLE* nivo izolacije transakcija štiti podatke od mnogih anomalija, poput **“*write skew”*.** *Write skew* se dešava kada dve transakcije čitaju iste podatke, istovremeno vrše ažuriranje i, konačno, upisuju podatke. Na primer, pretpostavimo da imamo tabelu koja sadrži jedinice i nule; prva transakcija želi da jedinice pretvori u nule, a druga transakcija želi da nule promeni u jedinice. Ako se transakcije izvršavaju serijski, trebali bismo dobiti ili jedinice ili nule u zavisnosti od toga koje se transakcija izvršava prva, u suprotnom nešto između. To ćemo demonstrirati i na primeru.

Prvo ćemo kreirati tabelu koja će imati naizmenično jedinice i nule.



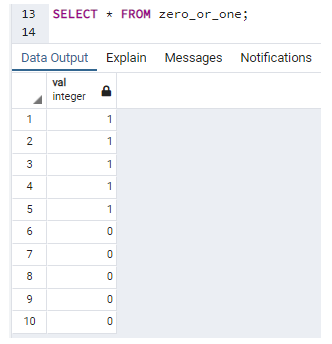
Slika - Kreiranje tabele

Nakon toga, nivo izolovanosti transakcije postavljamo na *REPEATABLE READ* i radimo sledeće:

|  |  |  |
| --- | --- | --- |
|  | Transakcija1 | Transakcija2 |
| T1 |  |  |
| T2 |  |  |
| T3 |  |  |

Slika 16 - Write skew primer

Pogledajmo sada kako izgledaju podaci u tabeli:



Slika - Podaci u tabeli nakon upisa

Da bismo videli kako će se isti primer ponašati SERIALIZABLE, prvo ćemo isprazniti tabelu na sledeći način:



Sada ponovimo isti primer sa *serializable:*

|  |  |  |
| --- | --- | --- |
|  | Transakcija1 | Transakcija2 |
| T1 |  |  |
| T2 |  |  |
| T3 |  |  |

Slika 18 - Primer sa slike 15 sa SERIALIZABLE nivoom izolacije

Rezultat commit naredbe u okviru prve transakcije je greška sa porukom:

ERROR:  could not serialize access due to read/write dependencies among transactions

DETAIL:  Reason code: Canceled on identification as a pivot, during commit attempt.

HINT:  The transaction might succeed if retried.

SQL state: 40001

U REPEATABLE režimu, obe transakcije izvršene su bez greške, međutim, krajnji rezultat je bio pogrešan. U slučaju SERIALIZABLE nivoa izolovanosti, kada postoji *write skrew*, transakcije će se odvijati dok se ne izvrši, odnosno potvrdi,  jedna transakcija. Transakcija koje se prva komituje će jedina biti uspešna dok će se ostale poništiti. Ono na šta treba obratiti pažnju je hint u primeru koji kaže da bi transakcija koja je poništena mogla biti uspešna ako se pokrene ponovo.

# Mehanizmi eksplicitnog zaključavanja

Do sada smo se bavili zaključavanjima podataka koje obezbeđuje MVCC. U slučajevima kada MVCC ne obezbeđuje ponašanje kakvo želimo, možemo eksplicitno kontrolisati zaključavanje. Generalno govoreći, PostgreSQL nudi tri mehanizma zaključavanja i to su:

* Table-level locks – zaključavanje na nivou tabela
* Row-level locks – zaključavnje na nivou reda
* Advisory locks – zaključavanje na nivou aplikacije/sesije

## Zaključavanje na nivou tabela

Tabela se može zaključati pomoću jednog od nekoliko mod-ova. Sintaksa je sledeća:

*LOCK [ TABLE ] [ ONLY ] name [ \* ] [, ...] [ IN lockmode MODE ] [ NOWAIT ]*

, gde je *lockmode* jedan od:

*ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE*

*| SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE*

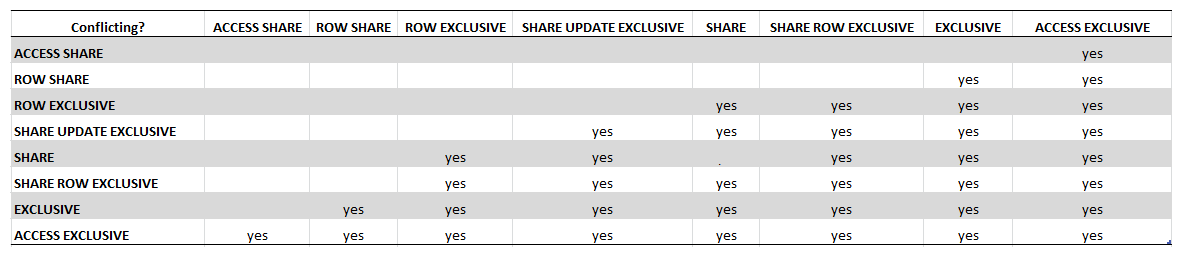
PostgreSQL tabelu implicitno zaključava kada poziva bilo koju SQL naredbu. Tabelu zaključava pomoću najmanje restriktivnog režima radi poboljšanja paralelnosti. Kada programer želi restriktivnije zaključavanje, tada se može koristiti LOCK komanda.

### Modeli zaključavanja tabela

* **ACCESS SHARE**: Ovaj režim se koristi kada se pokrene SELECT.
* **ROW SHARE:** koristi se kod naredbi SELECT FOR UPDATE i SELECT FOR SHARE
* **ROW EXCLUSIVE**: Koristi se kod UPDATE, DELETE i INSERT
* **SHARE UPDATE EXCLUSIVE**: Ovaj model se koristi za zaštitu tabele od istovremenih promena šeme. VACUUM(bez FULL-a),  ANALYZE,CREATE INDEX CONCURRENTLY, CREATE STATISTICS, ALTER TABLE, VALIDATE i varijante ALTER TABLE.

* **SHARE**: Ovaj model se koristi za zaštitu tabele od istovremenih promena podataka. CREATE INDEX koristi share.
* **SHARE ROW EXCLUSIVE**: Ovaj model štiti tabelu od istovremenih promena podataka i istovremeno ga može držati samo jedna sesija. Koristi se u CREATE COLLATION, CREATE TRIGGER i mnogim oblicima ALTER TABLE.
* **EXCLUSIVE**: Koristi ga REFRESH MATERIALIZED VIEW CONCURRENTLY. Ovaj režim omogućava samo čitanje podataka iz tabele.
* **ACCESS EXCLUSIVE**: Ovaj model garantuje da podatke drži jedini transakcija koja na bilo koji način pristupa tabeli. Koriste ga DROP TABLE, TRUNCATE, REINDEX, CLUSTER, VACUUM FULL i REFRESH MATERIALIZED VIEW. Mnogi oblici ALTER TABLE takođe koriste ovaj model. Ovo je takođe podrazumevani način zaključavanja za komandu LOCK TABLE kada se ne navode eksplicitno drugi režim.

Veoma važna stvar koju treba napomenuti za svaki mod je lista mod-ova koji su u sukobu sa njim. Transakcije ne mogu zaključati istu tabelu modelima koji su sukobljeni.



Slika 19 – Odnos modela zaključavanja tabele

Kao što se sa tabele vidi ACCESS EXCLUSIVE je u konfliktu sa  ACCESS SHARE,  što znači da se ne može uraditi SELECT nad tabelom koja je zaključana ACCESS EXCLUSIVE mode-om. Takođe, ne može se uraditi DROP tabele ako neka transakcija čita podatke iz nje. Vidimo i to da je SELECT moguće raditi u bilo kom drugom mode-u, osim u ACCESS EXCLUSIVE.

Na sledećem primeru videćemo šta će se desiti ako uradimo DROP tabele iz koje se čitaju podaci:

|  |  |  |
| --- | --- | --- |
|  | Transakcija1 | Transakcija2 |
| T1 |  |  |
| T2 |  |  |
| T3 |  |  |

Slika 20 - Primer DROP TABLE za tebelu iz koje se čitaju podaci

Primer pokazuje da je tabela zaključana kada se iz nje čitaju podaci i to traje do kraja transakcije. Za razumevanje kako zaključavanje funkcioniše veoma je koristan *pg\_locks.* Često se koristi za detektovanje kritičnih tačaka u vidu uskih grla, tzv. *bottleneck-ova,* kod sistema gde postoji konkurentno izvršavanje. Moguće je kreirati ***view***koji bi informacije o zaključavanju prikazao na način koji više odgovara programeru i to na sledeći način:

*CREATE OR REPLACE VIEW lock\_info AS*

*SELECT*

*lock1.pid as locked\_pid,*

*stat1.usename as locked\_user,*

*stat1.query as locked\_statement,*

*stat1.state as locked\_statement\_state,*

*stat2.query as locking\_statement,*

*stat2.state as locking\_statement\_state,*

*now() - stat1.query\_start as locking\_duration,*

*lock2.pid as locking\_pid,*

*stat2.usename as locking\_user*

*FROM pg\_catalog.pg\_locks lock1*

*JOIN pg\_catalog.pg\_stat\_activity stat1 on lock1.pid = stat1.pid*

*JOIN pg\_catalog.pg\_locks lock2 on*

*(lock1.locktype,lock1.database,lock1.relation,lock1.page,lock1.tuple,lock1.*

*virtualxid,lock1.transactionid,lock1.classid,lock1.objid,lock1.objsubid) IS*

*NOT DISTINCT FROM*

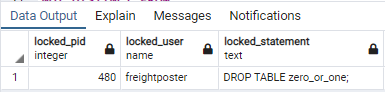
*(lock2.locktype,lock2.DATABASE,lock2.relation,lock2.page,lock2.tuple,lock2.*

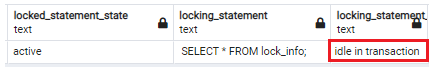
*virtualxid,lock2.transactionid,lock2.classid,lock2.objid,lock2.objsubid)*

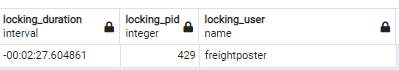
*JOIN pg\_catalog.pg\_stat\_activity stat2 on lock2.pid = stat2.pid*

*WHERE NOT lock1.granted AND lock2.granted****;***

Onda bi rezultat naredbe *SELECT \* FROM lock\_info ;* izgledao ovako:







Slika 21 - Informacije o lock-ovi za primer sa slike 19

View pokazuje da proces 480 pokušava da izvrši komandu DROP TABLE i čeka transakciju koju je izdao ID procesa 429. Proces 429 u tom trenutku ne radi ništa, pa je zbog toga status prve transakcije *idle in transaction,* zbog čega druga transakcija čeka.

## Zaključavanje na nivou reda

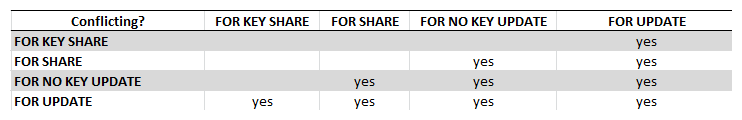
Zaključavanje na nivou reda uopšte ne zaključava *SELECT* naredbe, već se koristi kod *UPDATE* i *DELETE* izraza. Kao i kod zaključavanja na nivou tabele, dve transakcije ne mogu zaključati isti red u konfliktnim režimima. Zaključavanje redova korisno je u scenariju u kojem aplikacija proverava vrednost reda pre nego što ga ažurira. Drugi scenario je sprečavanje korisnika od ažuriranja vrednosti koja je zastarela. U *PostgreSQL 9.5* uvedena je opcija *SKIP LOCKED*. Ona menja ponašanje zaključavanja redova i korisna je za rukovanje skupom ažuriranja (batch) bez prevelikog vremena čekanja, ili rukovanja pool-ovima.

### Modeli zaključavanja reda

U starijim verzijama, npr. *PostgreSQL 9.3*, postoje samo dva načina zaključavanja, a to su *FOR UPDATE* i *FOR SHARE*. *FOR UPDATE* se koristi za zaključavanje reda i ne dozvoljava drugim transakcijama da ga ažuriraju ili izbrišu. *FOR SHARE* je manje restriktivan jer omogućava drugim transakcijama da zaključaju red, takođe sa FOR SHARE, ali ne dozvoljava drugim transakcijama da izbrišu ili ažuriraju red. *FOR SHARE* se koristi za *nonrepeatable read* scenaria kada je nivo izolovanosti *READ COMMITTED*. Na primer, može se zaključati red sa *FOR SHARE* i to će garantovati da se vrednost neće promeniti dok transakcija ne bude izvršena.

FOR SHARE ne rešava problem izgubljenog ažuriranja - *lost update-a*, jer omogućava drugim transakcijama da zaključaju red u istom režimu. Za rešavanje problema *lost update*-a može se koristiti *FOR UPDATE*. Malo detaljnije o problemu *lost update* videćemo u jednom od narednih poglavja.

U PostgreSQL-u postoji i *FOR NO KEY UPDATE* model koji je sličan kao *FOR UPDATE*, ali je manje restriktivan. Takođe, postoji i *FOR KEY SHARE* koji je slabija verzija zaključavanja *FOR SHARE*-a. Na sledećoj tabeli možemo da vidimo koji od ovih modela su u konfliktu:



Slika 22 - Konfliktni modeli kod zaključavanja reda

Za testiranje zaključavanja na nivou reda kreiraćemo tabelu “t*est*” i u nju dodati dve vrednosti:

***CREATE TABLE test (val int);***

***insert into test Values(1), (2);***

Nakon toga uradićemo sledeće:

|  |  |  |
| --- | --- | --- |
|  | Transakcija1 | Transakcija2 |
| T1 |  |  |
| T2 |  |  |

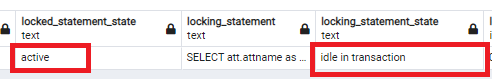
Slika 23 – Primer zaključavanja na nivou reda

Transakcija 2 čeka na Transakciju broj 1, zato što je ona pozvala *FOR UPDATE* lock. Kada se pozove

***SELECT \* FROM lock\_info ;***

dobijamo:

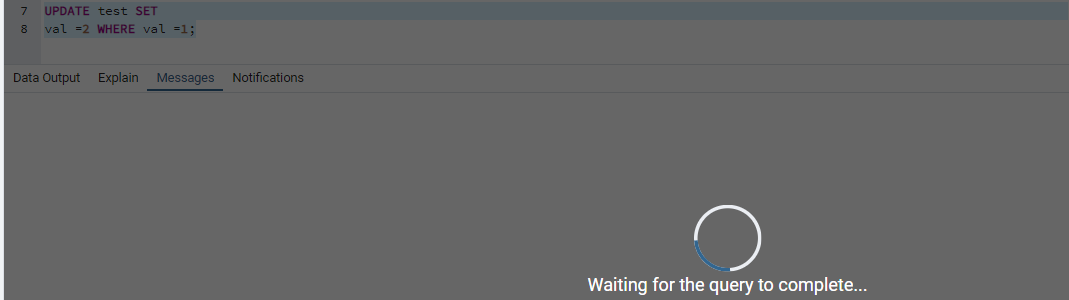
**

**

**

Slika 24 - Informacije o zaključavanju za primer sa slike 22

Transakcija koja “drži” lock ima status ***idle in transaction,*** zbog čega druga transakcija čeka.



Slika 25 - Transakcija koja je blokirana u pg Admin-u

### Deadlock-ovi

Deadlock-ovi se dešava kada dve ili više transakcija gde jedna od njih drži zaključane podatke koji su potrebni nekoj drugoj transakciji. Da bismo ovo pokazali na primeru koristićemo FOR SHARE model zaključavanja:

|  |  |  |
| --- | --- | --- |
|  | Transackcija 1 | Transakcija 2 |
| T1 |  |  |
| T2 |  |  |
| T3 |  |  |
| T4 |  |  |
| T5 |  |  |

Slika 26 – Primer DEADLOCK-a

Greška koju dobijemo u T4 je:

ERROR:  **deadlock detected**

DETAIL:  Process 663 waits for ShareLock on transaction 556; blocked by process 429.

**Process 429 waits for ShareLock on transaction 557; blocked by process 663**.

HINT:  See server log for query details.

CONTEXT:  while updating tuple (0,1) in relation "test"

SQL state: 40P01

Da bi se izbegli *deadlock*-ovi prvi lock koji se desi u transakciji treba da bude najrestriktivniji. Npr ako Transakcija 1 korisiti FOR UPDATE, onda će Transakcija 2 biti blokirana. Drugi način , kao što smo već pomenuli , bi bio da se koriste restriktivniji model koji bi forsirao transakcije da čekaju, ali bio lošiji po pitanju perfrormansi.

### Zaključavanja koja postavlja aplikacija (Advisory lock)

*Advisory lock*-ovi predstavljaju zaključavanja koja postavlja aplikacija. Postavljaju se na nivou sesije ili transakcije, tj podaci se zaključavaju na početku sesije ili transakcije i ostaju u tom stanju dok se sesija ili transakcija ne završi. Koriste se u situacijama kada želimo da ograničimo konkurentno izvršavanje na samo jedan proces. Na primeru bi to funkcionisalo ovako: kada novi proces započne sa izvršenjem, on zatraži *lock*. Ako dobije *lock*, proces nastavlja sa izvršenjem, a u suprotnom se stopira. Zbog takvog ponašanja, *advisory lock* omogućava programerima da bazu tretiraju kao *single user environment,* tj kao sistem u kome istovremeno može raditi samo jedan korisnik, što ih oslobađa od briga oko konkurentosti, zaključavanja, izolovanosti itd, ali po cenu performansi.

*Advisory lock*-ovi, kao i lock-ovi koje smo videli do sada, čuvaju se u deljenoj memoriji, u pool-u čiju veličinu definišu konfiguracione promenljive [max\_locks\_per\_transaction](https://www.postgresql.org/docs/current/runtime-config-locks.html#GUC-MAX-LOCKS-PER-TRANSACTION) i [max\_connections](https://www.postgresql.org/docs/current/runtime-config-connection.html#GUC-MAX-CONNECTIONS). Na programeru je da vodi računa o tome da ne iskoristi ceo kapacitet ove memorije, jer u tom slučaju neće moći da “dobije” ni jednu vrsu lock-a.

Na sledećem primeru videćemo kako se koristi *advisory lock:*

|  |  |
| --- | --- |
| Transakcija 1 | Transakcija 2 |
|  |  |
|  |  |
|  |  |
|  |  |

Slika 27 - Prmer upotrebe Advisory lock-a

U toku jedne iste sesije možemo pozvati *advisory lock* više, ali bi trebalo da oslobodimo lock onoliko puta koliko smo ga i zatražili:

***SELECT pg\_try\_advisory\_lock(1);***

***SELECT pg\_try\_advisory\_lock(1);***

***-- To release***

***select pg\_advisory\_unlock(1);***

***select pg\_advisory\_unlock(1);***

# Zaključak

Transakcije predstavljaju jedan od najvažnijih koncepata rada sa bazom podataka. Pomoću njih moguć je rad u višekorisničkom okruženju, gde se upiti izvršavaju konkurentno. PostgreSQL ima napredni podsistem koji se bavi transakcijama i korisnicima pruža mnogo funkcija koje olakšavaju rad sa njima. Na primerima smo videli da postoji mnogo različitih podešavanja sistema kada se radi sa transakcijama. Od izabrane strategije zavisi kako će se sam sistem ponašati, počeći od toga koje će promene biti uzete u obzir, preko toga koliko je otporan na greške, pa do samih performansi. Iako može biti moćno oružje, treba pažljivo rukovati podsistemom za transakcije jer se lako može ući u probleme.

Većina primera u ovom radu fokusirana je na konkurentno izvršavanje i probleme koji nastaju u vezi sa njim, izolaciju transakcija i zaključavanje. Videli smo na koje se načine podaci mogu zaključati, šta to znači u bazi, šta za progremara i koje su posledice. Ukratko su objašnjene prednosti i mane svakog od pristupa i rečeno je kada bi mogli biti od koristi. Primerima smo pokazali na koje sve načine programer može da utiče na ponašanje sistema. Dokazali smo da je znanje o tome kako treba raditi sa transakcijama, kao i to na koji način funckcioniše zaključavanje i konkurentost  od krucijalnog značaja za rad sa bazom.

# Literatura

[1] Ibrar Ahmed, Gregory Smith, Enrico Pirozzi , “PostgreSQL 10 High Performance Expert techniques for query optimization, high availability, and efficient database maintenance”, Packt Publishing, 2018

[2] Hans-Jurgen Schonig, “Mastering PostgreSQL 11 Expert techniques to build scalable, reliable, and fault-tolerant database applications (Secon edition) “, Packt publishing, October 2018

[3] Transaction Isolation <https://www.postgresql.org/docs/13/transaction-iso.html> [poslednji pristup 18.05.2021.]

[4] PostgreSQL **SSI** <https://wiki.postgresql.org/wiki/SSI> [poslednji pristup 18.05.2021.]

[5] Be careful with prepared transactions in PostgreSQL

<https://blog.dbi-services.com/be-careful-with-prepared-transactions-in-postgresql/> [poslednji pristup 11.05.2021.]

[6] Korry Douglas, Susan Douglas, “PostgreSQL: A Comprehensive Guide to Building, Programming, and administering PostgreSQL databases”, Sams Publishing, February 2003