TRANSAKCIJE

U situacijama kada jedan ili više korisnika pokušava da pristupi i promeni podatke u istoj tabeli u istom trenutku, dolazi do preplitanja akcija i mogućnosti nekonzistentne promene istih podataka ili netačnosti podataka. U cilju rešavanja ovog problema SQL podržava korišćenje transakcija da bi se obezbedilo konkurentno izvođenje akcija, ali i u cilju oporavka baze podataka.

Transakcija se definiše kao jedinica posla, koju čine jedna ili više SQL naredbi koje izvršavaju odgovarajući skup akcija.
NPR. Prijavljivanje ispita je transakcija, koja se sastoji od više akcija (čitanje i provera podataka u više tabela , ažuriranja i upisivanja podataka).
Problem predstavljaju konkurentne transakcije. One zahtevaju posebnu kontrolu.

Karakteristike transakcija, planiranje i izvršavanje

Sa stanovišta DBMS-a, transakcija je jedna atomska jedinica posla.
ACID (**A**tomicity, **C**onsistency, **I**solation, **D**urability) - svojstva transakcija koja DBMS mora da obezbedi u pogledu konkurentnog pristupa i sistemskih padova:

1. **Atomicity**
   Korisnici bi trebalo da posmatraju izvođenje svake transakcije kao **atomske**. Sve akcije u transakciji će se realizovati u celini ili neće ni jedna. Korisnici ne bi trebalo da brinu o efektima nekompletnih transakcija (recimo, kada se dogodi pad sistema).
2. **Consistency**
   Svaka transakcija, izvedena sama sa nekonkurentnim izvođenjem drugih transakcija mora da štiti konzistenciju baze podataka. Drugim rečima, ako je svaka transakcija konzistentna, a baza podataka bila u konzistentnom stanju ona će ostati u tom stanju i posle završetka transakcija. Ovo svojstvo se naziva **konzistencija**, i DBMS preuzima da je obezbedi za svaku transakciju. Garantovanje ovog svojstva transakcija je odgovornost korisnika.
3. **Isolation**
   DBMS prepliće akcije nekoliko transakcija zbog obezbeđivanja boljih performansi, ali i zbog usaglašenosti sa realnim sistemom. Uticaj na bazu podataka mora da bude isti kao da se transakcije izvršavanju sekvencijalno. Ovo svojstvo se naziva **izolacija**. Transakcije su izolovane, ili zaštićene, od uticaja istovremenog izvođenja drugih konkurentnih transakcija nad istim podacima. Nivo izolovanosti transakcija može da bude različit. Pošto SQL server zaključava podatke da bi obezbedio konzistentnost podataka, više zaključavanja znači manji nivo pristupa podacima i niži nivo istovremenog rada transakcija. Viši nivo konzistentnosti dovodi do nižeg nivoa konkurentnosti.
4. **Durability**
   Kada DBMS informiše korisnika da je transakcija uspešno završena (izvršen *commit*), njeni efekti bi trebalo da budu sačuvani, čak i ako sistem padne, pre nego što su se sve njene promene upisane na disku. Drugim rečima, rezultati transakcija koje su uspešno završene i kao takve potvrđene, pamti se u bazi. Ovo svojstvo se naziva **trajnost**.

Konzistencija i izolacija

Korisnik koji pravi transakciju mora da obezbedi da se izvođenje same transakcije završi na „konzistentnoj“ instanci baze podataka, tj. transakcija mora ostaviti bazu podataka u „konzistentnom“ stanju.
Svojstvo izolacije je obezbeđeno garantovanjem da će i pored akcija nekoliko transakcija koje mogu da se prepliću krajnji efekat da bude jednak efektu izvođenja transakcija jedne posle druge po nekom serijskom redosledu. Na primer, ako se dve transakcije *T1* i *T2* izvode istovremeno, garantuje se da kranji efekat bude isti sa izvođenjem *T1* posle *T2* ili izvođenjem *T2* posle izvođenja *T1*.
Konzistentnost baze podataka sledi iz transakcione atomičnosti, izolacije i transakcione konzistentnosti.

Atomičnost i trajnost

Transakcije mogu da budu nekompletne iz četiri razloga:
- Prvo, transakcija može da bude prekinuta (odbačena) od strane DBMS-a, zato što su se nastale neke anomalije tokom izvršavanja. Ako je transakcija prekinuta od DBMS-a iz nekih internih razloga, ona se automatski restartuje i izvršava ponovo.
- Drugo, sistem može da padne (na primer, zato što je prekinuto snabdevanje strujom) dok je jedna ili više transakcija u izvršavanju.
- Treće, transakcija može da naiđe na neočekivanu situaciju (na primer, čitanje nepredviđene vrednosti podataka ili nije sposobna da pristupi nekom disku) i "reši" da prekine samu sebe.
- Četvrto, može da dođe do fizičkog oštećenja na memorijskim uređajima, kada podaci na njima više nisu dostupni.

Transakcija koja je prekinuta može da ostavi bazu podataka u nekonzistentnom stanju. DBMS mora da odstrani efekte nepotpunih transakcija iz baze podataka, to jest, mora da obezbedi atomičnost transakcija: bilo da su sve akcije transakcija završene ili nisu. DBMS obezbeđuje atomičnost transakcija poništavanjem akcija *nekompletnih* transakcija. Da bi bio u stanju da ovo čini, DBMS održava *dnevnik (log)*, svih upisivanja u bazu podataka. Dnevnik se, takođe, koristi da obezbedi trajnost: Ako sistem padne pre neko što su promene učinjene kompletnim (transakcije upisane na disk), dnevnik se koristi da zapamti i vrati ove promene kada se sistem restartuje.
DBMS komponenta koja omogućava atomičnost i trajnost se naziva *rukovalac oporavkom (recovery manager)*.

Transakcije i vremensko planiranje

DBMS vidi transakcije kao seriju, ili spisak (listu), **akcija**. Akcije koje mogu da se izvrše transakcijama uključuju **čitanja** i **upisivanja** objekata baze podataka. Transakcije, takođe, mogu da se definišu kao skup akcija koje su delimično (parcijalno) uređene. To jest, relativni redosled (poredak) nekih akcija ne mora da bude značajan.

Zbog jednostavnosti zapisa i razmatranja pojedinih slučajeva uvode se sledeće pretpostavke i oznake:
Podrazmeva se da su objekti *O* nad kojima se izvršavaju operacije uvek učitani (u promenljivu koja se takođe naziva *O*).
RT(O) - akcija transakcije *T* koja čita objekat *O*; slično, upisivanje kao *WT* (*O*).
Kada je transakcija *T* izbrisana iz konteksta biće izostavljen indeks.

Kao dodatak čitanju i pisanju svaka transakcija *mora* da specificira svoju završnu akciju bilo **commit** (na primer, potpuno uspešno završena) ili **abort** (na primer, završavanje i poništavanje svih akcija koje su sprovedene). *Abort* označava akciju *T* kao prekinutu, a *Commit* označava *T* kao potvrđenu.

Vremensko planiranje je spisak akcija (čitanja, upisivanja, prekidanja ili potvrđivanja) iz skupa transakcija, a uređenje u kome se dve akcije transakcije *T* pojavljuju u vremenskoj raspodeli mora da budu isto kao uređenje u kome se one pojavljuju u *T*. Vremensko planiranje predstavlja neku stvarnu (aktuelnu) ili potencijalnu izvođačku sekvencu. Na primer, vremensko planiranje u donjoj tabeli prikazuje uređenje izvođenja dve transakcije *T1* i *T2*. Izvršavanje počinje od prve vrste. Vremensko planiranje opisuje akcije transakcija kako ih "vidi" DBMS. U dodatku ovim akcijama, transakcije mogu da izvedu druge transakcije, kao što su čitanje i pisanje od datoteka operativnog sistema, koje izračunavaju aritmetičke izraze, i tako dalje.

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

Tabela *18-1* Vremenski plan koji uključuje dve transakcije

Vremensko planiranje u tabeli 18-1 ne sadrži prekid ili potvrđivanje akcija za bilo koju transakciju. Vremensko planiranje koje sadrži bilo prekid ili potvrđivanje za svaku transakciju, čije akcije su nabrojane u njoj, se naziva **kompletno vremensko planiranje**. Kompletno vremensko planiranje mora da sadrži sve akcije svih transakcija koje se pojavljuju u njoj. Ako se akcije različitih transakcija ne prepliću – to jest, transakcije se izvršavaju od početka do kraja jedna po jedna – takvo vremensko planiranje nazivamo **serijsko vremensko planiranje**.

Konkurentno izvršavanje transakcija

DBMS prepliće (preklapa) akcije različitih transakcija da poveća performanse, tako što povećava propusnu moći ili povećava vreme odziva (odgovora) za kratke transakcije.

Motivacija za konkurentno izvršavanje

Vremensko planiranje prikazano na slici *18-1* predstavlja preklapanje izvršavanja dve transakcije. Potrebno je omogućavanje izolacije transakcija dok se dozvoljava takvo konkurento izvršavanje. Dok jedna transakcija čeka na stranicu koja će biti pročitana sa diska, CPU može da obradi drugu transakciju. Preklapanje aktivnosti I/O i CPU povećava **propusnu moć sistema** (prosečan broj transakcija koje su završene u datom periodu). Preplitanje izvršavanja kratkih transakcija sa dugim transakcijama obično omogućava kratkim transakcijama da se brže kompletiraju (završavaju). U serijskom izvršavanju, kratke transakcije bi mogle da se nađu iza dugih transakcija uvodeći nepredvidivo odlaganje u **vremenu odziva**, ili prosečnog vremena koje je uzeto za kompletiranje transakcija.

Serijabilnost

Polazna pretpostavka je da je projektant baze podataka definisao neku ideju **konzistentnog stanja baze podataka**. Na primer, jedan od kriterijuma konzistentnosti baze podataka univerziteta bi mogao da bude da suma zarade zaposlenih u svakom odeljenju treba da bude manja od 80 procenata budžeta za ovo odeljenje. Ako se zahteva da svaka transakcija mora da **očuva** konzistentnost baze podataka, sledi da će bilo koje serijsko vremensko planiranje (raspored) koje je završeno, takođe, čuvati konzistentnost baze podataka.

Serijabilno vremensko planiranje nad skupom *S* potvrđenih transakcija je vremensko planiranje čiji je efekt nad bilo kojom konzistentnom instancom baze podataka garantovano isti kao i neko završeno serijsko vremensko planiranje nad *S*.
To jest, instanca baze podataka koja rezultira iz izvršavanja datog vremenskog planiranja je identična sa instancom baze podataka koja rezultira iz izvršavanja transakcija po nekom serijskom uređenju (redosledu).
Napomene:

- Izvršavanje transakcija serijski i u različitim redosledima može da proizvede različite rezultate, ali sva ta izvršavanja pretenduju da budu prihvatljiva; DBMS ne pravi garancije o tome koja od njih će biti rezultat transakcija koje se prepliću.
- Gornja definicija serijabilnog vremenskog planiranja ne pokriva slučajeve vremenskih planiranja koja su sadržala prekinute transakcije. Zbog pojednostavljenja, nastavak razmatranja se tiče preplitanja izvršavanja skupa završenih, potvrđenih transakcija.
- Ako transakcija izračunava vrednost i prikazuje je na ekranu, ovo je „efekat“ koji ne utiče direktno na stanje baze podataka. Radi pojednostavljenja, podrazumevaće se da su sve takve vrednosti, takođe, upisane u bazu podataka.

Neke anomalije koje su povezane sa preplitanjem transakcija

Tri glavna slučaja, u kojima vremensko planiranje obuhvata dve potvrđene transakcije, koje bi mogle da se izvedu nad konzistentnom bazom podataka i ostave je u nekonzistentnom stanju (tri slučaja kada su akcije transakcija *T1* i *T2* sukobljene jedna sa drugom su):

- **writeread (WR),** *T2* čita podatke objekta pre upisivanja *T1*; slično se definišu konflikti
- **read-write (RW)** i
- **write-write (WW)**.

Dve akcije nad istim objektom podataka biće u konfliktu (sukobiće se) ako je najmanje jedna od njih upisana.

Čitanje nepotvrđenih podataka (WR konflikt) - prljavo čitanje

Prvi izvor anomalije je vremensko planiranje u kome transakcija *T2* može da pročita objekat *A* baze podataka koji nije bio promenjen od transakcije *T1*, koja još nije potvrđena.
Primer. Razmotrimo dve transakcije *T1* i *T2*, od kojih se svaka, izvodi sama, štiteći konzistentnost baze podataka: *T1* prenosi $100 iz *A* u *B*, a *T2* povećava i *A* i *B* za 6 procenata (na primer, godišnja kamata je pridodata na ova dva računa). Pretpostavimo da se njihove akcije prepliću tako da
(1) program prenosa računa *T1* odbija $100 od računa *A*, onda
(2) program za polaganje kamate *T2* čita tekuću vrednosti računa *A* i *B* i dodaje 6 procenata kamate na svaki, a onda

(3) program za prenos računa polaže $100 na račun *B*.
Odgovarajuće vremensko planiranje, koje je pogled od strane DBMS-a, je prikazano u tabeli *18-2*.

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

Tabela *18-2* Čitanje nepotvrđenih podataka

Rezultat ovog vremenskog planiranja je različit od bilo kog rezultata koji bi dobili izvršavanjem prvo jedne od dve transakcije a onda druge. Uzrok problema jeste činjenica da je vrednost *A* upisana sa *T1*, pročitana od *T2* pre nego što je *T1* završila sve svoje promene.
Opšti problem prikazan ovde je da *T1* može da upiše neku vrednost u *A* koja čini bazu podataka nekonzistentnom. Sva upisivanja transakcije *T1* sa „korektnim" vrednostima *A* pre potvrđivanja, ne škodi, ako *T1* i *T2* rade po nekom serijskom redosledu, zato što onda *T2* neće da vidi (privremeno) nekonzistentnost. U protivnom, preplitanje izvršavanja prouzrokuje ovu nekonzistentnost i dovodi bazu podataka u nekonzistentno stanje.
Transakcije (mada moraju da ostave bazu podataka u konzistentnom stanju posle njihovih završetka) ne zahtevaju da se baza podataka drži konzistentnom dok je ona još u razvijanju. Takvi zahtevi bi bili isuviše restriktivni: Za prenošenje novca sa jednog računa na drugi, transakcija *mora* da umanji jedan račun, privremeno ostavljajući bazu podataka nekonzistentnom, a onda uveća drugi račun, koji uspostavlja (vraća) ponovo konzistentnost.

Neponovljena čitanja (RW konflikt)

Do **RW konflikta** bi došlo kada transakcija T1 pročita vredost objekta A, transakcija *T2* može da promeni vrednost objekta *A* koja nije bila pročitana od transakcija *T1*, dok je *T1* još u razvoju.

Pisanje preko ne potvrđenih podataka (WW konflikt)

Treći izvor nepravilnog ponašanja mogućnost da transakcija *T2* može da prepiše preko vrednosti objekta *A*, koja je već bila promenjena transakcijom *T1*, dok je *T1* još u razvoju. Ako *T2* ne pročita vrednost *A* upisanu sa *T1*, nastaje potencijalni problem kao što ilustruje sledeći primer.
Pretpostavimo da su Janko i Marko dva službenika, a njihove plate moraju da se održavaju jednakim. Transakcija *T1* postavlja njihove plate na $1,000, a transakcija *T2* postavlja njihove plate na $2,000. Ako ih izvršavamo po serijskom uređenju da *T1* sledi posle *T2*, oba primaju platu $2,000; serijsko uređenje da *T2*

sledi posle *T1* daje svakom platu $1,000. Bilo šta od ovoga je konzistentno. Zapazimo da nijedna transakcija ne čita vrednost plate pre nego što je upiše-takvo upisivanje se naziva **slepo** upisivanje, iz očiglednih razloga.
Sada razmotrimo sledeće preplitanje akcija *T1* i *T2*: *T1* postavlja Jankovu platu na $1,000, *T2* postavlja Markovu platu na $2,000, *T1* postavlja Markovu platu na $1,000, i konačno *T2* postavlja Jankovu platu na $2,000. Rezultat nije identičan rezultatu bilo kog serijskog izvršavanja, i vremensko preplitanje nije zbog toga serijabilno. To krši željeni konzistentni kriterijum da dve plate moraju da budu jednake.

Vremenska planiranja koja obuhvataju neuspele transakcije

Podrazumevamo da će sve akcije neuspelih transakcija biti poništene i da neće biti obnovljene. Na bazi toga, proširujemo definiciju serijabilnog vremenskog planiranja na sledeći način: **Serijabilno vremensko planiranje** nad skupom *S* transakcija je planiranje čije dejstvo na bilo koju konzistentnu instancu baze podataka je identično sa nekim potpunim vremenskim planiranjem nad skupom *potvrđenih* transakcija u S.
Ova definicija serijabilnosti proširena na akcije neuspelih transakcija, koje su pri tome potpuno poništene, bi mogla, u nekim situacijama, da bude nemoguća. Na primer, da (1) neki program za prenos računa *T1* oduzima $100 sa računa *A*, a onda (2) neki program za obračun kamate *T2*, čita aktuelnu vrednosti računa *A* i *B* i dodaje 6 procenata kamate svakom, onda potvrđuje, a onda je (3) *T1* neuspela. Odgovarajuće vremensko planiranje je prikazano u tabeli *18-3*.

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
Abort	

Tabela *18-3* Vremenski plan bez mogućnosti oporavka

Tako je *T2* pročitala vrednost za *A*, koja nije rezultat potvrđene transakcije (Podsetimo da efekti neuspelih transakcija ne bi trebalo da budu vidljivi za druge transakcije). Ako *T2* nije još potvrđena, mogli bi da imamo posla sa situacijom koja kaskadno prekida *T1*, a takođe, prekida *T2*; takav proces će rekurzivno da prekine bilo koju transakciju koja, čita podatke koje je upisala transakcija *T2*, i tako dalje. Ali *T2* je već potvrđena i mi ne možemo da poništimo njene akcije. Takvo vremensko planiranje se naziva *nepovratnim* (*fatalnim)*.

Povratno vremensko planiranje je ono u kome se transakcije potvrđuju samo posle i ukoliko su sve transakcije čije promene one čitaju potvrđene. Ako transakcije čitaju samo promene potvrđenih transakcija, ne samo da je vremensko planiranje povratno, već se i prekidanje transakcije može izvršiti bez kaskadnog prekida drugih transakcija.

Moramo, takođe, da razmotrimo nekompletne transakcije za rigoroznu diskusiju sistemskih prekida, zato što transakcije koje su aktivne kada sistem padne nisu niti prekinute niti potvrđene. Naravno, oporavak sistema obično počinje sa prekidom svih aktivnih transakcija, i za našu neformalnu diskusiju, razmatranje vremenskog planiranja obuhvatanjem potvrđenih i prekinutih transakcija je dovoljno.

Postoji drugi potencijalni problem u poništavanju akcija transakcija. Pretpostavimo da transakcija *T2* prepisuje preko vrednosti nekog objekta *A* koji je bio promenjen transakcijom *T1*, dok je *T1* još u toku i *T1* se kasnije prekine. Sve promene *T1* nad objektima baze podataka su poništene ponovnim vraćanjem vrednosti bilo kog objekta koja je promenjena u vrednost objekta pre promena *T1*. Kada se *T1* prekine, i njene promene poništе na ovaj način, promene T2 se takođe gube, čak i ako T2 odluči da potvrdi transakciju. Tako, na primer, ako je *A* imao originalnu vrednost 5, koja je promenjena sa *T1* na 6, i sa T2 na 7, ako se *T1* sada prekine, vrednost A postaje ponovo 5. Čak ako *T2* potvrdi transakciju, izmena objekta *A* je izgubljena. Tehnika kontrole konkurencije nazvana Strict 2PL, može da spreči ovaj problem.

Zaključavanje bazirano na kontroli konkurencije

DBMS mora da obezbedi da bude dozvoljeno samo **serijabilno, povratno vremensko planiranje**. DBMS obično koristi *protokol zaključavanja* da to postigne. **Protokol zaključavanja** je skup pravila koja se primenjuju pri svakoj transakciji (sprovedenoj od DBMS-a), da bi obezbedio da u isti mah nekoliko transakcija može da se prepliće, a da efekat bude isti kao i pri pri njihovom serijskom izvođenju.

Strogo dvo-fazno zaključavanje (Strogo 2PL)

Najšire korišćen protokol zaključavanja, nazvan strogo *dvo-fazno zaključavanje* ili *Strogo 2PL*, ima dva pravila.

(1) Ako transakcija *T* želi da pročita (naizmenično, respektivno **menja**) neki objekat, ona zahteva **deljivo** (respektivno **ekskluzivno**) zaključavanje objekta. Naravno, transakcija koja ima ekskluzivno zaključavanje može da čita objekat; dodatno deljivo zaključavanje se ne zahteva. Transakcija koja zahteva zaključavanje je suspendovana dok DBMS ne bude sposoban da joj dozvoli zahtevano zaključavanje. DBMS održava trag zaključavanja koji je dozvolio i obezbeđuje da ako transakcija drži ekskluzivno zaključavanje nekog objekta, druge transakcije ne mogu da drže deljivo ili ekskluzivno zaključavanje istog objekta. Drugo pravilo Strogog 2PL je

(2) Sva zaključavanja držana od transakcije se oslobađaju kada se transakcija završi. Zahtevi za dobijanje ili oslobađanje zaključavanja se automatski obrađuju i beleže od strane DBMS-a.

U rezultatu, protokol zaključavanja omogućava „sigurno“ („bezbedno“) preplitanje transakcija. Ako dve transakcije pristupaju potpuno nezavisnim delovima baze podataka, one će moći istovremeno (konkurentno) da dobiju zaključavanja, koja su im potrebna i nastave sa radom. Ako dve transakcije pristupaju istom objektu, a jedna od njih želi da ga promeni, njihove akcije su stvarno uređene serijski - sve akcije jedne od ovih transakcija (ona koja dobija zaključavanje na zajedničkom objektu prva) su završene pre (ovo zaključavanje je oslobođeno i) druga transakcija može da nastavi.

Označimo akciju transakcije *T* koja zahteva deljivo (respektivno, *ekskluzivno*) zaključavanje na objektu *O* kao *ST(O)* (respektivno, *XT(O))*, i izostavimo indeks koji označava transakciju kada je to jasno iz konteksta. Na primer, razmotrimo vremensko planiranje prikazano na slici *18-2*.

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

Ovo preplitanje bi moglo da rezultira u stanje koje ne može da rezultira iz bilo kojeg serijskog izvršavanja ovih transakcija. Na primer, *T1* bi mogla da promeni *A* sa 10 na 20, onda *T2* (koja čita vrednost 20 za *A*), bi mogla da promeni *B* iz 100 u 200, a onda će *T1* da pročita 200 za *B*. Ako rade serijski, bilo *T1* bilo *T2* da se izvrši prva, čita vrednost 10 za *A* i 100 za *B*: Jasnije, preplitano izvršavanje nije jednako njihovom serijskom izvršavanju.
Ako se koristi Strogi 2PL protokol gornje preplitanje nije dozvoljeno. Da vidimo zašto. Pretpostavka je da transakcije postupaju na isti način. Transakcija *T1* će prva dobiti ekskluzivno zaključavanje na *A*, zatim će čitati i upisati novu vrednost u *A* (tabela *18-4*). Tada sledi zahtev T2 za zaključavanje na *A*. Ovaj zahtev neće biti dozvoljen

T1	T2
X(A)	
R(A)	
W(A)	

Tabela *18-4* Vremenski plan koji ilustruje tehniku *Strict 2PL*

dok *T1* drži ekskluzivno zaključavanje na *A*, i zbog toga DBMS zaustavlja (suspenduje) *T2*. *T1* nastavlja da drži ekskluzivno zaključavanje na *B*, čita i piše *B*, onda konačno potvrđuje, pa su u tom trenutku zaključavanja oslobođena. Zahtevi *T2* za zaključavanjima su sada dozvoljeni, pa ona nastavlja sa radom. U ovom primeru protokol zaključavanja rezultira serijskim izvršavanjem ove dve transakcije, prikazanom u tabeli *18-5*.

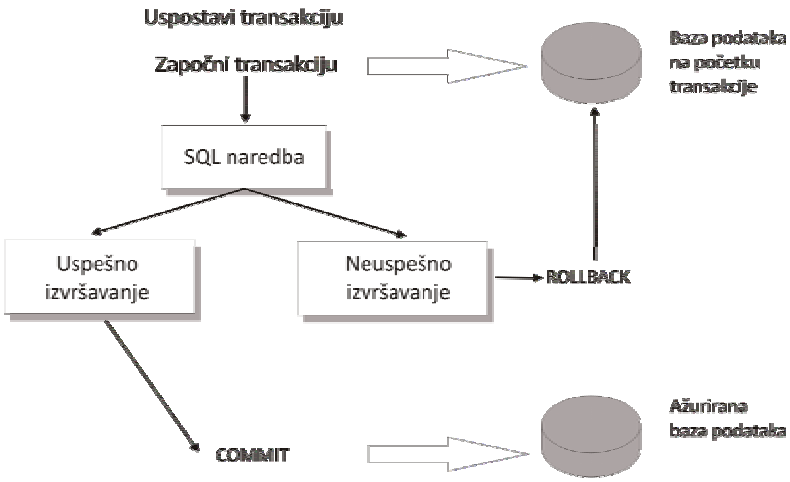
T1	T2
R(A)	
W(A)	
	R(B)
	W(B)
R(C)	

W(C)

Tabela *18-5* Vremenski plan koji ilustruje tehniku *Strict 2PL* sa serijskim izvršavanjem

Naredbe za upravljanje transakcijama

SQL obezbeđuje sledeće naredbe za upravljanje transakcijama:
SET TRANSACTION,
START TRANSACTION – BEGIN TRAN,
SET CONSTRAINTS – nema ga SQL server,
SAVEPOINT – SAVE TRAN,
RELEASE SAVEPOINT – nema ga,
ROLLBACK i
COMMIT.
Ove naredbe se koriste za početak i završetak transakcija, postavljanje njihovih svojstava, poštovanje ograničenja za vreme transakcije, određivanje mesta u okviru transakcije koja rade kao tačke na koje će se izvođenje transakcije da vrati u slučaju poništavanja nekih akcija. Na slici *16-1* je prikazana tipična transakcija.



Slika *16-1* Osnovna SQL transakcija

SET TRANSACTION

Naredba *SET TRANSACTION* kontroliše mnoge karakteristike promene podataka, najpre read/write karakteristike i nivo izolacije (izdvajanja) transakcija. Kada se transakcija izvršava sve karakteristike će biti uključene, i one koje su navedene u naredbi i one koje nisu. Karakteristike transakcije koje nisu navedene će preuzeti podrazumevane vrednosti. Sve karakteristike transakcije nisu raspoložive ni za jednu transakciju osim prve.

Sintaksa naredbe prema standardu SQL2003 ∩ SQL sever

```
SET TRANSACTION ISOLATION LEVEL {READ COMMITTED | READ UNCOMMITTED |
                                REPEATABLE READ | SERIALIZABLE | SNAPSHOT }
```

Postavlja izolacioni nivo za sledeću transakciju u sesiji. Izolacioni nivo definiše kako će se transakcija izolovati od akcija drugih transakcija.

READ COMMITTED

Omogućava transakciji da čita vrste upisane od drugih transakcija samo kada su one bile potvrđene.

This option is the SQL Server default.

The behavior of READ COMMITTED depends on the setting of the READ_COMMITTED_SNAPSHOT database option:

- If READ_COMMITTED_SNAPSHOT is set to OFF (the default), the Database Engine uses shared locks to prevent other transactions from modifying rows while the current transaction is running a read operation. The shared locks also block the statement from reading rows modified by other transactions until the other transaction is completed. The shared lock type determines when it will be released. Row locks are released before the next row is processed. Page locks are released when the next page is read, and table locks are released when the statement finishes.
- If READ_COMMITTED_SNAPSHOT is set to ON, the Database Engine uses row versioning to present each statement with a transactionally consistent snapshot of the data as it existed at the start of the statement. Locks are not used to protect the data from updates by other transactions.

When the READ_COMMITTED_SNAPSHOT database option is ON, you can use the READCOMMITTEDLOCK table hint to request shared locking instead of row versioning for individual statements in transactions running at the READ COMMITTED isolation level.

READ UNCOMMITTED

Omogućava transakciji da čita vrste koje su bile upisane ali nisu bile potvrđene od drugih transakcija.

REPEATABLE READ

Specifies that statements cannot read data that has been modified but not yet committed by other transactions and that no other transactions can modify data that has been read by the current transaction until the current transaction completes.

Shared locks are placed on all data read by each statement in the transaction and are held until the transaction completes. This prevents other transactions from modifying any rows that have been read by the

current transaction. Other transactions can insert new rows that match the search conditions of statements issued by the current transaction. If the current transaction then retries the statement it will retrieve the new rows, which results in phantom reads. Because shared locks are held to the end of a transaction instead of being released at the end of each statement, concurrency is lower than the default READ COMMITTED isolation level. Use this option only when necessary.

SERIALIZABLE

Specifies the following:

- Statements cannot read data that has been modified but not yet committed by other transactions.
- No other transactions can modify data that has been read by the current transaction until the current transaction completes.
- Other transactions cannot insert new rows with key values that would fall in the range of keys read by any statements in the current transaction until the current transaction completes.

Range locks are placed in the range of key values that match the search conditions of each statement executed in a transaction. This blocks other transactions from updating or inserting any rows that would qualify for any of the statements executed by the current transaction. This means that if any of the statements in a transaction are executed a second time, they will read the same set of rows. The range locks are held until the transaction completes. This is the most restrictive of the isolation levels because it locks entire ranges of keys and holds the locks until the transaction completes. Because concurrency is lower, use this option only when necessary. This option has the same effect as setting HOLDLOCK on all tables in all SELECT statements in a transaction.

SNAPSHOT

Specifies that data read by any statement in a transaction will be the transactionally consistent version of the data that existed at the start of the transaction. The transaction can only recognize data modifications that were committed before the start of the transaction. Data modifications made by other transactions after the start of the current transaction are not visible to statements executing in the current transaction. The effect is as if the statements in a transaction get a snapshot of the committed data as it existed at the start of the transaction.

Except when a database is being recovered, SNAPSHOT transactions do not request locks when reading data. SNAPSHOT transactions reading data do not block other transactions from writing data. Transactions writing data do not block SNAPSHOT transactions from reading data.

During the roll-back phase of a database recovery, SNAPSHOT transactions will request a lock if an attempt is made to read data that is locked by another transaction that is being rolled back. The SNAPSHOT transaction is blocked until that transaction has been rolled back. The lock is released immediately after it has been granted.

The ALLOW_SNAPSHOT_ISOLATION database option must be set to ON before you can start a transaction that uses the SNAPSHOT isolation level. If

a transaction using the SNAPSHOT isolation level accesses data in multiple databases, ALLOW_SNAPSHOT_ISOLATION must be set to ON in each database.

A transaction cannot be set to SNAPSHOT isolation level that started with another isolation level; doing so will cause the transaction to abort. If a transaction starts in the SNAPSHOT isolation level, you can change it to another isolation level and then back to SNAPSHOT. A transaction starts the first time it accesses data.

A transaction running under SNAPSHOT isolation level can view changes made by that transaction. For example, if the transaction performs an UPDATE on a table and then issues a SELECT statement against the same table, the modified data will be included in the result set.

Kada se izda, *SET TRANSACTION* postavlja svojstva sledeće dolazeće transakcije. Zbog toga, *SET TRANSACTION* je privremena naredba, koju bi trebalo izdavati posle jedne završene transakcije a pre nego što sledeća transakcija počne (za početak transakcije i postavljanje njenih karakteristika u istom trenutku, koristiti *START TRANSACTION*). Može da bude primenjena više od jedne opcije sa ovom naredbom ali samo u jednom načinu pristupa, izolacionom nivou, a dijagnostički broj (veličina) može da se specificira odvojeno.

Izolacioni nivo transakcija specificira koji stepen izolacije transakcija ima od drugih sesija koje rade istovremeno. Nivo izolacije kontroliše:

- Da li su vrste koje su pročitane i ažurirane vašom sesijom baze podataka raspoložive drugim konkurentnim sesijama baze podataka koje se izvode.
- Da li aktivnosti ažuriranja, čitanja i upisivanja drugih sesija baze podataka mogu da utiču na vašu sesiju baze podataka.

Klauzula *ISOLATION LEVEL* kontroliše broj ponašanja i anomalija u transakciji koja se bavi konkurentnim transakcijama, naročito: Prljavo čitanje (Dirty reads), Neponovljiva čitanja (Nonrepeatable reads), Fantomske vrste (Phantom records).

Tabela *16-3*. prikazuje značaj (uticaj) različitih postavljenja izolacionog nivoa na anomalije koje su upravo navedene.

Nivo izolacije	**Prljava čitanja**	**Neponovljiva čitanja**	**Fantomske vrste**
READ UNCOMMITTED	Omogućeno	Omogućeno	Omogućeno
READ COMMITTED	Nije omogućeno	Omogućeno	Omogućeno
REPEATABLE READ	Nije omogućeno	Nije omogućeno	Omogućeno
SERIALIZABLE	Nije omogućeno	Nije omogućeno	Nije omogućeno
SNAPSHOT	Nije omogućeno	Nije omogućeno	Nije omogućeno

BEGIN TRANSACTION

MySQL, PostgreSQL i SQL Server podržavaju sličnu naredbu, *BEGIN [TRAN[SACTION]]* i njen sinonim *BEGIN [WORK]*. *BEGIN TRANSACTION* deklariše eksplicitnu transakciju, ali ona ne postavlja (određuje) izolacione nivoe.

Većina platformi baza podataka sprovodi implicitnu kontrolu transakcija, korišćenjem nečega što se obično naziva *autocommit* način. U *autocommit* načinu, baza podataka tretira svaku naredbu kao transakciju kako unutar tako i izvan nje, upotpunjenu sa implicitnom naredbom *BEGIN TRAN* i *COMMIT TRAN*.

MySQL, PostgreSQL i SQL Server, omogućavaju da se eksplicitno deklariše transakcija eksplicitno potvrdi, vrati na kontrolnu tačku ili povrati transakcija.

Većina platformi koje su prethodno pomenute rade u autocommit načinu po dogovoru. Zbog toga, dobro je nepisano pravilo da se koriste samo eksplicitno deklarisane transakcije, ako se namerava da se tako radi sa svim transakcijama u sesiji. Drugim rečima, ne treba mešati implicitno deklarisane transakcije i eksplicitno deklarisane transakcije u jednoj sesiji. Svaka transakcija koja je eksplicitno deklarisana može da bude učinjena stalnom samo sa naredbom *COMMIT*. Slično, bilo koja transakcija koja je neuspešna ili je potrebno da bude odbačena mora da bude eksplicitno poništena sa naredbom *ROLLBACK*.

Napomena: proverite da je *BEGIN* u paru sa bilo *COMMIT* ili *ROLLBACK*. Inače, DBMS neće moći da završi transakciju(e) dok ne naiđe na *COMMIT* ili *ROLLBACK*. Ne blagovremeno postavljanje *COMMIT*s (ili *ROLLBACK*s) potencijalno može da dovede do ogromnih ili nedovoljno dobro planiranih transakcija.

Dobra ideja je da se izda eksplicitni rollback ili commit posle jedne ili malog broja naredbi zato što transakcije koje duže traju mogu da zaključaju resurse, tako sprečavajući druge korisnike da pristupaju ovim resursima. Transakcije koje dugo rade ili vrlo velike grupne transakcije mogu da ispune povratne segmente ili dnevnike transakcija baze podataka.

Naredba `COMMIT`

Naredba *COMMIT* eksplicitno završava otvorenu transakciju i čini promene stalnim u bazi podataka. Transakcije mogu da budu otvorene implicitno kao deo naredbe *INSERT*, *UPDATE* ili *DELETE*, ili otvorene eksplicitno sa naredbom *START* (*BEGIN*). U bilo kom slučaju, eksplicitno izdavanje naredbe *COMMIT* će da završi otvorenu transakciju.

Za jednostavne operacije, transakcije ćete da izvršavate (to jest, SQL kod rukuje ili menja podatke i objekte u bazi podataka) bez eksplicitnog definisanja transakcije. Naravno, sve transakcije su bolje upravljane eksplicitnim njihovim zaključivanjem sa naredbom *COMMIT*. Zato što vrste pa čak i cele tabele mogu da budu zaključane za vreme trajanja transakcije, veoma je značajno da se transakcije završe koliko je moguće brže. Tako, ručno izdavanje naredbe *COMMIT* sa transakcijom može da pomogne kontroli korisnika u pitanjima konkurentnosti i problemima zaključavanja na bazi podataka. Predlažemo uvek korišćenje eksplicitnih transakcija sa *START TRAN*, na platformama baza podataka koje je podržavaju, za početak transakcije i *COMMIT* ili *ROLLBACK* za završavanje transakcija.

BEGIN TRANSACTION i COMMIT MSDN primer

```
CREATE TABLE TestTran (Cola INT PRIMARY KEY, Colb CHAR(3))
GO
BEGIN TRANSACTION OuterTran -- @@TRANCOUNT set to 1.
GO
INSERT INTO TestTran VALUES (1, 'aaa')
GO
BEGIN TRANSACTION Inner1 -- @@TRANCOUNT set to 2.
GO
INSERT INTO TestTran VALUES (2, 'bbb')
GO
BEGIN TRANSACTION Inner2 -- @@TRANCOUNT set to 3.
GO
INSERT INTO TestTran VALUES (3, 'ccc')
GO
COMMIT TRANSACTION Inner2 -- Decrements @@TRANCOUNT to 2.
-- Nothing committed.
GO
COMMIT TRANSACTION Inner1 -- Decrements @@TRANCOUNT to 1.
-- Nothing committed.
GO
COMMIT TRANSACTION OuterTran -- Decrements @@TRANCOUNT to 0.
-- Commits outer transaction OuterTran.
GO
```

Najznačajnije pitanje za razmatranje je da neke platforme baze podataka izvršavaju automatski i implicitno transakcije, dok druge zahtevaju eksplicitne transakcije. Ako podrazumevamo da platforma koristi jedan metod transakcija više od drugog i isključivo se oslanjamo na taj metod, može da dođe do problema. Pre svega kada se prelazi između platformi baza podataka trebalo bi da sledite standardni, unapred određen način adresiranja transakcija.

U dodatku završavanju jedne ili grupe operacija za manipulisanje podacima, *COMMIT* ima neke interesantne efekte na druge aspekte transakcije. Prvo, zatvara bilo koje povezane kursore. Drugo, bilo koje(e) tabela specificirana sa *ON COMMIT DELETE ROWS* (opciona klauzula naredbe *ON COMMIT DELETE ROWS*) je očišćena od podataka. Treće, sva poštovana ograničenja su proverena. Ako je bilo koje poštovano ograničenje narušeno, transakcija se ponavlja. Konačno, sva zaključavanja koja su otvorena, transakcijom se oslobađaju. Zapazimo da SQL2003 zahteva da se transakcije otvore implicitno kada se izvrši jedna od sledećih naredbi:

- ALTER
- CLOSE
- COMMIT AND CHAIN
- CREATE
- DELETE
- DROP
- FETCH
- FREE LOCATOR
- GRANT
- HOLD LOCATOR
- INSERT
- OPEN
- RETURN
- REVOKE
- ROLLBACK AND CHAIN
- SELECT

- START TRANSACTION
- UPDATE

Ako transakcija nije eksplicitno otvorena i ako je započeta sa jednom od ovih naredbi, standard zahteva da DBMS platforma otvori transakciju za vas.

Naredba ROLLBACK

```
ROLLBACK [ TRAN [ SACTION ]
    [ transaction_name | @tran_name_variable
    | savepoint_name | @savepoint_variable ] ]
```

Naredba ROLLBACK poništava transakcije do njihovog početka ili do prethodno definisane *SAVEPOINT*. *ROLLBACK* takođe zatvara sve otvorene kursore.

U dodatku poništavanja jedne operacije za manipulaciju podacima kao što su naredbe *INSERT*, *UPDATE* ili *DELETE* (ili grupa od njih), naredba *ROLLBACK* poništava transakcije do poslednje izdate naredbe *START TRANSACTION*, *SET TRANSACTION* ili *SAVEPOINT*.

ROLLBACK se koristi za poništavanje transakcija. Ona može da se koristi da poništi eksplicitno deklarisane transakcije koje počinju sa naredbom *START TRAN* ili implicitno sa transakciono-započetom naredbom. Takođe može da se koristi za poništavanje implicitnih transakcija koje počinju bez naredbe *START TRAN*. *ROLLBACK* je međusobno isključiv sa naredbom *COMMIT*.

Naredba SAVEPOINT - SAVE TRAN

SAVE TRAN [SACTION] { *savepoint_name* | *@savepoint_variable* }

Postavlja tačku čuvanja koja se naziva *savepoint_name* u okviru tekuće transakcije. Transakcije mogu da budu parcijalno povraćene u oznaci tačke prekida korišćenjem naredbe *ROLLBACK*.

Neki proizvođači omogućavaju dupla imena tačaka prekida u okviru transakcije, ali se ovo ne preporučuje ANSI standardom.

Tačke prekida su postavljene u okviru delokruga cele transakcije u kojoj su definisane. Imena tačaka prekida bi trebalo da budu jedinstvena u okviru njihovih delokruga. Osim toga, treba obazrivo proveriti korišćenje naredbi *BEGIN* i *COMMIT* , zato što slučajno postavljanje naredbe *BEGIN* ranije (prerano) ili naredbe *COMMIT* prekasno može da ima dramatični uticaj na način kako su transakcije upisivane u bazu podataka. Uvek treba koristiti jasna imena za tačke čuvanja zato što će kasnije da budu pozivana u programima. Na slici *16-3* je prikazana upotreba naredbe *SAVEPOINT*, kao i ostalih naredbi koje se koriste za upravljanje transakcijama.

PRIMERI

```
-- Kreiranje tabela za demostriranje transakcija
create table magacin (
    proizvod smallint primary key,
    naziv char(20),
    cena decimal(10,2),
    stanje int)

create table racun (
    broj_racuna smallint primary key,
    datum datetime default getdate(),
```

```
    kupac smallint)

create table stavka_racuna (
    broj_racuna smallint references racun(broj_racuna),
    proizvod smallint references magacin(proizvod),
    kolicina int)
```

Primer 1.
Upisuju se podaci u tabeli Magacin. Korišćenem sistemske funkcije @@ERROR, se proverava ispravnost upisa. Ako je sve ispravno podaci se upisuju, ako nisu, transakcija se poništava do sledeće tačke čuvanja podataka.

```
BEGIN TRANSACTION t1

INSERT INTO magacin VALUES(100, 'proizvod1', 100.50, 40)
IF @@ERROR <> 0 ROLLBACK
save transaction tr1
INSERT INTO magacin VALUES(101, 'proizvod2', 110.50, 50);
IF @@ERROR <> 0 ROLLBACK TRANSACTION tr1
save transaction tr2
INSERT INTO magacin VALUES(102, 'proizvod3', 120.50, 50)
IF @@ERROR <> 0 ROLLBACK TRANSACTION tr2
save transaction tr3
INSERT INTO magacin VALUES(103, 'proizvod4', 130.50, 50)
IF @@ERROR <> 0 ROLLBACK TRANSACTION tr3
save transaction tr4
INSERT INTO magacin VALUES(104, 'proizvod5', 140.50, 50)
IF @@ERROR <> 0 ROLLBACK TRANSACTION tr4
save transaction tr5
INSERT INTO magacin VALUES(105, 'proizvod5', 150.50, 50)
IF @@ERROR <> 0 ROLLBACK TRANSACTION tr5
save transaction tr6
INSERT INTO magacin VALUES(106, 'proizvod6', 160.50, 50)
IF @@ERROR <> 0 ROLLBACK TRANSACTION tr6
save transaction tr7
INSERT INTO magacin VALUES(107, 'proizvod7', 170.50, 50)
IF @@ERROR <> 0 ROLLBACK TRANSACTION tr7

COMMIT TRANSACTION t1
```

Primer 2.
Ovaj primer se odnosi na najnerestriktiviniji nivo zaključavanja transakcija *READ UNCOMMITTED*. Istovremeno se pokreću dve transakcije. Prva transakcija vrši ažuriranje prvih pet vrsta, odnosno vraća stanje na 0. Koriste se naredba *WAITFOR DELAY* koja blokira izvršavanje transakcije za specificirani period vremena i sistemska funkcija *@@ROWCOUNT*, koja vraća broj vrsta iz poslednje naredbe.

```
-- transakcija 1
SELECT TOP 5 Proizvod, Stanje FROM magacin ORDER BY Proizvod
BEGIN TRAN
UPDATE magacin SET Stanje = 0
SELECT TOP 5  Proizvod, Stanje FROM Magacin ORDER BY Proizvod
WAITFOR DELAY '00:00:05'
ROLLBACK TRAN
SELECT TOP 5 Proizvod, Stanje FROM magacin ORDER BY Proizvod

-- transakcija 2
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

```
PRINT 'sada se promene vide…'
SELECT TOP 5  Proizvod, Stanje FROM Magacin
WHERE Stanje = 0
ORDER BY Proizvod
IF @@ROWCOUNT>0 BEGIN
WAITFOR DELAY '00:00:05'
PRINT '… sada se promene ne vide '
SELECT TOP 5  Proizvod, stanje FROM Magacin
WHERE Stanje = 0
ORDER BY Proizvod
END
```

Poruke iz druge transakcije:

```
'sada se promene vide…'
Proizvod Stanje
-------- -----------
100      0
101      0
102      0
103      0
104      0


… sada se promene ne vide
Proizvod Stanje
-------- -----------
```

Druga transakcija je nastupila, pošto je izvršena naredba *UPDATE* u prvoj transakciji, tako da ove promene vidi druga transakcija. Pošto je usledilo poništavanje promene u prvoj transakciji, sada druga transakcija vidi prvobitno stanje, odnosno stanje bez promena.

Primer 3.
Primer se odnosi na nivo izolacije *READ COMMITTED*, koji ne dozvoljava prljavo čitanje. Prva transakcija čita podatke iz tabele Magacin, jednom na početku i drugi put posle 5 sekundi i na kraju vrši poništavanje transakcija. U međuvremenu je druga transakcija izvršila promenu količine na jednom proizvodu.

```
-- Transakcija 1
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRAN
PRINT 'sada se promene ne vide…'
SELECT TOP 5 Proizvod, Stanje FROM magacin ORDER BY Proizvod
WAITFOR DELAY '00:00:05'
PRINT '… sada se promene vide '
SELECT TOP 5 Proizvod, Stanje FROM Magacin ORDER BY Proizvod
GO
ROLLBACK TRAN
-- Transakcija 2
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
UPDATE Magacin SET Stanje = 41 WHERE Stanje = 40
```

```
sada se promene ne vide…
Proizvod Stanje
-------- -----------
100      40
101      50
102      50
103      50
104      50

… sada se promene vide
```

```
Proizvod Stanje
-------- -----------
100      41
101      50
102      50
103      50
104      50
```

Primer 4.

Primer se odnosi na *REPEATABLE READ*, koji omogućava fantomske vrste. Prva transakcija prikazuje podatke u različitim fazama izvršavanja i na kraju poništavanje, a druga vrši upisivanje nove vrste u tabeli Magacin.

```
-- Transakcija 1
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRAN
PRINT 'Ne vadim ništa iz rukava...'
SELECT TOP 5 Proizvod, Stanje FROM Magacin ORDER BY Stanje
WAITFOR DELAY '00:00:05'
PRINT '...izuzev ovog zeca'
SELECT TOP 5 Proizvod, Stanje FROM Magacin ORDER BY Stanje
GO
ROLLBACK TRAN
-- Transakcija 2
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
INSERT Magacin VALUES(108, 'proizvod8', 180.50, 30)
```

```
Ne vadim ništa iz rukava...
Proizvod Stanje
-------- -----------
100      41
101      50
102      50
103      50
104      50

...izuzev ovog zeca'
Proizvod Stanje
-------- -----------
108      30
100      41
101      50
102      50
103      50
```

Primer 5.

Primer se odnosi na *SERIALIZABLE*, koji ne dozvoljava ni jednu od pojava nekonzistentnosti.

```
-- Transakcija 1
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRAN
PRINT 'Ne vadim ništa iz rukava…'
SELECT TOP 5 Proizvod, Stanje FROM Magacin ORDER BY Stanje
WAITFOR DELAY '00:00:05'
PRINT '…izuzev ovog zeca'
SELECT TOP 5 Proizvod, Stanje FROM Magacin ORDER BY Stanje
GO
ROLLBACK TRAN
-- Transakcija 2
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
INSERT Magacin VALUES(108, 'proizvod8', 180.50, 30)
```

```
Ne vadim ništa iz rukava...
Proizvod Stanje
-------- -----------
100      41
105      50
104      50
103      50
102      50

(5 row(s) affected)

...ili iz mog šešira
Proizvod Stanje
-------- -----------
100      41
105      50
104      50
103      50
102      50

(5 row(s) affected)

(1 row(s) affected)
```

Prvo će se izvršiti prva transakcija, pa potom i druga.

Primer 6.

Vrši se upisivanje u tabelama Račun i Stavka_računa. U slučaju greške transakije se poništavaju.

```
BEGIN TRANSACTION prodaja
INSERT INTO Racun VALUES (1,' ',111)
IF @@ERROR != 0
BEGIN
   ROLLBACK TRANSACTION prodaja
   PRINT 'Nije dozvoljeno upisivanje u RACUN! Transakcija je poništena'
   RETURN
END
UPDATE MAGACIN SET Stanje = Stanje - 10 WHERE Proizvod = 100
IF @@ERROR != 0
BEGIN
   ROLLBACK TRANSACTION prodaja
   PRINT 'Nije dozvoljeno upisivanje u MAGACIN! Transakcija je poništena'
   RETURN
END
INSERT INTO STAVKA_RACUNA VALUES (1, 100, 10)
IF @@ERROR != 0
BEGIN
   ROLLBACK TRANSACTION prodaja
   PRINT 'Nije dozvoljeno upisivanje u STAVKA_RACUNA! Transakcija je
poništena'
   RETURN
END
COMMIT TRANSACTION prodaja
IF @@ERROR != 0
   PRINT 'Transakcija ne može da se potvrdi'
ELSE
   PRINT 'Transakcija potvrđena'
```

Primer 7.

Primer se odnosi na ugnježdenu transakciju.

```
SELECT 'Pre BEGIN TRAN', @@TRANCOUNT
BEGIN TRAN
	SELECT 'Posle BEGIN TRAN', @@TRANCOUNT
	DELETE stavka_racuna
	BEGIN TRAN ugnježđena
		SELECT 'Posle BEGIN TRAN ugnježđena ', @@TRANCOUNT
		DELETE racun
	COMMIT TRAN ugnježđena -- Samo vrši smanjenje  @@TRANCOUNT
	SELECT 'After COMMIT TRAN nested', @@TRANCOUNT
ROLLBACK TRAN
SELECT 'Posle ROLLBACK TRAN',@@TRANCOUNT
IF @@TRANCOUNT>0 BEGIN
		-- COMMIT TRAN -- ne koristite to ovde zbog ROLLBACK
		SELECT 'Posle COMMIT TRAN', @@TRANCOUNT
		SELECT TOP 5 * FROM stavka_racuna
END
```

Pre BEGIN TRAN 1

Posle BEGIN TRAN 2

Posle BEGIN TRAN ugnježđena 3

After COMMIT TRAN nested 2

Posle ROLLBACK TRAN 0

broj_racuna proizvod kolicina
----------- -------- -----------
1	100	10