



Univerzitet u Nišu  
Elektronski Fakultet  
Katedra za računarstvo



Seminarski rad

# Database Transaction PostgreSQL

*Obrada transakcija, plan izvršavanja, izolacija i zaključivanje*  
Srđan Ognjanović | DBMS

Sistemi za upravljanje bazama podataka

*Profesori: Leonid Stoimenov, Aleksandar Stanimirović*

## Sadržaj

|  |    |
|--|----|
| 1. Opis .....  | 2  |
| 2. Teorijski deo – Procesiranje transakcija .....                  | 3  |
| 2.1 Uvod .....   | 3  |
| 2.2 Koncept transakcije .....                                      | 3  |
| 2.3 Stanja transakcije .....                                       | 4  |
| 2.4 Procesiranje transakcija .....                                 | 5  |
| 2.5 ACID svojstva .....  | 6  |
| 2.6 Rasporedi .....  | 8  |
| 2.7 Nepravilnosti prilikom konkurentnog izvršavanja .....          | 16 |
| 2.8 Protokoli kontrole konkurentnosti .....                        | 19 |
| 3. Praktični deo - Procesiranje transakcija kod PostgreSQL-a ..... | 25 |
| 3.1 Osnovne komande za kontrolu transakcija .....                  | 25 |
| 3.2 MVCC kod PostgreSQL-a.....                                     | 28 |
| 3.3 Nivoi izolacije.....   | 32 |
| 3.4 Eksplicitno zaključavanje .....                                | 38 |
| 4. Zaključak.....  | 42 |
| 5. Literatura.....   | 43 |

## 1. Opis

U seminarskom radu biće teorijski obrađene izabrane teme sa opisom praktične implementacije na primeru tehnologije PostgreSQL.

Seminarski rad na temu *“Data Transaction” - Obrada transakcija, plan izvršavanja, izolacija i zaključivanje kod PostgreSQL* iz predmeta *Sistemi za upravljanje bazama podataka (DBMS)* se sastoji iz dve celine. Prvi deo predstavlja teorijsku podlogu i ključne koncepte na kojima se zasnivaju transakcije uopšteno: ACID svojstva, rasporedi, anomalije prilikom preplitanja i zaključivanje. Drugi deo se odnosi na praktičnu realizaciju odnosno primenu navedenih pojmova kod PostgreSQL baze. Za potrebe rada podignuta je instanca baze podataka PostgreSQL na operativnom sistemu *Linux Ubuntu*.

## 2. Teorijski deo – Procesiranje transakcija

### 2.1 Uvod

Aplikacije koje se koriste u današnje vreme zahtevaju konstantnu dostupnost i brz odziv ka velikom broju korisnika koji istovremeno koriste istu. S' obzirom da su baze podataka nezaobilazan i neophodan deo ovakvih, modernih aplikacija, DBMS mora da poseduje mehanizme za konkurentnu obradu višekorisničkih zahteva. Svaki korisnički zahtev koji se treba izvršiti, DBMS interpretira kao jedan niz operacija (upis i čitanje), odnosno kao jednu **transakciju** [1]. Primer jednog ovakvog sistema bila bi javno dostupna aplikacija za licitaciju vozila, kojoj istovremeno pristupa na hiljade korisnika ili bankovni transfer novca gde se mora u celosti ostvariti balans (promena stanja na računima pošiljaoca i primaoca), što se može videti u *Primeru 1*.

Da bi DBMS na najefikasniji način iskoristio CPU i mehanizam protočnosti, transakcije se ne mogu i ne smeju izvršavati jedna po jedna tj. sekvencijalno (*sequentially*), 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 bazu, odnosno treba postići efekat kao kad bi se date transakcije izvršavale sekvencijalno.

Problemi koji se mogu javiti prilikom konkurentnog izvršavanja transakcija mogu dovesti do toga da baza ostane u nekonzistentnom stanju (*inconsistent state*). Komponenta DBMS-a koja se zove **kontrola konkurencije** (*concurency control*) je zadužena za rešavanje ovakvih problema. Kontrola konkurencije je u tesnoj vezi i sa **menadžerom oporavka** (*recovery manager*), koji je zadužen da, prilikom neuspelog izvršenja transakcije, poništi efekat iste i vrati bazu u prethodno stanje [1]. U nastavku rada će biti više detalja o konceptu transakcije, načinu procesiranja transakcije i svojstvima koje mora da poseduje svaka transakcija.

### 2.2 Koncept transakcije

Tradicionalno, transakcija je sporazum između kupca i prodavca koji razmenjuju sredstva za plaćanje. Ovo je definicija u poslovnom smislu [2, 3]. Sa tačke gledišta baze podataka, to se može shvatiti kao jedinica rada koja se izvodi u okviru sistema za upravljanje bazom podataka (ili sličnog sistema) nad bazom podataka i tretira kao koherentan i pouzdan način nezavisan od drugih transakcija [2, 4]. Dat je već pomenuti primer sa prenosom novca između bankovnih računa.

#### *Primer 1*

Prenos novca u iznosu od \$100 između dva bankovna računa A i B [5].

```

        read(A);
A := A - 100;
        write(A);
        read(B);
B := B + 100;
        write(B)

```

- Operacija *read(X)* – prenosi stavku podataka X iz baze podataka u lokalni bafer koji pripada transakciji
- Operacija *write(X)* – prenosi stavku podataka X iz lokalnog bafera nazad u bazu podataka
- Izvršava se kao jedna celina: ni u jednom trenutku između čitanja (A) i pisanja (B) korisnik ne može da upita bazu podataka, jer je ona možda u nedoslednom (nekonzistentnom) stanju

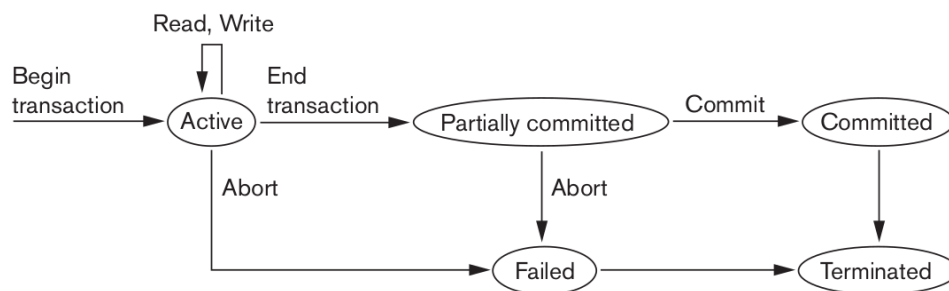
Transakcija u bazi podataka treba da ima dve namene. Prva je pružanje pouzdanih jedinica rada, koje omogućavaju ispravan oporavak od neuspeha (*recovery from failure*) [2]. Druga svrha je pružanje izolacija (*isolation*) između programa kako bi obezbedili istovremeni pristup bazi podataka, što znači da transakcija u bazi podataka mora imati **ACID** svojstva da bi pravilno pokrenula program. Na primer kod distribuiranih baza podataka, transakcije se implementiraju preko više aplikacija i hostova. Štaviše, distribuirane transakcije takođe primenjuju ACID svojstva na više skladišta podataka [2].

## 2.3 Stanja transakcije

U cilju oporavka, sistem mora da vodi evidenciju o promeni stanja odnosno o tome kada svaka transakcija započne, završi, potvrdi ili prekine (*start, terminate, commit, abort*) svoje izvršenje. Shodno tome, menadžer oporavka DBMS-a mora da vodi računa o sledećim operacijama [1]:

- BEGIN\_TRANSACTION
- READ ili WRITE
- END\_TRANSACTION
- COMMIT\_TRANSACTION
- ROLLBACK (ili ABORT)

Na slici 1 je prikazan dijagram tranzicije stanja koji ilustruje kako se transakcija kreće kroz svoja stanja izvršenja.



Slika 1

Prema datim izvorima transakcija u bazi podataka može biti u jednom od sledećih stanja [1, 8]:

- Aktivno (**Active**) - U ovom stanju se transakcija izvršava. Ovo je početno stanje svake transakcije
- Delimično potvrđeno (**Partially Committed**) - Kada transakcija izvrši svoju završnu operaciju, kaže se da je u delimično potvrđenom stanju
- Neuspelo (**Failed**) - Kaže se da je transakcija u neuspehom stanju ako bilo koja provera koju je izvršio sistem oporavka baze podataka ne uspe. Neuspela transakcija više ne može da se nastavi
- Prekinuto (**Terminated / Aborted**) - Ako bilo koja od provera ne uspe i transakcija dođe do neuspelog stanja, tada će menadžer oporavka vratiti sve svoje operacije upisivanja u bazu podataka da bi je vratio u prvobitno stanje u kojem je bila pre izvršenja transakcije. Transakcije u ovom stanju nazivaju se prekinutim. Modul za oporavak baze podataka može izabrati jednu od dve operacije nakon prekida transakcije:
  - Ponovo pokrenuti transakciju (**restart**)
  - Ubiti transakciju (**kill**)
- Potvrđeno / Komitovano (**Committed**) - Ako transakcija uspešno izvrši sve svoje operacije, kaže se da je izvršena. Svi njeni efekti su tada trajno uspostavljeni na sistemu baze podataka

## 2.4 Procesiranje transakcija

Pod pojmom transakcije, smatra se program u izvršenju DBMS-a koji je u stvari logička jedinica obrade baze podataka [6]. Transakcija je izvršni program koji uključuje neke operacije baze podataka, kao što su čitanje iz baze podataka, upis, ili dodavanje, brisanje ili ažuriranje baze podataka, modifikovanje ili pribavljanje podataka. Na kraju izvršenja transakcije, baza podataka se mora ostaviti u važećem ili doslednom stanju koje zadovoljava sva ograničenja navedena u šemi baze podataka [1]. 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 bila bi prethodno navedena aplikacija za licitaciju vozila, kod koje korisnici istovremeno šalju svoju ponudu i tim putem se

obraćaju bazi, koja za svaki od ovih zahteva pokreće isti program kojim se vrši čitanje prethodne vrednosti automobila, nakon čega se upisuje nova trenutna vrednost.

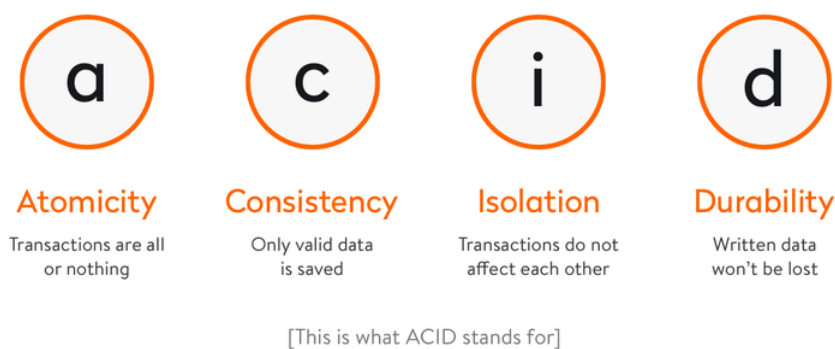
Komande upisa i čitanja predstavljaju gradivne blokove svake transakcije, gde svako **čitanje** obuhvata pronalaženje i prebacivanje odgovarajućeg bloka, koji sadrži traženi element, u bafer glavne memorije, iz koga se po potrebi prebacuje traženi element u odgovarajuću programsku promenljivu. Nasuprot tome prilikom **upisa** vrši se traženje i prebacivanje traženog bloka u bafer glavne memorije, nakon čega se vrši upis u odgovarajuću lokaciju bafera glavne memorije, a potom, odmah ili kasnije, se modifikovani blok prebacuje na disk. Svaka transakcija kao svoju završnu poslednju operaciju ima *commit*, ukoliko je uspešno izvršena, ili u suprotnom *abort*, kojom se poništavaju sve operacije te transakcije učinjeni do tada [7].

## 2.5 ACID svojstva

ACID svojstva predstavljaju:

- Atomičnost (*Atomicity*)
- Konzistentnost (*Consistency*)
- Izolacija (*Isolation*)
- Trajnost (*Durability*)

Svaka transakcija mora slediti ova četiri svojstva, odnosno ona su implementirana korišćenjem nekih metoda za transakciju [2].



Slika 2

### 2.5.1 Atomicity

Atomičnost (*Atomicity*) se odnosi na sposobnost DBMS-a da garantuje da će se svi zadaci transakcije izvršiti ili nijedan od njih. Atomična stanja koja modifikacije baze podataka moraju slediti se nazivaju pravilom „sve ili ništa“. Ako neki deo transakcije otkaže, onda zataji cela transakcija i obrnuto [2]. 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*), ukoliko dođe 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

### 2.5.2 Consistency

Svojstvo konzistencije (**Consistency**) osigurava da baza podataka ostane u doslednom stanju, uprkos uspehu ili neuspehu transakcije i pre početka i nakon završetka transakcije [2]. Uspešno izvršena transakcija prevodi bazu iz jednog konzistentnog stanja u drugo. Za očuvanje ovog svojstva je zadužen sam korisnik, iz razloga što DBMS ne zna šta se podrazumeva pod „konzistentnim“ stanjem. Primer ovog svojstva, koji se odnosi na prethodno pomenutu aplikaciju za licitaciju vozila, bi bio da korisnik mora sam da unese vrednost za kupovinu, koja ne sme biti negativna. Ovo bi moglo da se izvede uz pomoć specificiranja jednostavnih ograničenja konzistentnosti nad željenim podacima, koji bi bili sprovedeni od strane DBMS-a.

### 2.5.3 Isolation

Izolacija (**Isolation**) se odnosi na zahtev da druge operacije ne mogu pristupiti ili videti podatke u srednjem stanju tokom transakcije. Svojstvo izolacije može takođe pomoći u primeni istovremenosti baze podataka [2]. Istovremenošću se daje nalik da se svaka transakcija izvršava nezavisno od ostalih, iako se odvija preplitanje operacija iz više transakcija tj. transakcije se konkretno izvršavaju. Primer ovoga bi bio, da ukoliko imamo dve transakcije T i T' koje se konkurentno izvršavaju, konačni efekat na bazu mora da bude takav kao da su se T i T' sekvencijalno izvršile u nekom rasporedu (T pa T', ili T' pa T) [5]. Za ovo svojstvo su zadužene metode kontrola konkurencije. Ovo svojstvo takođe implicira da jedna transakcija ne može videti nekomitovane promene drugih transakcija.

### 2.5.4 Durability

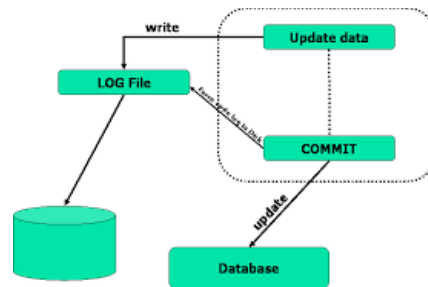
Trajnost (**Durability**) ili postojanost (kako se negde prevodi) navodi da kada se transakcija izvrši, zagantovano je da će se njeni efekti zadržati čak i u slučaju naknadnih neuspeha. To znači da će korisnici biti obavešteni o uspehu, a transakcije će trajati, neće biti poništene i preživeće od pada sistema [2]. Za ovo svojstvo je zadužen menadžer za oporavak.

### 2.5.5 Log

Menadžer transakcija obezbeđuje atomičnost tako što, ukoliko bi došlo do bilo kakvih nepravilnosti u toku izvršenja transakcije, što bi dalje 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 neuspešne transakcije. Ovo se obezbeđuje na taj način što sistem održava **log** u kome se nalaze sve operacije transakcija koje su izvršene [6].



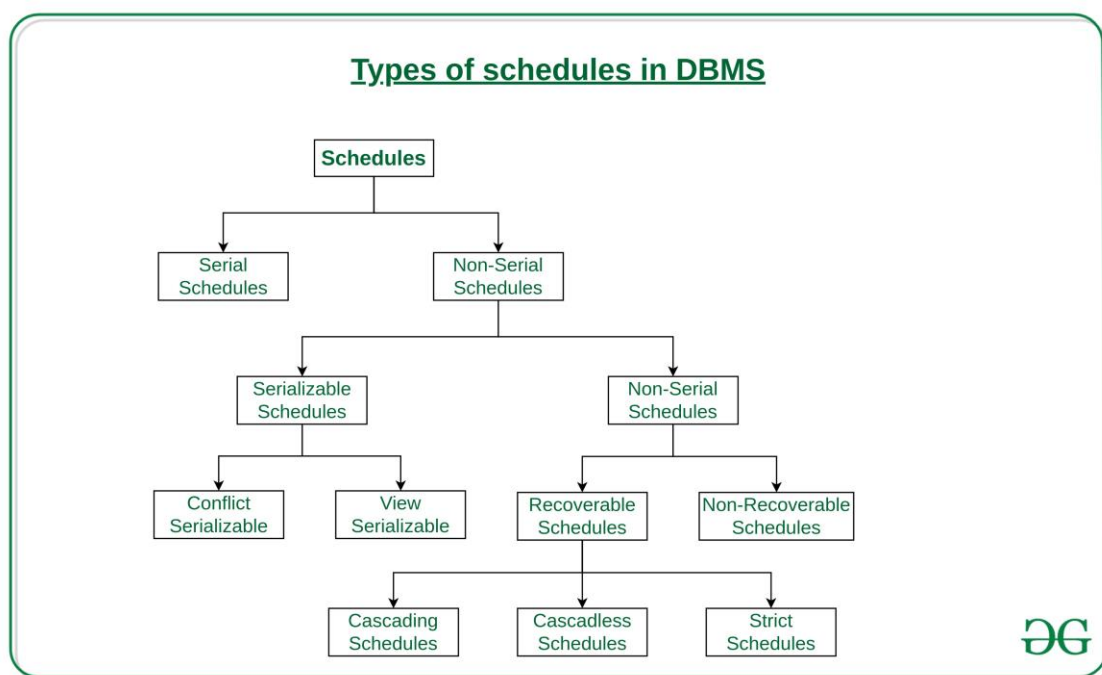
Slika 3

Ovaj *log* se ujedno koristi i kod svojstva trajnosti, tako što prilikom pada sistema, pre nego što su promene od potvrđene (*committed*) transakcije bile zapamćene na disk, *log* se koristi da bi se promene iste transakcije ponovo primenile i zapamtile u bazi. Za svojstvo izolacije je zadužena kontrola konkurencije, koja predstavlja proces upravljanja konkurentnim operacijama nad bazom, da bi se sprečila njihova međusobna interferencija. Jedan od najbitnijih pojmova kada je reč o kontroli konkurencije jeste raspored izvršenja operacija, o kojima će u nastavku biti više reči.

## 2.6 Rasporedi

Rasporedi (*Schedules*), kao što i samo ime govori, predstavljaju postupak postavljanja transakcija izvršavanjem ih jednu po jednu. Kada postoji više transakcija koje se izvršavaju istovremeno, redosled operacija je potrebno podesiti tako da se operacije ne preklapaju [9]. Raspored praktično predstavlja konkretnu, ili potencijalnu sekvencu izvršavanja odnosno listu operacija (*read*, *write*, *commit*, *abort*). Dve operacije unutar rasporeda se smatraju da su u konfliktu, ukoliko obe pristupaju istom elementu, pripadaju različitim transakcijama i ako je bar jedna od te dve operacije upis [1]. Primenjuje se raspoređivanje i transakcije se tempiraju u skladu sa tim [9]. U nastavku rada biće više o kontroli paralelnosti, nivoima izolacije transakcija i različitim vrstama rasporeda.

Tipovi rasporeda kod DBMS-a su dati na sledećoj slici [9].



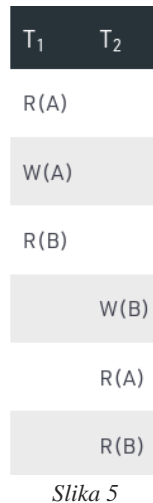
Slika 4

### 2.6.1 Serial Schedules

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 [1], tj. one u kojima nijedna transakcija ne započinje dok se ne izvrši završena transakcija nazivaju se serijski rasporedi (***Serial Schedules***) [9]. 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.

#### Primer 2

Razmotrite sledeći raspored koji uključuje dve transakcije T1 i T2 [9].



Gde R(A) označava da se vrši operacija čitanja na nekoj stavci podataka „A“, dok W(A) predstavlja upis. Ovo je serijski raspored, jer se transakcije obavljaju serijski po redosledu T<sub>1</sub> -> T<sub>2</sub>.

### 2.6.2 Non-serial Schedules

Vrsta raspoređivanja gde se operacije više transakcija prepliću naziva se neserijskim raspoređivanjem (**Non-Serial Schedules**). To bi moglo dovesti do porasta problema sa paralelnošću. Transakcije se izvršavaju na neserijski način, održavajući krajnji rezultat tačnim i u jednakom serijskom rasporedu. Za razliku od serijskog rasporeda gde jedna transakcija mora čekati da druga završi sve svoje operacije, u neserijskom rasporedu druga transakcija se nastavlja ne čekajući da se prethodna transakcija završi. Ovakav raspored ne pruža nikakve koristi od istovremene transakcije. Neserijski raspored se može dalje podeliti na “serijabilan” i “neserijabilan” raspored. [9].

#### 2.6.2.a Serializable Schedules

Serijabilni rasporedi (**Serializable Schedules**) se koriste za održavanje doslednosti baze podataka i uglavnom se primenjuje kod neserijskog raspoređivanja da bi se verifikovalo da li će raspoređivanje dovesti do bilo kakve nesaglasnosti ili ne. S druge strane, serijskom rasporedu nije potrebna serijalizacija, jer prati transakciju samo kada je prethodna transakcija završena. Kaže se da je neserijski raspored u serijabilnom rasporedu samo kada je ekvivalentan serijskim rasporedima, za n broj transakcija. Pošto je u ovom slučaju istovremeno dozvoljena istovremenost, više transakcija se može izvršavati istovremeno. Raspored koji se može serijalizovati pomaže u poboljšanju korišćenja resursa i protoka CPU-a [9]. 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 [1]:

- **Conflict equivalence** - Dva rasporeda se smatraju conflict ekvivalentnim ukoliko je raspored bilo koje dve konfliktne operacije isti u oba rasporeda

- **View equivalence** - Dva rasporeda se smatraju view ekvivalentnim ukoliko 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

#### 2.6.2.a.1 Conflict Serializable Schedules

Ukoliko je neki nesisjski raspored *conflict* ekvivalentan sa nekom od odgovarajućih serijskih rasporeda, radi se o *conflict* serijabilnom rasporedu (**Conflict Serializable Schedules**) [1]. Za dve operacije se kaže da su u suprotnosti ako su sledeći uslovi zadovoljeni [9]:

- Pripadaju različitim transakcijama
- Rade na istoj stavci podataka
- Najmanje jedna od njih je operacija pisanja

#### 2.6.2.a.1 View Serializable Schedules

U koliko se radi o *view* ekvivalenciji, pod istim uslovima, onda je reč o *view* serijabilnom (**View Serializable Schedules**) rasporedu. Raspored se naziva “pogledom koji se može serijalizovati” ako je prikaz jednak serijskom rasporedu (bez preklapanja transakcija). Raspored sukoba je pogled koji je moguće serijalizovati, ali ako serijabilnost sadrži slepe (*blind*) upise, tada se serijabilni pogled ne može serijalizovati [9].

Definicija *view* i *conflict* serijabilnih rasporeda je ista ukoliko 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* serijabilnost je manje restriktivna od *conflict* serijabilnosti. Generalno, svaki konflikt serijabilni raspored je ujedno i pogled serijabilan, dok obrnuta tvrdnja ne važi [1].

#### 2.6.2.b Non-Serializable Schedules

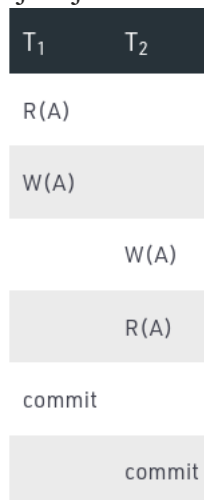
Rasporedi koji se ne mogu serijalizovati (**Non-Serializable Schedules**) sa aspekta oporavka podeljeni su na dve vrste, na one kod kojih je oporavak moguć **Recoverable** “obnovljivi” - kad se jednom transakcija komituje nema potrebe za njenim *rollback*-om i **Non-Recoverable** “nepovratni” rasporedi kod kojih je oporavak nemoguć [1, 9].

#### 2.6.2.b.1 Recoverable Schedules

Rasporedi u kojima se transakcije potvrđuju (*commit*-uju) tek nakon što se sve transakcije čije se promene pročitaju, nazivaju se obnovljivim rasporedima (**Recoverable Schedules**). Drugim rečima, ako neka transakcija T<sub>j</sub> očitava vrednost koju je ažurirala ili napisala neka druga transakcija T<sub>i</sub>, tada se i T<sub>j</sub> mora dogoditi nakon potvrđivanja T<sub>i</sub> [9].

### Primer 3

Razmotrite sledeći raspored koji uključuje dve transakcije T1 i T2 [9].



Slika 6

Ovo je raspored koji se može oporaviti, jer se T1 potvrđuje pre T2, što čini vrednost koju čita T2 tačnom.

Kod nekih *recoverable* rasporeda može doći do pojave kao što je **cascading rollback**, kada nekomitovana transakcija mora da obavi *rollback* zato što je obavila čitanje elementa 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 *recoverable* rasporedima. Raspored se smatra otpornim na *cascading rollback* tj. **cascadeless**, ukoliko svaka transakcija čita elemente koji su upisivani od strane već komitovanih transakcija.

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

#### 2.6.2.b.1.1 Cascading Schedules

Kaskadni raspored (**Cascading Schedules**) se takođe naziva i izbegavanje kaskadnog prekidanja / vraćanja (**Avoids Cascading Aborts/Rollbacks - ACA**). Kada dođe do greške u jednoj transakciji i to dovodi do vraćanja unazad ili prekida drugih zavisnih transakcija, tada se takvo raspoređivanje naziva kaskadno vraćanje (*rolling*) ili kaskadno prekidanje (*abort*) [9].

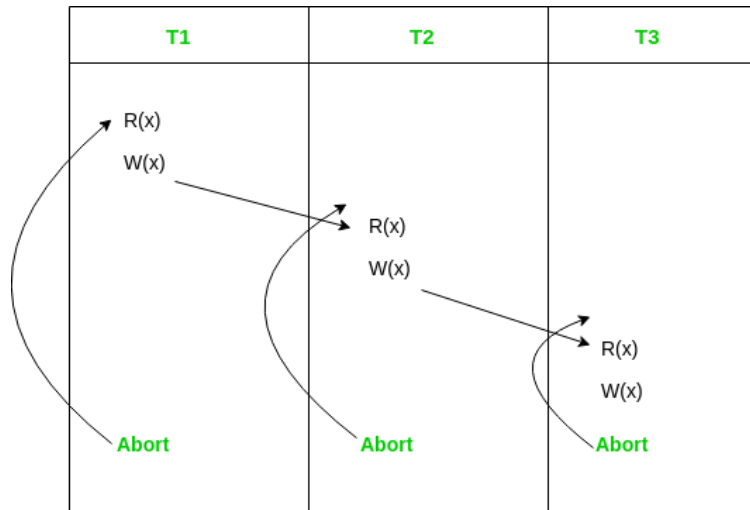


Figure - Cascading Abort

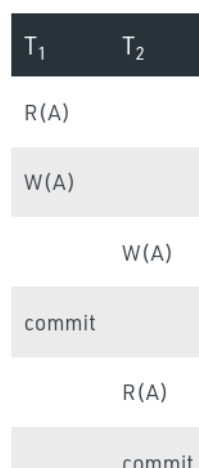
Slika 7

#### 2.6.2.b.1.2 Cascadless Schedules

Rasporedi u kojima transakcije čitaju vrednosti samo nakon svih transakcija čije će promene pročitati, nazivaju se bezkaskadni rasporedi (*Cascadeless Schedules*). Izbegavanje da prekid jedne transakcije dovodi do niza vraćanja transakcija. Strategija za sprečavanje kaskadnog prekida je zabraniti transakciji čitanje neizvezenih promena iz druge transakcije u istom rasporedu. Drugim rečima, ako neka transakcija  $T_j$  želi da pročita vrednost koju je ažurirala ili napisala neka druga transakcija  $T_i$ , tada se  $T_j$  potvrđivanje mora pročitati nakon potvrđivanja  $T_i$  [9].

#### Primer 4

Razmotrite sledeći raspored koji uključuje dve transakcije  $T_1$  i  $T_2$  [9].

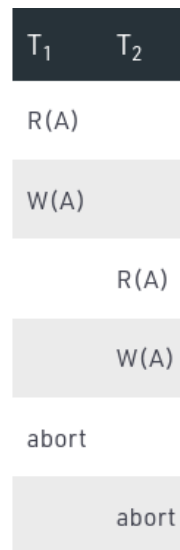


Slika 8

Ovaj raspored je bez kaskada. Pošto  $T_2$  čita ažuriranu vrednost A tek nakon transakcije ažuriranja, tj.  $T_1$  potvrđivanja.

### Primer 5

Razmotrite sledeći raspored koji uključuje dve transakcije T1 i T2 [9].



Slika 9

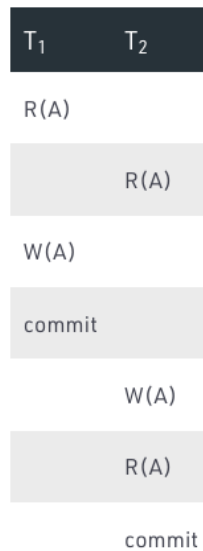
To je raspored koji se može oporaviti, ali ne izbegava kaskadni prekid. Može se videti da ako se T1 prekine, i T2 će morati da se prekine da bi se održala ispravnost rasporeda jer je T2 već pročitao nepotvrđenu vrednost koju je napisao T1 [9].

#### 2.6.2.b.1.3 Strict Schedules

Raspored je strog (**Strict Schedules**) ako za bilo koje dve transakcije  $T_i$ ,  $T_j$ , operacija pisanja  $T_i$  prethodi sukobljenoj operaciji  $T_j$  (bilo čitanje ili pisanje), onda je događaj potvrđivanja ili prekida  $T_i$  takođe takođe prethodi toj sukobljenoj operaciji  $T_j$  [9].

### Primer 6

Razmotrite sledeći raspored koji uključuje dve transakcije T1 i T2 [9].



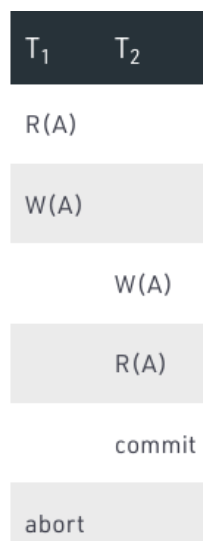
Slika 10

Ovo je strog raspored jer T<sub>2</sub> čita i piše A koji je upisan od strane T<sub>1</sub> tek nakon potvrđivanja T<sub>1</sub>.

#### 2.6.2.b.2 Non-Recoverable Schedules

##### Primer 7

Razmotrite sledeći raspored koji uključuje dve transakcije T<sub>1</sub> i T<sub>2</sub> [9].



Slika 11

T<sub>2</sub> je pročitala vrednost A koju je napisala T<sub>1</sub> i T<sub>2</sub> potvrđuje. T<sub>1</sub> je kasnije prekinuta, stoga je vrednost koju je pročitala T<sub>2</sub> pogrešna, ali pošto je T<sub>2</sub> potvrđena, ovaj raspored se ne može povratiti (**Non-Recoverable Schedules**).

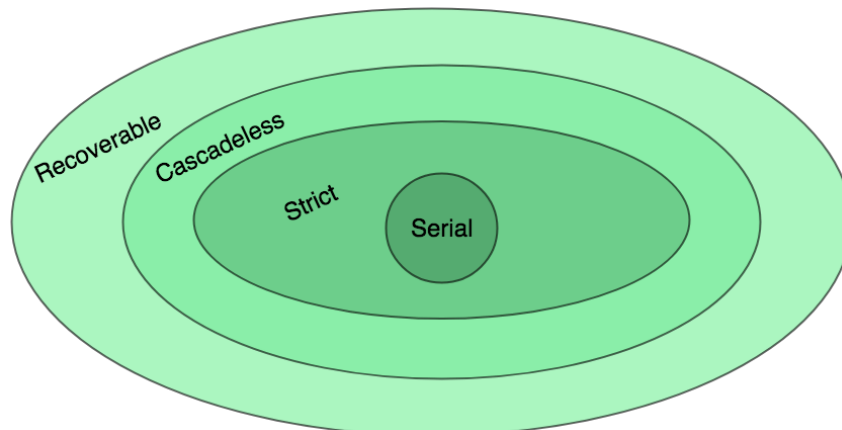
##### Sumirano

Možemo zaključiti da:



- Kaskadni rasporedi su stroži od rasporeda koji se mogu obnoviti ili su podskup rasporeda koji se mogu obnoviti
- Strogi rasporedi su stroži od kaskadnih rasporeda ili su podskup kaskadnih rasporeda
- Serijski rasporedi zadovoljavaju ograničenja svih nadoknadivih, kaskadnih i strogih rasporeda, pa su stoga podskup strogih rasporeda

Veza između različitih vrsta rasporeda može se prikazati kao [9]:



Slika 12

Cilj DBMS-a je da putem metoda i protokola (odnosno skupa pravila), osigura serijabilnost bez testiranja rasporeda na isti, što bi bilo nepraktično. Ukoliko se svaka transakcija pridržava protokola 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 će biti detaljno objašnjenje ove metode, važno je navesti kakve su posledice i nepravilnosti moguće ukoliko se koristi raspored koji nije serijabilan.

## 2.7 Nepravilnosti prilikom konkurentnog izvršavanja

Iako pojedinačno transakcije prevode bazu iz jednog konzistentnog stanja u drugo, nesorijabilni raspored može da dovede do toga da se, nakon izvršenja, baza nađe u nekonzistentnom stanju. Kada se više transakcija izvršava istovremeno na nekontrolisan ili neograničen način, tada to može dovesti do nekoliko problema. Ovi problemi se obično nazivaju konkurentnim problemima u okruženju baza podataka.

Postoje pet problema sa konkurentnošću koji se mogu javiti [10]:

- **Temporary Update Problem / Dirty Read Problem** (problem privremenog ažuriranja / problem prljavog čitanja)
- **Incorrect Summary Problem** (netačan rezime problem)
- **Lost Update Problem** (problem izgubljenog ažuriranja)

- **Unrepeatable Read Problem** (neponovljiv problem čitanja)
- **Phantom Read Problem** (fantomski problem čitanja)

### 2.7.1 Temporary Update Problem / Dirty Read

Problem privremenog ažuriranja ili prljavog čitanja (**Temporary Update / Dirty Read Problem**) javlja se kada jedna transakcija ažurira stavku i ne uspe. Ali ažuriranu stavku koristi druga transakcija pre nego što je stavka promeni ili vrati na zadnju vrednost [10] odnosno ovaj problem se javlja kada jedna transakcija čita vrednost upisanu od strane druge transakcije koja se još uvek izvršava tj. nije komitovana.

*Primer*

| T1   | T2  |
|--|---|
| <code>read_item(X)</code><br><code>X = X - N</code><br><code>write_item(X)</code><br><br><code>read_item(Y)</code> | <br><code>read_item(X)</code><br><code>X = X + M</code><br><code>write_item(X)</code> |

Slika 13

U gornjem primeru, ako transakcija 1 iz nekog razloga ne uspe, X će se vratiti na svoju prethodnu vrednost. Ali transakcija 2 je već pročitala netačnu vrednost X-a.

### 2.7.2 Incorrect Summary Problem

Ukoliko 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**). Razmotrimo situaciju u kojoj jedna transakcija primenjuje agregatnu funkciju na neke zapise, dok druga transakcija ažurira te zapise. Agregatna funkcija može izračunati neke vrednosti pre nego što su vrednosti ažurirane, a druge nakon što su ažurirane [10].

*Primer*

| T1  | T2   |
|---|--|
| <code>read_item(X)</code><br><code>X = X - N</code><br><code>write_item(X)</code> | <code>sum = 0</code><br><code>read_item(A)</code><br><code>sum = sum + A</code>                                    |
| <code>read_item(Y)</code><br><code>Y = Y + N</code><br><code>write_item(Y)</code> | <code>read_item(X)</code><br><code>sum = sum + X</code><br><code>read_item(Y)</code><br><code>sum = sum + Y</code> |

Slika 14

U gornjem primeru, transakcija 2 izračunava sumu nekih zapisa dok ih transakcija 1 ažurira. Stoga agregatna funkcija može izračunati neke vrednosti pre nego što su ažurirane, a druge nakon što su ažurirane.

### 2.7.3 Lost Update Problem

Kod problema gubitka pri ažuriranju (**Lost Update Problem**), ažuriranje izvršeno za neku stavku podataka transakcijom se gubi jer je prepisuje ažuriranje izvršeno drugom transakcijom [10]. 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.

Primer

| T1  | T2  |
|---|---|
| <code>read_item(X)</code><br><code>X = X + N</code> | <code>X = X + 10</code><br><code>write_item(X)</code> |

Slika 15

U gornjem primeru, transakcija 1 menja vrednost X, ali je prepisuje ažuriranje izvršeno transakcijom 2 na X. Stoga se ažuriranje izvršeno transakcijom 1 gubi.

### 2.7.4 Unrepeatable Read Problem

Neponovljivi problem čitanja (**Unrepeatable Read Problem**) nastaje kada dve ili više operacija čitanja iste transakcije čitaju različite vrednosti iste promenljive [10]. Kod ovog problema jedna transakcija izvrši *update* nad nekim elementom, koji je prethodno pročitao od strane neke druge aplikacije koja se još uvek izvršava, tako da ukoliko prva transakcija pokuša sa ponovnim čitanjem, dobiće izmenjenu vrednost, iako je nije ona sama u međuvremenu izmenila.

Primer

| T1       | T2      |
|----------|---------|
| Read(X)  |         |
| Write(X) | Read(X) |
|          | Read(X) |

Slika 16

U gornjem primeru, kada transakcija 2 pročita promenljivu X, operacija pisanja u transakciji 1 menja vrednost promenljive X. Dakle, kada se transakcijom 2 izvrši druga operacija čitanja, ona čita novu vrednost X koju je ažurirala transakcija 1.

### 2.7.5 Phantom Read Problem

Problem sa fantomskim čitanjem javlja se kada transakcija jednom pročita promenljivu, ali ako zatim ponovo pokuša da pročita tu istu promenljivu, javlja se greška koja kaže da promenljiva ne postoji [10]. Kod ovog tipa problema se u okviru jedne transakcije dva puta izvršava isti upit nad bazom, pri čemu se kao rezultat, svakiput dobije različiti broj redova. Razlog za ovo je najčešće taj, što neka druga transakcija unese novi red u bazu između dva izvršenja upita u prvoj transakciji.

*Primer*

| T1        | T2      |
|-----------|---------|
| Read(X)   |         |
| Delete(X) | Read(X) |
|           | Read(X) |

Slika 17

U gornjem primeru, kada transakcija 2 pročita promenljivu X, transakcija 1 briše promenljivu X bez znanja transakcije 1. Dakle, kada transakcija 2 pokuša da pročita X, onda to nije u moguće.

## 2.8 Protokoli kontrole konkurentnosti

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

Uvedeni su sledeći tipovi protokola [11]:

- **Lock Based Protocols**
- **Time-Stamp Based Protocol**

- *Validation Based Protocol*
- *Multiversion Concurrency Control Protocol*

### 2.8.1 Lock Based Protocols

Pod terminom **lock** smatra se promenljiva koja se dodeljuje nekom podatku 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 odgovarajući *lock* [1]. Zaključavanje se drugim rečima naziva pristup (*access*). U ovoj vrsti protokola nijedna transakcija neće biti obrađena sve dok transakcija ne zaključa zapis. To znači da bilo koja transakcija neće povući, umetnuti ili ažurirati ili izbrisati podatke ukoliko ne dobije pristup tim podacima. Zaključavanja su široko kvalifikovana i poznata kao binarna (*Binary locks*) i zajednička/ekskluzivna (*Shared/Exclusive locks*) tehnika zaključavanja [11].

Kod binarnog *lock*-a podaci se mogu zaključati ili otključati, dakle postoje samo dva stanja. Mogu se zaključati za preuzimanje, umetanje ili ažuriranje ili brisanje podataka ili otključati zbog neiskorišćenja podataka. [11].

U zajedničkoj/ekskluzivnoj tehnici zaključavanja podaci se zaključavaju isključivo ak ose radi o umetanju / ažuriranju / brisanju. Kada je zaključan, nijedna druga transakcija ne može čitati ili pisati podatke. Kada se podaci čitaju iz baze podataka, tada se zaključavanje deli tj. podatke mogu čitati i druge transakcije, ali se ne mogu menjati dok se preuzimaju [11]. Matrica kompatibilnosti zaključavanja za zajedničke/ekskluzivne *lock*-ove data je u nastavku gde je i objašnjena ova tehnika detaljnije kod *Two-phase locking* protokola.

#### 2.8.1.1 Simplistic Lock Protocol

Kao što i samo ime govori, ovo je najjednostavniji način zaključavanja podataka tokom transakcije. Ovaj protokol omogućava svim transakcijama da zaključaju podatke pre nego što ih ubace / ažuriraju / izbrišu. Nakon završetka transakcije otključaće podatke [11].

#### 2.8.1.2 Pre-claiming Protocol

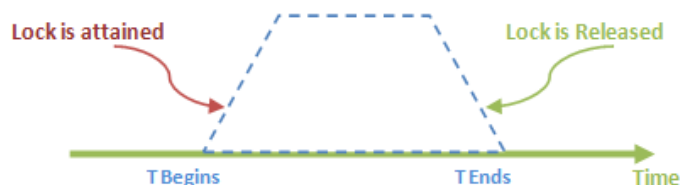
Ovaj tip protokola procenjuje transakciju da bi nabrojao sve stavke podataka za koje je potrebno zaključati transakciju. Zatim zahteva od DBMS-a da *lock*-uje sve te stavke podataka pre početka transakcije. Ako DBMS obezbedi zaključavanje svih podataka, tada ovaj protokol omogućava započinjanje transakcije. Kada je transakcija završena, ona otpušta sve *lock*-ove. Ako DBMS obezbedi sva zaključavanja, on vraća transakcije i čeka zaključavanje [11].



Slika 18

### 2.8.1.3 Two-Phase Locking Protocol (2PL)

Jedna od glavnih tehnika koja se koristi za kontrolu izvršavanja konkurentnih transakcija je **Two-Phase Locking (2PL)** tehnika. Ova tehnika je bazirana na zaključavanju elemenata baze prilikom pristupa istim [1].



Slika 19

2PL protokol je baziran na korišćenju **shared/exclusive lock**-ova, pri čemu, ukoliko transakcija želi da obavi upis nad elementom, mora da pribavi **exclusive (write) lock**, kojim je omogućeno da samo jedna transakcija pristupa elementu radi upisa. Ukoliko transakcija treba da izvrši čitanje, mora da pribavi **shared (read) lock**, pri čemu je dozvoljeno da više transakcija čita element istovremeno, bez blokiranja i čekanja. Na slici je dat prikaz odnosa između **read** i **write lock**-a, pri čemu, ukoliko 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 transakcija koja drži **lock** oslobodi isti. Takođe se vidi da ukoliko 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].

| LOCKS     | Shared | Exclusive |
|-----------|--------|-----------|
| Shared    | TRUE   | FALSE     |
| Exclusive | FALSE  | FALSE     |

Slika 20

Samo korišćenje **Shared/Exclusive lock**-ova nije dozvoljeno da bi se garantovala serijabilnost rasporeda. **Two-phase locking** protokol 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**) - prva faza u kojoj se pribavljaju novi **lock**-ovi za konkretne elemente, pri čemu u ovoj fazi nema oslobađanja **lock**-ova
- skupljanje (**shrinking**) - u kojoj se oslobađaju zauzeti **lock**-ovi, bez pribavljanja novih

Ukoliko svaka transakcija poštuje ovakav protokol, raspored je sa sigurnošću serijabilan [1].

#### 2.8.1.4 Strict Two-Phase Locking Protocol (Strict 2PL)

Najčešće korišćena varijanta 2PL protokola je striktni 2PL (**Strict 2PL**), koji pored serijabilnosti omogućava i to da rasporedi koji poštuju ovaj protokol budu i striktni. Kod striktnog 2PL protokola transakcija oslobađa 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 [1].



Slika 21

#### 2.8.1.5 Nedostatak kod protokola zasnovanih na lock mehanizmu - Deadlock

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 dve ili više transakcija, čeka na pristup nekom elementu, koji je zaključen od neke druge transakcije iz tog seta [1].

| T1   | T2  |
|--|---|
| <b>Exclusive_Lock(X)</b><br>$X := X + 10$<br>WRITE (X)<br><br><b>Exclusive_Lock(Y)</b> | <br><b>Shared_Lock(Y)</b><br><br>READ (Y)<br><br><b>Shared_Lock (X)</b> |

Slika 22

Pretpostavimo da transakcija T1 ažurira vrednost X. Istovremeno transakcija T2 pokušava da pročita X zaključavanjem. Pored toga, T1 će pokušati da zaključa Y koji se drži zaključanim od strane T2. Može se predstaviti jasnije kao u nastavku: Ovde T2 čeka da T1 otpusti bravu na X, dok T1 čeka da T2 pusti bravu na Y. T1 će biti potpun tek kada se zaključa na Y, a zatim će otpustiti bravu na X i Y. Ali bravica na Y drži T2 i čeka da se T1 dovrši, tako da može osloboditi Y. Ovde obe transakcije čekaju da se završe sve druge. Ova vrsta situacije naziva se zastoje. Ovo čekanje se nikada neće završiti ako jedan od njih ne bude ubijen / prekinut, tako da se sve brave koje drže u njima oslobode i druge transakcije mogu da se nastave i završe. Stoga, prilikom

dodeljivanja brava, mora se paziti maksimalno da se izbegne zastoje. U gornjem slučaju, ako se T1 zaključa na Y mnogo pre nego što se T2 pokrene, tada neće doći do mrtve tačke, tj. ako T1 zaključa na Y, tada ne treba čekati da T2 otpusti bilo kakve brave, pa će stoga nastaviti sa svojom transakcijom i otpustiti zaključavanje na X i Y. Dok T2 počne, na T1 neće biti zaključavanja [11].

### 2.8.2 Timestamp Control Protocol

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 mlađe transakcije. Kod *timestamp ordering (TO)* protokola 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 transakcije koja je uspešno pročitala konkretan element, *write timestamp* – *timestamp* najmlađe transakcije koja je uspešno izvršila upis nad konkretnim elementom [1]. Prema ovom protokolu, ako je transakcija T nad podatkom X onda važi sledeći algoritam prikazan na slici [11].

```
Code
IF TS (T) < W_TS(X) THEN
  Reject T
ELSE IF TS (T) >= W_TS(X) THEN
  Execute T
  Update W_TS(X) and R_TS(X)
END
```

Slika 23

Gde su:

- TS (T) → *timestamp* transakcije T
- W\_TS (X) → *timestamp write* operacije nad podatkom X
- R\_TS (X) → *timestamp read* operacije nad podatkom X

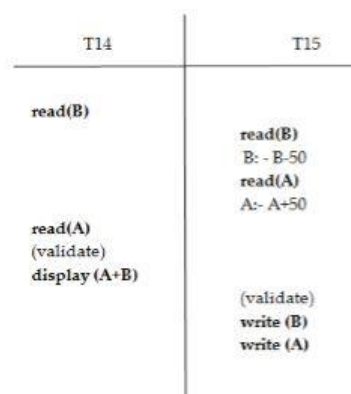
TO algoritam forsira serijabilnost konkurentnih transakcija na sledeći način: kada transakcija T želi da izvrši upis nad nekim elementom X, algoritam proverava promenljive *write timestamp* i *read timestamp* elementa X (*write* može da bude u konfliktu sa *read* i *write* operacijama) i ukoliko se utvrdi da je bilo koja od ove dve promenljive veća od *timestamp*-a transakcije T, što bi značilo da je neka mlađa 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. Ukoliko se utvrdi da je transakcija T mlađa od najmlađih transakcija koje su izvršile upis i čitanje elementa X, transakciji T je dozvoljen upis, a *write timestamp* za element X uzima vrednost *timestamp*-a T. Ukoliko transakcija T želi da izvrši čitanje elementa X,



algoritam vrši poređenje *timestamp*-a T sa *write timestamp*-om od X, ukoliko 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 dozvoljava čitanje transakciji T, a *read timestamp* za X se postavlja na *timestamp* T-a ukoliko je ovaj veći od *read timestamp* vrednosti, dok ukoliko nije, ostaje nepromenjen [1].

### 2.8.3 Validation Based Protocol

**Validation Based Protocol** se takođe naziva i *Optimistic Concurrency Control Technique* [12]. Kod ovog tipa protokola, transakcija se izvršava u tri faze - početak (čitanje i izvršenje), validacija i završetak (ažuriranje i pisanje). U prvoj fazi se čita i izvršava transakcija T. Rezultati se zapisuju u privremene promenljive. Ove vrednosti će se validirati prema stvarnim podacima kako bi se utvrdilo da li krši serijalibilnost. Ako je potvrđen, privremeni rezultati se zapisuju u bazu podataka; u suprotnom se transakcija vraća nazad. Faze se mogu kombinovati u jednu poput kombinovanja provere ispravnosti i upisivanja podataka u DB u jedan proces, ali sve tri faze moraju se izvršiti istim redosledom.



Slika 24

Ovde svaka faza ima različite vremenske žigove: start (T), validacija (T) i završetak (T). U ovom protokolu vremenski žig za transakciju za serijalizaciju određuje se vremenskim žigom faze validacije, jer je stvarna faza koja određuje da li će se transakcija izvršiti ili vratiti. Otuda je TS (T) = validacija (T). Stoga se serializacija utvrđuje u procesu validacije i ne može se unapred odlučiti. Dakle osigurava veći stepen podudarnosti tokom izvršavanja transakcije i takođe manji broj sukoba i ima manji broj povraćaja transakcija [11].

### 2.8.4 Multiversion Concurrency Control Protocol (MVCC)

Još jedan često korišćen protokol za kontrolu konkurentnosti je **Multiversion Concurrency 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 kreirala nova verzija tog elementa, pri čemu bi se stara sačuvala. Jedina mana ovakvog 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 naglasiti da MVCC sprečava blokiranja i čekanja 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. Praktični deo - Procesiranje transakcija kod PostgreSQL-a

Kao i većina modernih DBMS-a, i PostgreSQL podržava konkurentno izvršavanje više transakcija, kao i razne alate koji omogućavaju korisnicima da eksplicitno definišu na koji način će njihova transakcija da interaguje sa drugim transakcijama. Jedna od pogodnosti PostgreSQL-a, što predstavlja efikasno rešenje, jeste osiguravanje izolacije transakcije uz pomoć korišćenja jedne od varijanti MVCC protokola, baziranog na pribavljanju *snapshot*-a baze podataka prilikom početka transakcije, što dovodi do toga da svaka transakcija radi sa svojom verzijom baze. U nastavku ovog poglavlja biće više detalja o načinu kreiranja i kontrole transakcija, o MVCC mehanizmu i nivoima izolacije koje PostgreSQL obezbeđuje kako bi rešio razne nepravilnosti kod konkurentnog izvršavanja, kao i mehanizme za eksplicitno zaključavanje i kontrolu konkurentnosti koje ova baza pruža.

#### 3.1 Osnovne komande za kontrolu transakcija

PostgreSQL omogućava kombinovanje više komandi za kontrolu transakcija nad elementima baze podataka, kako bi se izvršili zajedno u obliku jedne transakcije. Važno je obezbediti konzistentnost baze nakon izvršenja transakcije od strane korisnika.

Sledeće komande se koriste za kontrolu transakcija [13]:

- **BEGIN TRANSACTION** – za pokretanje transakcije, takođe se može upotrebiti **BEGIN** ili **BEGIN WORK**
- **COMMIT** – da bi se sačuvala promena, alternativno se može koristiti i komanda **END TRANSACTION**, **COMMIT TRANSACTION** ili **COMMIT WORK**
- **ROLLBACK** – za vraćanje promena nazad, odnosno **ROLLBACK TRANSACTION** ili **ROLLBACK WORK**

Naredbe za upravljanje transakcijama koriste se samo sa DML naredbama **INSERT**, **UPDATE** i **DELETE**. Ne mogu se koristiti prilikom kreiranja ili brisanja tabele, jer se ove operacije automatski komituju u bazu podataka [13].

Na slikama 25 i 26 je data test baza podataka *testdb* sa tabelom *accounts*, koja će biti korišćena u narednim primerima [14].

```

postgres=# \c testdb
You are now connected to database "testdb" as user "postgres".
testdb=# \l

```

| Name      | Owner    | Encoding | Collate     | Ctype       | Access privileges                      |
|-----------|----------|----------|-------------|-------------|--|
| postgres  | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |  |
| template0 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres +<br>postgres=CTc/postgres |
| template1 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres +<br>postgres=CTc/postgres |
| testdb    | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |  |

```

(4 rows)

testdb=# \dt

```

| Schema | Name     | Type  | Owner    |
|--------|----------|-------|----------|
| public | accounts | table | postgres |

```

(1 row)

```

Slika 25 – Konekcija i baza

```

testdb=# select * from accounts;

```

| id | name    | balance  |
|----|---------|----------|
| 1  | Bob     | 10000.00 |
| 2  | Alice   | 10000.00 |
| 3  | Sarah   | 20000.00 |
| 4  | Mike    | 1500.00  |
| 5  | Srdjan  | 12000.00 |
| 6  | Nikola  | 15000.00 |
| 7  | Jessica | 9000.00  |
| 8  | Alex    | 5800.00  |

```

(8 rows)

```

Slika 26 - Tabela

Ukoliko se izvrši smanjenje novca u vrednosti od 1000 sa Bob-ovog naloga, a poveća za istu sumu tj. prebaci na Alice-in račun (ukupan budžet mora ostati isti, to je uslov konzistentnosti), dobija se novo stanje tabele kao na slici 24. 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 27 se takođe može uočiti i upotreba funkcije *txid\_current()*, čijim se pozivom nakon starta transakcije, prikazuje id trenutne transakcije – *txid*, koja je dodeljena od strane menadžera transakcija.

```

testdb=# BEGIN;
BEGIN
testdb=# SELECT txid_current();
 txid_current
-----
          496
(1 row)

testdb=# UPDATE accounts SET balance = balance - 1000 WHERE id = 1;
UPDATE 1
testdb=# UPDATE accounts SET balance = balance + 1000 WHERE id = 2;
UPDATE 1
testdb=# COMMIT;
COMMIT
testdb=# SELECT * FROM accounts;
 id |  name  | balance
-----+-----+-----
  3 | Sarah  | 20000.00
  4 | Mike   |  1500.00
  5 | Srdjan | 12000.00
  6 | Nikola | 15000.00
  7 | Jessica |  9000.00
  8 | Alex   |  5800.00
  1 | Bob    |  9000.00
  2 | Alice  | 11000.00
(8 rows)

```

Slika 27 – Commit

U slučaju da su unete pogrešne izmene koje nisu poželjne tj. 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 samog izvršenja transakcije. Primer upotrebe ove komande data je na slici 28, gde se u okviru transakcije vrši uklanjanje reda iz tabele, pri čemu se krajnji efekat ove transakcije poništava ROLLBACK komandom.

```

testdb=# BEGIN;
BEGIN
testdb=# DELETE FROM accounts WHERE name = 'Srdjan';
DELETE 1
testdb=# SELECT * FROM accounts;
 id |  name  | balance
-----+-----+-----
  3 | Sarah  | 20000.00
  4 | Mike   |  1500.00
  6 | Nikola | 15000.00
  7 | Jessica |  9000.00
  8 | Alex   |  5800.00
  1 | Bob    |  9000.00
  2 | Alice  | 11000.00
(7 rows)

testdb=# ROLLBACK;
ROLLBACK
testdb=# SELECT * FROM accounts;
 id |  name  | balance
-----+-----+-----
  3 | Sarah  | 20000.00
  4 | Mike   |  1500.00
  5 | Srdjan | 12000.00
  6 | Nikola | 15000.00
  7 | Jessica |  9000.00
  8 | Alex   |  5800.00
  1 | Bob    |  9000.00
  2 | Alice  | 11000.00
(8 rows)

```

PostgreSQL obezbeđuje bolju kontrolu transakcije korišćenjem **SAVEPOINT** i **ROLLBACK TO** komandi. Komandom **ROLLBACK TO** se na neku skladišnu tačku (*savepoint*) 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 stanju izvršavanja i da bi komande pre *savepoint*-a imale efekta, potrebno je završiti transakciju **COMMIT** naredbom. Na slici 29 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 [15].

```
testdb=# BEGIN;
BEGIN
testdb=# INSERT INTO accounts(name, balance) VALUES ('Mark','99999999');
INSERT 0 1
testdb=# SAVEPOINT facebook;
SAVEPOINT
testdb=# UPDATE accounts SET balance = balance * 2 WHERE name = 'Mark';
UPDATE 1
testdb=# ROLLBACK TO SAVEPOINT facebook;
ROLLBACK
testdb=# UPDATE accounts SET balance = 100000000 WHERE name = 'Mark';
UPDATE 1
testdb=# COMMIT;
COMMIT
testdb=# SELECT * FROM accounts WHERE name = 'Mark';
 id | name |  balance
-----+-----
  9 | Mark | 100000000.00
(1 row)
```

Slika 29 – Rollback to Savepoint

## 3.2 MVCC kod PostgreSQL-a

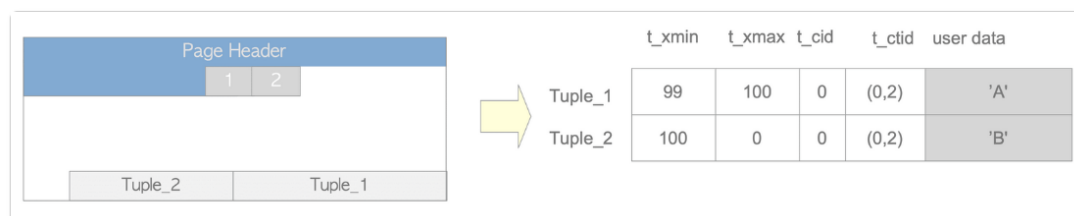
### 3.2.1 Zaglavlje

Kod 2PL protokola zaključivanja je poznato da se kontrola konkurentnosti obezbeđuje uz minimalno čekanja i blokiranja. Za razliku od 2PL-a, gde se transakcije koje se čitaju, mogu da blokiraju one koje vrše upis i obrnuto, MVCC kod PostgreSQL-a omogućava da svaka komanda radi sa jednom konkretnom verzijom baze podataka – *snapshot*-om, pri čemu 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.



Slika 30 - Zaglavlje

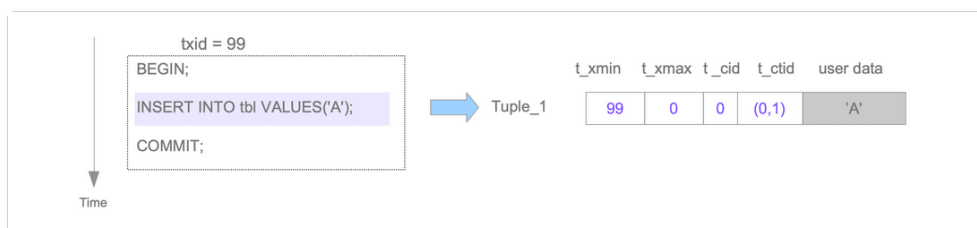
Zahteva pamćenje više verzija jednog reda (*tuple*), s tim što se uz vrednost reda, pamte još i dodatna polja kao što su *t\_xmin*, koje označava *id* transakcije koja je unela red u bazu i *t\_xmax*, koje označava *id* transakcije koja je izvršila update nad tim redom ili brisanje reda (ukoliko je ovo polje 0, radi se o aktuelnom redu), *t\_cid*, koje sadrži naredbu *id* (*cid*), što znači koliko je SQL naredbi izvršeno pre izvršavanja ove naredbe u tekućoj transakciji koja počinje od 0, *t\_ctid*, koje sadrži identifikator reda (*tid*) koji pokazuje na sebe ili novi red. Kada se ovaj red ažurira, *t\_ctid* ovog reda ukazuje na novi red; u suprotnom, *t\_ctid* pokazuje na sebe [16].



Slika 31 - Zaglavlje

### 3.2.2 Insert

Na slici 31 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 [16].

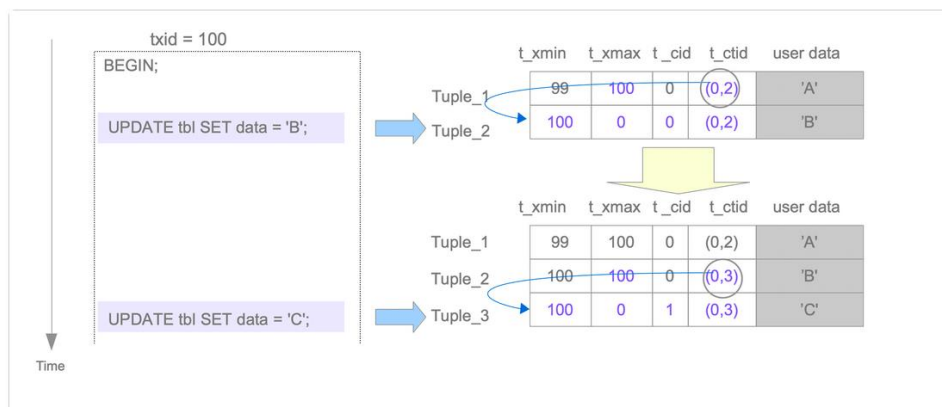


Slika 32 - Dodavanje

### 3.2.3 Update

Ukoliko se izvrši **UPDATE** nad ovim redom od strane neke transakcije, što se vidi na slici 32, *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 [16].

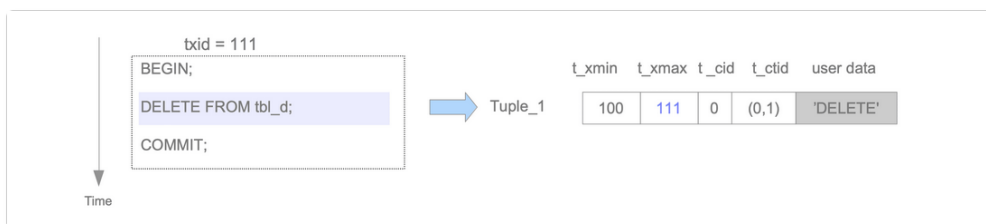




Slika 33 - Ažuriranje

### 3.2.4 Delete

Što se tiče operacije uklanjanja reda **DELETE**, odvija se po sličnom principu kao i *update*, odnosno vrši se logičko uklanjanje, postavljanjem *t\_xmax* vrednosti na *id* transakcije koja je izvršila brisanje, što se može videti na slici 33 [16].



Slika 34 - Brisanje

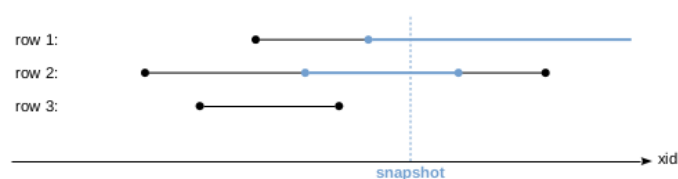
### 3.2.5 Snapshot

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.

Consistent slice of data at a certain point

transaction id — determines the point in time

list of active transactions — not to look on not-yet-committed changes



Slika 35 - Snapshot

*Snapshot* omogućava da se uzmu u obzir jedino one transakcije koje su komitovale pre pribavljanja *snapshot*-a, odnosno da se ne uzimaju obzir transakcije koje su započele pre pribavljanja *snapshot*-a a nisu još uvek komitovane, ili one transakcije koje su startovale posle pribavljanja *snapshot*-a [16].

### 3.2.6 Commit Log (clog)

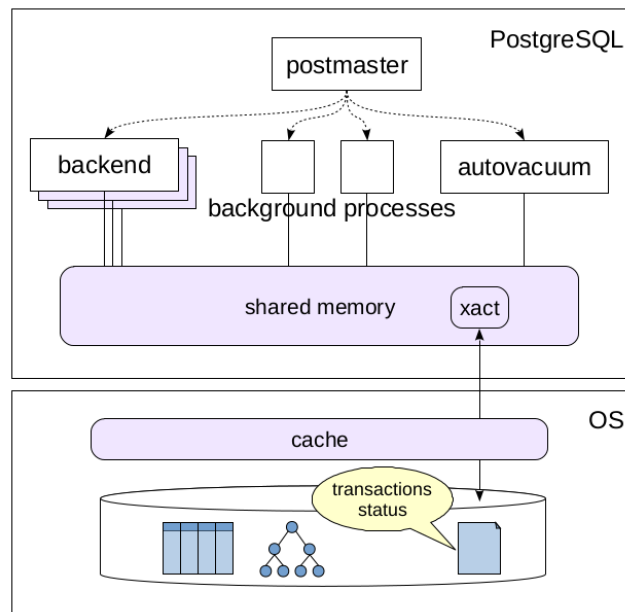
*Log* je zapravo istorija akcija, koje su se izvršile u okviru DBMS-a [17]. Još jedna bitna informacija koja je PostgreSQL-u potrebna kod određivanja koja je verzija reda vidljiva od strane konkretne transakcije, jeste **Commit Log (clog)** koji sadrži status svake transakcije (izvršava se, komitova 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 [18].

Kada se PostgreSQL isključi ili kad god se pokrene proces kontrolne tačke, podaci začepljenja zapisuju se u datoteke uskladištene u poddirektoriju *pg\_clog*. Ove datoteke se nazivaju 0000, 0001 itd. Maksimalna veličina datoteke je 256 KB. Na primer, kada *clog* koristi osam stranica (prva stranica do osme stranice; ukupna veličina je 64 KB), njeni podaci se zapisuju u 0000 (64 KB), a sa 37 stranica (296 KB) podaci se zapisuju na 0000 i 0001, čije su veličine 256 KB, odnosno 40 KB. Kada se PostgreSQL pokrene, podaci uskladišteni u datotekama *pg\_clog* (datoteke *pg\_xact*) učitavaju se radi inicijalizacije *clog*-a [16].

### 3.2.7 Vacuum, Autovacuum

Za održavanja *clog*-a, kao i uklanjanje verzije redova koji se više ne koriste, zadužen je **Vacuum** proces. Da li je neka verzija reda vidljiva ili ne, u određenoj transakciji, određuje se uz pomoć polja verzije reda (*t\_xmin*, *t\_xmax*), *clog*-a i pribavljenog *snapshot*-a. *Vacuum* obično radi automatski i konfigurisan je od strane administratora tako da podatke briše na vreme, izbegavajući veliko povećanje veličine datoteka. Da bi to uradio, **Autovacuum** reaguje na aktivnost promene podataka u tabelama, a ne samo na pokretanje po rasporedu: što se češće menjaju podaci u tabeli, tabela se češće vakumira [18].





Slika 36 - Autovacuum

Autovakum se sastoji od nekoliko procesa [18]:

- *Autovacuum launcher* (pokretač) - pozadinski proces reaguje na aktivnost promene podataka
- *Autovacuum worker* (radnik) - pokreće se od strane *launcher*-a po potrebi i obavlja čišćenje

### 3.3 Nivoi izolacije

SQL standard definiše četiri nivoa izolacije, pri čemu svaki naredni nivo nameće strožija ograničenja kada je reč o tome koje anomalije, kod konkurentnog izvršavanja transakcija, se mogu tolerisati. Nivoi izolacije od najslabijeg do najrestriktivnijeg, koje korisnik može prema potrebi da koristi, su [19, 20]:

- *Read Uncommitted*
- *Read Committed*
- *Repeatable Read*
- *Serializable*

| Isolation Level  | Dirty Read             | Nonrepeatable Read | Phantom Read           | Serialization Anomaly |
|------------------|------------------------|--------------------|------------------------|-----------------------|
| Read uncommitted | Allowed, but not in PG | Possible           | Possible               | Possible              |
| Read committed   | Not possible           | Possible           | Possible               | Possible              |
| Repeatable read  | Not possible           | Not possible       | Allowed, but not in PG | Possible              |
| Serializable     | Not possible           | Not possible       | Not possible           | Not possible          |

Slika 37 – Nivoi izolacije

**Read Uncommitted** 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*.

**Read Committed** je podrazumevani nivo izolacije kod PostgreSQL-a. Kod ovog nivoa, transakcija pribavlja *snapshot* baze, pre izvršenja svakog upita. Iako sprečava *Dirty Read* nepravilnost, 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.

#### Primer – Read Committed

Na slikama 38 i 39 je dat primer izvršenja dve transakcije čiji je nivo izolacije *Read Committed*, pri čemu je strelicama označen redosled izvršavanja komandi transakcije. Za primer je uzeta *accounts* tabela koja inicijalno sadrži 9 redova. Ukoliko se prvo izvrši dodavanje novog reda u transakciji T1, a zatim u transakciji T2 izvrši upit koji vraća broj redova u tabeli, uočava se da T2 ne vidi komitovanu izmenu dodavanja novog reda od strane transakcije T1, što onemogućava *Dirty Read*.

```
testdb=# BEGIN;
BEGIN
testdb=# INSERT INTO accounts(name,balance) VALUES ('Milica',10000);
INSERT 0 1
testdb=# COMMIT;
COMMIT
```

Slika 38 – Transakcija T1

Međutim, nakon što T1 komituje transakciju, T2 ponovo izda istu komandu i nakon toga se može primetiti da je broj redova uvećan za jedan, što zapravo pokazuje da je došlo do *Phantom Read*-a, koji je moguć kod *Read Committed* nivoa izolacije.

```
testdb=# BEGIN;
BEGIN
testdb=# SELECT COUNT(*) FROM accounts;
count
-----
      9
(1 row)

testdb=# SELECT COUNT(*) FROM accounts;
count
-----
     10
(1 row)
```

Slika 39 – Transakcija T2

**Repeatable Read** nivo izolacije, za razliku od *Read Committed* nivoa, pribavlja *snapshot* baze jednom pre početka izvršavanja transakcije, što znači da sve komande u okviru transakcije vide identično stanje baze. Posledica ovog jeste da se nepravilnosti poput *Unrepeatable Read*-a i *Phantom Read*-a ne mogu desiti na ovom nivou izolacije.

#### Primer – Repeatable Read

Na slikama 40 i 41 date su dve transakcije od kojih prva koristi *Repeatable Read* nivo izolacije. Može se primetiti da ukoliko se izvrši čitanje konkretnog reda od strane transakcije T1, a potom transakcija T2 izvrši *update* nad tim istim redom pa onda *commit*. Kada transakcija T1 ponovo izvrši čitanje, zateći će potpuno isto stanje koje je prethodilo, iz razloga što je *snapshot* baze pribavljen jednom prilikom početka izvršavanja transakcije.

```
testdb=# BEGIN
testdb=# ISOLATION LEVEL REPEATABLE READ;
BEGIN
testdb=# SELECT * FROM accounts WHERE name = 'Milica';
 id | name | balance 
----+-----+-----
 11 | Milica | 10000.00
(1 row)

testdb=# SELECT * FROM accounts WHERE name = 'Milica';
 id | name | balance 
----+-----+-----
 11 | Milica | 10000.00
(1 row)

testdb=# COMMIT;
COMMIT
```

Slika 40 – Transakcija T1

```
BEGIN
testdb=# UPDATE accounts SET balance = balance + 5000 WHERE name = 'Milica';
UPDATE 1
testdb=# COMMIT;
```

Slika 41 – Transakcija T2

Prilikom konkurentnog upisa istog elementa, može doći do blokiranja i čekanja usled zaključavanja. PostgreSQL sledi *first-updater-win* princip, kod koga jedna transakcija, prilikom *update*-a, mora da sačeka da transakcija koja trenutno vrši *update* završi i ukoliko je ova završila sa *commit*-om, na osnovu nivoa izolacije se odlučuje da li će novi *update* biti dozvoljen. Ukoliko se radi o *Read Committed* nivou izolacije transakcije koja pokušava da ažurira, može doći do **Lost Update** problema.

#### *Primer – Lost Update problem kod Read Committed nivoa izolacije*

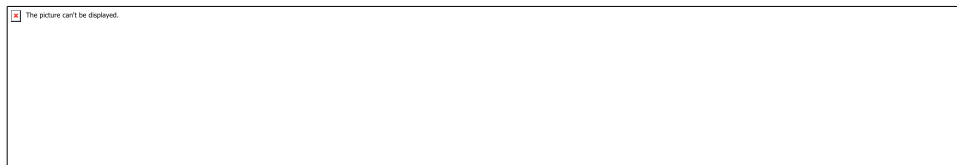
Na slikama 42 i 43 se može videti da transakcija T1 u trenutku 1 vrši *update* nad konkretnim redom, dok transakcija T2 takođe pokušava da izmeni isti red u trenutku 2, čime se ona zapravo blokira, sve dok T1 ne izvrši *commit* u trenutku 3, nakon čega T2 izvršava *update* i *commit* u trenutku 4, brišući samim tim vrednost koju je T1 prethodno upisala.

```

testdb=# BEGIN;
BEGIN
testdb=# UPDATE accounts SET balance = 5000 WHERE name = 'Nikola';
UPDATE 1
testdb=# COMMIT;
COMMIT
testdb=# SELECT * FROM accounts WHERE name = 'Nikola';
 id | name | balance
-----+-----+-----
   6 | Nikola | 55555.00
(1 row)

```

Slika 42 – Transakcija T1



Slika 43 – Transakcija T2

### Primer – Sprečavanje *Lost Update*-a korišćenjem *Repeatable Read* nivoa izolacije

Kod transakcija sa nivoom izolacije *Repeatable Read* ili *Serializable*, *Lost Update* anomalija je otklonjena tako što se obustavlja transakcija koja je pokušala ažuriranje nad elementom, nad kojim se vrši *update* od strane neke druge transakcije. 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 obustavlja (uz obaveštenje o grešci) da bi se sprečila *Lost Update* nepravilnost, što se može videti na slikama 44 i 45.

```

testdb=# BEGIN
testdb=# ISOLATION LEVEL READ COMMITTED;
BEGIN
testdb=# UPDATE accounts SET balance = 0 WHERE name = 'Srdjan';
testdb=# ;
UPDATE 1
testdb=# COMMIT;
COMMIT

```

Slika 44 – Transakcija T1

```

testdb=# BEGIN
testdb=# ISOLATION LEVEL REPEATABLE READ;
BEGIN
testdb=# UPDATE accounts SET balance = 33333 WHERE name = 'Srdjan';
ERROR: could not serialize access due to concurrent update
testdb=#

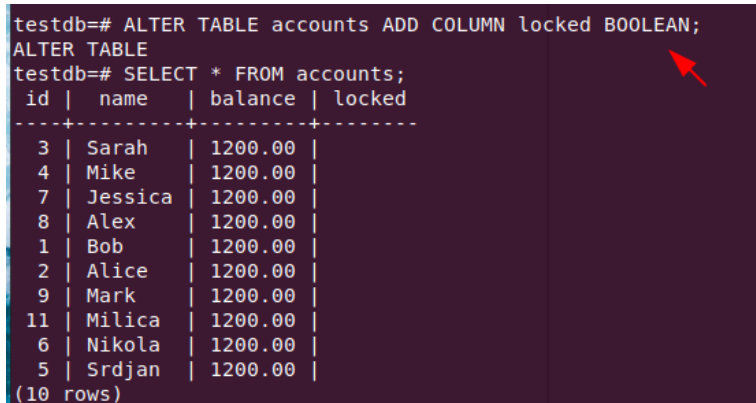
```

Slika 45 – Transakcija T2

Problem pod nazivom *Write Skew* je nemoguće otkloniti sa *Repeatable Read* nivoom i javlja se kada dve konkurentne transakcije vrše čitanje redova koji se preklapaju, nakon čega vrše ažuriranje i komitovanje [16]. Kao krajnji rezultat dobijamo stanje baze, koje nije isto kao nakon bilo kog sekvencijalnog izvršavanja istih transakcija. Pobeđuje transakcija koja prva komituje, dok se druga rolbekuje [21].

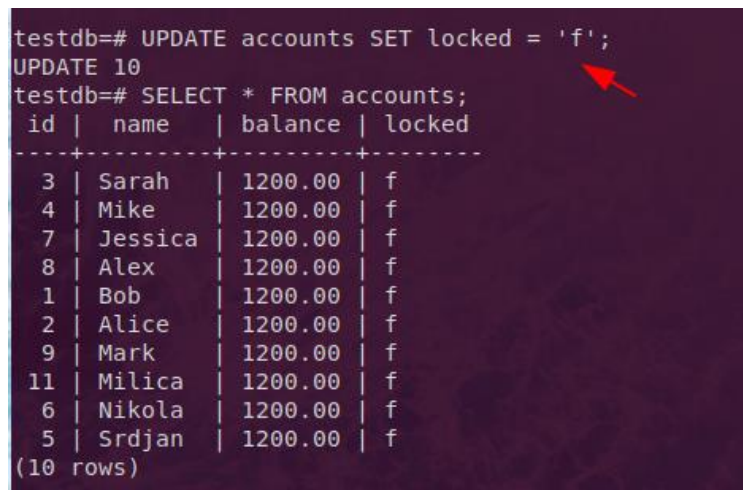
Za potrebe ilustracije *Write Skew* problema tabela *accounts* je dopunjena kolonom *locked* tipa *boolean* što zapravo predstavlja da li je nalog zaključan ili ne. Izmene su date na slikama 46 i 47 u nastavku.

```
testdb=# ALTER TABLE accounts ADD COLUMN locked BOOLEAN;
ALTER TABLE
testdb=# SELECT * FROM accounts;
 id |  name  | balance | locked 
----+-----+-----+-----
  3 | Sarah  | 1200.00 | 
  4 | Mike   | 1200.00 | 
  7 | Jessica| 1200.00 | 
  8 | Alex   | 1200.00 | 
  1 | Bob    | 1200.00 | 
  2 | Alice  | 1200.00 | 
  9 | Mark   | 1200.00 | 
11 | Milica | 1200.00 | 
  6 | Nikola | 1200.00 | 
  5 | Srdjan | 1200.00 | 
(10 rows)
```



Slika 46 – Alter table

```
testdb=# UPDATE accounts SET locked = 'f';
UPDATE 10
testdb=# SELECT * FROM accounts;
 id |  name  | balance | locked 
----+-----+-----+-----
  3 | Sarah  | 1200.00 | f
  4 | Mike   | 1200.00 | f
  7 | Jessica| 1200.00 | f
  8 | Alex   | 1200.00 | f
  1 | Bob    | 1200.00 | f
  2 | Alice  | 1200.00 | f
  9 | Mark   | 1200.00 | f
11 | Milica | 1200.00 | f
  6 | Nikola | 1200.00 | f
  5 | Srdjan | 1200.00 | f
(10 rows)
```



Slika 47 – Update table

### Primer – Write Skew kod Repeatable Read nivoa izolacije

Transakcija T1 želi da izvrši ažuriranje nad tabelom *accounts* tako što postavlja sve redove polja *locked* sa *false* na *true*, dok transakcija T2 pokušava obrnuto. Kada bi se ove transakcije izvršile jedna za drugom, krajnji rezultat bi bio da svi redovi imaju istu vrednost za polje *locked*, s tim što bi vrednost tih redova bila *false* ili *true* u zavisnosti od toga koja će se prvo izvršiti.

```

testdb=# SELECT name, locked FROM accounts;
 name | locked
-----+-----
 Jessica | t
   Alex | t
   Mark | t
  Milica | t
  Nikola | t
   Sarah | f
   Mike | f
   Bob  | f
   Alice | f
  Srdjan | f
(10 rows)

testdb=# BEGIN
testdb=# ISOLATION LEVEL REPEATABLE READ;
BEGIN
testdb=# UPDATE accounts SET locked = true WHERE locked = false;
UPDATE 5
testdb=# COMMIT;
COMMIT

```

Slika 48 – Transakcija T1

Međutim, iz razloga što je MVCC baziran na različitim verzijama redova, ovo dovodi do toga da je krajnji rezultat zamena vrednosti što se može videti na slikama 48 i 49.

```

testdb=# BEGIN
testdb=# ISOLATION LEVEL REPEATABLE READ;
BEGIN
testdb=# UPDATE accounts SET locked = false WHERE locked = true;
UPDATE 5
testdb=# COMMIT;
COMMIT
testdb=# SELECT name, locked FROM accounts;
 name | locked
-----+-----
  Sarah | t
   Mike | t
   Bob  | t
  Alice | t
  Srdjan | t
 Jessica | f
   Alex | f
   Mark | f
  Milica | f
  Nikola | f
(10 rows)

```

Slika 49 – Transakcija T2

**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 nepravilnostima, 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 [19, 20].

#### *Primer - Sprečavanje Write Skew-a korišćenjem Serializable 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 slikama 50 i 51.

```
testdb=# SELECT name, locked FROM accounts;
 name | locked
-----+-----
 Sarah | f
  Mike | f
   Bob | f
  Alice | f
 Srdjan | f
 Jessica | t
   Alex | t
   Mark | t
  Milica | t
  Nikola | t
(10 rows)

testdb=# BEGIN
testdb=# ISOLATION LEVEL SERIALIZABLE;
BEGIN
testdb=# UPDATE accounts SET locked = true WHERE locked = false;
UPDATE 5
testdb=# COMMIT;
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.
```

Slika 50 – Transakcija T1

```
testdb=# BEGIN
testdb=# ISOLATION LEVEL SERIALIZABLE;
BEGIN
testdb=# UPDATE accounts SET locked = false WHERE locked = true;
UPDATE 5
testdb=# COMMIT;
testdb=# SELECT name, locked FROM accounts;
 name | locked
-----+-----
 Sarah | f
  Mike | f
   Bob | f
  Alice | f
 Srdjan | f
 Jessica | f
   Alex | f
   Mark | f
  Milica | f
  Nikola | f
(10 rows)
```

Slika 51 – Transakcija T2

### 3.4 Eksplicitno zaključavanje

PostgreSQL obezbeđuje mehanizme kojima korisnik može eksplicitno da kontroliše konkurentan pristup podacima u bazi podataka. Za kontrolu konkurencije se

koriste različiti režimi (*mods*) zaključavanja. Glavna razlika između ovih režima (modova) je u tome sa kojima od ostalih režima 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 strane 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 omogućava 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 dođe u konflikt sa sobom. Na slici 52 dat je prikaz svih modova zaključavanja tabela kao i u kakvom su međusobnom odnosu (konfliktnom ili nekonfliktnom). Sa X je označeno da su modovi u konfliktu dok prazno polje označava suprotno [22].

| Requested Lock Mode    | Current Lock Mode |           |               |                        |       |                     |           |                  |
|------------------------|-------------------|-----------|---------------|------------------------|-------|---------------------|-----------|------------------|
|                        | 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 52

### Primer eksplicitnog zaključavanja tabele

Na slici 53 dat je primer eksplicitnog zaključavanja tabele, gde transakcija T1 najpre izvrši zaključavanje (1), nakon čega T2 izdaje komandu za čitanjem tabele, na slici 54 (2) 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.

```
testdb=# BEGIN;
BEGIN
testdb=# LOCK accounts IN ACCESS EXCLUSIVE MODE NOWAIT;
LOCK TABLE
testdb=# INSERT INTO accounts(name,balance,locked) VALUES ('Jovana',12000,true);
INSERT 0 1
testdb=# INSERT INTO accounts(name,balance,locked) VALUES ('Milos',24000,false);
INSERT 0 1
testdb=# COMMIT;
COMMIT
```

Slika 53 – Transakcija T1

Nakon što T1 transakcija izvrši upis (3 i 4) i komituje (5), dolazi do oslobađanja *lock*-a od strane T1 i T2 može pročitati tabelu. Sa slike 53 se još vidi da transakcija T2 vidi novododate redove od strane T1.



```

testdb=# BEGIN;
BEGIN
testdb=# SELECT * FROM accounts;
 id | name   | balance | locked
----+-----+-----+-----
  3 | Sarah  | 1200.00 | t
  4 | Mike   | 1200.00 | t
  1 | Bob    | 1200.00 | t
  2 | Alice  | 1200.00 | t
  5 | Srdjan | 1200.00 | t
  7 | Jessica| 1200.00 | f
  8 | Alex   | 1200.00 | f
  9 | Mark   | 1200.00 | f
 11 | Milica | 1200.00 | f
  6 | Nikola | 1200.00 | f
 12 | Jovana | 12000.00 | t
 13 | Milos  | 24000.00 | f
(12 rows)

testdb=# COMMIT;
COMMIT

```

Slika 54 – Transakcija T2

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 55 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.

|                   | Current Lock Mode |           |                   |            |
|-------------------|-------------------|-----------|-------------------|------------|
|                   | 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 55

#### Primer eksplicitnog zaključavanja reda

Na slikama 56 i 57 je dat primer zaključavanja redova i uočava se da transakcija T1 najpre zaključava red sa FOR UPDATE lock-om (1), nakon čega T2 pokušava da izvrši upis u isti red, što dovodi do blokiranja ove transakcije (2).

```

testdb=# BEGIN;
BEGIN
testdb=# SELECT * FROM accounts WHERE name = 'Srdjan' FOR UPDATE;
  id | name  | balance | locked
-----+-----+-----+-----
   5 | Srdjan | 1200.00 | t
(1 row)
testdb=# COMMIT;
COMMIT

```

Slika 56 – Transakcija T1

Transakcija T2 nastavlja sa radom tek nakon što T1 komitiruje i oslobodi *lock* nad redom koji T2 želi da modifikuje (3).

```

testdb=# BEGIN;
BEGIN
testdb=# UPDATE accounts SET balance = balance + 5000 WHERE name = 'Srdjan';
UPDATE 1
testdb=# SELECT * FROM accounts WHERE name = 'Srdjan';
  id | name  | balance | locked
-----+-----+-----+-----
   5 | Srdjan | 6200.00 | t
(1 row)
testdb=# COMMIT;
COMMIT

```

Slika 57 – Transakcija T2

## 4. Zaključak

Baza podataka predstavlja zajednički resurs koji mogu da koriste više procesa i korisnika konkurentno. Jedna transakcija predstavlja logičku jedinicu procesiranja u bazi, odnosno sve operacije koje pristupaju bazi i nalaze se između početne i završne oznake za transakciju se smatraju delom jedne, nedeljive, logičke celine. Kod aplikacija kao što su: bankarski sistemi, softveri za licitaciju vozila, sistemi za rezervaciju avio, autobuskih ili železničkih karata, baza podataka mora da bude u stanju da istovremeno procesira veliki broj transakcija koje joj pristižu od strane korisnika, a da pri tome konzistentnost baze ostane očuvana. Konkurentno izvršavanje transakcija bez kontrole bi moglo da dovede do nekonzistentnosti baze, *deadlock*-ova, loše performanse, pa i do otkaza samog sistema. Da ne bi došlo do ovih nepravilnosti i da bi se garantovalo svojstvo izolacije transakcije, koriste se protokoli za kontrolu izolacije kao što su na primer: 2PL protokol, *timestamp* protokol ili MVCC. Takođe u nekim sistemima je moguće definisati nivoe izolacije, kojima se podešava nivo tolerancije sistema na različite nepravilnosti prouzrokovane konkurentnim izvršavanjem. Kao i većina naprednih DBMS-ova, PostgreSQL obezbeđuje konkurentan pristup bazi, dok se istovremeno brine o ACID svojstvima transakcija. PostgreSQL omogućuje korisnicima, pored osnovne kontrole, da definišu ponašanje transakcija kada je reč o svojstvu izolacije. Takođe koristi MVCC protokol kada se radi o konkurentnom 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, prilikom pokretanja neke transakcije. Ukoliko je potrebno, korisnik uvek može da eksplicitno kontroliše pristup tabeli ili redovima kroz sistem eksplicitnog zaključavanja.

## 5. Literatura

- [01] Ramez Elmasri and Shamkant Navathe. (2010). Fundamentals of Database Systems (6th. ed.). Addison-Wesley Publishing Company, USA.
- [02] Yu, S. (2009). ACID properties in distributed databases. Advanced eBusiness Transactions for B2B-Collaborations.
- [03] Gray, Jim (September 1981). "The Transaction Concept: Virtues and Limitations
- [04] Gray, Jim; and Reuter, Andreas; Distributed Transaction Processing: Concepts and Techniques, Morgan Kaufman, 1993
- [05] L. Libkin. Database Systems. Transactions and Concurrency Control. The University of Edinburgh. School of Informatics. <https://homepages.inf.ed.ac.uk>
- [06] Prof.dr Leonid Stoimenov. Katedra za računarstvo, EFN [https://cs.elfak.ni.ac.rs/nastava/pluginfile.php/18034/mod\\_resource/content/1/Oporavak2016.pdf](https://cs.elfak.ni.ac.rs/nastava/pluginfile.php/18034/mod_resource/content/1/Oporavak2016.pdf)
- [07] Raghu Ramakrishnan and Johannes Gehrke. (2002). Database Management Systems (3rd. ed.). McGraw-Hill, Inc., USA.
- [08] States of Transactions  
[https://www.tutorialspoint.com/dbms/dbms\\_transaction.htm](https://www.tutorialspoint.com/dbms/dbms_transaction.htm)
- [09] Types of Schedules in DBMS  
<https://www.geeksforgeeks.org/types-of-schedules-in-dbms/>
- [10] <https://www.geeksforgeeks.org/concurrency-problems-in-dbms-transactions/>
- [11] <https://www.tutorialcup.com/dbms/concurrency-control-protocols.htm>
- [12] <https://www.geeksforgeeks.org/validation-based-protocol-in-dbms/>
- [13] [https://www.tutorialspoint.com/postgresql/postgresql\\_transactions.htm](https://www.tutorialspoint.com/postgresql/postgresql_transactions.htm)

- [14] <https://www.postgresqltutorial.com/postgresql-transaction/>
- [15] <https://www.postgresql.org/docs/8.3/tutorial-transactions.html>
- [16] <http://www.interdb.jp/pg/pgsql05.html>
- [17] <https://www.quora.com/What-is-a-DB-commit-log>
- [18] [https://edu.postgrespro.com/2dintro/o3\\_arch\\_mvcc.pdf](https://edu.postgrespro.com/2dintro/o3_arch_mvcc.pdf)
- [19] <https://www.postgresql.org/docs/9.1/transaction-iso.html>
- [20] <https://pgdash.io/blog/postgres-transactions.html>
- [21] <https://wiki.postgresql.org/wiki/SSI>
- [22] <https://www.postgresql.org/docs/9.4/explicit-locking.html>