

Obrada transakcija, planovi izvršavanja transakcija, izolacija i zaključavanje kod PostgreSQL baze podataka

Studijski program: Računarstvo i informatika

Modul: Softversko inženjerstvo

Predmet: Sistemi za upravljanje bazama podataka

|  |  |
| --- | --- |
| Student: | Profesor: |
|  |  |
| Jovan Vukadinović,  broj indeksa 1654 | Prof. dr Aleksandar Stanimirović |

Niš, maj 2024. godina

Sadržaj

[Uvod 3](#_Toc167649604)

[Izolacija i konzistentnost 4](#_Toc167649605)

[Nivoi izolacije i moguće anomalije 5](#_Toc167649606)

[Izgubljena izmena (lost update) 6](#_Toc167649607)

[Dirty reads i read uncommitted 6](#_Toc167649608)

[Non-Repeatable reads i read commited 6](#_Toc167649609)

[Phantom reads i repeatable read 6](#_Toc167649610)

[No anomalies i serializable 7](#_Toc167649611)

[Zbog čega nastaju anomalije? 8](#_Toc167649612)

[Nivoi izolacije u PostgreSQL-u 8](#_Toc167649613)

[Read Committed 9](#_Toc167649614)

[Repeatable Read 16](#_Toc167649615)

[Serializable 21](#_Toc167649616)

[MVCC 23](#_Toc167649617)

[Zaključavanja 27](#_Toc167649618)

[Zaključavanja na nivou relacija 27](#_Toc167649619)

[Teška zaključavanja 28](#_Toc167649620)

[Zaključavanja na nivou redova 28](#_Toc167649621)

[Upravljanje konkurentnim pristupom 29](#_Toc167649622)

[Zaključavanja bez čekanja 29](#_Toc167649623)

[Sprečavanje zastoja 30](#_Toc167649624)

[Multitransakcije i zastoji 30](#_Toc167649625)

[Ostala zaključavanja 30](#_Toc167649626)

[Neobjektna zaključavanja 30](#_Toc167649627)

[Zaključavanja za proširenje relacija 31](#_Toc167649628)

[Zaključavanja na memorijskim strukturama 31](#_Toc167649629)

[Zaključak 31](#_Toc167649630)

[Literatura 32](#_Toc167649631)

# Uvod

Razumevanje osnovnih principa izolacije i zaključavanja je ključno za dizajn i implementaciju baza podataka koje mogu da izdrže visoke zahteve konkurentnog pristupa podacima. Osnovna svojstva koja se najčešće pominju su ACID svojstava (Atomicity, Consistency, Isolation, Durability) koji doprinose pouzdanosti sistema. Atomičnost obezbeđuje da se transakcije izvršavaju u celosti ili se uopšte ne izvršavaju, dok doslednost osigurava da svaka transakcija vodi bazu iz jednog validnog stanja u drugo. Izolacija sprečava da jedna transakcija utiče na druge istovremene transakcije, a trajnost garantuje da će rezultati uspešnih transakcija biti trajno zabeleženi čak i u slučaju kvara sistema.

Videćemo specifične protokole zaključavanja i izolacije koje koristi PostgreSQL. On koristi viševerzijsku kontrolu konkurencije (MVCC) koja omogućava istovremeni pristup različitim verzijama podataka, čime se smanjuje potreba za zaključavanjem i povećava efikasnost sistema. Ovaj pristup omogućava čitanje starih verzija podataka dok se nove transakcije izvršavaju, što značajno poboljšava performanse i skalabilnost baze podataka.

Potom i različiti nivoe izolacije koje PostgreSQL podržava uključuju Read Uncommitted, Read Committed, Repeatable Read i Serializable. Svaki od ovih nivoa izolacije pruža različite stepene zaštite od anomalija koje se mogu pojaviti tokom konkurentnog izvršavanja transakcija. Read Uncommitted omogućava čitanje podataka koji nisu još uvek potvrđeni, što može dovesti do prljavih čitanja. Read Committed obezbeđuje da se čitaju samo potvrđeni podaci, čime se eliminiše mogućnost prljavih čitanja, ali ne i neponovljivih čitanja. Repeatable Read dodatno eliminiše neponovljiva čitanja, dok Serializable pruža najviši nivo izolacije, garantujući da će transakcije biti izvršene kao da su sekvencijalno izvršavane, čime se eliminišu sve vrste anomalija.

A na kraju ćemo se upoznati i sa naprednim mehanizmom zaključavanja koji služi da bi upravljao konkurentnim pristupom deljenim resursima, osiguravajući doslednost i integritet podataka. Zaključavanja su ključna za sprečavanje konflikata koji nastaju kada više procesa pokušava da pristupi ili modifikuje isti resurs istovremeno. U PostgreSQL-u su neophodna za koordinaciju akcija između konkurentnih transakcija, a upravo oni sprečavaju anomalije kao što su "dirty reads", "non-repeatable reads" i "phantom reads".

# Izolacija i konzistentnost

Ključna karakteristika relacionih baza podataka je njihova sposobnost da obezbede doslednost podataka, odnosno, ispravnost podataka. Poznato je da na nivou baze podataka mogu biti postavljeni integritetni uslovi, kao što su NOT NULL ili UNIQUE. Sistem baze podataka se brine da ovi uslovi nikada ne budu narušeni, tako da integritet podataka nikada nije ugrožen.

Da su svi potrebni uslovi mogli biti formulisani na nivou baze podataka, doslednost bi bila zagarantovana. Međutim, neki uslovi su suviše kompleksni za to, na primer, oni se tiču nekoliko tabela istovremeno. Čak i ako se uslov može definisati u bazi podataka, ali iz nekog razloga nije, to ne znači da ovaj uslov ne može biti prekršen. Dakle, doslednost podataka je strožija od integriteta, ali sistem baze podataka ne zna šta zapravo znači "doslednost". Ako aplikacija naruši doslednost, a da ne naruši integritet, nema načina da sistem baze podataka to otkrije.

Stoga je aplikacija ta koja mora da postavi kriterijume za doslednost podataka, i moramo verovati da je napisana ispravno i da nikada neće imati greške. Ali ako aplikacija uvek izvršava samo ispravne nizove operacija, gde se onda uklapa sistem baze podataka? Pre svega, ispravan niz operacija može privremeno narušiti doslednost podataka, i koliko god to čudno zvučalo to je potpuno normalno. Jedan od često korišćenih, ali jasnih primera je prenos sredstava sa jednog računa na drugi.

Pravilo doslednosti može zvučati ovako: transfer novca nikada ne sme promeniti ukupni saldo pogođenih računa. Prilično je teško, iako moguće, formulisati ovaj uslov kao integritetni uslov u SQL-u, pa pretpostavimo da je definisan na nivou aplikacije i ostaje neprovidan za sistem baze podataka. Transfer se sastoji od dve operacije: prva povlači određeni iznos novca sa jednog od računa, dok druga dodaje ovaj iznos na drugi račun. Prva operacija narušava doslednost podataka, dok je druga vraća.

Ako prva operacija uspe, ali druga ne uspe (zbog nekog kvara), doslednost podataka će biti narušena. Takve situacije su neprihvatljive, ali je potrebno mnogo napora da se otkriju i reše na nivou aplikacije. Srećom, to nije potrebno, jer problem može potpuno rešiti sam sistem baze podataka ako zna da ove dve operacije čine nedeljivu celinu, odnosno transakciju.

Ali ovde takođe postoji i suptilniji aspekt. Iako su apsolutno ispravne same po sebi, transakcije mogu početi da se ponašaju neispravno kada se izvršavaju istovremeno. To je zato što se operacije koje pripadaju različitim transakcijama često mešaju. Ne bi bilo takvih problema da sistem baze podataka prvo završi sve operacije jedne transakcije, pa tek onda pređe na sledeću, ali izvršenje po ovom principu bi bilo veoma sporo. Stvarna istovremena izvršenja transakcija mogu se postići samo na sistemima sa odgovarajućim hardverom tojest višejezgarni procesor, disk array, itd. Ali isto razmišljanje važi i za server koji izvršava naredbe sekvencijalno u režimu deljenja vremena. Radi opšteg opisa, ove situacije se ponekad nazivaju konkurentnim izvršenjem. Ispravne transakcije koje se ponašaju neispravno kada se izvršavaju zajedno rezultuju konkretnim anomalijama ili fenomenima konkurencije.

Evo jednostavnog primera. Da bi dobila dosledne podatke iz baze podataka, aplikacija ne sme videti nikakve promene koje su napravljene od strane drugih transakcija koje nisu završene, kao osnovni zahtev. Inače (ako neke transakcije budu poništene), videla bi stanje baze podataka koje nikada nije postojalo. Takva anomalija se naziva prljavo čitanje (dirty reads). Postoji i mnogo drugih anomalija, koje su složenije.

Kada se transakcije izvršavaju istovremeno, baza podataka mora da garantuje da će rezultat takvog izvršenja biti isti kao i ishod jednog od mogućih sekvencijalnih izvršenja. Drugim rečima, mora izolovati transakcije jednu od druge, i tako se brinuti o svim mogućim anomalijama. Ukratko, transakcija je skup operacija koji bazu podataka vodi iz jednog ispravnog stanja u drugo ispravno stanje (doslednost), pod uslovom da je izvršena u potpunosti (atomičnost) i bez uticaja drugih transakcija (izolacija). Ova definicija kombinuje zahteve koji se implicitno podrazumevaju prvim tri slova akronima ACID. Oni su toliko isprepleteni da ima smisla razmatrati ih zajedno. Zapravo, zahtev za izdržljivošću je teško odvojiti: nakon pada sistema, sistem i dalje može sadržati neke promene napravljene od strane transakcija koje nisu završene, i morate nešto preduzeti kako biste vratili doslednost podataka.

Dakle, sistem baze podataka pomaže aplikaciji da održava doslednost podataka uzimajući u obzir granice transakcija, iako nema pojma o podrazumevanoj doslednosti. Nažalost, potpuna izolacija je teško sprovesti i može negativno uticati na performanse. Većina sistema u stvarnom životu koristi slabije nivoe izolacije, koji sprečavaju neke anomalije, ali ne i sve. To znači da deo odgovornosti za održavanje doslednosti podataka pada na aplikaciju. I to je upravo razlog zašto je veoma važno razumeti koji nivo izolacije se koristi u sistemu, šta je zagarantovano na ovom nivou, a šta ne, i kako osigurati da vaš kod bude ispravan u takvim uslovima.

# Nivoi izolacije i moguće anomalije

Standard SQL definiše četiri nivoa izolacije. Ovi nivoi su definisani listom anomalija koje mogu ili ne moraju da se pojave tokom istovremenog izvršavanja transakcija. Dakle, kada govorimo o nivoima izolacije, moramo početi sa anomalijama.

Trebalo bi imati na umu da je standard samo teorijski, on utiče na praksu, ali praksa i dalje odstupa od njega na mnogo načina. Baveći se transakcijama na bankovnim računima, ovi primeri su prilično jasni, ali treba razumeti da nemaju nikakve veze sa stvarnim bankarskim operacijama. Zanimljivo je da se stvarna teorija baza podataka takođe razlikuje od standarda, razvijena je nakon što je standard usvojen, a praksa je već bila daleko ispred.

## Izgubljena izmena (lost update)

Izgubljena izmena se javlja kada dve transakcije čitaju isti red tabele, zatim jedna od transakcija ažurira ovaj red, i na kraju druga transakcija ažurira isti red ne uzimajući u obzir bilo kakve promene koje je napravila prva transakcija.

Na primer, pretpostavimo da dve transakcije treba da povećaju saldo istog računa za $100. Prva transakcija čita trenutnu vrednost ($1000), zatim druga transakcija čita istu vrednost. Prva transakcija povećava saldo (čineći ga $1100) i upisuje novu vrednost u bazu podataka. Druga transakcija radi isto: dobija $1100 nakon povećanja salda i upisuje tu vrednost. Kao rezultat, klijent gubi $100. Izgubljene izmene su zabranjene standardom na svim nivoima izolacije.

## Dirty reads i read uncommitted

Prljava čitanja (Dirty read) se javljaju kada transakcija čita nepotvrđene promene (Uncommitted read) napravljene od strane druge transakcije.

Na primer, prva transakcija prenosi $100 na prazan račun ali ne potvrđuje tu promenu. Druga transakcija čita stanje računa (koje je ažurirano ali nije potvrđeno) i dozvoljava klijentu da podigne novac. Iako je prva transakcija prekinuta i njene promene su poništene, tako da je račun prazan. Standard dozvoljava prljava čitanja na nivou čitanja nepotvrđenih.

## Non-Repeatable reads i read commited

Neponovljena čitanja se javljaju kada transakcija čita isti red dva puta, dok druga transakcija ažurira (ili briše) taj red između ova dva čitanja i potvrđuje promenu. Kao rezultat, prva transakcija dobija različite rezultate.

Na primer, pretpostavimo da postoji pravilo doslednosti koje zabranjuje negativan saldo na bankovnim računima. Prva transakcija treba da smanji saldo računa za $100. Proverava trenutnu vrednost, dobija $1000, i odlučuje da je ova operacija moguća. U isto vreme, druga transakcija povlači sav novac sa ovog računa i potvrđuje promene. Ako bi prva transakcija ponovo proverila saldo u ovom trenutku, dobila bi $0 (ali je odluka o povlačenju novca već doneta, i ova operacija izaziva prekoračenje). Standard dozvoljava neponovljena čitanja na nivou čitanja nepotvrđenih i čitanja potvrđenih.

## Phantom reads i repeatable read

Lažna čitanja se javljaju kada ista transakcija izvrši dva identična upita koji vraćaju skup redova koji zadovoljavaju određeni uslov, dok druga transakcija dodaje neke druge redove koji zadovoljavaju ovaj uslov i potvrđuje promene u vremenskom intervalu između ova dva upita. Kao rezultat, prva transakcija dobija dva različita skupa redova.

Na primer, pretpostavimo da postoji pravilo doslednosti koje zabranjuje klijentu da ima više od tri računa. Prva transakcija treba da otvori novi račun, pa proverava koliko računa trenutno ima (recimo da ih ima dva) i odlučuje da je ova operacija moguća. U ovom trenutku, druga transakcija takođe otvara novi račun za ovog klijenta i potvrđuje promene. Ako bi prva transakcija ponovo proverila broj otvorenih računa, dobila bi tri (ali već otvara drugi račun, i klijent na kraju ima četiri). Standard dozvoljava lažne podatke na nivou čitanja nepotvrđenih, čitanja potvrđenih i ponovljivog čitanja.

## No anomalies i serializable

Nivo Serializable, definisan standardom, ne dozvoljava nikakve anomalije. To nije isto kao zabrana izgubljenih izmena i prljavih, neponovljivih i lažnih čitanja. Zapravo, postoji mnogo veći broj poznatih anomalija od onih koje standard navodi, i veliki broj onih koje nisu poznate.

Nivo Serializable mora sprečiti sve anomalije. To znači da programer aplikacije ne mora da uzima u obzir izolaciju. Ako transakcije izvrše ispravne nizove operacija kada se izvršavaju samostalno, istovremeno izvršavanje ne može narušiti doslednost podataka.

Da bi ovo bilo ilustrovano, koristiće se poznatu tabelu koja je dostupna u standardu, a poslednja kolona je dodata radi jasnoće:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Lost update | Dirty read | Non-repeatable read | Phantom read | Other anomalies |
| Read Uncommited | - | Da | Da | Da | Da |
| Read Commited | - | - | Da | Da | Da |
| Repeatable Read | - | - | - | Da | Da |
| Serializable | - | - | - | - | - |

## Zbog čega nastaju anomalije?

Od svih mogućih anomalija, standard pominje samo ove, i niko ne zna siguran razlog. Nije neobično da druge anomalije jednostavno nisu uzete u obzir kada su prve verzije standarda usvojene, jer je teorija bila daleko iza prakse u to vreme.

Pored toga, pretpostavljeno je da izolacija mora biti zasnovana na “Locks”. Široko korišćeni protokol dvofaznog zaključavanja (2PL) zahteva od transakcija da zaključaju pogođene redove tokom izvršavanja i da oslobode lock-ove po završetku. Jednostavno rečeno, što više lock-ova transakcija stekne, to je bolje izolovana od drugih transakcija. I zbog toga su lošije performanse sistema, jer transakcije počinju da čekaju da dobiju pristup istim redovima umesto da se izvršavaju istovremeno. Razlika između standardnih nivoa izolacije u velikoj meri definisana je brojem lock-ova potrebnih za njihovu implementaciju.

Ako su redovi za ažuriranje zaključani samo za pisanje, ali ne i za čitanje, dobijamo nivo izolacije čitanja nepotvrđenih, koji dozvoljava čitanje podataka pre nego što se potvrde.

Ako su redovi za ažuriranje zaključani i za čitanje i za pisanje, dobijamo nivo izolacije čitanja potvrđenih, zabranjeno je čitanje nepotvrđenih podataka, ali upit može vratiti različite vrednosti ako se izvrši više puta (neponovljena čitanja). Zaključavanje redova za čitanje i ažuriranje za sve operacije daje nam nivo izolacije Ponovljivog čitanja: ponovljeni upit će vratiti isti rezultat. Međutim, nivo Serializable postavlja problem, nemoguće je zaključati red koji još ne postoji. To ostavlja mogućnost za lažna čitanja da se dese, transakcija može dodati red koji zadovoljava uslov prethodnog upita, i ovaj red će se pojaviti u sledećem rezultatu upita.

Dakle, redovni lock-ovi ne mogu obezbediti potpunu izolaciju: da bismo to postigli, moramo zaključati uslove (predicate) umesto redova. Takvi predicate lock-ovi su uvedeni još 1976. godine kada je razvijan System R, međutim, njihova praktična primenljivost je ograničena na jednostavne uslove za koje je jasno da li dva različita predikata mogu da se sukobe.

# Nivoi izolacije u PostgreSQL-u

Vremenom, protokoli zasnovani na lock-ovima za upravljanje transakcijama su zamenjeni protokolom izolacije snapshot-a (S1). Ideja iza ovog pristupa je da svaka transakcija pristupa doslednom snapshot-u podataka onako kako je izgledala u određenom trenutku. Snapshot uključuje sve trenutne promene koje su potvrđene pre nego što je snapshot napravljen.

Snapshot izolacije minimizira broj potrebnih lock-ova. Zapravo, red će biti zaključan samo prilikom pokušaja istovremenog ažuriranja. U svim drugim slučajevima, operacije mogu biti izvršene istovremeno, pisanja nikada ne zaključavaju čitanja, a čitanja nikada ne zaključavaju ništa. PostgreSQL koristi viševerzijski oblik S1 protokola. Viševerzijska kontrola konkurencije podrazumeva da baza podataka može sadržati nekoliko verzija istog reda u svakom trenutku, tako da PostgreSQL može uključiti odgovarajuću verziju u snapshot umesto da poništi transakcije koje pokušavaju da čitaju zastarele podatke.

Na osnovu snapshot-a, izolacija PostgreSQL-a se razlikuje od zahteva specificiranih u standardu, zapravo, ona je čak i strožija. Prljava čitanja su zabranjena po dizajnu. Tehnički, možete specificirati nivo čitanja nepotvrđenih, ali će se njegovo ponašanje podudarati sa nivoom čitanja potvrđenih. Ponovljivo čitanje ne dozvoljava ni neponovljiva ni lažna čitanja (iako ne garantuje potpunu izolaciju). Ali u nekim slučajevima, postoji rizik od gubitka promena na nivou čitanja potvrđenih.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Lost update | Dirty read | Non-repeatable read | Phantom read | Other anomalies |
| Read Commited | Da | - | Da | Da | Da |
| Repeatable Read | - | - | - | - | Da |
| Serializable | - | - | - | - | - |

Pre nego što istražimo unutrašnje mehanizme izolacije, razmotrimo svaki od tri nivoa izolacije sa stanovišta korisnika.

U tu svrhu, kreiraćemo tabelu računa, Alisa i Bob će imati po $1,000, ali Bob će imati dva računa:

=> **CREATE TABLE** accounts(

id integer **PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY**,

client text,

amount numeric

);

=> **INSERT INTO** accounts **VALUES**

(1, 'alice', 1000.00), (2, 'bob', 100.00), (3, 'bob', 900.00);

## Read Committed

Nema Dirty reads. Lako je proveriti da čitanje prljavih podataka nije dozvoljeno. Hajde da započnemo transakciju. Podrazumevano, koristi se nivo izolacije Read Commited:

=> **BEGIN**;

=> **SHOW** transaction\_isolation;

transaction\_isolation

−−−−−−−−−−−−−−−−−−−−−−−

read committed

(1 row)

Da budemo precizniji, podrazumevani nivo je podešen pomoću sledećeg parametra, koji se može promeniti po potrebi:

=> **SHOW** default\_transaction\_isolation;

default\_transaction\_isolation

−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

read committed

(1 row)

Otvorena transakcija povlači neka sredstva sa računa klijenta, ali još uvek ne potvrđuje ove promene. Ipak, ona će videti svoje promene, jer joj je to uvek dozvoljeno:

=> **UPDATE** accounts **SET** amount = amount - 200 **WHERE** id = 1;

=> **SELECT** \* **FROM** accounts **WHERE** client = 'alice';

Id | client | amount

---−−−−+−−−−−−−−+−−−−−−−−

1 | alice | 800.00

(1 row)

U drugoj sesiji, započinjemo još jednu transakciju koja će takođe raditi na nivou čitanje potvrđenih:

=> **BEGIN**;

=> **SELECT** \* **FROM** accounts **WHERE** client = 'alice';

Id | client | amount

−−−−+−−−−−−−−+−−−−−−−−−

1 | alice | 1000.00

(1 row)

Kao što se moglo očekivati, druga transakcija ne vidi bilo kakve nepotvrđene promene, prljava čitanja su zabranjena.

**Non-repeatable reads**. Sada dozvolimo prvoj transakciji da potvrdi promene. Zatim će druga transakcija ponoviti isti upit:

=> **COMMIT**;

=> **SELECT** \* **FROM** accounts **WHERE** client = 'alice';

id | client | amount

−−−−+−−−−−−−−+−−−−−−−−

1 | alice | 800.00

(1 row)

=> **COMMIT**;

Upit dobija ažuriranu verziju podataka, i to je upravo ono što se podrazumeva pod neponovljivim čitanjem, što je dozvoljeno na nivou čitanje potvrđenih.

U transakciji ne smete donositi odluke na osnovu podataka koje je pročitao prethodni operator, jer se sve može promeniti između. Evo primera čije varijacije se često pojavljuju u kodu aplikacije toliko često da se može smatrati klasičnim anti-pattern-om:

**IF** (**SELECT** amount **FROM** accounts **WHERE** id = 1) >= 1000 **THEN**

**UPDATE** accounts **SET** amount = amount - 1000 **WHERE** id = 1;

**END** **IF**;

Tokom vremena koje prođe između provere i ažuriranja, druge transakcije mogu slobodno menjati stanje računa, pa je takva "provera" potpuno beskorisna. Za bolje razumevanje, možete zamisliti da su nasumični operatori drugih transakcija ubačeni između operatora trenutne transakcije. Na primer ovako:

**IF** (**SELECT** amount **FROM** accounts **WHERE** id = 1) >= 1000 **THEN**

**UPDATE** accounts **SET** amount = amount - 200 **WHERE** id = 1;

**COMMIT**;

**UPDATE** accounts **SET** amount = amount - 1000 **WHERE** id = 1;

**END** **IF**;

Ako sve krene naopako čim se operatori preurede, onda je kod neispravan. Verovatno će se desiti situacija kada ćete se naći u ovakve probleme, “sve što može poći naopako, poći će naopako”. Takve greške su veoma teške za reprodukciju, a samim tim je i njihovo popravljanje pravi izazov.

Postoji nekoliko opcija kako se ovaj kod može ispraviti:

* Zamena proceduralnog koda deklarativnim.

Na primer, u ovom slučaju lako je pretvoriti IF statement u CHECK ograničenje:

**ALTER** **TABLE** accounts

**ADD CHECK** amount >= 0;

Sada vam nije potrebna nikakva provera u kodu, dovoljno je samo pokrenuti komandu i obraditi izuzetak koji će biti podignut ako se pokuša kršenje ograničenja integriteta.

* Korišćenje jednog SQL operatora.

Doslednost podataka može biti ugrožena ako se transakcija potvrdi u vremenskom razmaku između operatora druge transakcije, čime se menja vidljivost podataka. Ako postoji samo jedan operator, nema takvih rupa.

PostgreSQL ima dovoljno mogućnosti da reši kompleksne zadatke jednom SQL naredbom. Konkretno, nudi zajedničke tabelarne izraze (CTE) koji mogu sadržati operatore poput INSERT, UPDATE, DELETE, kao i operator INSERT ON CONFLICT koji implementira sledeću logiku, ubaci red ako ne postoji, inače izvrši ažuriranje.

* Primena eksplicitnog zaključavanja.

Poslednji način je ručno postavljanje ekskluzivnog zaključavanja na sve potrebne redove (SELECT FOR UPDATE) ili čak na celu tabelu (LOCK TABLE). Ovaj pristup uvek funkcioniše, ali poništava sve prednosti MVCC-a, neke operacije koje bi mogle biti izvršene istovremeno, izvršiće se sekvencijalno.

**Read skew**. Međutim, nije sve tako jednostavno. PostgreSQL implementacija omogućava druge, manje poznate anomalije koje nisu regulisane standardom.

Pretpostavimo da je prva transakcija započela transfer novca između Bobovih računa:

=> **BEGIN**;

=> **UPDATE** accounts **SET** amount = amount - 100 WHERE id = 2;

U međuvremenu, druga transakcija počinje da prolazi kroz sve Bobove račune kako bi izračunala njihov ukupni saldo. Počinje sa prvim računom (videći naravno njegovo prethodno stanje):

=> **BEGIN**;

=> **SELECT** amount **FROM** accounts **WHERE** id = 2;

amount

−−−−−−−−

100.00

(1 row)

U ovom trenutku, prva transakcija uspešno se završava:

=> **UPDATE** accounts **SET** amount = amount + 100 WHERE id = 3;

=> **COMMIT**;

Druga transakcija čita stanje drugog računa (i vidi već ažuriranu vrednost):

=> **SELECT** amount **FROM** accounts **WHERE** id = 3;

amount

−−−−−−−−−

1000.00

(1 row)

=> **COMMIT**;

Kao rezultat, druga transakcija dobija $1,100 jer je pročitala netačne podatke. Takva anomalija se naziva "read skew" (izobličenje čitanja).

Ova anomaliju na nivou čitanja potvrđenih može se izbeći na primer ovako:

**SELECT** sum(amount) **FROM** accounts **WHERE** client = 'bob';

Do sada je tvrđeno da se vidljivost podataka može promeniti samo između operatora, međutim da li je to stvarno tako. Šta ako upit traje dugo? Može li videti različite delove podataka u različitim stanjima u tom slučaju?

Dobar način da to proverimo je da dodamo kašnjenje operatoru pozivom funkcije pg\_sleep. Tada će prvi red biti pročitan odmah, ali će drugi red morati da sačeka dve sekunde:

=> **SELECT** amount, pg\_sleep(2) -- two seconds

**FROM** accounts **WHERE** client = 'bob';

Dok se ovaj kod izvršava, započećemo još jednu transakciju da vratimo novac:

=> **BEGIN**;

=> **UPDATE** accounts **SET** amount = amount + 100 **WHERE** id = 2;

=> **UPDATE** accounts **SET** amount = amount - 100 **WHERE** id = 3;

=> **COMMIT**;

Rezultat pokazuje da je operator video sve podatke u stanju koje odgovara početku njegovog izvršenja, što je svakako ispravno:

Amount | pg\_sleep

−−−−−−−−+−−−−−−−−−−

0.00 |

1000.00 |

(2 rows)

Ali ni ovo nije tako jednostavno. Ako upit sadrži funkciju koja je deklarisana kao VOLATILE, a ta funkcija izvršava drugi upit, tada podaci koje vidi ovaj ugnježdeni upit neće biti dosledni sa rezultatom glavnog upita.

Proverimo saldo na Bobovim računima koristeći sledeću funkciju:

=> **CREATE FUNCTION** get\_amount(id integer) **RETURNS** numeric **AS** $$

**SELECT** amount **FROM** accounts a **WHERE** a.id = get\_amount.id;

$$ **VOLATILE** **LANGUAGE** sql;

=> **SELECT** get\_amount(id), pg\_sleep(2)

**FROM** accounts **WHERE** client = 'bob';

Ponovo ćemo preneti novac između računa dok se naš odloženi upit izvršava:

=> **BEGIN**;

=> **UPDATE** accounts **SET** amount = amount + 100 **WHERE** id = 2;

=> **UPDATE** accounts **SET** amount = amount - 100 **WHERE** id = 3;

=> **COMMIT**;

U ovom slučaju, dobićemo nedosledne podatke, $100 je izgubljeno:

get\_amount | pg\_sleep

−−−−−−−−−−−−+−−−−−−−−−−

100.00 |

800.00 |

(2 rows)

Mora se naglasiti da je ovaj efekat moguć samo na nivou izolacije čitanja potvrđenih, i samo ako je funkcija VOLATILE. Problem je u tome što PostgreSQL koristi upravo ovaj nivo izolacije i ovu kategoriju volatilnosti podrazumevano. Možemo primetiti da je ova zamka veoma nepredvidiva.

**Read skew** umesto **lost updates**. Anomalija čitanja odstupanja (read skew anomaly) može se pojaviti i unutar jednog operatora tokom ažuriranja, iako na pomalo neočekivan način. Pogledajmo šta se dešava ako dve transakcije pokušaju da modifikuju isti red.

Bob trenutno ima ukupno $1,000 na dva računa:

=> **SELECT** \* **FROM** accounts **WHERE** client = 'bob';

id | client | amount

−−−−+−−−−−−−−+−−−−−−−−

2 | bob | 200.00

3 | bob | 800.00

(2 rows)

Pokrenimo transakciju koja će smanjiti Bobovo stanje:

=> **BEGIN**;

=> **UPDATE** accounts **SET** amount = amount - 100 **WHERE** id = 3;

Istovremeno, druga transakcija će izračunati kamatu za sve račune klijenata sa ukupnim stanjem od $1,000 ili više:

=> **UPDATE** accounts **SET** amount = amount \* 1.01

**WHERE** client IN (

**SELECT** client

**FROM** accounts

**GROUP** **BY** client

**HAVING** sum(amount) >= 1000

);

Izvršenje operatora UPDATE praktično se sastoji od dve faze. Prvo, redovi koji treba da se ažuriraju se biraju na osnovu navedenog uslova. Pošto prva transakcija još nije potvrđena, druga transakcija ne može videti njen rezultat, tako da izbor redova za obračun kamata nije pogođen. Tako, Bobovi računi zadovoljavaju uslov, i njegovo stanje treba da se poveća za $10 nakon što se operacija UPDATE završi.

U drugoj fazi, odabrani redovi se ažuriraju jedan po jedan. Druga transakcija mora da čeka jer je red sa id = 3 zaključan, jer ga prva transakcija ažurira.

U međuvremenu, prva transakcija potvrđuje svoje izmene:

=> **COMMIT**;

=> **SELECT** \* **FROM** accounts **WHERE** client = 'bob';

id | client |amount

−−−−+−−−−−−−−+−−−−−−−−−−

2 | bob | 202.0000

3 | bob | 707.0000

(2 rows)

S jedne strane, komanda UPDATE ne bi smela videti nikakve promene koje je napravila prva transakcija. S druge strane, ne sme izgubiti nikakve potvrđene promene. Kada se zaključavanje oslobodi, operator UPDATE ponovo čita red koji treba ažurirati (ali samo taj red). Kao rezultat, Bob dobija 9 dolara kamata, na osnovu ukupnog iznosa od 900 dolara. Ali ako je imao 900 dolara, njegovi računi ne bi trebali biti uključeni u rezultate upita uopšte.

Dakle, naša transakcija je vratila netačne podatke, odnosno različiti redovi su pročitani iz različitih snapshot-ova. Umesto izgubljenog ažuriranja, ponovo posmatramo anomaliju čitanja odstupanja (read skew anomaly).

**Lost updates.** Međutim, trik ponovnog čitanja zaključanog reda neće pomoći protiv izgubljenih ažuriranja ako se podaci modifikuju različitim SQL operatorima.

Evo primera koji smo već videli. Aplikacija čita i beleži (izvan baze podataka) trenutno stanje Alice-inog računa:

=> **BEGIN**;

=> **SELECT** amount **FROM** accounts **WHERE** id = 1;

amount

−−−−−−−−

800.00

(1 row)

U međuvremenu, druga transakcija radi isto:

=> **BEGIN**;

=> **SELECT** amount **FROM** accounts **WHERE** id = 1;

amount

−−−−−−−−

800.00

(1 row)

Prva transakcija povećava prethodno zabeleženu vrednost za $100 i potvrđuje ovu izmenu:

=> **UPDATE** accounts **SET** amount = 800.00 + 100 **WHERE** id = 1

**RETURNING** amount;

amount

−−−−−−−−

900.00

(1 row)

**UPDATE** 1

=> **COMMIT**;

Druga transakcija radi isto:

=> **UPDATE** accounts **SET** amount = 800.00 + 100 **WHERE** id = 1

**RETURNING** amount;

amount

−−−−−−−−

900.00

(1 row)

**UPDATE** 1

=> **COMMIT**;

Nažalost, Alice je izgubila $100. Sistem baze podataka ne zna da je zabeležena vrednost od $800 nekako povezana sa accounts.amount, tako da ne može sprečiti anomaliju izgubljenog ažuriranja. Na nivou izolacije Read Committed, ovaj kod je netačan.

## Repeatable Read

Bez **non-repeatable** i bez **phantom reads**. Kao što naziv sugeriše, nivo izolacije Repeatable Read mora garantovati ponovljivo čitanje. Proverimo to i uverimo se da lažna čitanja ne mogu da se dogode. U tu svrhu, započećemo transakciju koja će vratiti Bobove račune u prethodno stanje i kreirati novi račun za Čarlija:

=> **BEGIN**;

=> **UPDATE** accounts **SET** amount = 200.00 **WHERE** id = 2;

=> **UPDATE** accounts **SET** amount = 800.00 **WHERE** id = 3;

=> **INSERT** **INTO** accounts **VALUES**

(4, 'charlie', 100.00);

=> **SELECT** \* **FROM** accounts **ORDER BY** id;

id | client | amount

−−−−+−−−−−−−−−+−−−−−−−−

1 | alice | 900.00

2 | bob | 200.00

3 | bob | 800.00

4 | charlie | 100.00

(4 rows)

U drugoj sesiji, započećemo još jednu transakciju, sa eksplicitno navedenim nivoom Repeatable Read u komandi BEGIN (nivo prve transakcije nije bitan):

=> **BEGIN** **ISOLATION** **LEVEL** **REPEATABLE READ**;

=> **SELECT** \* **FROM** accounts **ORDER BY** id;

id | client | amount

−−−−+−−−−−−−−+−−−−−−−−−−

1 | alice | 900.00

2 | bob | 202.0000

3 | bob | 707.0000

(3 rows)

Sada prva transakcija potvrđuje svoje promene, a druga transakcija ponavlja isti upit:

=> **COMMIT**;

=> **SELECT** \* **FROM** accounts **ORDER BY** id;

id | client | amount

−−−−+−−−−−−−−+−−−−−−−−−−

1 | alice | 900.00

2 | bob | 202.0000

3 | bob | 707.0000

(3 rows)

=> **COMMIT**;

Druga transakcija i dalje vidi iste podatke kao pre, ni novi redovi, ni ažuriranja redova nisu vidljivi. Na ovom nivou izolacije, ne morate brinuti da će se nešto promeniti između operacija.

**Serialization failures** umesto **lost updates**. Kao što smo već videli, ako dve transakcije ažuriraju isti red na nivou Read Committed, to može izazvati read skew anomaliju, čekajuća transakcija mora ponovo da pročita zaključani red, pa vidi stanje tog reda u različitim vremenskim tačkama u odnosu na ostale redove. Takva anomalija nije dozvoljena na nivou izolacije Repeatable Read, i ako se dogodi, transakcija može biti prekinuta samo sa greškom serijalizacije. Proverimo to ponavljanjem scenarija sa obračunom kamate:

=> **SELECT** \* **FROM** accounts **WHERE** client = 'bob';

id | client | amount

−−−−+−−−−−−−−+−−−−−−−−

2 | bob | 200.00

3 | bob | 800.00

(2 rows)

=> **BEGIN**;

=> **UPDATE** accounts **SET** amount = amount - 100.00 **WHERE** id = 3;

=> **BEGIN ISOLATION LEVEL REPEATABLE READ**;

=> **UPDATE** accounts **SET** amount = amount \* 1.01

**WHERE** client IN (

**SELECT** client

**FROM** accounts

**GROUP** **BY** client

**HAVING** sum(amount) >= 1000

);

=> **COMMIT**;

**ERROR**: Could not serialize access due to concurrent update

=> **ROLLBACK**;

Podaci ostaju konzistentni:

=> **SELECT** \* **FROM** accounts **WHERE** client = 'bob';

id | client | amount

−−−−+−−−−−−−−+−−−−−−−−

2 | bob | 200.00

3 | bob | 700.00

(2 rows)

Ista greška će se pojaviti kod bilo kojih paralelnih ažuriranja redova, čak i ako utiču na različite kolone.

Takođe ćemo dobiti ovu grešku ako pokušamo da ažuriramo saldo na osnovu prethodno uskladištene vrednosti:

=> **BEGIN ISOLATION LEVEL REPEATABLE READ**;

=> **SELECT** amount **FROM** accounts **WHERE** id = 1;

amount

−−−−−−−−

900.00

(1 row)

=> **BEGIN ISOLATION LEVEL REPEATABLE READ**;

=> **SELECT** amount **FROM** accounts **WHERE** id = 1;

amount

−−−−−−−−

900.00

(1 row)

=> **UPDATE** accounts **SET** amount = 900.00 + 100.00 **WHERE** id = 1

**RETURNING** amount;

amount

−−−−−−−−−

1000.00

(1 row)

**UPDATE** 1

=> **COMMIT**;

=> **UPDATE** accounts **SET** amount = 900.00 + 100.00 **WHERE** id = 1

**RETURNING** amount;

**ERROR**: Could not serialize access due to concurrent update

=> **ROLLBACK**;

Ukoliko vaša aplikacija koristi nivo izolacije Repeatable Read za transakcije sa upisivanjem, mora biti spremna da ponovo pokuša transakcije koje su završene sa greškom serijalizacije. Za transakcije koje su samo za čitanje, takav ishod je nemoguć.

**Write skew**. Kao što smo videli, PostgreSQL implementacija nivoa izolacije Repeatable Read sprečava sve anomalije opisane u standardu. Ali ne sve moguće anomalije, niko ne zna koliko ih zapravo postoji. Međutim, jedan važan podatak je dokazan, snapshot isolation ne sprečava samo dve anomalije, bez obzira na to koliko drugih anomalija postoji.

Prva je write skew.

Definišimo sledeće pravilo konzistentnosti: dozvoljeno je imati negativan saldo na nekim od računa korisnika sve dok je ukupan saldo nenegativan.

Prva transakcija dobija ukupan saldo Bobovih računa:

=> **BEGIN ISOLATION LEVEL REPEATABLE READ**;

=> **SELECT** sum(amount) **FROM** accounts **WHERE** client = 'bob';

sum

−−−−−−−−

900.00

(1 row)

Druga transakcija dobija isti zbir:

=> **BEGIN ISOLATION LEVEL REPEATABLE READ**;

=> **SELECT** sum(amount) **FROM** accounts **WHERE** client = 'bob';

sum

−−−−−−−−

900.00

(1 row)

Prva transakcija sa pravom pretpostavlja da može da zaduži jedan od računa za $600:

=> **UPDATE** accounts **SET** amount = amount - 600.00 **WHERE** id = 2;

Druga transakcija dolazi do istog zaključka i zadužuje drugi račun:

=> **UPDATE** accounts **SET** amount = amount - 600.00 **WHERE** id = 3;

=> **COMMIT**;

=> **COMMIT**;

=> **SELECT** \* **FROM** accounts **WHERE** client = 'bob';

id | client | amount

−−−−+−−−−−−−−+−−−−−−−−−

2 | bob | −400.00

3 | bob | 100.00

(2 rows)

Bobov ukupni saldo je sada negativan, iako bi obe transakcije bile ispravne da su izvršene zasebno.

**Read-only transaction anomaly.** Anomalija transakcije samo za čitanje je druga i poslednja dozvoljena na nivou izolacije Repeatable Read. Da bismo posmatrali ovu anomaliju, moramo pokrenuti tri transakcije: dve će ažurirati podatke, dok će treća biti samo za čitanje.

Ali prvo, vratimo Bobov saldo:

=> **UPDATE** accounts **SET** amount = 900.00 **WHERE** id = 2;

=> **SELECT** \* **FROM** accounts **WHERE** client = 'bob';

id | client | amount

−−−−+−−−−−−−−+−−−−−−−−

3 | bob | 100.00

2 | bob | 900.00

(2 rows)

Prva transakcija obračunava kamatu na Bobov ukupni saldo i dodaje ovu sumu na jedan od njegovih računa:

=> **BEGIN ISOLATION LEVEL REPEATABLE READ**;

=> **UPDATE** accounts

**SET** amount =

amount + (**SELECT** sum(amount) **FROM** accounts **WHERE** client = 'bob') \* 0.01

**WHERE** id = 2;

Zatim druga transakcija povlači nešto novca sa Bobovog drugog računa i potvrđuje ovu promenu:

=> **BEGIN ISOLATION LEVEL REPEATABLE READ**;

=> **UPDATE** accounts **SET** amount = amount - 100.00 **WHERE** id = 3;

=> **COMMIT**;

Ako se prva transakcija potvrdi u ovom trenutku, neće biti anomalija: mogli bismo pretpostaviti da je prva transakcija potvrđena pre druge (ali ne i obrnuto—prva transakcija je videla stanje računa sa id = 3 pre nego što je druga transakcija napravila bilo kakva ažuriranja).

Ali zamislimo da u ovom trenutku pokrenemo transakciju samo za čitanje da upitamo račun koji nije pogođen prvim dvema transakcijama:

=> **BEGIN ISOLATION LEVEL REPEATABLE READ**;

=> **SELECT** \* **FROM** accounts **WHERE** client = 'alice';

id | client | amount

−−−−+−−−−−−−−+−−−−−−−−−

1 | alice | 1000.00

(1 row)

I tek sada će prva transakcija biti potvrđena:

=> **COMMIT**;

Koje stanje treba da vidi treća transakcija u ovom trenutku? Počevši, mogla je videti promene koje je napravila druga transakcija (koja je već bila potvrđena), ali ne i prva (koja još nije bila potvrđena). Ali, kao što smo već ustanovili, druga transakcija treba da se tretira kao da je započeta nakon prve. Svako stanje koje vidi treća transakcija biće nekonzistentno, ovo je tačno ono što znači read-only anomalija transakcije:

=> **SELECT** \* **FROM** accounts **WHERE** client = 'bob';

Id | client | amount

−−−−+−−−−−−−−+−−−−−−−−

2 | bob | 900.00

3 | bob | 0.00

(2 rows)

=> **COMMIT**;

## Serializable

Nivo izolacije Serializable sprečava sve moguće anomalije. Ovaj nivo je praktično izgrađen na vrhu snapshot isolation. Anomalije koje se ne pojavljuju na nivou izolacije Repeatable Read (kao što su dirty, non-repeatable ili phantom reads) ne mogu se pojaviti ni na nivou Serializable. A one dve anomalije koje se pojavljuju (write skew i read-only transaction anomalies) otkrivaju se na poseban način kako bi prekinule transakciju, uzrokujući već poznatu grešku serijalizacije.  
  
**No anomalies**. Uverimo se da će naš scenario sa write skew na kraju završiti sa greškom serijalizacije:

=> **BEGIN ISOLATION LEVEL SERIALIZABLE**;

=> **SELECT** sum(amount) **FROM** accounts **WHERE** client = 'bob';

sum

−−−−−−−−−−

910.0000

(1 row)

=> **BEGIN ISOLATION LEVEL SERIALIZABLE**;

=> **SELECT** sum(amount) **FROM** accounts **WHERE** client = 'bob';

sum

−−−−−−−−−−

910.0000

(1 row)

=> **UPDATE** accounts **SET** amount = amount - 600.00 **WHERE** id = 2;

=> **UPDATE** accounts **SET** amount = amount - 600.00 **WHERE** id = 3;

=> **COMMIT**;

**COMMIT**

=> **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.

Scenarij sa anomalijom transakcije samo za čitanje dovodi do iste greške.

**Odlaganje read-only transakcije**. Da bi se izbegle situacije kada transakcija samo za čitanje može uzrokovati anomaliju koja ugrožava konzistentnost podataka, PostgreSQL nudi interesantno rešenje, ova transakcija može biti odložena dok njeno izvršenje ne postane sigurno. To je jedini slučaj kada SELECT upit može biti blokiran zbog ažuriranja redova.

Proverićemo ovo ponavljanjem scenarija koji je demonstrirao anomaliju transakcije samo za čitanje:

=> **UPDATE** accounts **SET** amount = 900.00 **WHERE** id = 2;

=> **UPDATE** accounts **SET** amount = 100.00 **WHERE** id = 3;

=> **SELECT** \* **FROM** accounts **WHERE** client = 'bob' **ORDER BY** id;

id | client | amount

−−−−+−−−−−−−−+−−−−−−−−

2 | bob | 900.00

3 | bob | 100.00

(2 rows)

=> **BEGIN ISOLATION LEVEL SERIALIZABLE**;

=> **UPDATE** accounts **SET** amount = amount + ( **SELECT** sum(amount) **FROM** accounts **WHERE** client = 'bob' ) \* 0.01

**WHERE** id = 2;

=> **BEGIN ISOLATION LEVEL SERIALIZABLE**;

=> **UPDATE** accounts **SET** amount = amount - 100.00 **WHERE** id = 3;

=> **COMMIT**;

Izričito ćemo deklarisati treću transakciju kao READ ONLY i DEFERRABLE:

=> **BEGIN** **ISOLATION LEVEL SERIALIZABLE READ ONLY DEFERRABLE**;

=> **SELECT** \* **FROM** accounts **WHERE** client = 'alice';

Pokušaj izvršavanja upita blokira transakciju, inače bi uzrokovala anomaliju.

I tek kada se prva transakcija potvrdi, treća može nastaviti svoje izvršenje:

=> **COMMIT**;

id | client | amount

−−−−+−−−−−−−−+−−−−−−−−−

1 | alice | 1000.00

(1 row)

=> **SELECT** \* **FROM** accounts **WHERE** client = 'bob';

id | client | amount

−−−−+−−−−−−−−+−−−−−−−−−−

2 | bob | 910.0000

3 | bob | 0.00

(2 rows)

=> **COMMIT**;

Dakle, ako aplikacija koristi nivo izolacije Serializable, mora biti spremna da ponovo pokuša transakcije koje su završile sa greškom serijalizacije. Nivo Repeatable Read zahteva isti pristup osim ako je aplikacija ograničena na transakcije samo za čitanje.

Nivo izolacije Serializable donosi lakše programiranje, ali cena za to je dodatno opterećenje uzrokovano otkrivanjem anomalija i prisilnim prekidom određenog broja transakcija. Možete smanjiti ovaj uticaj eksplicitnim korišćenjem klauzule READ ONLY prilikom deklarisanja transakcija samo za čitanje. Ali glavno pitanje je, koliki je deo prekinutih transakcija, pošto će ove transakcije morati ponovo da se izvršavaju. Bilo bi dobro daPostgreSQL prekida samo one transakcije koje rezultiraju sukobima podataka i koje su zaista nekompatibilne. Ali takav pristup bi neizbežno bio previše resursno intenzivan, jer bi uključivao praćenje operacija na svakom redu.

Trenutna implementacija dozvoljava lažno pozitivne rezultate. PostgreSQL može prekinuti neke apsolutno sigurne transakcije koje jednostavno nemaju sreće. Njihova "sreća" zavisi od mnogih faktora, kao što su prisustvo odgovarajućih indeksa ili količina dostupnog RAM-a, tako da je stvarno ponašanje teško predvideti unapred.

Ako se koristi nivo Serializable, moraju se poštovati sve transakcije aplikacije. Kada se kombinuje sa drugim nivoima, Serializable se ponaša kao Repeatable Read bez ikakvog obaveštenja. Dakle, ako odluka bude da se koristite nivo Serializable, ima smisla promeniti vrednost parametra default\_transaction\_isolation u skladu s tim, iako neko još uvek može eksplicitno postaviti drugačiji nivo.

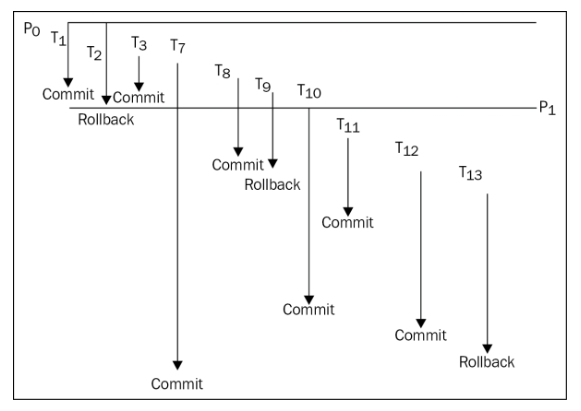
Postoje i druga ograničenja, na primer, upiti koji se izvršavaju na nivou Serializable ne mogu se izvršavati na replikama. Iako se funkcionalnost ovog nivoa stalno poboljšava, trenutna ograničenja i dodatno opterećenje čine ga manje privlačnim.

# MVCC

PostgreSQL koristi MVCC (Multi-Version Concurrency Control) da pruži različite poglede na bazu podataka različitim sesijama, koji se zasnivaju na podešenom nivou izolacije. Sa MVCC-om, postiže se visok nivo konkurencije bez žrtvovanja performansi. Ključno pravilo je da čitanje ne sme blokirati pisanje, a pisanje ne sme blokirati čitanje. MVCC se koristi i u mnogim drugim bazama podataka (na primer, Oracle i Berkeley DB).

PostgreSQL koristi koncept identifikatora transakcije (transaction ID) da postigne MVCC. Svaka transakcija će videti efekte svih transakcija koje su započete i završene pre nego što je trenutna transakcija započeta. To je osnovno pravilo koje se poštuje kako bi se dobili konzistentni pogledi na podatke u transakciji.

Na sledećoj slici, P0 je tačka u vremenu. T1, T2, T3 do T13 su transakcije koje počinju jedna za drugom. P1 je tačka u vremenu kada je transakcija T10 započela.



Slika 1. Primer trke transakcija

Sve naredbe u T10 će videti efekte transakcija T1, T3 i ostalih transakcija koje su započete i završene pre nego što je T10 započela. Nijedna naredba u T10 neće videti efekat T7, jer je ona završena nakon što je T10 završila. Nekoliko naredbi u T10 će videti efekat T8 ako je nivo izolacije "read committed". Nekoliko naredbi u T10 će videti efekat T11 ako je nivo izolacije "read committed". Nijedna naredba u T10 neće videti efekat T13.

PostgreSQL koristi nekoliko sistemskih kolona za ovo upravljanje. Postoji prilično mnogo sistemskih kolona prisutnih u svim tabelama u PostgreSQL-u. Dve najrelevantnije kolone za generisanje snapshot-a podataka su:

* **xmin**: Ova kolona čuva identitet transakcije koja je kreirala određenu verziju reda. Napomena je da govorimo o verziji reda. Kada ažuriramo zapis, prilikom usisavanja (vacuuming), postojeći zapis nije prepisan. Kreira se nova verzija.
* **xmax**: Ova kolona čuva identitet transakcije koja je obrisala određeni red. Ponovo, zapis zapravo nije uklonjen iz baze podataka, samo je označen kao obrisan. Ako zapis više nije koristan, s obzirom da nema transakcija sa ID-om manjim od kolone xmax reda, usisivač (vacuum) će se pobrinuti za zapis i osigurati da se prostor ponovo koristi za nove unose.

accounts => **BEGIN**;

**BEGIN**

accounts => **CREATE TABLE** emp(id integer,

first\_name varchar, last\_name varchar);

**CREATE TABLE**

accounts => **INSERT INTO** emp(first\_name,

last\_name) **VALUES** ('SCOTT','TIGER');

**INSERT** 0 1

accounts => **SELECT** xmin,xmax, \* FROM

emp;

xmin | xmax | id | first\_name |last\_name

------+------+----+------------+-----------

6644 | 0 | | SCOTT | TIGER

(1 row)

accounts => **SELECT** txid\_current();

txid\_current

--------------

6644

(1 row)

accounts => COMMIT;

**COMMIT**

Može se videti da je kolona xmin popunjena, dok kolona xmax nije, jer nijedna transakcija još uvek nije obrisala zapis. Kolona xmin je ista kao trenutni ID transakcije. Pokušaćemo da obrišemo i ažuriramo zapis. Sledećom komandom u jednoj sesiji:

accounts=> **BEGIN**;

**BEGIN**

accounts => **SELECT** txid\_current();

txid\_current

--------------

6645

(1 row)

accounts => **UPDATE** emp **SET** last\_name = 'TGR';

**UPDATE** 1

accounts => **SELECT** xmin, xmax, \* **FROM**

emp;

xmin | xmax | id | first\_name |last\_name

------+------+----+------------+-----------

6645 | 0 | | SCOTT | TGR

(1 row)

Vidimo samo novi zapis kreiran od strane trenutne transakcije. Ukoliko ne uradimo COMMIT već pokrenemo još jednu sesiju i probamo sledeće:

accounts => **SELECT** xmin,xmax, \* **FROM** emp;

xmin | xmax | id | first\_name |last\_name

-----+------+----+------------+-----------

6644 | 6645 | | SCOTT | TIGER

(1 row)

Vidimo da postoji istekla verzija zapisa sa prezimenom TIGER. Sada se možemo vratiti na prvu sesiju i ukucati sledeću komandu:

accounts=>**COMMIT**;

**COMMIT**

U obe ove sesije rezultat bi bio

accounts => **SELECT** xmin,xmax, \* **FROM** emp;

xmin | xmax | id | first\_name |last\_name

-----+------+----+------------+-----------

6645 | 0 | | SCOTT | TGR

(1 row)

Sada vidimo novu verziju zapisa, koja je kreirana od strane identifikatora transakcije 6645. Originalni zapis koji je bio unet i njegov identifikator transakcije su prošlost.

Pogledajmo primer transakcije koja je započela tokom trajanja trenutne transakcije:

accounts => **DELETE** **FROM** emp;

**DELETE** 1

accounts => **BEGIN**;

**BEGIN**

accounts => **SELECT** txid\_current();

txid\_current

--------------

6649

(1 row)

accounts => **SELECT** xmin, xmax, \* **FROM**

emp;

xmin | xmax | id | first\_name |last\_name

-----+------+----+------------+-----------

(0 rows)

U drugoj sesiji, unesite zapis:

accounts => **INSERT** **INTO** emp(first\_name,

last\_name) **VALUES**

('SCOTT','TIGER');

Potom u prvoj sesiji:

accounts => **SELECT** xmin, xmax, \* **FROM** emp;

xmin | xmax | id | first\_name |last\_name

-----+------+----+------------+-----------

6650 | 0 | | SCOTT | TIGER

(1 row)

accounts =>**COMMIT**;

**COMMIT**

Sada SELECT naredba u transakciji ID 6649 može videti efekat transakcije ID 6650. To zavisi od nivoa izolacije postavljenog za transakciju.

# Zaključavanja

PostgreSQL koristi napredan mehanizam zaključavanja kako bi upravljao konkurentnim pristupom deljenim resursima, osiguravajući doslednost i integritet podataka. Zaključavanja su ključna za sprečavanje konflikata koji nastaju kada više procesa pokušava da pristupi ili modifikuje isti resurs istovremeno.

U PostgreSQL-u su neophodna za koordinaciju akcija između konkurentnih transakcija. Oni sprečavaju anomalije kao što su "dirty reads", "non-repeatable reads" i "phantom reads". Ove anomalije mogu nastati u odsustvu odgovarajuće sinhronizacije, što dovodi do nekonzistentnosti podataka. Mehanizmi zaključavanja PostgreSQL-a su dizajnirani da podrže visok nivo konkurentnosti i efikasno procesuiranje transakcija, što je ključno za performanse u okruženjima sa više korisnika.

Na visokom nivou, zaključavanja se mogu kategorizovati u nekoliko tipova na osnovu njihove preciznosti i tipa resursa koji štite. Ovi uključuju "heavyweight" zaključavanja, "lightweight" zaključavanja i “advisory” zaključavanja, između ostalih. Svaka vrsta zaključavanja služi specifičnoj svrsi i pogodna je za različite scenarije, osiguravajući da baza podataka ostane konzistentna i da se operacije obavljaju bez konflikta.

## Zaključavanja na nivou relacija

PostgreSQL pruža čak osam modova zaključavanja relacija (tabela, indeks ili bilo koji drugi objekat). Ovakva raznovrsnost omogućava maksimalan broj konkurentnih komandi koje se mogu izvršiti nad relacijom.

Mod Access Share je najslabiji, može se koristiti sa bilo kojim drugim modom osim Access Exclusive, koji je nekompatibilan sa svim modovima. Tako, SELECT komanda može da se izvršava paralelno sa gotovo bilo kojom operacijom, ali ne dozvoljava da se obriše tabela koja se trenutno upituje.

Prva četiri moda omogućavaju istovremene modifikacije heap-a, dok ostala četiri ne. Na primer, CREATE INDEX komanda koristi Share mod, koji je kompatibilan sa samim sobom (tako da možete kreirati nekoliko indeksa na tabeli istovremeno) i sa modovima koje koriste operacije samo za čitanje. Kao rezultat, SELECT komande mogu da se izvršavaju paralelno sa kreiranjem indeksa, dok će INSERT, UPDATE i DELETE komande biti blokirane.

Nasuprot tome, nedovršene transakcije koje modifikuju podatke u heap-u blokiraju komandu CREATE INDEX. Umesto toga, mogu se pozvati CREATE INDEX CONCURRENTLY, koji koristi slabiji mod Share Update Exclusive, potrebno je više vremena za kreiranje indeksa (i ova operacija može čak i da ne uspe), ali zauzvrat su omogućene istovremene modifikacije podataka.

Komanda ALTER TABLE ima više verzija koje koriste različite modove zaključavanja (Share Update Exclusive, Share Row Exclusive, Access Exclusive). Sve one su opisane u dokumentaciji.

### Teška zaključavanja

Teška zaključavanja su dugoročna zaključavanja. Preuzimaju se na nivou objekta i uglavnom se koriste za relacije, ali se mogu primeniti i na druge tipove objekata. Teška zaključavanja obično štite objekte od konkurentnih ažuriranja ili zabranjuju njihovo korišćenje tokom restrukturiranja, ali mogu služiti i drugim potrebama. Ovakve definicija je zbog toga jer se ova zaključavanja koriste za različite svrhe. Jedina stvar koju imaju zajedničku je njihova interna struktura. Njima upravlja menadžer zaključavanja, koji prati zaključavanja u sistemskom katalogu pg\_locks.

Sistemski katalog pg\_locks pruža informacije o svim aktivnim zaključavanjima u sistemu, uključujući tip zaključavanja, mod i transakciju koja drži zaključavanje. Ovaj katalog je od velikog značaja za administratore baze podataka pri dijagnostikovanju problema sa zaključavanjem i razumevanju ponašanja zaključavanja njihovih aplikacija.

Sledeća lista predstavlja imena tipova lock-ova na način na koji su prikazana u pg\_locks:

* transactionid and virtualxid — lock za transaction ID
* relation — lock na nivou veze
* tuple — lock primenjen na tuple-u
* object — lock na objektima koji nisu relacije
* extend — lock proširenja relacije
* page —page-level lock korišćen od strane nekog indeksnog tipa
* advisory —advisory lock

## Zaključavanja na nivou redova

Zaključavanja na nivou redova pružaju detaljniju kontrolu od zaključavanja na nivou relacija, omogućavajući više transakcija da rade na različitim redovima iste tabele istovremeno. Ova kontrola je ključna za okruženja sa visokim nivoom konkurentnosti, gde sposobnost zaključavanja pojedinačnih redova umesto celih tabela značajno poboljšava performanse.

U okruženjima gde se mnoge transakcije istovremeno izvode nad istom tabelom, mogućnost zaključavanja samo potrebnih redova umesto cele tabele značajno smanjuje potencijalne konflikte i poboljšava ukupnu efikasnost sistema. Na primer, ako dva korisnika pristupaju različitim redovima u istoj tabeli, njihovi zahtevi neće biti u sukobu, što omogućava brže i efikasnije obavljanje transakcija.

PostgreSQL koristi različite modove zaključavanja na nivou redova, zavisno od specifičnih potreba transakcija:

Za Update Lock: Sprečava druge transakcije da dobiju zaključavanja na istim redovima. Ovo zaključavanje je korisno kada se red treba ažurirati, jer osigurava da drugi procesi ne mogu izvršiti promene dok je red zaključan za ažuriranje.

Za No Key Update Lock: Slično zaključavanju za ažuriranje, ali ne dozvoljava promene primarnog ključa reda. Ovo je korisno kada se red ažurira, ali se ne menja njegov primarni ključ, omogućavajući veću fleksibilnost dok se i dalje održava integritet podataka.

Za Key Share Lock: Omogućava drugim transakcijama da čitaju red, ali sprečava promene primarnog ključa. Ovo zaključavanje se često koristi u scenarijima gde je važno omogućiti čitanje podataka bez rizika od promene primarnog ključa.

### Upravljanje konkurentnim pristupom

Zaključavanja na nivou redova takođe koriste mehanizam reda čekanja za upravljanje konkurentnim pristupom. Kada više transakcija zahteva pristup istim redovima, one se stavljaju u red čekanja, čime se osigurava da zaključavanja budu dodeljena na uredan način i sprečavajući zastoj.

Mehanizam reda čekanja funkcioniše tako da, kada transakcija ne može odmah dobiti traženo zaključavanje, ona se stavlja u red čekanja dok zaključavanje ne postane dostupno. Ovaj pristup pomaže u održavanju stabilnosti i integriteta sistema, osiguravajući da transakcije budu obrađene po redosledu prijema zahteva.

### Zaključavanja bez čekanja

U scenarijima sa visokim performansama, mogu se koristiti zaključavanja bez čekanja. Ova zaključavanja odmah ne uspevaju ako ne mogu biti dobijena, čime se izbegava vreme čekanja i omogućava transakciji da nastavi bez kašnjenja.

Zaključavanja bez čekanja su posebno korisna u aplikacijama koje zahtevaju brzu obradu podataka i gde je prihvatljivo odbacivanje zahteva umesto zadržavanja procesa u čekanju. Na primer, u visoko interaktivnim aplikacijama kao što su onlajn transakcioni sistemi, gde korisnici očekuju brze odgovore, zaključavanja bez čekanja mogu značajno poboljšati korisničko iskustvo smanjujući latenciju.

### Sprečavanje zastoja

Efikasan mehanizam upravljanja zaključavanjima na nivou redova takođe pomaže u sprečavanju zastoja (deadlocks), situacija u kojima dve ili više transakcija čekaju jedna na drugu da oslobode zaključavanja. PostgreSQL automatski detektuje i rešava zastoje tako što prekida jednu od transakcija, omogućavajući drugim transakcijama da nastave.

Detekcija zastoja se vrši pomoću algoritama koji redovno proveravaju redove čekanja na zaključavanja. Kada se otkrije ciklična zavisnost (gde transakcije formiraju zatvoreni krug čekanja), PostgreSQL automatski interveniše, prekidajući jednu od transakcija i oslobađajući zaključavanja, čime se omogućava ostatku sistema da nastavi s radom bez većih problema.

## Multitransakcije i zastoji

Multitransakcije omogućavaju više transakcija da drže zaključavanja na istom redu. Ova funkcionalnost je posebno korisna za složene operacije koje uključuju više koraka ili podtransakcija. Multitransakcije povećavaju fleksibilnost i efikasnost upravljanja transakcijama, omogućavajući PostgreSQL-u da rukuje naprednim transakcijskim tokovima rada.

Zastoji se javljaju kada dve ili više transakcija čekaju jedna na drugu da oslobode zaključavanja. Na primer, Transakcija A drži zaključavanje koje je potrebno Transakciji B, a Transakcija B drži zaključavanje koje je potrebno Transakciji A. Ova situacija uzrokuje da obe transakcije budu zaglavljene, čekajući beskonačno. PostgreSQL uključuje mehanizme za automatsko otkrivanje i rešavanje zastoja. Kada se otkrije zastoj, sistem prekida jednu od transakcija (obično onu sa najmanje uloženih resursa) kako bi prekinuo zastoj i omogućio drugoj transakciji da nastavi.

## Ostala zaključavanja

Pored standardnih zaključavanja na objektima baze podataka, PostgreSQL podržava još nekoliko tipova različitih zaključavanja, uključujući neobjektna zaključavanja i zaključavanja za proširenje relacija.

### Neobjektna zaključavanja

Neobjektna zaključavanja se koriste za svrhe koje nisu direktno povezane sa specifičnim objektima baze podataka. Na primer, advisory zaključavanja omogućavaju aplikacijama da implementiraju prilagođene strategije zaključavanja. Ova zaključavanja se mogu koristiti za sinhronizaciju aktivnosti koje nisu direktno vezane za tabelu ili red, pružajući veću fleksibilnost u dizajnu aplikacija.

### Zaključavanja za proširenje relacija

Zaključavanja za proširenje relacija su neophodna kada se tabela proširuje, kao što je dodavanje nove stranice za smeštaj više redova. Ova zaključavanja osiguravaju da samo jedan proces može proširiti relaciju u isto vreme, sprečavajući oštećenja i osiguravajući integritet podataka.

### Zaključavanja na memorijskim strukturama

Zaključavanja nisu ograničena na trajne objekte baze podataka, oni se takođe primenjuju na razne memorijske strukture kako bi se osigurao konzistentan pristup i modifikacija. Na primer, zaključavanja na strukturama deljene memorije kao što su keš bafera i podaci menadžera zaključavanja su ključna za održavanje integriteta internog stanja PostgreSQL-a. Ova zaključavanja osiguravaju da konkurentni procesi mogu bezbedno čitati i pisati u deljene memorijske oblasti bez uzrokovanja nekonzistentnosti.

# Zaključak

Detaljna analiza mehanizama izolacije i zaključavanja u PostgreSQL bazi podataka pokazuje kako se efikasno postiže doslednost podataka čak i u uslovima visokog nivoa konkurencije. Implementacija viševerzijske kontrole konkurencije (MVCC) omogućava paralelno izvršavanje transakcija sa minimalnim zaključenjem redova, čime se smanjuje mogućnost pojave anomalija i poboljšava performanse sistema. MVCC omogućava korisnicima da pristupaju različitim verzijama podataka bez potrebe za zaključavanjem, što značajno doprinosi efikasnosti i skalabilnosti sistema.

Različiti nivoi izolacije koje nudi PostgreSQL pružaju fleksibilnost u odabiru odgovarajuće mere zaštite podataka u skladu sa specifičnim potrebama aplikacije. Nivo izolacije Read Committed omogućava visoke performanse uz osnovnu zaštitu od anomalija, dok nivo Serializable obezbeđuje najviši stepen izolacije i doslednosti, ali po cenu smanjenja performansi zbog većeg broja zaključavanja. Razumevanje ovih kompromisa je ključno za optimizaciju performansi baza podataka i očuvanje integriteta podataka.

Ova analiza doprinosi dubljem razumevanju teorijskih i praktičnih aspekata izolacije i zaključavanja u PostgreSQL bazama podataka. Efikasno dizajniranje i implementacija transakcija omogućava očuvanje doslednosti i integriteta podataka, čak i u složenim scenarijima konkurentnog pristupa. Optimalna primena ovih mehanizama osigurava stabilnost i pouzdanost sistema za upravljanje bazama podataka, čime se omogućava njihova upotreba u širokom spektru aplikacija i industrija.

# Literatura

[1] PostgreSQL dokumentacija - <https://www.postgresql.org/files/documentation/pdf/16/postgresql-16-A4.pdf>

[2] Ibrar Ahmed, Gregory Smith, Enrico Pirozzi - PostgreSQL 10 High Performance-Packt Publishing (2018)

[3] Jayadevan Maymala - PostgreSQL for Data Architects-Packt Publishing (2015)

[4] Joshua D. Drake, John C. Worsley - Practical PostgreSQL

[5] Baji Shaik, Avinash Vallarapu - Beginning PostgreSQL on the Cloud\_ Simplifying Database as a Service on Cloud Platforms-Apress (2018)