UNIVERZITET U NISU

ELEKTRONSKI FAKULTET

**IZOLACIJA I KONKURENTNI PRISTUP KOD POSTGRESQL BAZE PODATAKA**

Seminarski rad

Studijski program: Racunarstvo i informatika

Modul: Softversko inzenjerstvo

Mentor: Student:

Prof. dr Aleksandar Stanimirovic Zeljko Vasic, br.ind. 1808

Sadrzaj

[1. Uvod 2](#_Toc8562)

[2. Nivoi izolacije u bazama podataka 3](#_Toc16532)

[2.1. Read uncommitted 3](#_Toc15708)

[2.2. Read commited 3](#_Toc10088)

[2.3. Repeatable read 3](#_Toc15179)

[2.4. Serializable 4](#_Toc28668)

[3. Razlozi za primenu izolacije u bazama podataka 4](#_Toc3464)

[4. Mehanizmi implementiranja nivoa izolacije 8](#_Toc7162)

[4.1. MVCC 8](#_Toc11665)

[4.2. Snapshot mehanizam 8](#_Toc11130)

[4.3. Row locking 9](#_Toc7526)

[4.4. Table locking 9](#_Toc14744)

[4.5. Predicate locking 9](#_Toc12366)

[5. Strategije upravljanja konkurentim pristupom 9](#_Toc18104)

[5.1. Pessimistic locking 9](#_Toc16008)

[5.2. Optimistic locking 12](#_Toc26076)

[6. Lock-ovi i mehanizmi zakljucavanja 14](#_Toc17352)

[7. Konkurentnost i performanse 18](#_Toc28088)

# Uvod

Izolacija obezbedjuje da transakcije koje se izvrsavaju istovremeno uticu jedna na drugu na predvidiv nacin. Fokusira se na to kako ce se transakcije ponasati kada istovremeno pristupaju istim resursima. Izolacija odredjuje koliko su transakcije medjusobno izolovane, odnosno odvojene jedne od drugih i u kojoj meri im je dozvoljeno da vide promene drugih transakcija koje se trenutno izvrsavaju.

Vecina danasnjih baza podataka omogucava konkurentno izvrsavanje transakcija nad bazom. To sa sobom donosi rizik da se podaci promene izmedju dva citanja kada ne bi trebalo da se promene, da se citaju nepotvrdjene vrednosti iz drugih transakcija ili da se podaci menjaju istovremeno od strane vise transakcija. Izolacija omogucava kontrolisanje anomalija koje ugrozavaju konzistentnost baze podataka.

Najcesce anomalije su:

* Dirty read - nastaje kada jedna transakcija cita necommitovane podatke druge transakcije, odnosno podatke koji jos uvek nisu potvrdjeni i potencijalno mogu biti ponisteni.
* Non-repetable read - nastaje kada jedna transakcija izvrsava isti upit za citanje vise puta, ali dobija razlicite rezultate. To se desava zato sto je druga transakcija, dok se prva jos uvek izvrsavala, izmenila te podatke u bazi i prva transakcija dobija drugaciji rezultat nego prilikom prvog citanja.
* Lost update - nastaje kada dve transakcije menjaju istovremeno isti red, i jedna transakcija pregazi promene druge transakcije.
* Phantom read - nastaje slicno kao i Non-repetable read osim sto druga transakcija u medjuvremenu dodaje redove koji odgovaraju filterima upita za citanje, i prva transakcija dobija vise redova u drugom citanju nego u prvom citanju.
* Write skew - nastaje kada dve transakcije istovremeno menjaju razlicite redove i nastaje logicki problem. Nema direktnog fizickog konflikta ali kada se spoje promene te dve transakcije baza gubi integritet.

Izolacija je jedno od cetiri osnovna svojstva koje cine poznati ACID model(Atomocity, Consistency, Isolation, Durability).

Atomocity(Atomicnost) je svojstvo koje osigurava da ce se transakcija izvrsiti u celini ili se nece izvrsiti uopste. Transakcija se gleda kao celina, i ne moze se deliti, odnosno nije moguce izvrsiti samo jedan deo transakcije.

Consistency(Konzistentnost) je svojstvo da se baza prevodi iz jednog konzistentnog stanja u drugo konzistentno stanje. Odnosno baza nikada ne moze da se nadje u nekonzistentnom stanju. Nekonzistentna baza je ona baza ciji su podaci nepotpuni, kontradiktorni ili nevazeci.

Durability(Trajnost) je svojstvo da promene u bazi koje su jednom potvrdjene ostaju trajno zabelezene u bazi, ne gube se cak ni ako baza padne.

# Nivoi izolacije u bazama podataka

Nivoi izolacije odredjuju koliko su transakcije izolovane jedna od druge, odnosno koje anomalije konkurentnog pristupa su dozvoljene a koje ne.

SQL standard definise 4 standardna nivoa izolacije:

1. Read uncommitted
2. Read committed
3. Repetable Read
4. Serializable

## Read uncommitted

Najslabiji nivo. Transakcije mogu da vide nepotvrdjene(unccommited) promene drugih transkacije.

To znaci da se moze desiti dirty read, odnosno da jedna transakcija cita podatke koja druga transakcija jos nije commitovala i moze da izvrsi rollback. Takodje je moguce da podaci koje koristi jedna transkacija budu izmenjeni od strane druge transkacije, u toku trajanja prve transakcije, i tada dolazi do nonrepetable reads (isti red se menja izmedju dva citanja) ili phantom reads (novi redovi se pojave izmedju dva citanja sa istim kriterijumom).

PostgreSQL ne implementira ovaj nivo.

## Read commited

Read commited je podrazumevani nivo izolacije u vecini baza podataka (ukljucujuci i PostgreSQL) . Svaki SQL upit vidi samo potvrdjene podatke,odnosno upit u trenutnoj transakciji ne moze da procita podatke koji su izmenjeni od strane druge transakcije a jos uvek nisu commitovan (dirty read je sprecen na ovom nivou izolacije). Ipak,i dalje se desava problem da ako se u jednoj transakciji jedan upit izvrsi vise puta rezultat se moze promeniti ako neka druga transkacija u medjuvremenu upise i potvrdi nove vrednosti.

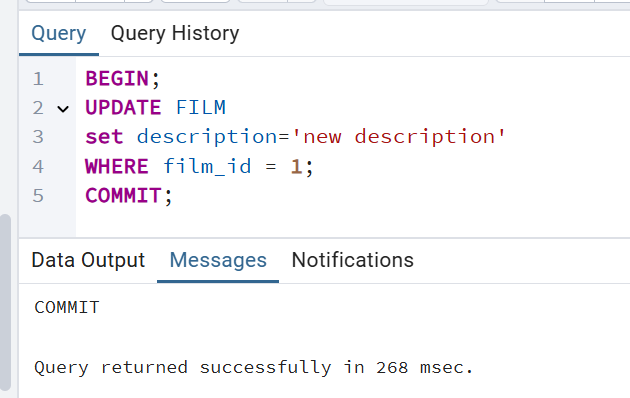
Dozvoljene anomalije su i dalje non-repetable reads, lost-update, phantom reads i write-skew.

Non-repetable read primer:

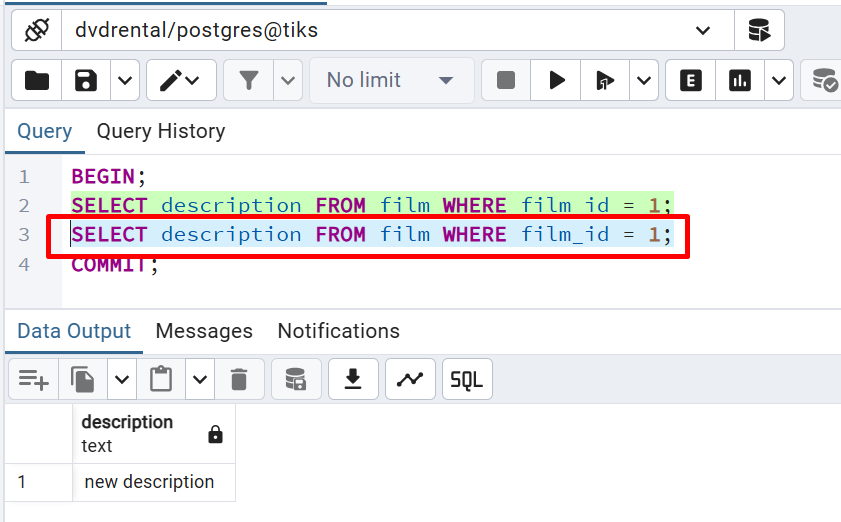
Prva transakcija pocela sa izvrsavanjem, i izvrsio se samo oznaceni deo, vidimo da je upit vratio odredjeni podatak:



U medjuvremenu druga transakcija se izvrsava u celosti i menja bas taj podatak koji prva transakcija cita:



Prva transakcija nastavlja sa izvrsavanjem, opet izvrsava identican upit kao malopre ali sada dobija drugaciji rezultat, odnosno rezultat koji je upisala druga transakcija u medjuvremenu.

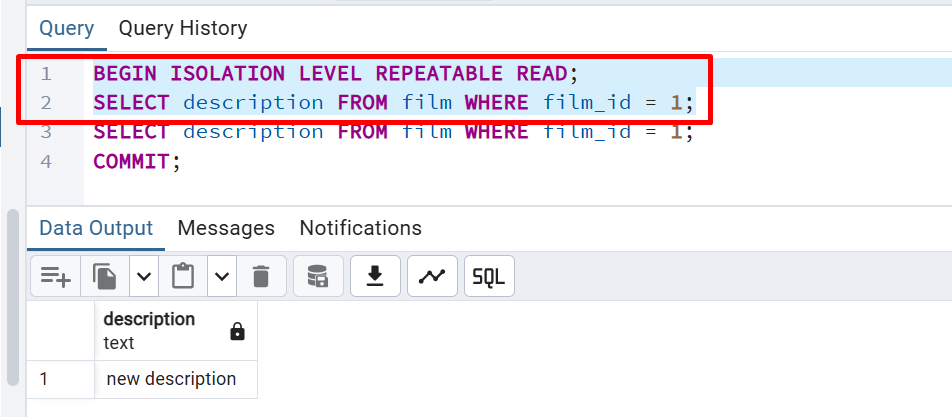


## Repeatable read

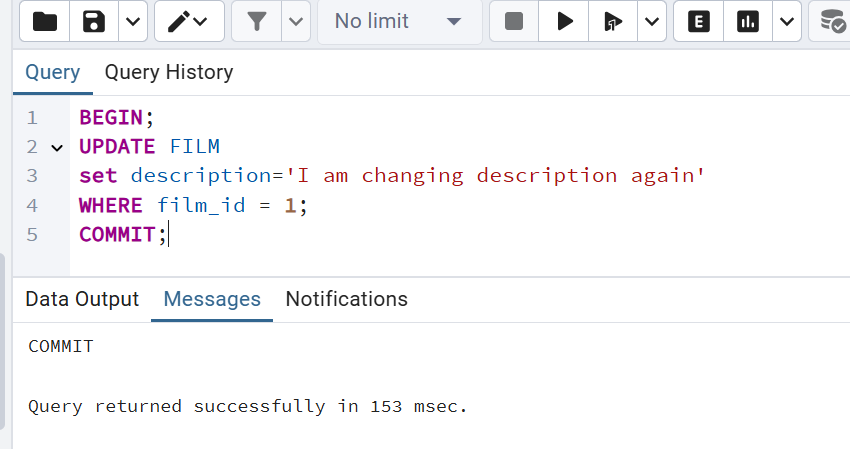
Kao i u prethodnom nivou sprecen je dirty read ali je dodatno **sprecen non-repetable read i lost update**. Nijedna druga transkacija ne moze da menja podatke koji se citaju u trenutnoj transakciji dok se ona ne zavrsi. Garantuje da u okviru jedne transakcije sva ponovljena citanja vracaju iste podatke.

Primer prethodnih transakcija ali na izolacionom nivou repetable read

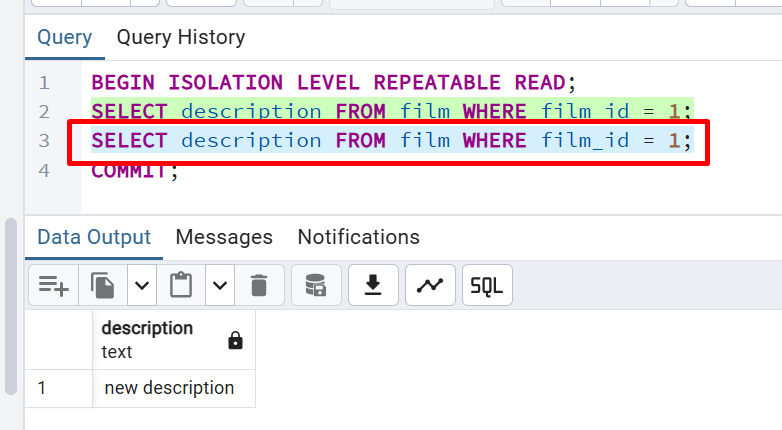
Prva transakcija pocela sa izvrsavanjem, i izvrsio se samo oznaceni deo, vidimo da je upit vratio trenutni podatak:



U medjuvremenu druga transakcija se izvrsava u celosti i menja bas taj podatak koji prva transakcija cita:



Ali ovog puta kada prva transakcija nastavi sa radom dobija se razultat koji je transakcija videla na pocetku izvrsavanja:



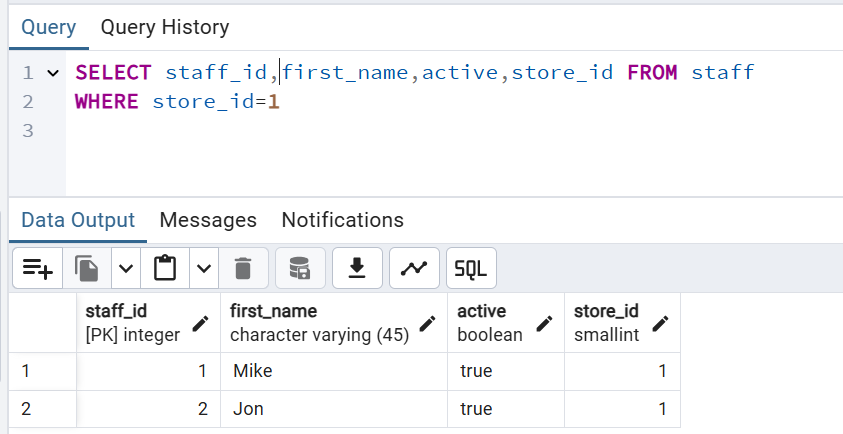
U nekim bazama moze da se desi phantom read na repetable read nivou izolacije,ali u bazama koje implementiraju MVCC(kao sto je i PostgreSQL) phantom read je sprecen zahvaljujuci snapshot mehanizmu. PostgreSQL koristi MVCC mehanizam pa svaka transakcija vidi konzistentan snapshot baze u trenutku kada je ta transakcija pocela.

Write skew je jedina preostala anomalija, ostale anomalije su sprecene na nivou Repetable read.

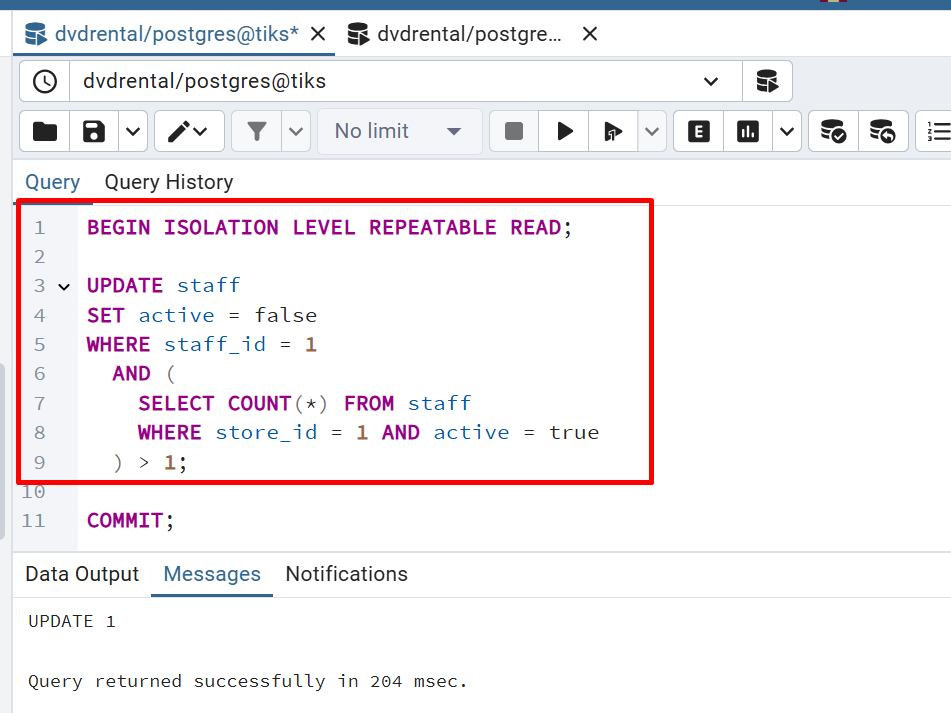
Primer write skew:

Imamo dve transakcije i obe proveravaju da li u radnji postoji vise od 1 aktivnog radnika, ako postoji taj radnik moze zavrsiti smenu, odnosno stavlja se active=false.

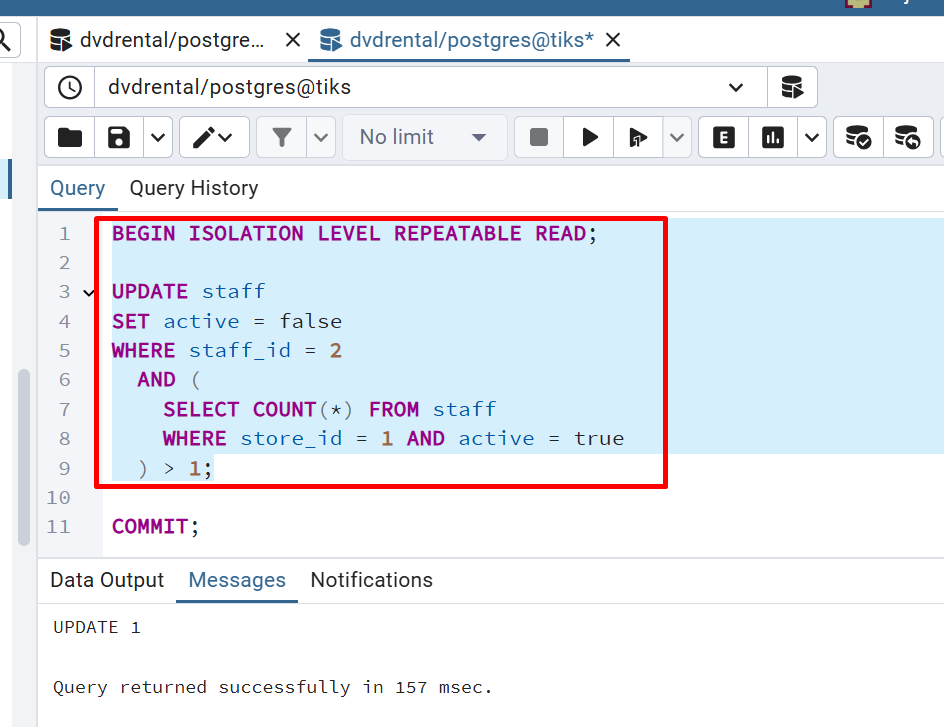
Pre pocetka obe transakcije postoje dva aktivna radnika, pretpostavimo da je potrebno da samo jedan radnik bude aktivan u jednom trenutku



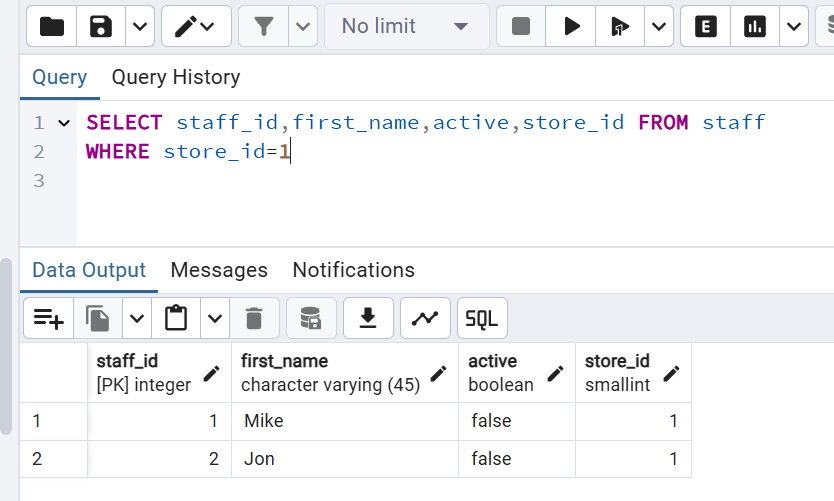
Prva transakcija proverava stanje, vidi da trenutno postoji vise od 1 radnika u toj radnji i izvrsava se, odnosno radniku sa staff\_id=1 zavrsava smenu.



Druga transakcija istovremeno proverava to isto, takodje vidi da ima vise od jednog radnika u radnji ali zavrsava smenu radniku sa staff\_id=2.



Nakon commitovanja obe transakcije, ukoliko se opet izvrsi pocetni upit za proveru stanja u bazi imamo:



Za oba radnika je postavljeno active=false, sto ostavlja prodavnicu bez nadzora sto u slucaju da je to radnja koja radi 24/7 ne sme da se desi.

## Serializable

Transakcije se izvrsavaju kao da su serijalizovane jedna za drugom, iako se fizicki preklapaju. Nema dirty read-a, ni non-repeatable read ni phantom read-a. Negativna strana je sto su ovde performanse najslabije u poredjenju sa ostalim nivoima izolacije.

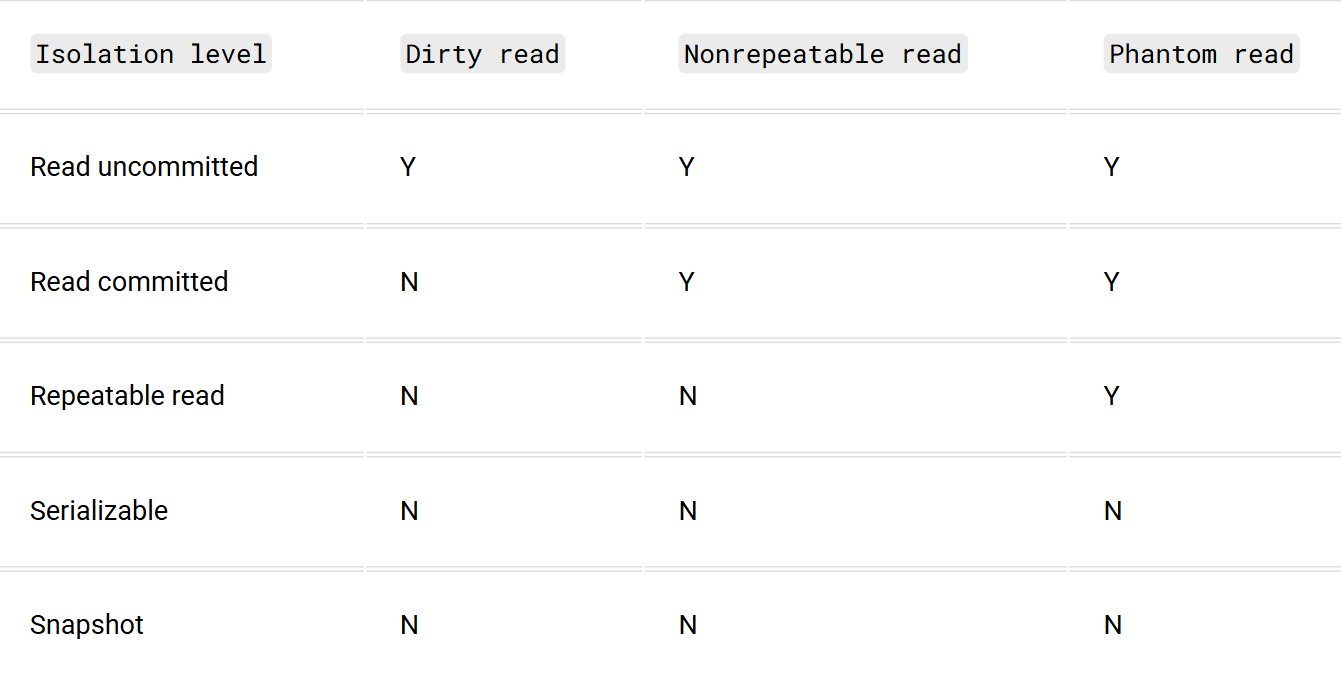
Resava i write skew.

# Primeri anomalija zbog kojih je izolacija potrebna

**Race condition** je situacija u konkurentnim sistemima kada vise procesa ili transakcija pristupa nekim podacima a krajnji rezultat zavisi od redosleda izvrsavanja tih procesa ili transakcija. Ako se procesi izvrsavaju u neocekivanom redosledu krajnji rezultat moze biti pogresan ili nekonzistentan. Izolacija transakcija sluzi da spreci ili detektuje race conditione cime se obezbedjuje konzistentnost podataka, odnosno da je krajnji rezultat transkacija predvidljiv i da ne zavrsi od toga koja transakcija ce se prva izvrsiti.

Tipicni pimeri race condition-a su:

* Dirty read
* Dirty write
* Non-repetable read
* Phantom read
* Lost update



Na slici iznad je tabela sa nivoima izolacije i da li je moguce da se jave odredjene anomalije u njima.

*Dirty read scenario*

Pretpostavimo za dve transakcije pristupaju tabeli customer istovremeno.

Prva transakcija je:

BEGIN

UPDATE customer

SET email = 'newemail@gmail.com'

WHERE customer\_id = 1;

Druga transakcija je:

SELECT email

FROM customer

WHERE customer\_id = 1;

Kod dirty read-a bi druga transakcija videla novi email koji upisuje prva transakcija iako ga ona jos uvek nije potvrdila. Ukoliko prva transkacija odradi rollback, to bi znacilo da je transakcija 2 procitala nevazeci podataka.

Ovo je primer dirty read-a ali nije moguce da se dirty-read pojavi u PostgreSQL-u zbog njegovog MVCC sistema.

*Non-repetable read scenario*

Ukoliko imamo transakciju 1

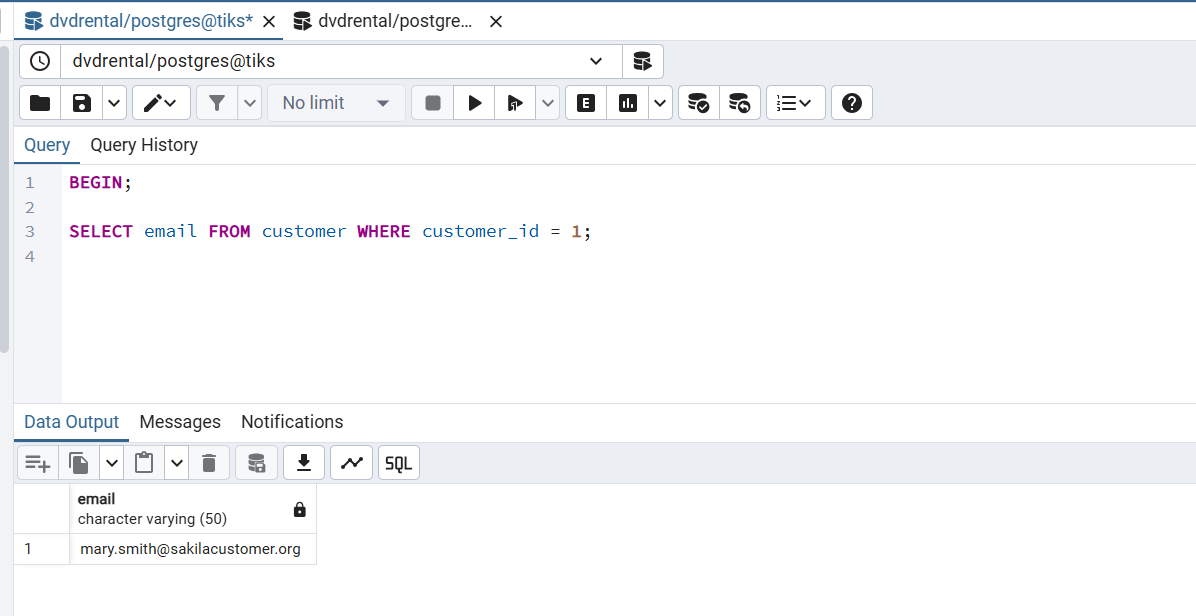
BEGIN;

SELECT email FROM customer WHERE customer\_id = 1;

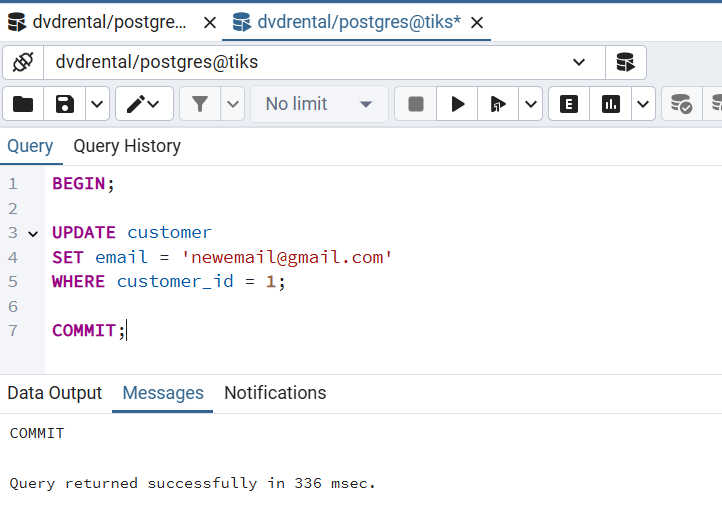
…

SELECT email FROM customer WHERE customer\_id = 1;

Transakcija 1 je pocela sa izvrsavanjem i izvrsio se deo transkcije sa slike ispod, vidimo da je vracen email customera sa customer\_id=1

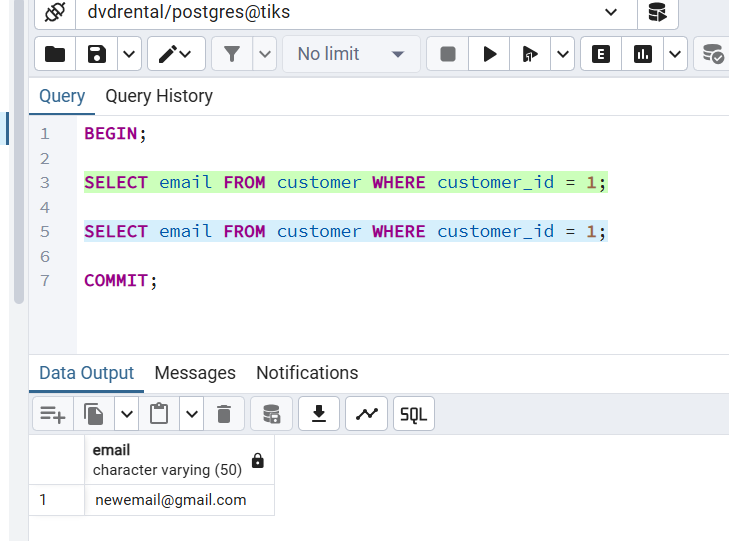


A u medjuvremenu transkacija 2 krece i je izvrsena u potpunosti.



Transkacija 2 je izmenila onaj podatak koji transkacija 1 cita.

Sada kada transakcija 1 nastavlja sa izvrsavanjem i opet izvrsava identican upit ali dobija razlicit rezultat.

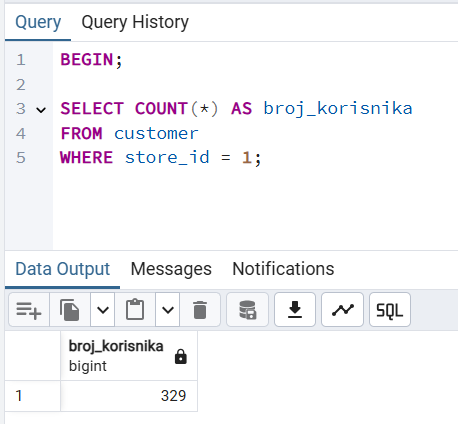


Ova anomalija se zove non-repetable reads, zato sto jedan te isti upit vraca 2 razlicita rezultata u okviru iste transkacije.

*Phantom-read scenario*

Ukoliko transkacija 1 procita skup redova koji zadovoljaju neki uslov,druga transkacija u medjuvremenu ubaci nove redove koji zadovoljaju taj isti uslov, a transkacija 2 opet cita te redove, desice se da transakcija 1 vidi dodatne redove.

Transakcija 1 je pocela sa izvrsavanjem i izvrsio se ovaj deo transkacije:



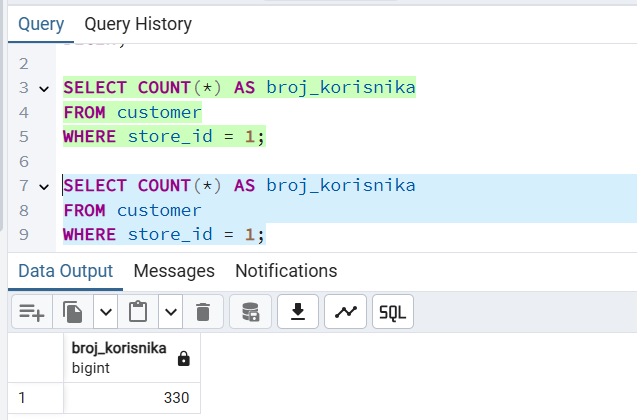
Vidimo da je broj korisnika koji vidi transkacija 1 trenutno 329.

U medjuvremenu se izvrsila kompletna transakcija 2:



Transakcija 1 nastavlja za izvrsenjem i opet izvrsava identican upit ali sada dobija drugaciji rezultat

Odnosno, sada postoji dodatan red koji zadovoljava uslov. To je phantom-read.



# Mehanizmi implementiranja nivoa izolacije

U PostgeSQL-u mehanizmi koji implementiraju razlicite nivoe izolacije se zasnivaju na kombinaciji MVCC i dodatnih mehanizama zakljucavanja.

## MVCC

Multi-Version Concurrency Control je osnonvni mehanizam koji PostgreSQL automatski koristi da bi omogucio konkurentni pristup bazi podataka bez zakljucavanja za citanje. Svaka transkacija vidi snapshot baze koji je vazio u trenutku kada je ta transakcija pocela.

Koristi se da bi baza podrzala visoke nivoe izolacije, poput Repetable read i Serializable.

Osnovna ideja MVCC je da umesto da transakcije blokiraju jedna drugu prilikom citanja i pisanja, da svaka promena u tabeli stvara novu verziju reda. Svaki red u tabeli sada ima i dodatne kolone **xmin** i **xmax**.

Xmin predstavlja ID transakcije koja je kreirala red, a xmax predstavlja ID transkacije koja je obrisala ili zamenila red. Kada transakcija cita red proverava se vrednost xmin. Ako je xmin manji od ID transakcije a xmax je veci od ID transakcije ili je NULL onda je red vidljiv. U suprotnom se taj red ignorise (nije deo snapshota). Na ovaj nacin svaka transkacija vidi konzistentan snapshot baze cak i kada druge transakcije vrse promene. Transakcije mogu paralelno da citaju podatke bez cekanja drugih transakcija.

## Snapshot mehanizam

Snapshot je mehanizam rada koji koriste moderne baze zasnovane na MVCC(Multi Version Concurrecny Control) principu, ukljucujuci i PostgreSQL.

Upit moze da koristi podatke samo ako ce baza biti u konzistentnom stanju u toku cele transakcije. Ako neka druga transakcija izmeni podatke dok se oni jos koriste u trenutnoj transkaciji, to trenutna transkacija ne vidi. Trenutna transakcija ce raditi sa snapshotom podataka koji su postojali na pocetku transkacije. Snaphost transakcije ne zahtevaju zakljucavanje podataka prilikom citanja podataka niti zabranu ostalim transkacijama da menjaju te podatke. S obzirom da transakcije ne blokiraju citanja i upise, vreme cekanja se smanjuje u odnosu na serializable nivo. Medjutim, moze doci do konflikata pri pisanju (write-write conflict) koji se resavaju abortovanjem jedne transkacije. Takodje moze doci i do write screw-a. Write screw nastaje kada razlicite transakcije menjanju razlicite redove ili kolone u bazi ali zajednici rezultat tih transakcija krsi integritet podataka.

## Row locking

Row locking je osnovni mehanizam kontrole konkurentnosti u PostgreSQL-u. Omogucava da vise transakcija istovremeno pristupa istoj tabeli, ali razlicitim redovima. Ukoliko pokusaju da pristupe istim redovima te tabele dolazi do blokiranja transakcija. Omogucava paralelne upise u istu tabelu i predstavlja balans izmedju izolacije i performansi baze podataka.

Kada transakcija izvrsava komandu za upis podataka u tabelu, sistem automatski zakljucava te redove.

## Table locking

Obuhvata citavu tabelu, ukljucujuci njene redove, indekse i metapodatke. Zakljucavanje na nivou tabele se koristi redje nego zakljucavanje na nivou redova. Koristi se kada se izvrsavaju operacije nad tabelom kao sto su ALTER TABLE ili DROP TABLE. PostgreSQL automatski primenjuje table locking prilikom izvrsavanja tih operacija. Kada se cela tabela zakljuca ostale transakcije koje pokusavaju da pristupe podacima iz te tabele ce biti blokirane sve dok se transakcija ne zavrsi. Ovo obezbedjuje konzistentnost podataka ali i moze narusiti performanse zato sto smanjuje nivo paralelizma.

## Predicate locking

Predicate locking je oblik zaklujcavanja koji ne zakljucava konretne redove u tabeli vec zakljucava uslov koji se definise nad tabelom. Transakcija zakljucava sve redove koji zadovoljvaju neki uslov i sve buduce redove koji zadovoljavaju taj uslov. Ukoliko neka transakcija cita podatke koji zadovoljavaju odredjeni uslov i stavi lock nad tim uslovom, a neka druga transakcija pokusa da doda red koji bi odgovarao tom uslovu, prva transakcija ce i dalje videti nepromenjene podatke sve dok se ne zavrsi. Na ovaj nacin se sprecavaju phantom reads.

# Strategije upravljanja konkurentim pristupom

## Pessimistic locking

Pessimistic locking se u PostgreSQL-u koristi samo po potrebi, i mora se upotrebiti rucno(implicitno). Transakcija zakljucava red koji cita ili menja, kako bi druge transakcije cekale dok se trenutna transakcija ne zavrsi.

Pessimistic locking pretpostavlja da ce do konflikata izmedju transakcija dolaziti cesto. Kada transakcija pristupi podacima za citanje ili pisanje, baza odmah zakljucava te podatke. Ostale transackije koje zele da pristupe tim podacima moraju da cekaju dok se trenutna transkacija ne zavrsi da bi se podaci otkljucali. Koristi se u sistemima gde je konzistentnost vrlo vazna i prihvatljivo je da se transakcije duze izvrsavaju kao sto su bankarski sistemi, sistemi za rezervacije. Lockovi koje pessimistis locking koristi mogu da budu Exclusive lock i Shared lock (obradjeni u poglavlju 6).

Ukoliko je cela transakcija 1:

BEGIN;

SELECT \* FROM customer

WHERE customer\_id = 1

FOR UPDATE;

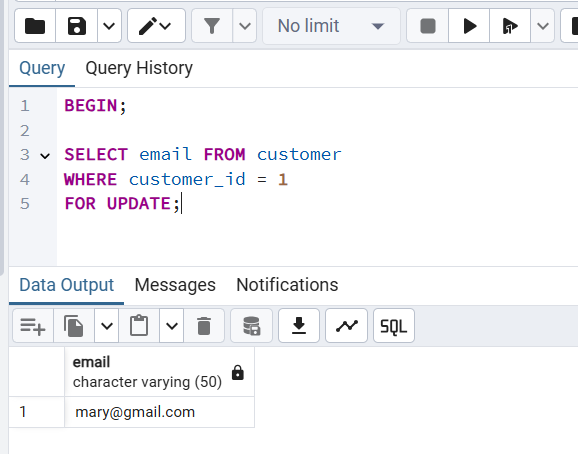
UPDATE customer

SET email = 'newEmailSecured@gmail.com'

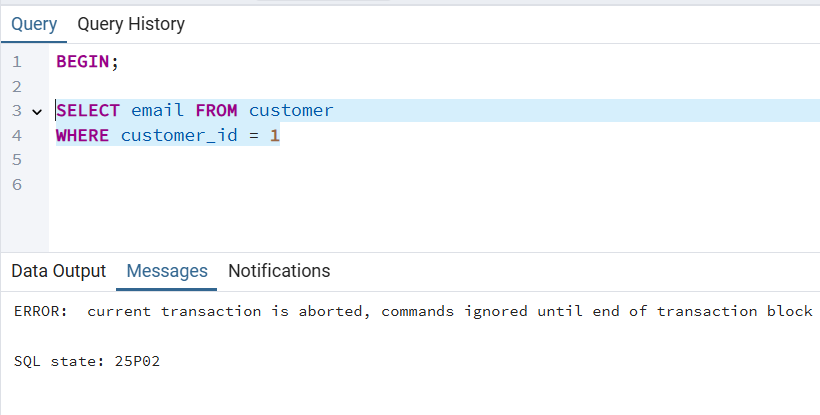
WHERE customer\_id = 1;

COMMIT;

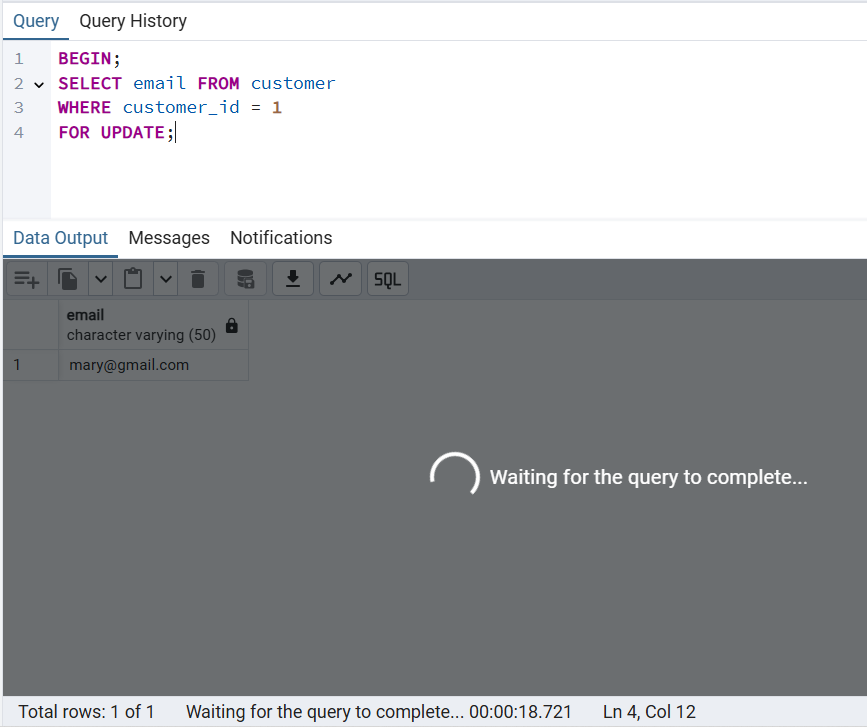
Cak iako se izvrsio samo deo te transakcije



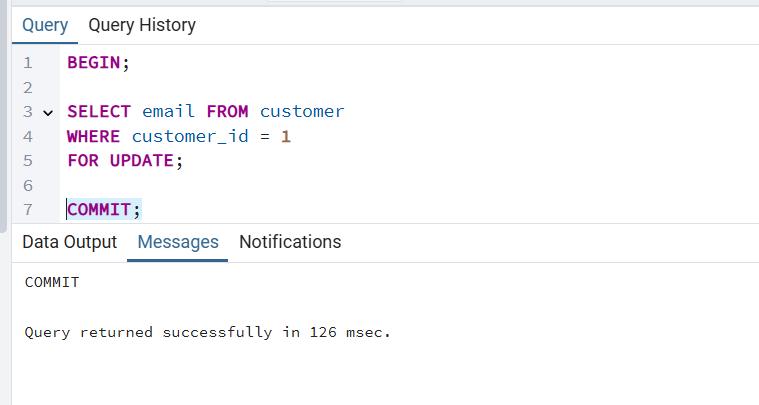
A druga transakcija koja menja isti podatak se izvrsi u potpunosti



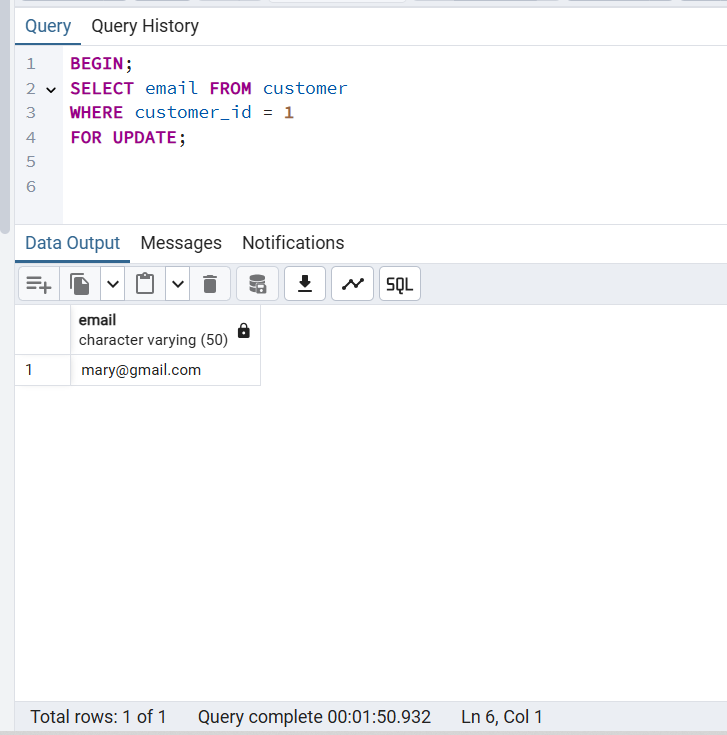
Druga transakcija je blokirana, sve dok se u potpunosti ne zavrsi transakcija 1



Tek kada se izvrsi COMMIT iz prve transakcije i transakcija se kompletira, onda moze da se izvrsi druga transakcija.



Druga transakcija se automatski izvrsila nakon zavrsetka prve transakcije, do tada je jednostavno cekala na izvrsenje

.

## Optimistic locking

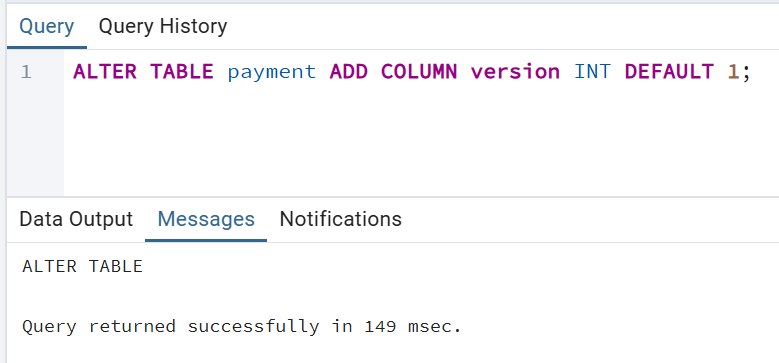
Optimistic lokcing ne koristi fizicke lockove. Optimistic locking se kao i Pessimistic lokcing koristi samo po potrebi i mora se upotrebiti rucno. Transakcija ne zakljucava red dok ga cita ili menja, vec proverava verziju redova prilikom komitovanja. Ako se verzije razlikuju u odnosu na verzije sa pocetka transakcije, transakcija se abortira.

Optimisitic lockingpretpostavlja da konflikti izmedju transakcija nastaju retko. Transakcija ne zakljucava podatke unapred kada ih cita. Kod ovog pristupa je prioritet konkurentnost a konflikti se resavaju pri upisu.

Osnovna ideja optimistic lockinga je da se svakoj tabeli ili zapisu u bazi doda verziona kolona. Najcesce je to celobrojni brojac (version) ili vremenska oznaka (timestamp).

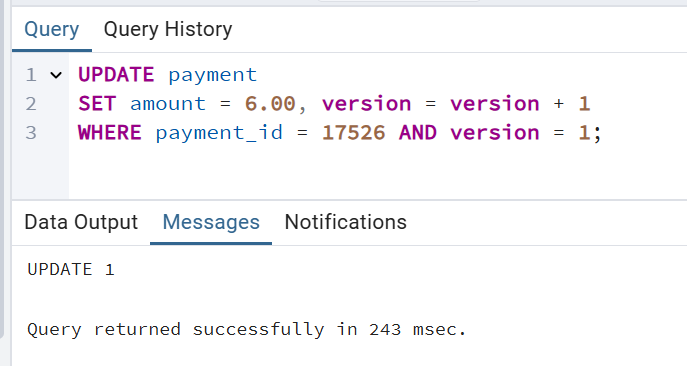
Kada aplikacija ucita neki red iz baze podataka, ona pamti i vrednost verzione kolone. Kada je potrebno da se taj red azurira aplikacija proverava da li je trenutna vrednost verzione kolone ista kao i kada se red ucitao. Ako je vrednost ista to znaci da taj red nije menjan u medjuvremenu i moze se jednostavno izvrsiti azuriranje. Ako je vrednost razlicita to znaci da je u medjuvremenu neka druga transakcija izmenila taj red i trenutna transakcija mora da se ponisti i da opet izvrsi ispocetka nakon uzimanja nove verzije.

Dodavanje kolone version u tabeli payment koja ce sluziti za kontrolu verzija:

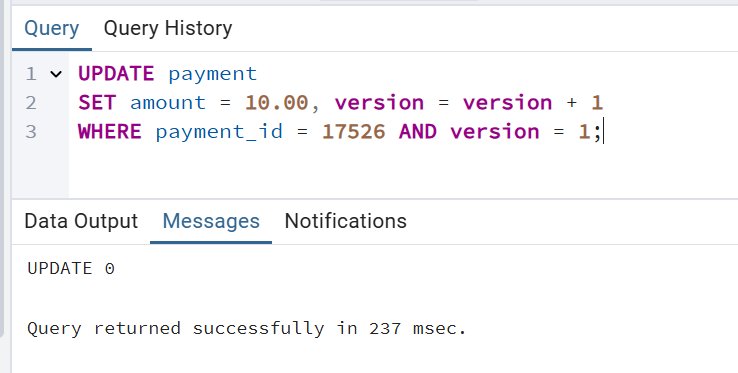


Pretpostavimo da postoje dve paralelne transkacije koje obe menjaju isti red iz tabele payment.

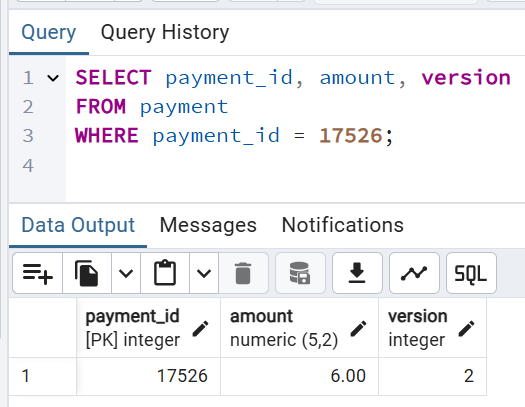
Transkacija 1 se uspesno izvrsila i azurira je kolonu version, odnosno povecala je za 1



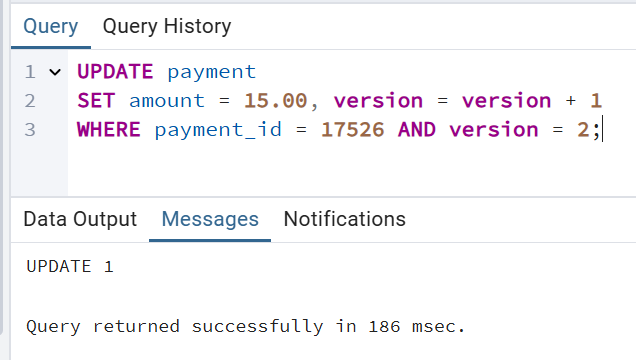
Transkacija 2 pokusava da azurira taj isti red ali i dalje koristi version = 1 sto je zastarela verzija. Baza ne pronalazi odgovarajuci red, s obzirom da kolona version nije vise 1 nego 2.



Transakcija 2 mora da prvo ucita red sa payment\_id = 17526 da bi dobila podatak o trenutnoj verziji a tek onda da azurira taj red.



Ukoliko se sada izvrsi query sa version = 2, baza pronalazi odgovarajuci red i azurira ga.



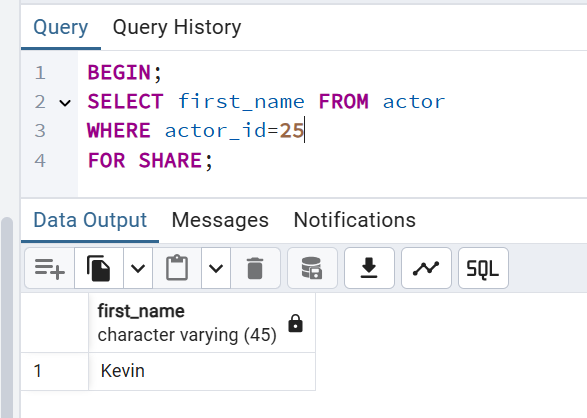
# Lock-ovi i mehanizmi zakljucavanja

Jedan od najvaznijih mehanizama kojima se odrzava kontrola konkurentnosti u bazama podataka gde vise korisnika istovremeno pristupa podacima su **lock-ovi(zakljucavanje)**. Zakljucavanje predstavlja nacin da baza kontrolise pristup odredjenom resursu kako bi se srpecilo da istovremeno izvrsavanje transkacija dovede do nekonzistenih rezultata ili gubitka podataka.

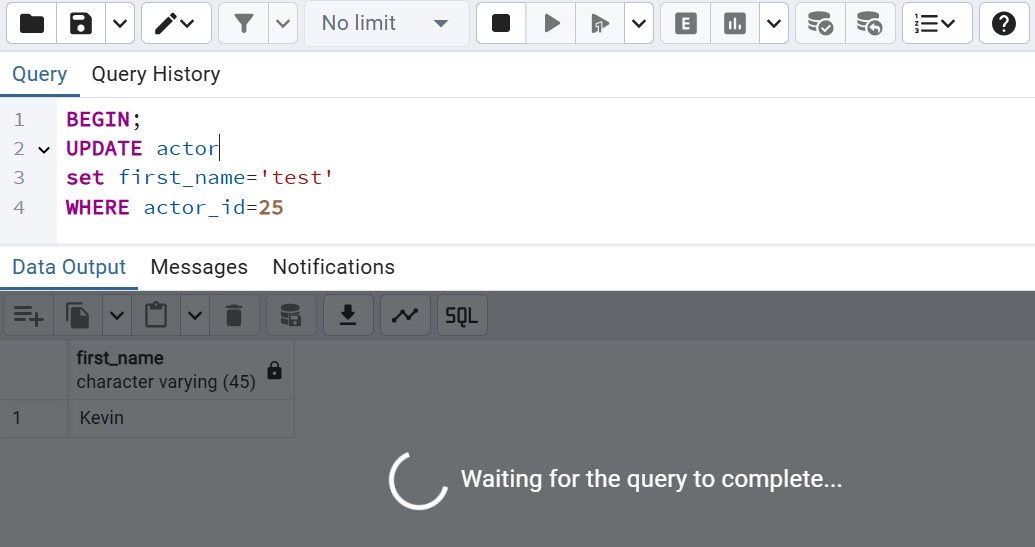
U bazi podataka postoje dve osnovne vrste zaklujcavanja:

* Shared Lock (S-lock)
* Exslusive Lock (X-lock)

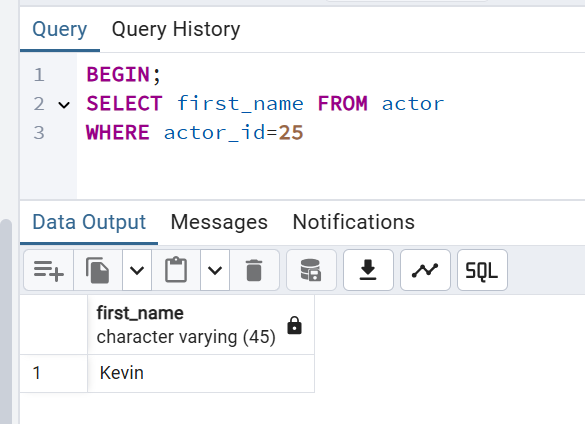
**Shared Lock** se koristi kada transkacija samo cita podatke iz baze ali ih ne menja. Vise transkacija moze istovremeno drzati shared lock nad istim redom ili tabelom s obzriom da ctanje ne remeti konzistentnost podataka. Komandom **FOR SHARE** se stavlja Shared lock na procitane redove.



Ovom transakcijom je red iz tabele actor sa actor\_id=25 shared zakljucan i ostale transakcije mogu da citaju taj podatak ali ne mogu da ga menjaju sve dok se ne izvrsi COMMIT iz trenutne transkacije. Transakcija koja zeli da azurira taj podatak mora ceka u redu i izvrsava se tek kada se trenutna transkacija zavrsi.



Ukoliko neka transkacija zeli samo da procita taj podatak to moze da uradi bez cekanja.

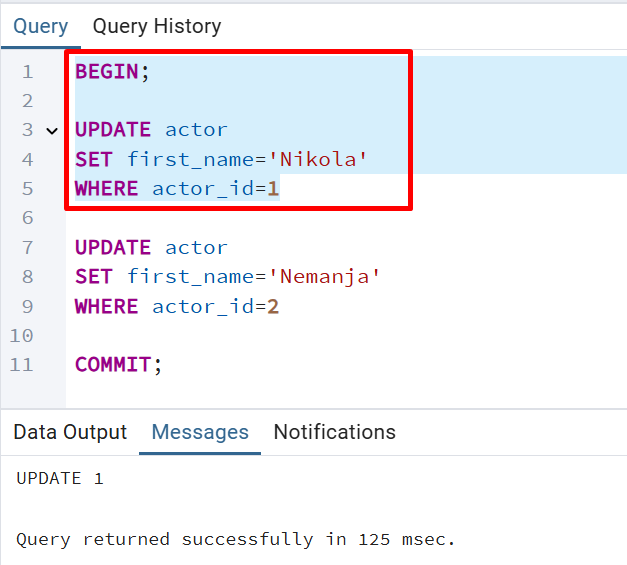


**Exclusive lock** se koristi kada transakcija menja podatke iz baze. Da ne bi dolazilo do nekonzistentnosti neophodno je da samo jedna transkcija ima pristup podacima koji se menjaju. Dok traje X-lock nijedna druga transkacija ne moze dobiti ni X-lock a ni S-lock nad tim podacima.

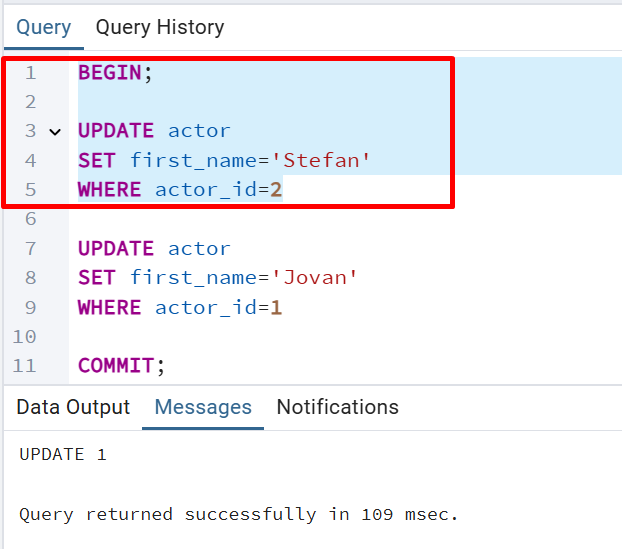
-primer

**Deadlock** se desava kada su dve ili vise transakcija medjusobno blokirane zato sto cekaju jedna drugu i nijedna transakcija ne moze da nastavi sa izvrsavanjem dok je neka transakcija ne odblokira a to se nece desiti nikada. Deadlock obicno nastaje kada transakcije zakljucavaju iste resurse u razlicitom redosledu.

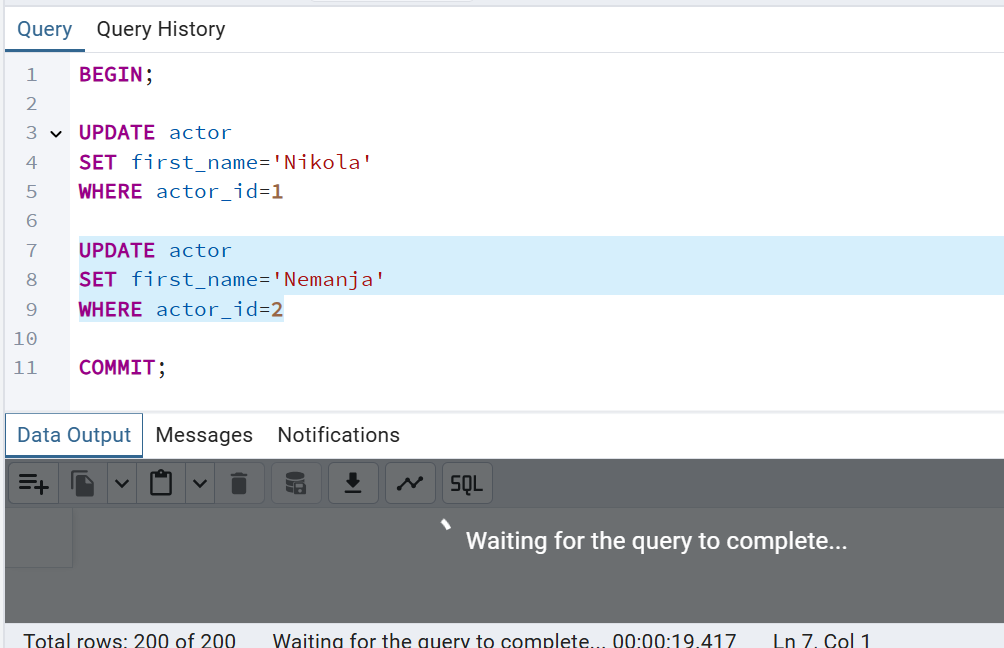
Ukoliko imamo 2 transakcije koje se izvrsavaju paralelno, a transkacija 1 se izvrsila delimicno, i prva transakcija je za sada izvrsila samo:



I imamo transkaciju 2 koja je izvrsila



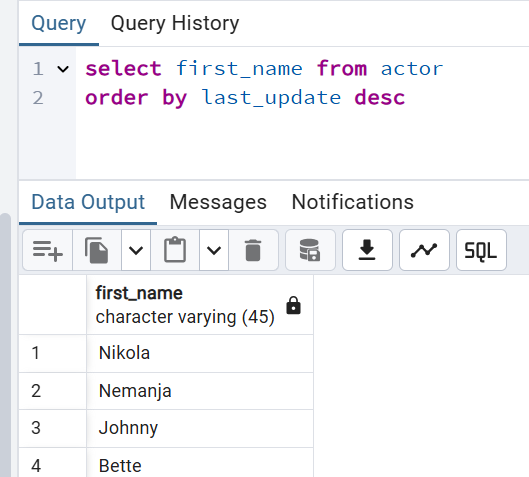
Sada kada transakcija 1 pokusa da pristupi sledecem upitu, odnosno da azurira red kojem je actor\_id=2 onda ceka da ga transakcija 2 oslobodi zato sto ona trenutno drzi lock na njemu



A transakcija 2 da bi ga oslobodila mora da se izvrsi u potpunosti, odnosno da se izvrsi naredba COMMIT, medjutim transkacija 2 se nece izvrsiti do kraja zato sto ona u isto vreme ceka da transkacija1 oslobodi red kojem je actor\_id=1



PostgreSQL ima **deadlock detektor** i ova poruka govori da je PostgreSQL primetio da trenutno postoji deadlock. Transakcija koja je mladja, odnosno ona koja je kasnije krenula sa izvrsavanjem se prekida(abort) da bi starija mogla da nastavi sa izvrsavanjem. Krajnji rezultat ove dve transkacije je ustvari rezultat samo transkacije 1, transakcija 2 je prekinuta.



Prevencija deadlocka:

* Dosledan redosled zakljucavanja - sve transakcija zakljucavaju resurse u istom redosledu
* Kratke transakcije - sto krace traje transkacija to su manje sanse za deadlock
* Minimalno zakljucavanje - zakljucati samo one podatke koji su potrebni
* Optimistic locking moze smanjiti pojavu deadlocka
* Koristiti Shared lock ukoliko je moguce, a Exslusive lock koristiti samo ako je neophodan
* Postaviti timeout za cekanje na lock

**Lock escalation** je proces u kojem baza podataka automatski konvertuje vise manjih zakljucavanja (uglavnom redova) u jedno veliko zakljucavanje (uglavnom tabelu). Desava se kada je veliki broj redova iste tabele zakljucan a to iziskuje veliku potrosnju resursa jer svaki poseban lock zauzima deo memorije i procesora. Baza tada radi lock escalation odnosno umesto da drzi mnogo manjih redova onda ih spaja u jedan veliki lock i jednim lockom drzi celu tabelu.

Pojam slican deadlocku je **livelock**. To je situacija kada transakcije ne napreduju, ali za razliku od deadlocka one ne cekaju, vec non stop nesto rade ali bez pravog pomaka sa izvrsenjem. **Starvation** je situacija u konkurentnim sistemima kada neki proces ili transkacija nikada ne dobija pristup resursu koji mu je potreban iako je taj resurs dostupan i druge transakcije normalno rade. Javlja se zbog loseg rasporeda izvrsavanja procesa, kada jedni procesi stalno imaju prioritet u odnosu na ostale.

# Konkurentnost i performanse

Konkurentnost u bazama podataka oznacava mogucnost da vise transakcija istovremeno pristupa bazi, sto je vece ogranicenje odnosno veci stepen izolacije to je utrosak resursa veci.

Visok nivo izolacije(Serializable) moze da spreci sve oblike race condition-a i da garantuje maksimalnu konzistentnost podataka. Ipak, veca izolacija znaci vise cekanja i zakljucavanja transkacija sto moze povecati vreme izvrsavanja.

Nizi nivo izolacija(Read commited) omogucava brze izvrsavanje ali povecavaju rizik od anomalija.

U zavisnosti od tipa aplikacije odredjuje se da li je prioritet na konzistentnosti kao kod finansijskih transakcija (implementiraju se visi nivoi izolacije) ili je prioritet na performansama kao kod sistema sa mnogo citanja i retkim konfliktima (implementiraju se nizi nivoi izolacije). Pozeljno je izabrati najnizi nivo izolacije koji je prihvatljiv za aplikaciju. Dobra praksa je minimizirati kriticnu sekciju sto dovodi do manje zakljucanih redova.

Kljucni faktori konkurentnosti koji uticu na performanse su:

* Optimistic i pessimistic locking - Koristiti optimisticko zakljucavanje ako je moguce, a pesimisticko samo kada je neophodno, odnosno kada su konflikti cesti i kriticni.
* Locks - Odnosi se na vrste pesimistickog zakljucavnja shared i exslusive lock. Koristiti exclusive lock samo kada je neophodan, a kad god je moguce koristi shared lock, jer vise transakcija moze istovremeno da cita podatak sto povecava konkurentnost i smanjuje vreme izvrsavanja.