UNIVERZITET U NIŠU
ELEKTRONSKI FAKULTET

Seminarski rad

Predmet: Sistemi za upravljanje bazama podataka

Obrada transakcija, planovi izvršavanja, izolacija i zaključavanje

Student:

Dimitrije Mitić 822

Mentor:

Doc. dr Aleksandar Stanimirović

Sadržaj

1.Uvod

Današnje aplikacije zahtevaju konstantnu dostupnost i brz odziv za veliki broj korisnika koji istovremeno koriste istu. S obzirom da su baze podataka nezaobilazan deo ovakvih, modernih aplikacija, DBMS mora da poseduje mehanizme za konkurentnu obradu više korisničkih zahteva. Svaki korisnički zahtev koji se treba izvršiti, DBMS interpretira kao jedan niz *read* i *write* operacija, odnosno kao jednu **transakciju** [1]. Primer jednog ovakvog sistema bi bila aplikacija za rezervaciju avio karata kojoj istovremeno pristupa na hiljade korisnika. Da bi DMBS na najefikasniji način iskoristio procesorsku jedinicu (CPU) putem protočnosti, transakcije se ne mogu izvršavati jedna po jedna (sekvencijalno), već mora doći do izvesnog preplitanja (*interlancing*) između operacija koje pripadaju različitim transakcijama. Kod ovakvog preplitanja, treba voditi računa da krajnji rezultat bude ekvivalentan, po svom krajnjem efektu na bazu, kao kad bi se transakcije izvršavale sekvencijalno u nekom od redosleda. Nekoliko problema se može pojaviti prilikom konkuretnog izvršavanja transakcija, koji mogu dovesti do toga da baza ostane u nekonzistentnom stanju. Komponenta DBMS-a koja se zove **kontrola konkurencije** (*concurency control*) je zadužena za rešavanje ovog problema [1]. Kontrola konkurencije je u tesnoj vezi i sa **menadžerom oporavka** koji je zadužen da, prilikom neuspelog izvršenja transakcije, poništi efekat iste i vrati bazu u predjašnje stanje.

Ovaj rad se bavi načinom na koji PostgeSQL DBMS upravlja transakcijama, omogućuje njihovo konkuretno izvršavanje, kao i problemima koji se pri tome javaljaju i načinima za njihovo prevazilaženje. Sam rad je podeljen u dve celine, pri čemu se prva bavi uopšteno transakcijama: ACID svojstva, rasporedi(*schedules*), anomalije prilikom preplitanja, zaključavanje, dok se drugi deo bavi primenom ovih stvari kod PostgreSQL baze.

2.Procesiranje transakcija

Pod pojmom transakcije, smatra se program u izvršenju DBMSa koji je ustvari logička jedinica obrade baze podataka [1][3]. Transakcija obuhvata jednu ili više operacija pristupa bazi, koje mogu biti: dodavanje, brisanje, modifikovanje ili pribavljanje podataka. S obzirom da transakcija predstavlja izvršenje jednog DBMS programa, više različitih korisnika može da pokrene izvršavanje jednog istog programa, pri čemu je svako od tih izvršavanja zasebna transakcija. Primer ovoga bi bila prethodno navedena aplikacija za rezerveraciju avio karata, kod koje se više korisnika, kada žele da rezervišu mesto za željeni let, putem aplikacije obraćaju bazi, koja za svaki od ovih zahteva pokreće isti „program" kojim se vrši čitanje broja slobodnih mesta za dati let, umanjuje se za željeni broj mesta, nakon čega se nova vrednost upisuje. Kada je reč o samim komandama upisa i čitanja, koje su gradivni blokovi svake transakcije, svako čitanje obuhvata pronalaženje i prebacivanje odgovarajućeg bloka, koji sadrži traženi element, u bafer glavne memorije (u koliko se već ne nalazi tu), iz koga se po potrebi prebacuje traženi element u odgovarajuću programsku promenljivu, dok se prilikom upisa vrši traženje i prebacivanje traženog bloka u bafer glavne memorije (u koliko se već ne nalazi tamo), nakon čega se vrši upis u odgovarajuću lokaciju bafera glavne memorije, nakon čega se, odmah ili kasnije, modifikovani blok prebacuje na disk. Svaka transakcija kao svoju zadnju operaciju ima **commit**, u koliko se je uspešno izvršila, ili u suprotnom **abort**, kojom se poništava svi efekti te transakcije učinjeni do tada [2]. U nastavku ovog polgavlja će više reči biti o svojstvima koje mora da poseduje svaka transakcija.

2.1 ACID svojstva

Svaka transakcija bi trebalo da poseduje nekoliko svojstava poznatijih kao ACID, čije bi ispunjenje bilo obezbeđeno od strane menadžera transakcija DBMS-a [3] U ova svojstva spadaju:

- Atomičnost (**A**tomicity) – Transakcija mora biti atomična jedinica obrade, što znači da ili će se sve operacije od kojih se sastoji transakcija, obaviti, ili neće ni jedna. Menadžer za oporavak, kao deo DBMS-a, mora da obezbedi da korisnik ne mora da vodi računa o nepotpunim transakcijama. Transakcija se smatra nepotpunom tj. obustavljenom (*aborted*), u koliko dodje do neke od sledećih pojava: DBMS je obustavio transakciju iz razloga što je došlo do neke anomalije prilikom izvršenja iste, došlo je do pada sistema u toku izvršenja transakcije, sama transakcija je naišla na neki neočekivani problem i obustavila samu sebe, ili je korisnik obustavio transakciju.
- Konzistentnost (**C**onsistency) – Uspešno izvršena transakcija prevodi bazu iz jednog konzistengog stanja u drugo. Za očuvanje ovog svojstva je zadužen sam korisnik, iz razloga što DBMS ne zna šta se podrazumeva pod „konzistentnim" stanjem. Primer za ovo svojstvo, koji se ondosi na prethodno pominjanu bazu sa slobodnim letovima, bi bio da korisnik sam mora da nemtne ograničenje da broj slobodnih mesta na letu, ne sme da bude negativan. Ovo bi moglo da se izvede uz pomoć specificiranja jednostavnih ograničenja konzistentnosti nad željenim podacima, koji bi bili sprovođeni od strane DBMS-a.

- Izolacija (**I**solation) – Treba da izlgeda da se svaka transakcija izvršava nezavisno od ostalih, iako se odvija preplitanje operacija iz više transakcija tj. iako se transakcije konkuretno izvršavaju. Primer ovoga bi bio, da ukoliko imamo dve transakcije T1 i T2 koje se konkurentnno izvršavaju, konačni efekat na bazu mora da bude takav kao da su se T1 i T2 sekvencijalno izvršile u nekom rasporedu (T1 pa T2, ili T2 pa T1). Za ovo svojstvo su zadužene metode kontrola konkurencije. Ovo svojstvo takođe implicira da jedna transakcija ne vidi nekomitovane promene drugih transakcija.
- Postojanost (**D**urability) – Promene od strane uspešno izvršene transakcije treba da budu trajne. Ovo znači da ove promene moraju da budu zapamćene i da se odražavaju na bazu u koliko dođe do bilo kakvog otkaza sistema. Za ovo svojstvo je zadužen menadžer za oporavak.

Menadžer transakcija obezbeđuje atomičnost time što, u koliko je došlo do bilo kakve anomalije u toku izvršenja transakcije, što je dovelo do obustavljanja iste, poništava efekat te transakcije i vraća bazu u stanje pre njenog izvršenja. Da bi ovo bilo moguće, DBMS mora da poseduje mehanizam kojim će da poništi efekat svih obavljenih operacija neuspšne transakcije. Ovo je moguće na taj način što sistem održava **log** u kome se nalaze sve operacije transakcija koje su izvršene [2]. Ovaj log se takođe koristi i da bi se forsiralo svojsto postojanosti, na taj način, što prilikom pada sisitema, pre nego što su promene od strane potvrđene (*commited*) transakcije bile zapamćene na disk, log se koristi da bi se promene ove transakcije ponovo primenile i zapamtile u bazi. Što se tiče svojstva izolacije, za to je zadužena kontrola konkurencije koja predstavlja proces upravljanja konkurentnim operacijama nad bazom, da bi se sprečila njihova medjusobna interferencija. Jedan od nabitnijih pojmova kada je reč o kontroli konkurencije jeste raspored izvršenja operacija, o kojima će u nastavku biti više reči.

2.2 Rasporedi (Schedules)

Raspored (*Schedule*) je lista operacija (čitanje, upis, commit, abort) koje pripadaju odgovarajućim transakcijama, s tim da redosled operacija u okviru jedne transakcije mora da bude isti i u raporedu [1]. Raspored praktično predstavla konkretnu, ili potencijalnu, sekvencu izvršavanja. Dve operacije unutar rasporeda se smatraju da su u **konfliktu**, u koliko obe pristupaju istom elementu, pripadaju različitim transakcijama i ako je bar jedna od te dve operacije upis. Primer jednog rasporeda je dat na slici 1. Pri čemu se sa R(A) označava operacija čitanja nekog elementa A, dok se sa W(B) označava operacija upisa nekog elementa B. Krećemo se napred kroz vreme, kako prelazimo sa reda na red naniže.

T1	T2
R(A)	
W(A)	
	R(B)
	W(B)
	commit
R(C)	
commit	

Slika 1 Primer jednog rasporeda dveju transakcija

Sa aspekta oporavka, rasporedi se mogu podeliti na: one kod kojih je oporavak moguć - ***recoverable schedules***, kod istih, jednom kada se transakcija komituje, nema potrebe za njenim *roll back*-om, odnosno, svojstvo postojanosti je očuvano i ***nonrecoverable schedules*** kod kojih je oporavak nemoguć [1]. Raspored *S* je *recovarable* ako ni jedna transakcija *T* iz *S*, ne komituje sve dok, pre toga, sve transkacije *T'* koje su prethodno upisivale u neku vrednost *X*, koju *T* kasnije čita, ne komituju (ili budu obustavljene) . Na slici 2a je dat prikaz jednog *recovarable* rasporeda, kod koga imamo da T1 komituje pre T2, što pročitanu vrednost A u transakciji T2 čini validnom. Dok na slici 2b imamo raspored koji nije *recovarable (nonrecoverable schdeule)* što se može videti iz toga što je transakcija T2 komitovala pre T1, što je dovelo do toga da, nakon što je T1 završena sa *abort*, vrednost A koju je T2 pročitala, ostaje nevalidna.

a)

T1	T2
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)
W(B)	
commit	
	commit

b)

T1	T2
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)
	commit
abort	

Slika 2 a) Recovarable b) Non recoverable schedule

Kod nekih *recovarable* rasporeda može doći do pojave kao što je ***cascading rollback***, kada nekomitovana transakcija mora da obavi *rollback* zato što je obavila čitanje elemnta koji je upisala transakcija koja je u međuvremenu obustavljena [1][2]. *Cascading rollback* može da bude veoma vremenski zahtevan proces, zato što se može desti da je potrebno poništiti veliki broj transakcija. Da bi se ovaj problem prevazišao potrebno je uvesti još neka ograničenja nad *recovarable* rasporedima. Raspored se smatra otpornim na *cascading rollbacki* tj. ***cascadeless***, ukoliko svaka transakcija čita elemente koji su upisivani od strane već komitovanih transakcija.

a)

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
abort	
	abort

b)

T1	T2
R(A)	
W(A)	
	W(B)
commit	
	R(A)
	W(A)
	commit

Slika 3 a) Cascading rollback b) Cascadeless schedule

Na slici 3a je dat primer rasporeda kod koga se javlja *cascading rollback* problem, dok je na slic 3b dat primer *cascadeless* rasporeda.

Rasporedi koji su najjednostavniji za opravak su tzv **striktni rasporedi** kod kojih transakcija ne obavlja ni čitanje niti uspis nekog elementa, sve dok poslednja tranakcija koja je izvršila upis u taj element ne komituje [1]. Važno je naglasiti da su striktni rasporedi podskup *cascadeless* rasporeda, koji su pak podskup *recovarable* rasporeda.

Rasporedi se, pored kriterijuma oporavka, mogu klasifikovati i na osnovu toga u kakvom stanju ostavljaju bazu nakon izvršenja. Rasporedi kod kojih se operacije svake transakcije izvršavaju uzastopno, bez preplitanja, nazivaju se **serijski rasporedi (*serial schedules*)** [1]. Kod ovakvih rasporeda, u svakom trenutku je samo jedna transakcija aktivna. Iako ovakav raspored garantuje konzistentan rezultat na kraju izvršenja, nameće veliko ograničenje koje se tiče iskorišćenosti resursa kroz nemogućnost konkurentnog izvršenja. Iz ovog razloga se serijski rasporedi smatraju neprihvatljivim u praksi. U koliko neserijski raspored ima ekvivalentan efekat na bazu kao i neki serijski raspored koji, sadrži iste transakcije, onda se radi o serijabilnom rasporedu (***serializable schedule***). Sam pojam **serijabilnosti** podrazumeva nalaženje neserijskog rasporeda čiji će krajnji efekat na bazu da bude ekvivalentan nekom od serijskih rasporeda, sa istim transakcijama. Pod pojmom ekvivalentnosti između dva rasporeda se može smatrati:

- **Conflict equivalence** – Dva rasporeda se smatraju *conflict* ekvivalentnim u koliko je raspored bilo koje dve konfliktne operacije isti u oba rasporeda.
- **View equivalence** – Dva rasporeda se smatraju *view* ekvivalentnim u koliko bilo koja operacija čitanja kod neke transakcije, čita rezultat upisa iste transakcije u oba rasporeda. Takođe, zadnja operacija upisa nekog elementa mora da bude ista (da pripada istoj transakciji) u oba rasporeda.

U koliko je neki neserisjki raspored *conflict* ekvivalentan sa nekom od odgovarajućih serisjkih rasporeda, reč je o *conflict* serijablinom (***conflict serializable***) rasporedu. U koliko se radi o *view* ekvivalenciji, pod istim uslovima, onda je reč o *view* serijabilnom (***view serializable***) rasporedu. Definicija *view* i *conflict* serijabilnih rasporeda je ista u koliko se nametne ograničenje da u transakcijama nema tzv ***blind* upisa**, tj. da svakom upisu, u okviru neke

transakcije, mora da prethodi njeno čitanje (u okviru iste transakcije) i da taj upis direktno zavisi od te prethodno pročitane vrednosti. U koliko ne postoji prethodno navedeni preduslov, odnosno mogući su *blind* upisi, *view* seriabilnost je manje restriktivna od *conflict* serijabilnosti. Generalno, svaki konflikt serijabilni raspored je ujedno i *view* serijabilan, dok obrnuta tvrdnja ne važi.

Cilj DBMS-a je da putem metoda i protokola (set pravila) osigura serijabilnost, bez testiranja rasporeda na isti, što bi bilo nepraktično. U koliko se svaka transakcija pridržava protokola, ili je isti forsiran od strane kontrole konkurentnosti, svaki raspored u kome učestvuju takve transakcije biće serijabilan. Neki od protokola kontrole konkurentnosti koji obezbeđuju ovakav efekat su: *two-phase* zaključavanje, uređivanje na osnovu *timestamp*-a, kao i *multiversion* protokoli. Pre nego što se detaljno objasne ove metode kontrole konkurentnosti, važno je navesti kakve su sve posledice i anomalije moguće u koliko se koristi raspored koji nije serijabilan.

2.3 Moguće anomalije prilikom konkuretnog izvršavanja

Iako pojedinačno transakcije prevode bazu iz jednog konzistentnog stanja u drugo, neserijabilni (*not serilizable*) raspored može da dovede do toga da se, nakon izvršenja, baza nadje u nekonzistentnom stanju. Na primeru rasporeda koji se sastoji od 2 transakcije biće opisane u nastavku anomalije koje se mogu javiti prilikom nekontrolisanog preplitanja operacija.

Jedna od anomalija koja se može javiti je **problem gubitka pri ažuriranju** (***lost update problem***). Kod ove vrste problema jedna transakcija vrši *update (write)* nad nekim elementom, nakon čega druga transakcija takođe vrši neki update nad istim elementom, iako je prethodna transakcija još uvek u toku, što dovodi do toga da *update* koji je izvršila prva transakcija bude izgubljen. Na slici 4a dat je primer ovakvog poblema.

Još jedna anomalija koja se može javiti je **problem privremenog ažuriranja** (***Dirty Read***). Ovaj problem se javlja kada jedna transakcija čita vrednost upisanu od strane druge transakcije koja se još uvek izvršava tj nije komitovana. Na slici 4b dat je primer ovog problema, koji se javlja usled toga što bi transakcija T1 prilikom upisa vrednosti A, mogla da ostavi bazu u privremeno nekonzistentnom stanju, koje bi se, u ovakvom rasporedu, prilikom čitanja te nevalidne vrednosti od strane transakcije T2, prenela na celu bazu. Takođe, u koliko bi se transakcija T1 završila sa *abort, dirty read* bi dovelo do nemogućnosti oporavka s obzirom da se radi o *nonrecoverable* rasporedu.

Treći izvor anomalije je **problem neponovljivog čitanja** (***Unrepeatable Read***) kod koga imamo da jedna transakcija izvrši *update* nad nekim elementom, koji je prethodno pročitan od strane neke druge aplikacije koja se još uvek izvršava, tako da u koliko prva transakcija pokuša sa ponovnim čitanjem, dobiće izmenjenu vrednost, iako je nije ona sama u medjuvremenu izmenila. Na slici 4c dat je prikaz ovog problema u slučaju rasporeda koji sadrži dve transakcije. Problem veoma sličan *Unrepeatable Read*-u je ***Phanntom Read*** kod koga se u okviru jedne transakcije dva puta izvršava isti upit nad bazom, pri čemu se kao rezultat, svaki

put dobije različiti broj redova. Razlog za ovo je najčešće taj, što neka druga transakcija unese novi red u bazu izmedju dva izvršenja upita u prvoj transakciji.

U koliko imamo raspored kod koga jedna transakcija računa neku sumarnu funkciju nad nekim brojem elemenata baze, dok neke druge transakcije vrše *update* nad nekim od tih elemenata, radi se o **problemu nekorektnog sabiranja** (***Incorrect Summary Problem***). Ovo može dovesti do toga da funkcija agregacije može da čita vrednosti nekih elemenata pre nego što su ovi modifikovani, ili nakon što su modifikovani, što može dovesti do izračunavanja nevalidniog rezultata.

a)

T1	T2
R(A)	
	R(A)
W(A)	
R(B)	
	W(A)
	commit
W(B)	
commit	

b)

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	commit
R(B)	
W(B)	
commit	

c)

T1	T2
R(A)	
	R(A)
	W(A)
	commit
R(A)	
W(A)	
commit	

Slika 4 a) Lost Update b)Dirty Read c) Unrepeatable Read

2.4 Protokoli kontrole konkurencije

Da bi se sprečile anomalije navedene u prethodnom poglavlju i da bi se obezbedilo svojstvo izolacije transakcija koje se konkuretno izvršavaju, mora da postoji set pravila – protokol kontrole konkuretnosti, kojeg će se pridržavati sve transakcije i time obezbediti serijabilnost rasporeda.

Jedna od glavnih tehnika koja se koristi za kontrolu izvršavanja konkuretnih transakcija je ***two-phsase locking (2PL)*** tehnika. Ova tehnika je bazirana na zaključavanju elemenata baze prilikom pristupa istim. Pod terminom ***lock*** smatra se promenljiva koja se dodeljuje nekom podataku i kojom se opisuje status tog podataka u odnosu na operacije koje su dozvoljene da se obavljaju nad njime. Kod bilo kog protokola koji koristi zaključavanje, transakcija ne može da pristupi elementu, dok pre toga ne pribavi odgovarjući *lock.* 2PL protokol je baziran na korišćenju ***shared/exclusive*** *lock*-ova, pri čemu, u koliko transakcija želi da obavi upis nad elementom, mora da pribavi *exclusive (write) lock,* kojim je omogućeno da samo jedna transakcija pristupa elementu radi upisa. U koliko transakcija želi da obavi čitanje, mora da pribavi *shared (read) lock*, pri čemu je dozvoljeno da više transakcija čita element istovremeno, bez blokiranja i čekanja. Na slici 5 je dat prikaz odnosa između *read* i *write lock*-a, pri čemu, u koliko transakcija *T1* ima pribavljeni *lock* čiji je tip naveden u koloni, nad nekim elementom, a transakcija *T2* zahteva *lock,* čiji je tip naveden u redu tabele, takođe nad istim elementom, *Yes* označava da transakcija T2 može da pribavi *lock,* dok *No* označava da transakcija T2, ne može da pribavi *lock,* već mora da sačeka dok transakcija T1 ne oslobodi

lock. Iz date tabele se vidi da ukoliko neka transakcija vrši upis tj pribavila je *write lock*, bilo koja druga transakcija koja želi da vrši upis ili čitanje, mora da sačeka da transkacija koja drži *lock* oslobodi isti. Takođe se vidi da u koliko je neka transakcija pribavila *read lock,* drugim transakcijama je dozvoljeno pribavljanje *read lock*-a, dok pribavljanje *write lock*-a nije moguće bez čekanja na oslobađanje [1].

	Read	Write
Read	Yes	No
Write	No	No

Slika 5 Kompatibilnost lockova

Samo korišćenje *Shared/Exclusive lock*-ova nije dovoljno da bi se garantovala serijabilnost rasporeda. *Two-phase locking protocol* nameće ograničenje da se sve operacije koje se odnose na pribavljanje *lock*-ova u transakciji, moraju naći pre **prve** operacije oslobađanja *lock*-ova. Ovo znači da je transakcija podeljena na dve faze: **širenje (*expanding, growing)*** koja je prva i u kojoj se pribavljaju novi *lock*-ovi za konkretne elemente, pri čemu u ovoj fazi nema oslobadjanja *lock*-ova i druga faza **skupljanje (*shrinking*)** u kojoj se oslobađaju zauzeti *lock*-ovi, bez pribavljanja novih. U koliko svaka transakcija u rasporedu poštuje ovakav protokol, raspored je serijabilan [1][4].

Najčešće korišćena varijanta 2PL protokola ja **striktni 2PL**, koji pored serijabilnosti omogućava i to da rasporedi koji poštuju ovaj protokol budu i striktni. Kod striktnog 2PL protokola transakcija oslobadja svoje *write lock*-ove tek prilikom komitovanja ili abortovanja iste, na ovaj način se postiže striktnost rasporeda, zato što nijedna transakcija ne može da čita ili upisuje element (pribavi *read* ili *write lock*) nad kojim je vršen upis od neke transakcije T, sve dok T ne komituje.

Jedan od problema koji se može javiti prilikom upotreba protokola baziranih na mehanizmu zaključavanja je ***deadlock***, koji se javlja kada **svaka** transakcija, iz nekog seta od 2 ili više transakcije, čeka na pristup nekom elementu, koji je zaključan od neke druge transakcije iz tog seta. Na slici 6 je dat primer *deadlock* problema, gde imamo da je T1 blokirana i čeka da T2 oslobodi element X, dok je T2 takođe blokirana i čeka da T1 oslobodi element Y. Postoje nekoliko metoda za rešavanje ovog problema, od kojih je najjednostavnija ona bazirana na *timeout*-u, kod koje u koliko transakcija čeka na neki element duže od sistemski predviđenog vremena, smatra se da je došlo do *deadlock*-a i ta transakcija se poništava. Drugi problem koji može da se javi jeste ***starvation***, kada je jedna transakcija blokirana i ne može da nastavi rad, dok druge transakcije nastavljaju sa radom normalno. Do ovoga može doći iz razloga što imamo šemu za dodelu pristupa *lock*-u, koja daje veći prioritet nekim transakcijama. Ovaj problem se može prevazići tako što bi se obezbedila fer šema pristupa, putem FIFO reda, kojim

će transakcijama biti omogućeno da pristupe elementu u redosledu u kome su pribavile *lock* za taj element.

T1	T2
Read_lock(Y)	
R(Y)	
	Read_lock (X)
	R(X)
Write_lock(X)	
W(X)	
	Write_lock(Y)
	W(Y)

Slika 6 Primer Deadlock-a

2PL protokol iako garantuje serijabilnost i striktnost (u slučaju striktnog 2PL-a), negativno utiče na konkuretnost. Do ovoga dolazi iz razloga što blokirane transakcije mogu da, pre toga i same zaključaju neke elemente, koje su potrebne drugim transakcijama da bi nastavile sa radiom, takođe, da bi se prevazišli *deadlock*-ovi, može doći do više obustavljanja i restarta transakcija.

Drugi često korišćeni protokol za kontrolu konkurentnosti je protokol baziran na **vremeskim oznakama (*timestamp*)**. Pod *timestamp*-om se podrazumeva jedinstvena vremenska oznaka koja se dodeljuje svakoj transakciji od strane DBMS-a. Uređenje *timestamp*-ova zavisi od starosti transakcije, tako da transakcija započeta pre neke druge, imaće manji *timestamp* od te mladje transakcije. Kod ***timestamp ordering*** **protokola (TO)** cilj je da raspored bude serijabilan tako da jedini dozvoljeni ekvivalentan serijski raspored bude onaj čije se transakcije izvršavaju u redosledu njihovih *timestamp*-a. Protokol mora da osigura da redosled pristupa nekom elementu od strane konfliktnih operacija u rasporedu mora da prati redosled *timestamp*-a. Da bi ovo bilo moguće, protokol za svaki element baze čuva dve promenljive: ***read timestamp*** – *timestamp* najmlađe transkacije koja je uspešno pročitala konkretan element, ***write timestamp*** – *timestamp* najmlađe transakcije koja je uspešno izvršila upis nad konkretnim elementom. TO algoritam forsira serijabilnost konkurentnih transakcija na sledeći način: kada transakcija *T* želi da izvrši upis nad nekim elemntom *X*, algoritam proverava promenljive *write timestamp* i *read timestamp* elementa *X* (*write* može da bude u konfliktu sa *read* i *write* operacijama) i u koliko se utvrdi da je bilo koja od ove dve promelnjive veća od *timestamp*-a transakcije *T*, što bi značilo da je neka mladja transakcija prethodno izvršila upis ili čitanje tog elementa i da je željeni redosled narušen, transakcija *T* se obustavlja i restartuje ali sa novim *timestamp*-om. U koliko se utvrdi da je transakcija *T* mlađa od najmladjih transakcija koje su izvršile upis i čitanje elementa *X*, transakciji *T* je dozvoljen upis, a *write timestamp* za element *X* uzima vrednost *timesatmpa T*. U koliko transakcija *T* želi da izvši čitanje elementa *X*, algoritam vrši poređenje *timestamp*-a *T* sa *write timestamp*-om od *X*, u koliko se utvrdi da je neka mlađa transakcija prethodno izvršila upis u *X*, transakcija *T* se obustavlja i restartuje sa

novim *timestamp*-om, u suprotnom se dozvolja čitanje transakciji *T*, a *read tiemstamp* za *X* se postavlja na *timestamp T*-a u koliko je ovaj veći od *read timestamp* vrednosti, u koliko nije, ostaje nepromenjen [1].

Iako su rasporedi bazirani na TO protokolu serijabilni, i dalje se mogu javiti problemi kaskadnih obustavljanja, takodje TO ne može da garantuje oporavak (*recoverability*) rasporeda. Da bi se smanjio broj obustavljanja transakcija, moguće je korišćenje modifikacije TO-a poznatije kao ***Thomas's Write Rule***. Ovaj protokol iako smanjuje broj obustavljanja transakcija prilikom upisa, ne obezbeđuje *conflict* serijabilnost već *view* serijabilnost. Kod ovog protokola, prilikom upisa od strane neke transakcije *T*, u koliko se utvrdi da je neka mladja transakcija prethodno izvršila upis, ne dolazi do obustavljanja transakcije *T*, već ista ne obavlja upis, ali nastavlja dalje sa izvršenjem. Transakcija *T* ne obavlja upis, zato što se smatra da je neka mladja transakcija već izvršila upis konkretnog elementa i tako prebrisala ono što bi *T* upisala.

Još jedan često korišćen protokol za kontrolu konkurentnosti je ***Multiversion Concurency Control (MVCC)***. Kod ovog protokola, osnovna ideja je da postoje više verzija elemenata baze, tako da čitanje elementa bude uvek omogućeno bez blokiranja i čekanja, na taj način što bi se čitale različite verzije elementa, na ovaj način bi se takođe očuvala serijabilnost rasporeda. Takođe, prilikom upisa elementa, bi se u stvari kreirala nova verzija tog elementa, pri čemu bi se stara sačuvala. Jedina mana ovoakvog pristupa je ta, što zahteva dodatan prostor za čuvanje više verzija elemenata, ali s obzirom da kod većine baza mehanizmi za oporavak već čuvaju prethodne verzije, ovaj nedostatak je time prevaziđen. Treba naglasiit ida MVCC sprečava blokiranja i čekanje kod *read – write* i *write - read* operacija, dok prilikom konkurentnog upisa (*write – write* operacija) i dalje može doći do blokiranja. MVCC se može implementirati uz pomoć *timestamp*-a ili uz pomoć 2PL protokola [1].

3. Procesiranje transakcija kod PostgreSQL-a

Kao i većina modernih DBMS-a, i PostgreSQL podržava konkuretno izvršavanje više transakcija istovremeno, kao i razne alate koji omogućavaju korisnicima baze da eksplicitno specifikuju na koji način će njihova transakcija da interaguje sa drugim transakcijama. Jedna od stvari za koje je PostgeSQL našao izuzetno efikasno rešenje jeste osiguravanje same izolacije transakcije uz pomoć korišćenja jedne od varijanti MVCC protokola, baziranog na pribavljanju *snapshot*-a baze podataka prilikom starta transakcije, što dovodi do toga da svaka transakcija radi sa svojom verzijom baze. U nastavku ovog poglavlja biće više reči o samom načinu kreiranja i kontrole transakcija, o samom MVCC mehanizmu i nivoima izolacije koje PostgreSQL obezbedjuje kako bi rešio razne anomalije kod konkuretnog izvršavanja, kao i mehanizme za eksplicitno zaključavanje i kontrolu konkurentnosti koje ova baza pruža. Uz sve ovo će biti dati i adekvatni praktični primeri.

3.1 Osnovne komande za kontrolu transakcija

PostgreSQL omogućava kombinovanje više komandi za manipulacijom nad elementima baze, kako bi se iste izvršili zajedno u obliku jedne transakcije. Jedina stvar koju korisnik baze mora da garantuje jeste konzistentnost baze nakon izvršenja transakcije. Na slici 7 je dat primer testne tabele *workers* koja će biti korišćena u primerima kroz ovo poglavlje.

```
testdb=# select * from workers;
 id |  name  | salary | on_duty
----+--------+--------+---------
  1 | Pera   |  14000 | t
  2 | Laza   |  10000 | t
  3 | Marko  |  12000 | f
  4 | Luka   |  18000 | f
  5 | David  |  20000 | t
  6 | Stefan |  25000 | f
  7 | Ana    |  17000 | t
  8 | Maria  |  18000 | f
  9 | Robert |  15000 | f
 10 | Milos  |  20000 | t
(10 rows)
```

Slika 7 Workers test tabela

U koliko bismo želeli da izvršimo smanjenje plate jednom radniku, a povećanje plate drugom, za istu sumu (ukupan budžet mora ostati isti, to je uslov konzistentnosti), upotrebili bismo transakciju koja je prikazana na slici 8a. Kao što se sa date slike može videti, transakcija se startuje sa komandom BEGIN, što je takođe moguće i sa komandama: BEGIN TRANSACTION ili BEGIN WORK. Da bi promene nad bazom bile vidljive od strane drugih transakcija, kao i da bi se osiguralo svojstvo postojanosti, potrebno je transakciju završiti sa COMMIT komandom (moguće je i sa COMMIT TRANSACTION ili COMMIT WORK komandama). Na slici 8 se takođe može videti i upotreba funkcije *txid_current( )*, čijim se pozivom nakon starta transakcije, prikazuje id trenutne transakcije – ***txid***, koja je dodeljena od strane menadžera transakcija.

U koliko smo u transakciju uneli neke izmene koje ne želimo da se odraze na bazu, transakciju je moguće završiti sa komandom ROLLBACK (isto je moguće i sa ROLLBACK

TRANSACTION ili ROLLBACK WORK), čime bi se transakcija obustavila i baza vratila u stanje pre njenoog izvršenja. Primer upotrebe ove komande data je na slici 8b, gde se u okviru transakcije vrši uklanjanje reda iz tabele, pri čemu se krajnji efekat ove transakcije poništava ROLLBACK komandom.

a)

```
testdb=# BEGIN;
BEGIN
testdb=# SELECT txid_current();
 txid_current
--------------
          770
(1 row)


testdb=# UPDATE workers SET salary = salary - 1000 WHERE name = 'Pera';
UPDATE 1
testdb=# UPDATE workers SET salary = salary + 1000 WHERE name = 'Laza';
UPDATE 1
testdb=# COMMIT;
COMMIT
testdb=# SELECT * FROM workers
testdb-# ;
 id |  name  | salary | on_duty
----+--------+--------+---------
  3 | Marko  |  12000 | f
  4 | Luka   |  18000 | f
  5 | David  |  20000 | t
  6 | Stefan |  25000 | f
  7 | Ana    |  17000 | t
  8 | Maria  |  18000 | f
  9 | Robert |  15000 | f
 10 | Milos  |  20000 | t
  1 | Pera   |  13000 | t
  2 | Laza   |  11000 | t
(10 rows)
```

b)

```
testdb=# BEGIN;
BEGIN
testdb=# DELETE FROM workers WHERE name = 'David';
DELETE 1
testdb=# SELECT * FROM workers;
 id |  name  | salary | on_duty
----+--------+--------+---------
  3 | Marko  |  12000 | f
  4 | Luka   |  18000 | f
  6 | Stefan |  25000 | f
  7 | Ana    |  17000 | t
  8 | Maria  |  18000 | f
  9 | Robert |  15000 | f
 10 | Milos  |  20000 | t
  1 | Pera   |  13000 | t
  2 | Laza   |  11000 | t
(9 rows)


testdb=# ROLLBACK;
ROLLBACK
testdb=# SELECT * FROM workers;
 id |  name  | salary | on_duty
----+--------+--------+---------
  3 | Marko  |  12000 | f
  4 | Luka   |  18000 | f
  5 | David  |  20000 | t
  6 | Stefan |  25000 | f
  7 | Ana    |  17000 | t
  8 | Maria  |  18000 | f
  9 | Robert |  15000 | f
 10 | Milos  |  20000 | t
  1 | Pera   |  13000 | t
  2 | Laza   |  11000 | t
(10 rows)
```

Slika 8 a) Komitovana transakcija *b)Obustavljena transakcija*

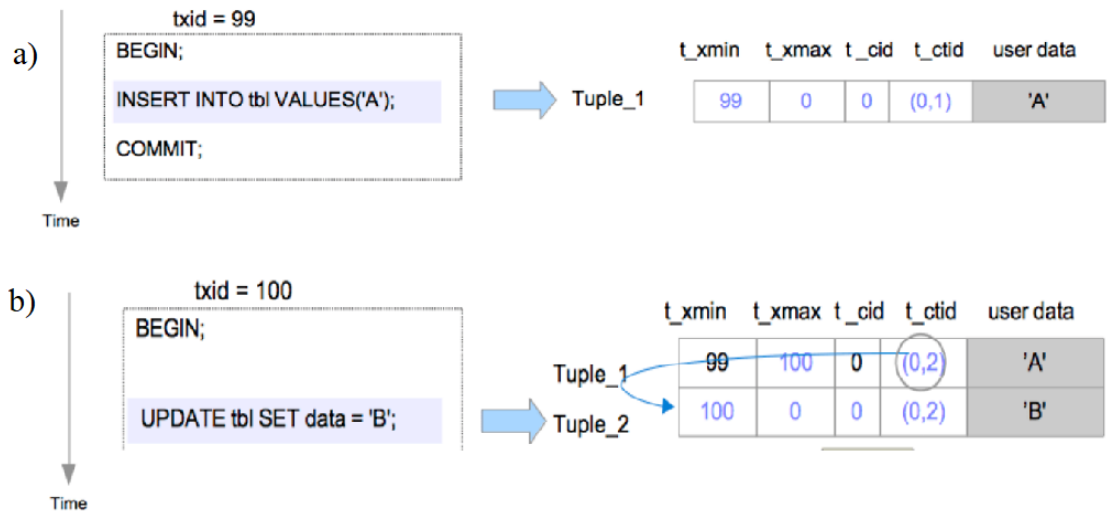
PostgreSQL omugućava i granularniju kontrolu transakcije korišćenjem SAVEPOINT i ROLLBACKA TO komandi. Komandom ROLLBACK TO na neki *savepoint* se poništavaju efekti svih komandi izdatih **nakon** definisanja SAVEPOINT-a. Takođe ROLLBACK TO komanda implicitno uništava sve *savepoint*-e nakon onog na koji je transakcija vraćena. Treba imati u vidu da se nakon izvršavanja ROLLBACK TO komande, transakcija i dalje nalazi u statnju izvršavanja i da bi komande pre *savepoint*-a imale efekta, potrebno je završiti transakciju sa COMMIT. Na slici 9 dat je prikaz transakcije koja koristi SAVEPOINT i ROLLBACK TO komande. Sa slike se vidi da je data transakcija napravila *savepoint* nakon upisa novog reda, nakon čega je isti red modifikovan. ROLLBACK TO komandom koja je usledila, stanje transakcije je vraćeno u ono neposredno pre modifikovanja reda. Nakon ovoga se transakcija izvršava normalno do kraja.

```
testdb=# BEGIN;
BEGIN
testdb=# INSERT INTO workers(name,salary,on_duty) VALUES ('Adam',25000,'yes');
INSERT 0 1
testdb=# SAVEPOINT my_first_savepoint;
SAVEPOINT
testdb=# UPDATE workers SET salary = salary * 2 WHERE name = 'Adam';
UPDATE 1
testdb=# ROLLBACK TO SAVEPOINT my_first_savepoint;
ROLLBACK
testdb=# UPDATE workers SET on_duty = 'no' WHERE name = 'Adam';
UPDATE 1
testdb=# COMMIT;
COMMIT
testdb=# SELECT * FROM workers WHERE name = 'Adam';
 id | name | salary | on_duty
----+------+--------+---------
 15 | Adam |  25000 | f
(1 row)
```

Slika 9 Primer kontrole subtransakcije

3.2 MVCC kod PostgreSQL-a

U nameri da obezbedi kontrolu konkurentnosti uz minimum čekanja i blokiranja, koji su česti prilikom upotrebe 2PL zaključavanja, PostgreSQL se oslanja na varijaciju MVCC-a. Za razliku od 2PL-a gde transakcije koje čitaju mogu da blokiraju one koje vrše upis i obrnuto, MVCC omogućava da svaka komanda radi sa jednom konkretnom verzijom baze – *snapshot*, pri čemu ova verzija ne mora da odražava trenutno stanje baze. Ovakav pristup omogućava da transakcije koje vrše upis nad nekim elementom ne blokiraju one koje vrše čitanje i obrnuto. Ovakav pristup takođe zahteva pamćenje više verzija jednog reda (*tuple*), s tim što se uz samu vrednost reda, pamte još i dodatna polja kao što su *t_xmin* koje označava id transakcije koja je unela red u bazu i *t_xmax* koje označava id transakcije koja je izvršila update nad tim redom ili brisanje reda (u koliko je ovo polje 0, radi se o aktuelnom redu). Na slici 10a dat je prikaz unosa novog reda komandom INSERT, ovom prilikom se polje *t_xmin* postavlja na id transakcije koja izvršava unos, u ovom slučaju to je 99, dok je *t_xmax* postavljen na 0, što znači da red nije brisan niti je vršen update nad njim. U koliko se izvrši update nad ovim redom od strane neke transkakcije, što se vidi na slici 10b, *t_xmax* polje se postavlja na id transakcije koja vrši update - 100, čime se označava da je ovaj red logički obrisan. Takođe, update-om se dodaje novi, aktuelni, red u bazu, kome se *t_xmin* postavlja na id transakcije koja je izvršila update, a *t_xmax* na 0 [5].



Slika 10 Format redova i primer: a) Insert-a b) Update-a [5]

Kada je reč o operaciji uklanjanja reda, ovija se po sličnom principu kao i update, odnosno vrši se logičko uklanjanje, postavljanjem *t_xmax* vrednosti na id transakkcije koja je izvršila brisanje.

Kod MVCC-a svaka transakcija mora da vidi najviše jednu verziju reda, da bi ovo bilo moguće koristi se ***snapshot* transakcije** koji se kreira u određenom momentu. Pod *snapshot*-om se porazumeva dataset, odnosno par brojeva kojima se označava koje su transakcije u jednom trenutku aktivne (u stanju izvršavanja, ili još nisu startovane) ili neaktivne (komitovane ili abortovane) iz aspekta konkretne transakcije. *Snapshot* omogućava da se uzmu u obzir jedino one transakcije koje su komitovale pre pribavljanja *snaphot*-a, odnosno da se ne uzimaju obzir transakcije koje su započele pre pribavljanja *snaphot*-a a nisu još uvek komitovane, ili one transakcije koje su startovale posle pribavljanja *snapshot*-a.

Još jedna bitna informacija koja je PostgreSQL-u potrebna kod doređivanja koja je verzija reda vidljiva od strane konkretne transakacije, jeste ***Commit Log (clog)*** koji sadrži status svake transakcije (izvršava se, komitova je ili abortovana). *Clog* je smešten u okviru posebnog fajla na disku (PGDATA/pg_xact), s tim što su često korišćeni podaci smešteni na zajedničkoj memoriji, zbog efikasnosti. Za održavanja *clog*-a, kao i uklanjanje verzije redova koji se više ne koriste, zadužen je ***Vacuum*** proces [6].

Da li je neka verzija reda vidljiva ili ne, određenoj transakciji, određuje se uz pomoć: polja same verzije reda (*t_xmin, t_xmax*), *clog*-a i pribavljenog *snapshoot*-a.

3.3 Nivoi izolacije

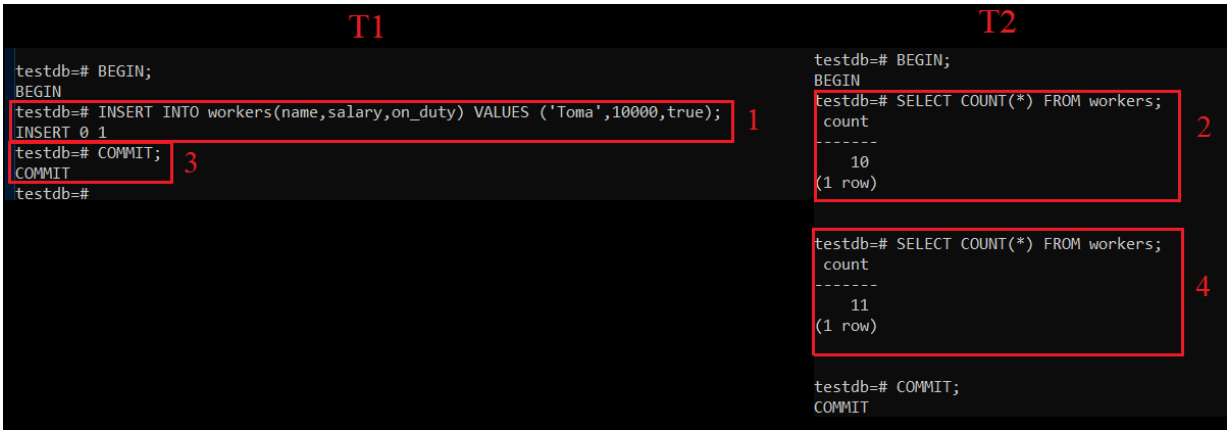
Sam SQL standard definiše 4 nivoa izolacije, pri čemu svaki naredni nivo nameće strožija ograničenja kada je reč o tome koje anomalije, kod konkuretnog izvršavanja transakcija, se

mogu tolerisati. Nivoi izolacije od najslabijeg do najrestriktvnijeg, koje korisnik može prema potrebi da koristi, su [7] [8]:

- ***Read Uncommited***
- ***Read Commited***
- ***Repeatable Read***
- ***Serializable***

Read Uncommited nivo ne sprečava transakciju da čita nekomitovane promene drugih transakcija, tj ne sprečava ***Dirty Read*** anomaliju. PostgreSQL ne implementira ovaj nivo, iz razloga što MVCC mehanizam sam po sebi sprečava *dirty read* anomaliju.

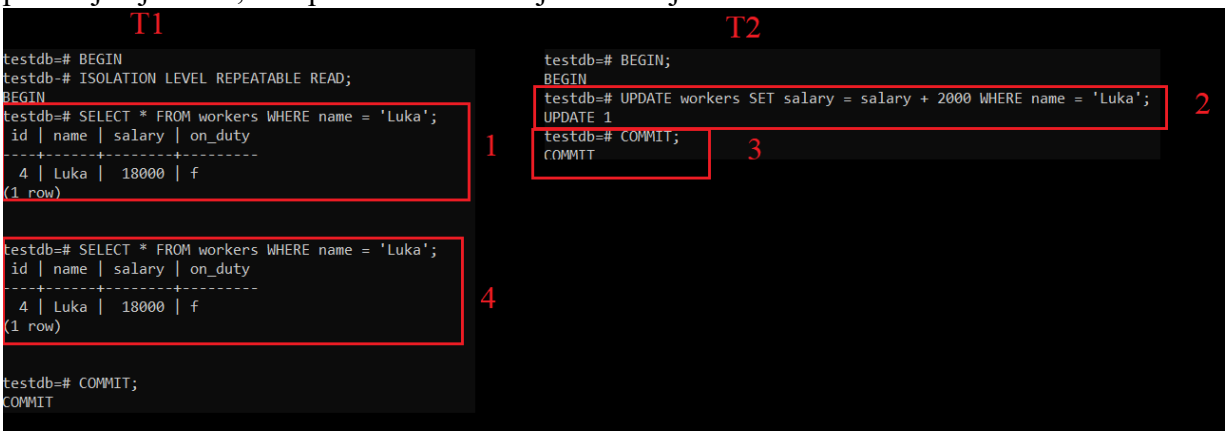
Read Commited je *default*-ni nivo izolacije kod PostgreSQL-a. Kod ovog nivoa, transakcija pribavlja *snapshot* baze, **pre izvršenja svakog upita**. Iako sprečava *dirty read* anomaliju, kod ovog nivoa, dve iste SELECT komande u okviru jedne transakcije mogu da daju različite rezultate, usled komitovane promene od strane neke druge transakcije između ova dva čitanja. Na slici 11 dat je primer izvršenja 2 transakcije čiji je nivo izolacije *Read Commited*, pri čemu je brojevima označen redosled izvršenja komandi transakcije. Za primer je uzeta *workers* tabela sa slike 7, koja inicijalno sadrži 10 redova. U koliko u momentu 1 u transakciji T1, izvršimo dodavanje novog reda, a u transakciji T2, u momentu 2, izdamo upit koji vraća broj redova u tabeli, vidimo da T2 ne vidi nekomitovanu izmenu dodavanja reda od T1, što onemogućava *dirty read*, ali nakon što T1 komituje u trenutku 3, a T2 ponovo izda istu komandu za pribavljanja broja redova u tabeli u trenutku 4, vidimo da je broj redova uvećan za 1, što praktično pokazuje da je došlo do *Phantom Read* anomalije koja je moguća kod *Read Commited* nivoa izolacije.



Slika 11 Primer Read Commited transakcije

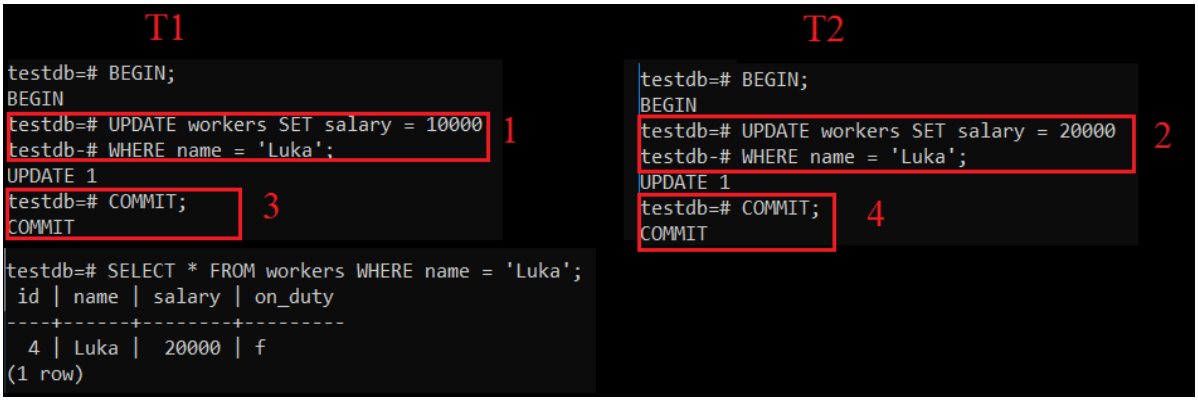
Repeatable Read nivo izolacije, za razliku od *Read Commited* nivoa, pribavlja *snapshot* baze jednom, **pre početka izvršavanja transakcije**, što znači da sve komande u okviru transakcije vide isto stanje baze. Posledica ovoga je da se anomalije poput *unrepeatable read*-a i *phantom read*-a ne mogu desiti na ovom nivou izolacije. Na slici 12 dat je primer transakcije koja koristi *Repeatable Read* nivo izolacije. Sa slike se vidi da u koliko se u momentu 1 izvrši čitanje

konkretnog reda od strane transakcije T1, čiji je nivo izolacije *Reapeatable Read*, a zatim transkcija T2 izvrši *update* nad tim istim redom i komituje (momenti 2 i 3), kada transkcija T1 ponovo izvrši čitanje, zateći će isto stanje kao i momentu 1, zato što je *snaphsot* baze pribavljen jednom, kod početka izvršavanja transkcije.



Slika 12 Primer funkcionisanja Repeatable Read nivoa izolacije

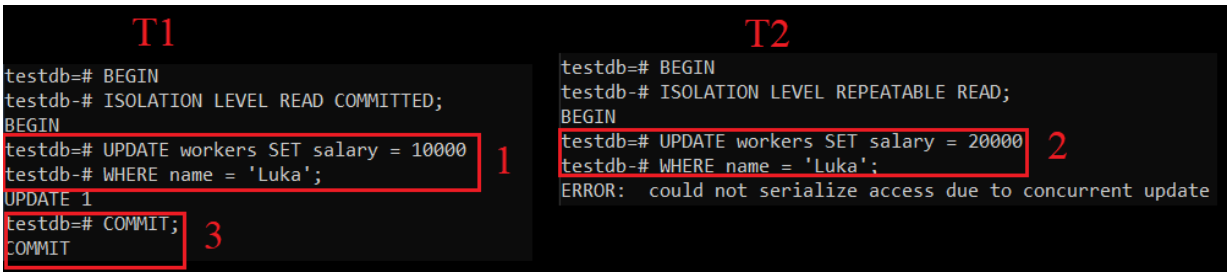
Kada je reč o konkurentnom upisu istog elementa, može doći do blokiranja i čekanja usled zaključavanja. PostgreSQL sledi *first-updater-win* pricip, kod koga jedna transakcija, prilikom *update*-a, mora da sačeka da transakcija koja trenutno vrši *update* završi i u koliko je ova završila sa *commit,* na osnovu nivoa izolacije se odlučuje da li će novi *update* biti dozvoljen. U koliko se radi o *Read Commited* nivou izolacije transakcije koja pokušava *update*, može doći do *Lost Update* problema kao što je prikazano na slici 13.



Slika 13 Lost Update problem kod Read Commited nivoa izolacije

Na slici 13 se vidi da transakcija T1 u trenutku 1 vrši update nad konkretnim redom, takođe transakciaj T2 takođe pokušava da modifikuje isti red u trenutku 2, što dovodi do blokiranja iste, sve dok T1 ne izvrši komit u trenutku 3, nakon čega T2 izvršava *update* i komituje u trenutku 4, brišući samim tim vrednost koju je T1 prethodno upisala. Kod transakcija sa nivoom izolacije *Repeatable Read* ili *Serializable, Lost Update* anomalija je otklonjena tako što se obustvalja transakcija koja je pokušala update nad elementom, nad kojim se vrši update od strane neke druge transakcije. Na slici 14 je dat primer ovoga. Kada transakcija T2 pokuša da izvrši *update* nad redom koji trenutno *update*-uje transakcija T1, T2 se blokira sve dok T1

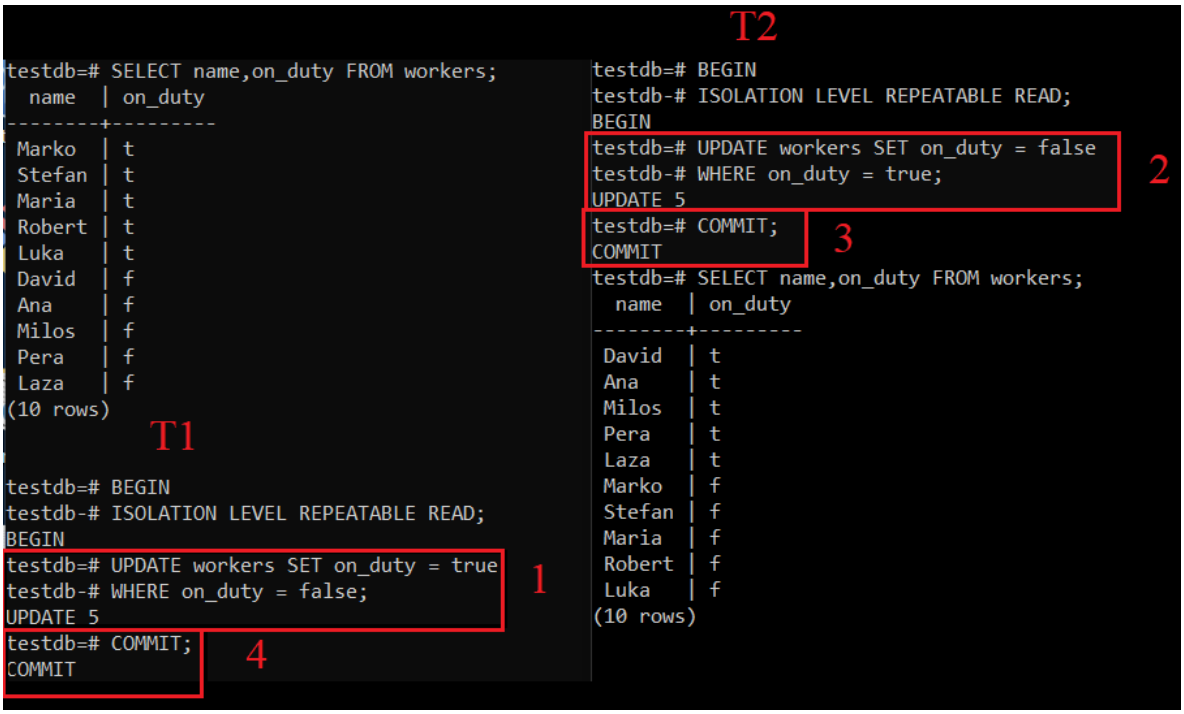
ne komituje, nakon čega se T2 obustavalja (uz obaveštenje o grešci) da bi se sprečila *Lost Update* anomalija.



Slika 14 Primer sprečavanja Lost Update-a korišćenjem Repeatable Read nivoa izolacije

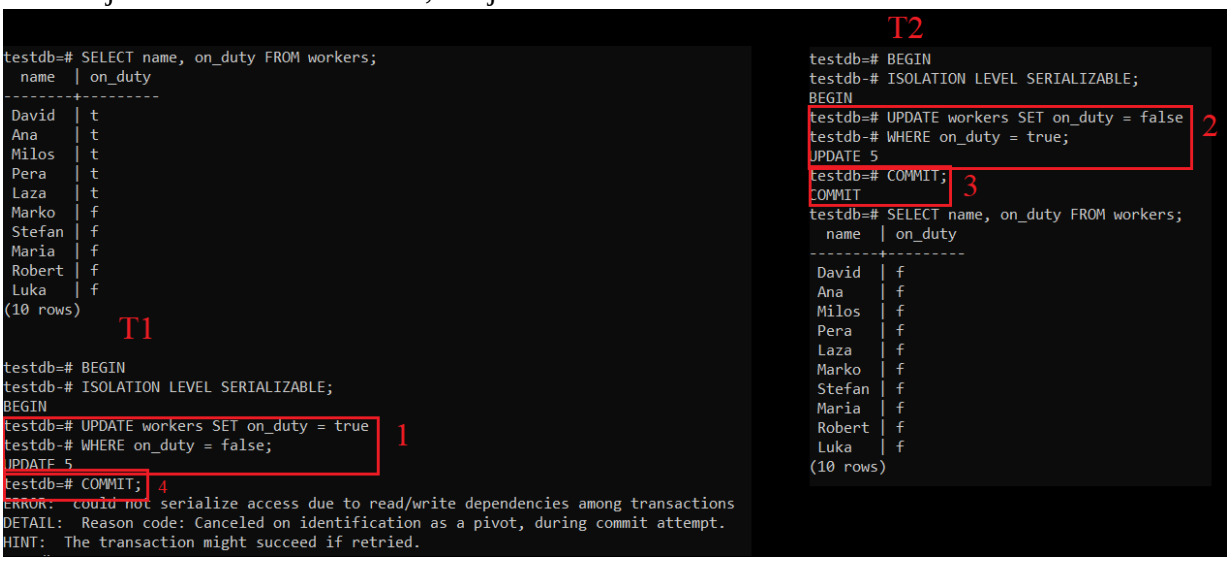
Serializable nivo je najrestriktivniji nivo izolacije koji sprečava ne samo anomalije koje sprečavaju nivoi ispod njega, već i tzv anomalije serijalizacije (*serialization anomaly*). Pod anomalijama serijalizacije podrazumeva se da prilikom komitovanja više transakcija, baza ostaje u nekonzistentnom stanju, u kome se ne bi našla da su transakcije izvršavane jedna za drugom u bilo kom redosledu. *Serializable* nivo praktično emulira serijsko izvršavanje transakcija, samim time otklanjajući sve anomalije koje se mogu javiti kod konkurentnog izvršavanja. Iako korisnik ne mora da vodi računa o anomalijama, treba imati u vidu da će neke transakcije morati da se restartuju usled obustavljanja istih radi obezbeđivanja serijabilnosti. Ovaj nivo izolacije je zasnovan na praćenju uslova koji mogu dovesti do toga da se naruši serijabilnost izvršavanja transakcija.

Kao primer anomalije izolacije, koju je nemoguće otkloniti sa *Repeatable Read* nivoom, uzet je problem *write-skew*-a. Ovaj problem se javlja kada dve konkuretne transakcije vrše čitanje redova koji se preklapaju, nakon čega vrše update i komitovanje [5]. Kao krajnji rezultat dobijamo stanje baze, koje nije isto kao nakon bilo kog sekvencijalnog izvršavanja istih transakcija. Kao primer ove anomalije uzet je slučaj kada imamo transakciju T1 koja želi da izvrši update nad tabelom *workers* (slika 7) na taj način što postavlja sve redove čije je polje *on_duty false* na *true*, dok transakcija T2 ima za cilj suprotan update, tj da svim redovima čije je polje *on_duty true* da se postavi na vrednost *false*. Kada bi se ove transakcije izvršile jedna za drugom, krajnji rezultat bi bio da svi redovi imaju istu vrednost za polje *on_duty* (*true* ili *false*, u zavisnosti od redosleda). Međutim, iz razloga što je MVCC baziran na različitim verzijama redova, ovo dovodi do toga da je krajnji rezultat zamena vrednosti, kao što se može videti sa slike 15.



Slika 15 Primer Write Skew anomalija kod Repeatable Read nivoa izolacije

Write skew anomalija se može uspešno sprečiti korišćenjem *Serializable* nivoa izolacije na taj način što prva transakcija koja komituje, njene promene su važeće, dok se druga transakcija, prilikom njenog komitovanja,obustavlja uz obaveštenje o grešci. Primer rešavanja *write skew* anomalije kod *Serializable* nivoa, dat je na slici 16.



Slika 16 Primer sprečavanja Write skew anomalije korišćenjem Serailizable nivoa izolacije

3.4 Eksplicitno zaključavanje

Pored implicitnog zaključavanja koje obezbedjuje MVCC mehanizam, PostgreSQL omogućuje i mehanizme kojima korisnik može eksplicitno da kontroliše konkurentan pristup podacima u

bazi. Različiti modovi zaključavanja se koriste za kontrolu konkurencije. Glavna razlika između ovih modova je u tome sa kojima od ostalih modova je konkretan mod u konfliktu. Dve transakcije ne mogu da drže *lock*-ove koji pripadaju konfliktnim modovima nad istom tabelom u isto vreme, dok sa druge strane nekonfliktni *lock*-ovi mogu da budu pribavljeni za istu tabelu od strnae više transakcija. Većina PostgreSQL komandi automatski pribavlja *lock*-ove odgovarajućih modova pre izvršavanja, da ne bi došlo do ne željenih modifikovanja tabela od strane drugi transakcija, dok se komanda izvršava. PostgreSQL omogućuje zaključavanje tabela, redova i stranica. Važno je naglasiti da kada transakcija pribavi neki *lock*, ona ga oslobađa tek prilikom komitovanja ili obustavljanja (abortovanja), kao i to da jedna transakcija nikad ne može da dodje u konflikt sa samim sobom. Na slici 17 dat je prikaz svih modova **zaključavanja tabela** kao i u kakvom su međusobnom odnosu (konfliktnom ili nekonfliktnom). Sa *X* se označava da su modovi u konfliktu, prazno polje označava suprotno.

TRAŽENI LOCK MOD	TRENUTNI LOCK MOD							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

Slika 17 Modovi zaključavanja tabela [9]

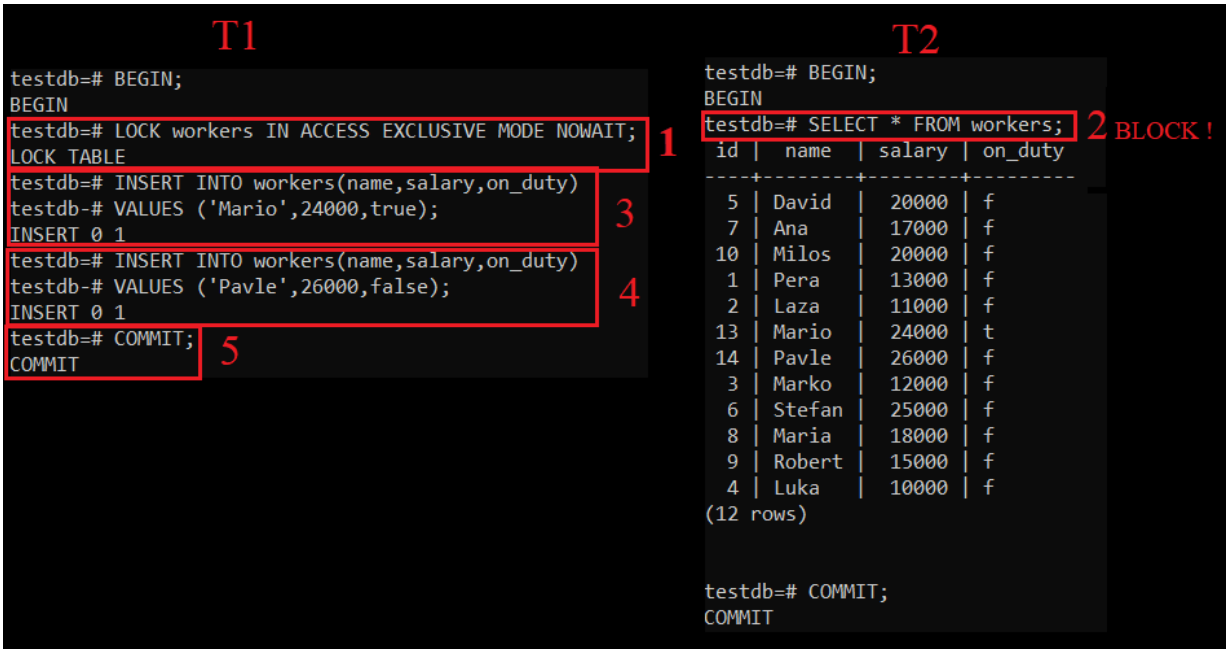
U nastavku će biti date sonvone karakteristike savakog od modova navedenih na slici 17 kao i koje ga komande PostreSQL-a implicitno koriste [9] [10]:

- ACCESS SHARE – najmanje restriktivan mod, u konfliktu je samo sa ACCESS EXCLUSIVE modom. SELECT komanda automatski pribavlja *lock* ovog moda na referenciranu tabelu, svaki upit koji vrši samo čitanje bez *update*-a pribavlja *lock* ovog moda.
- ROW SHARE – U konfliktu je sa EXCLUSIVE i ACCESS EXCLUSIVE modovima. Komande SELECT FOR UPDATE i SELECT FOR SHARE implicitno pribavljaju *lock* ovog moda na referentnu tabelu.
- ROW EXCLUSIVE – u konfliktu je sa: SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE i ACCESS EXCLUSIVE modovima. Komande UPDATE, DELETE i INSERT pribavljaju *lock* ovog moda nad referenciranom tabelom. Sve komande koje vrše *update* nad tabelom pribavlajju ovu vrstu *lock*-a.
- SHARE UPDATE EXCLUSIVE – u konfliktu je sa: *lock*-om istog moda, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE i ACCESS EXCLUSIVE modovima. Glavana namena ovog moda je da štiti od promena šeme baze, kao i od VACUUM prolaza kojima se uklanjaju nepotrebne verzije redova. Komande: VACUUM, ANALYZE, CREATE INDEX CONCURENTLY i neke varijante ALTER TABLE komande, koriste *lock*-ove ovog moda.

- SHARE – U konkfliktu je sa svim ostalim modovima osim sa: ACCESS SHARE, ROW SHARE i SHARE modovima. Komanda CREATE INDEX pribavlja *lock* ovog moda nad referenciranom tabelom.
- SHARE ROW EXCLUSIVE – u konfliktu je sa svim modovima osim sa ACCESS SHARE i ROW SHARE. Ovaj mod štiti tabelu od konkurentnih *update*-a, a takođe je i samo isključiv, što znači da *lock* ovog moda nad referenciranom tabelom može da drži samo jedna transakcija u jednom trenutku. Komande: CREATE COLLATION, CREATE TRIGGER i mnoge varijante ALTER TABLE komande pribavljaju *lock* ovog tipa.
- EXCLUSIVE – u konfliktu je sa svim modovima osim sa ACCESS SHARE, što znači da omogućava samo konkurentno čitanje dok konkrenta transakcija drži *lock* ovog tipa. Komanda REFRESH MATERIALIZED VIEW CONCURRENTLY koristi *lock* ovog moda.
- ACCESS EXCLUSIVE – najrestriktivniji mod, u konfliktu je sa svim modovima, što znači da je transakciji koja pribavi *lock* ovog tipa, zagrantovano da je jedina transkcija koja pristupa tabeli, dok se ne izvrši. Komande: DROP TABLE, TRUNCATE, REINDEX, CLUSTER, VACUUM FULL, REFRESH MATERIALIZED VIEW, kao i mnoge varijante ALTER TABLE komande pribavljaju *lock* ovog tipa. Prilikom eksplicitnog navođenja LOCK komande, podrazumeva se pribavljanje *lock*-a ovog moda.

Korisniku je eksplicitno zaključavanje kod PostgreSQL-a omogućeno korišćenjem LOCK komande čija je sintaksa: LOCK [TABLE] [ONLY] *name* [IN *lock_mode* MODE] [NOWAIT]. Sa *name* se navodi naziv tabele čije se zaključavanje želi. U koliko se ne navede ključna reč ONLY, izvršiće se zaključavanje željene tabele ali i svih tabela koje je nasleđuju. Sa *lock_mode* se specificira mod zaključavanja, dok ključna reč NOWAIT specifikuje da, u koliko se desi blokada transakcije usled zaključavanja, da konkretna transakcija ne čeka na oslobadjanje *lock*-a, već da se obustavi.

Na slici 18 dat je primer eksplicitne upotrebe zaključavanja, gde transakcija T1 najpre izvrši zaključavanje u momentu 1, nakon čega T2 izdaje komandu za čitanjem tabele, što dovodi do implicitnog pokušaja za pribavljanjem ACCESS SHARE *lock*-a, koji je u konfliktu sa trenutnim ACCESS EXCLUSIVE *lock*-om, što dovodi do blokiranja T2 transakcije. Nakon što T1 transkacija izvrši upis (momenti 3 i 4) i komituje (momenat 5), dolazi do oslobađanja *lock*-a od strane T1 i T2 može pročitati tabelu. Sa slike 18 se još vidi da transakcija T2 vidi novo dodate redove od strane T1.



Slika 18 Primer eksplicitnog zaključavanja tabele

Pored zaključavanja tabela, PostgreSQL obezbeđuje i **zaključavanje na nivou redova**. Zaključavanje redova se odnosi samo na operacije zaključavanja i upisa istih redova tabele, dok se čitanje sprovodi neometano. Na slici 19 data je tabela na kojoj su prikazani modovi zaključavanja redova, kao i odnosi ovih *lock*-ova kada je reč o konfliktima (sa *X* je označeno prisustvo konflikta, prazno polje je odsustvo istog). Dve transakcije ne mogu da drže *lock*-ove koje su u konfliktu nad istim redom istovremeno.

TRAŽENI LOCK MOD	TRENUTNI LOCK MOD			
	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE				X
FOR SHARE			X	X
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

Slika 19 Modovi zaključavanja redova [9]

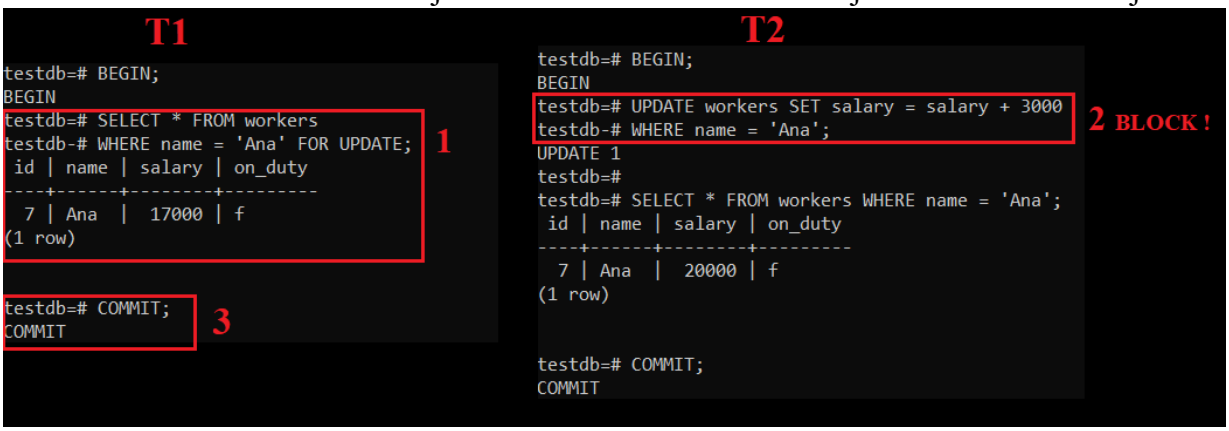
U nastavku će biti dat opis svakog od modova, kao i komande koje implicitno pribavljaju *row lock* datog moda pre početka izvršavanja [9]:

- FOR UPDATE – najrestriktivniji mod, zaključava sve redove pribavljene kao rezultat SELECT komande, što znači da se isti redovi ne mogu zaključati od strane neke druge

transakcije kao i da se nad njima ne mogu izvršavati komande kao što su: UPDATE, DELETE, SELECT FOR UPDATE, SELECT FOR NO KEY UPDATE, SELECT FOR SHARE, SELECT FOR KEY SHARE. Takođe, u koliko je, pre zaključavanja, bilo koji drugi mod (uključujući i FOR UPDATE) već pribavljen od strane neke druge transakcije, doći će do blokiranja. Kod komande brisanja reda – DELETE, implicitno se zaključava red *lock*-om ovog moda.

- FOR NO KEY UPDATE – slabiji *lock* mod od FOR UPDATE, iz razloga što nije u konfliktu sa FOR KEY SHARE, što znači da neće doći do blokiranja ako neka druga transkacija izda komandu SELECT FOR KEY SHARE nad istim redom. Ova vrsta *lock*-a se pribavlja kod nekih varijanti UPDATE komande, koja ne zahteva FOR UPDATE *lock*.
- FOR SHARE – Sličan mod sa FOR NO KEY UPDATE, s tim što ne pribavlja *exclusive*, već *shared lock* nad redom. Ovo znači da druge transakcije ne mogu da izdaju komande kao što su: UPDATE, DELETE, SELECT FOR UPDATE ili SELECT FOR NO KEY UPDATE, nad istim redom koji je zaključan u ovom modu, ali mogu da izdaju komande nad istim kao što su: SELECT FOR SHARE i SELECT FOR KEY SHARE
- FOR KEY SHARE – najslabiji *lock* mod. U konfliktu je samo sa FOR UPDATE modom. Blokiranje nastaje samo u slučaju kada druge transakcije pokušaju sa operacijama UPDATE, DELETE ili SELECT FOR UPDATE komande nad istim redom.

Na slici 20 je dat primer zaključavnja redova. Sa slike se vidi da T1 transakcija najpre zaključava red sa FOR UPDATE *lock*-om u momentu 1, nakon čega T2 pokušava da izvrši upis u isti red, što dovodi do blokiranja ove transakcije u momentu 2. Transakcija T2 nastavlja sa radom tek nakon što T1 komitijue i oslobodi *lock* nad redom koji T2 želi da modifikuje.



Slika 20 Primer zaključavanja reda

PostgreSQL takođe obezbeđuje i **zaključavanje na nivou stranica** (blokova). Ovo zaključavanje se vrši uz pomoć *exclusive/share lock*-ova i odnosi se na stranice pribavljene u *buffer pool*-u. Čim se odgovarajući red pronadje ili *update*-uje u okviru odgovarajućeg bloka, *lock* nad blokom se oslobađa.

PostgreSQL još obezbeđuje mogućnost da korisnici kreiraju i koriste *lock*-a koji je svojstven za aplikaciju koja koristi bazu – ***advisory lock***. O radu ovakve vrste *lock*-a zadužena je sama aplikacija, a ne RDBMS. *Advisory lock* se može pribaviti na nivou sesije ili na nivou transakcije, takođe oslobadjanje se i vrši prilikom završetka sesije, odnosno transakcije.

4. Zaključak

Baza podataka predstavlja zajednički resurs koji mogu da koriste više procesa i korisnika konkuretno. Jedna transakcija predstavlja logičku jedinicu procesiranja u bazi, odnsosno sve operacije koje pristupaju bazi i nalaze se između startne i završne oznake za transakciju se smatraju delom jedne, nedeljive, logičke celine. Kod aplikacija kao što su: bankraski sistemi, sistemi za rezervaciju avio ili železničkih karata, sistemi za upravljanje akcijama na berzi, baza podataka mora da bude u stanju da istovremeno procesira veliki broj transakcija koje joj pristižu od korisnika, a da pri tome konzistentnost baze ostane očuvana. Konkuretno izvršavanje transakcija bez kontrole bi moglo da dovede do nekonzistentnosti baze, *deadlock*-ova, loše performanse, pa i otkaza samog sistema. Da ne bi do ovih anomalija došlo i da bi se garantovalo svojstvo izolacije transakcije, koriste se protokoli za kontrolu izolacije kao što su: 2PL (*Two Phase Locking*) protokol, *timestamp* protokol, ili MVCC (*Multi Vesrion Concurency Control*). Takođe u nekim sistemima je moguće definisati nivoe izolacije, kojima se specifikuje nivo tolerancije sistema na razne anomalije prouzrokovane konkurentnim izvršavanjem.

Kao i većina naprednih RDBMS-ova, PostgreSQL obezbeđuje konkurentan pristup bazi, dok se istovremeno stara o ACID svojstvima transakcija. PostgreSQL omogućuje korisnicima, pored osnovne kontrole, da definišu ponašanje transakcija kada je reč o svojstvu izolacije. PostgreSQL koristi MVCC protokol kada se radi o konkuretnom pristupu redovima. MVCC pristup podrazumeva da se u bazi čuvaju više verzija istog reda i da se na taj način operacije čitanja i upisa nad istim elementom odvijaju bez blokiranja, tako što bi svaka transakcija radila sa adekvatnom verzijom reda bez potrebe za zaključavanjem (osim u slučaju konkurentnog upisa). Takođe PostgreSQL daje mogućnost korisniku da na efikasan način, definiše nivo izolacije koji mu je potreban, prilikom pokretanja neke transakcije. U koliko je potrebno, korisnik uvek može da eksplicitno kontroliše pristup tabeli ili redovima kroz sistem eksplicitnog zaključavanja.

5. Literatura

[1] Ramez Elmasri and Shamkant Navathe. 2010. *Fundamentals of Database Systems* (6th. ed.). Addison-Wesley Publishing Company, USA.

[2] Raghu Ramakrishnan and Johannes Gehrke. 2002. Database Management Systems (3rd. ed.). McGraw-Hill, Inc., USA.

[3] https://cs.elfak.ni.ac.rs/nastava/pluginfile.php/18034/mod_resource/content/1/Oporavak2016.pdf

[4] https://blog.gojekengineering.com/on-concurrency-control-in-databases-1e34c95d396e

[5] http://www.interdb.jp/pg/pgsql05.html

[6] https://edu.postgrespro.com/2dintro/03_arch_mvcc.pdf

[7] https://www.postgresql.org/docs/9.1/transaction-iso.html

[8] https://pgdash.io/blog/postgres-transactions.html

[9] https://www.postgresql.org/docs/9.1/explicit-locking.html

[10] https://engineering.nordeus.com/postgres-locking-revealed/