Oporavak od kvara

Kompjuterski sistem, kao i bilo koji drugi električni ili mehanički uređaj podložan je kvarovima. Postoji mnogo razloga koji dovode do kvara: kvar diska (gubi se sva informacija), prekid napajanja (gubi se sadržaj operativne memorije), kvar zbog softverskih grešaka (rezultati obrade mogu biti pogrešni pa baza može ući u nekonzistentno stanje).

Postoji nekoliko tipova skladištenja informacije:

1. privremeno – informacija smještena u ovakvo skladište najčešće ne može da preživi padove sistema (OM i keš).
2. polustabilno – informacija smještena na uređajima ovog tipa najčešće preživljava padove sistema. Primjeri takvog smještaja su diskovi i magnetne trake. Diskovi se koriste za interaktivno smještanje, dok se trake više koriste za arhiviranje. Diskovi su pouzdaniji od glavne memorije, ali manje pouzdani od magnetnih traka. I jedno i drugo je podložno kvarovima i oba mogu rezultirati gubitkom informacije (na primjer, kod diska može da se slomi glava).
3. stabilno – informacija smještena na ovakvim uređajima nikad se ne gubi. Da bi se realizovali takvi uređaji potrebno je vršiti replikaciju informacija na više polustabilnih uređaja, najčešće su to diskovi koji imaju nezavisne režime kvara. Ažuriranje informacije se mora vršiti na strogo kontrolisan način.

U računarskom sistemu mora postojati postupak (*šema*) za oporavak koji je odgovoran za detekciju kvarova i za restauraciju baze u konzistentno stanje koje je postojalo prije nastanka kvara. Postoje razne klasifikacije kvarova. Najbezazleniji su oni koji ne izazivaju gubitak informacije. Ostale je znatno teže otkloniti. Mogu se izdvojiti sljedeći tipovi kvara:

1. logičke greške – program ne može da nastavi izvršavanje zbog nekih unutrašnjih uslova, recimo, loš unos, podaci nisu pronadjeni, over flow, predjeni limiti na nekim resursima i dr.
2. sistemske greške – sistem je ušao u neželjeno stanje, npr. ćorsokak i kao rezultat toga program ne može nastaviti svoje normalno izvršavanje. Ipak, program se može izvršiti kasnije iznova.
3. padovi sistema – su kvarovi u hardveru koji rezultuju gubitkom sadržaja glavne memorije, međutim sadržaj polustabilne memorije ostaje netaknut.
4. kvar diska – blokovi na disku gube svoj sadržaj zbog npr. loma glave ili greške u transferu podataka. Najveći broj šema za oporavak radi na greškama prva 3 tipa.

Kod diskova input/output operacije se obavljaju u blokovima. To je zato što operativna memorija i diskovi, s jedne strane, i baza podataka, s druge strane, su podijeljeni na blokove. Pri tome, blokovi koji se nalaze na disku su fizički blokovi, a oni u memoriji su baferski blokovi. Postoje dvije osnovne operacije za premještanje podataka: *input(X),* koja vrši transfer fizičkog bloka na kome se nalazi podatak *X* u glavnu memoriju, i *output (X),* koja prebacuje baferki blok u kome se nalazi podatak *X* na disk i zamjenjuje odgovarajući fizički blok na disku. Programi se najčešće obraćaju SUBP naredbama *read* i *write*.

Naredba ***read (X, X_i)*** dodjeljuje vrijednost podatka *X* lokalnoj promjenljivoj X_i. Ova operacija se najčešće realizuje na sljedeći način:

1. Ako blok na kome se nalazi X nije u operativnoj memoriji izvrši *input (X)*.
2. Dodijeli promjenljivoj X_i vrijednost *X* iz baferskog bloka.

Naredba ***write (X, X_i)*** dodjeljuje vrijednost lokalne promjenljive X_i podatku *X* u baferskom bloku. Operacija se izvršava u dvije faze:

1. Ako blok na kome se *X* nalazi nije u operativnoj memoriji tada *input (X)*.
2. Dodijeli *X*–u vrijednost X_i.

Obije operacije zahtijevaju prebacivanje bloka sa diska u operativnu memoriju, ali ne zahtijevaju transfer iz memorije na disk. Baferski blok se upisuje na disk ili iz razloga što OS, tj. upravljač memorijom želi da iskoristi taj memorijski prostor za druge namjene, ili zato što SUBP želi da promjene nastale na baferskom bloku primijeni i na fizički blok, tj. na disk. Ova druga situacija se naziva *forsirani izlaz* baferskog bloka X i to se vrši operacijom *output (X)*.

Ako program želi da pristupi podacima *X* prvi put, on mora da uradi *read(X, X_i)*. Sva ažuriranja na *X* se vrše na X_i, a kada program poslednji put pristupi podacima on mora izvršiti *write(X, X_i)*, zato što blok na kome se nalazi *X* može sadržati druge podatke kojima se trenutno pristupa. Znači, stvarni upis na disk se odvija kasnije. Ukoliko se desi pad sistema posle *write(X, X_i)* naredbe, a prije *output(X)* nova vrijednost *X*–a će biti izgubljena zato što nikad nije upisana na disk.

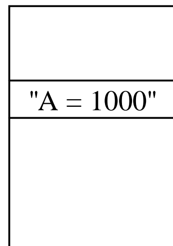
Posmatrajmo bazu podataka za banku koja sadrži nekoliko računa i nekoliko programa koji pristupaju tim računima i ažuriraju ih. Neka program p prebacuje 50$ sa računa A na račun B.

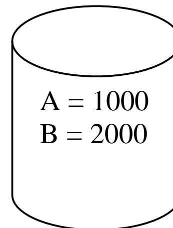
Program:

read (A, a_1)
$a_1 = a_1 - 50$
write (A, a_1)
read (B, b_1)
$b_1 = b_1 + 50$
write (B, b_1)

Pretpostavimo da su prije izvršavanja programa računi A i B imali A = 1000$ i B = 2000$, i da OM – ja sadrži baferski blok za A, a ne i za B.

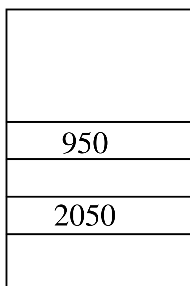
a_1 = 1000 read (A, a_1)

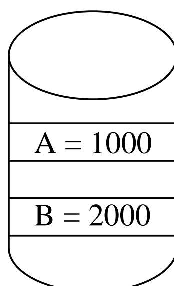




OM BP

Kada se izvršava *read (B, b_1)* sistem mora prebaciti fizički blok koji sadrži *B* u memoriju i tek nakon toga promjenljivoj b_1 pridružiti 2000. Nakon izvršenih operacija čitanja i pisanja imamo sljedeće stanje:





output (A) i *output (B)* još nisu izvršene i sadržaj memorije je različit u odnosu na sadržaj diska. Mogu se desiti neke greške zbog kojih se ovaj program neće izvršiti u potpunosti, npr. računi A + B ne postoje u bazi. Sistemske greške: greška u bitu parnosti, pad sistema (gubi se sadržaj memorije, nove vrijednosti se gube), kvar diska (nekoliko fizičkih blokova su oštećeni što rezultira čitanjem pogrešnih podataka, npr. kada se izvršava *input(A)*). Poslije izvršenja našeg programa stanje bi trebalo da bude *A = 1000, B = 2000.*

Pretpostavimo da se tokom izvršenja desila greška i izvršavanje je prekinuto. Pretpostavimo da se taj kvar desio nakon *output (A)* a prije izvršenja *output (B)* operacije. U ovoj situaciji stanja računa su *A = 950, B = 2000*, pa je došlo do gubitka 50$ i ovakvo stanje se naziva **nekonzistentno**. Moramo osigurati da se takve nekonzistentnosti ne vide. Postoji trenutak kada se baza mora naći u takvom stanju koje će biti zamijenjeno konzistentnim stanjem.

Transakcija je programska jedinica čije izvršenje čuva konzistentnost baze. Ukoliko se baza prije izvršenja transakcije nalazi u konzistentnom stanju, nakon izvršenja mora se, takođe, nalaziti u konzistentnom stanju. Da bi se ovo obezbijedilo, transakcija mora biti atomična operacija, tj. ili su sve instrukcije u njoj izvršene ili nije nijedna. Zadatak programera je da obezbijedi konzistentnost baze. Transakcija je dio programa koji pristupa nekim podacima i mijenja ih. Ukoliko transakcija ne završi svoje izvršavanje, da bi se sačuvalo svojstvo atomičnosti, transakcija ne smije imati nikakvog efekta na

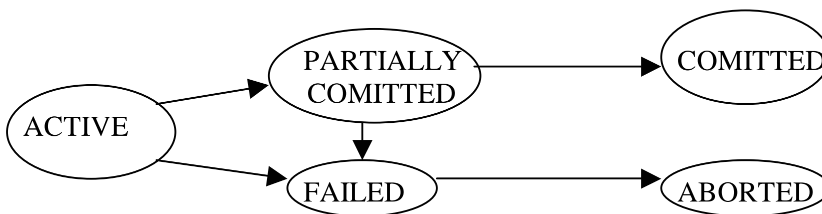
stanje baze. Zato stanje baze podataka mora biti vraćeno na stanje koje je bilo prije početka izvršavanja ove transakcije. Za takvu transakciju kažemo da je **otkazana**, to jest izvršena operacija *ROLLBACK*. Odgovornost za otkazivanje transakcije ima šema za oporavak, tj. SUBP. Transakcija koja uspješno završi svoje operacije naziva se **potvrdjenom** ili *COMMITED* transakcija.

START TRANSACTION (ili BEGIN TRANSACTION)
transakcija
COMMIT TRANSACTION

ROLLBACK TRANSACTION u slučaju greške u izvršenju.

Transakcija može biti u nekom od sljedećih stanja:

1. aktivna (*ACTIVE*),
2. parcijalno potvrdjena (*PARTIALLY COMMITED*),
3. *FAILED* – nakon konstatovanja činjenice da normalno izvršenje nije moguće,
4. *ABORTED* – nakon što se desio *ROLLBACK* i baza je restaurirana u stanje prije početka izvršavanja transakcije,
5. *COMMITED* – nakon uspješnog izvršavanja kada se izvrši poslednja naredba.



Stanja transakcije

Transakcija počinje u aktivnom stanju a kada izvrši svoju poslednju instrukciju ulazi u *PARTIALLY COMITTED* stanje. U ovom trenutku, ona je završila svoje izvršavanje, ali i dalje je moguće da će ona biti otkazana zato što stvarni izlaz., tj. upis na disk, još nije izvršen, a može se desiti hardverski kvar prije izvršenja.

Transakcija ulazi u *FAILED* stanje, nakon što se utvrdilo da se ne može više nastaviti njeno izvršenje, npr. zbog hardverske ili logičke greške. Za takvu transakciju mora se izvršiti *ROLLBACK*. Tada ona ulazi u *ABORTED* stanje. U ovom trenutku sistem ima dvije opcije:

1. Restartovati transakciju – ovo je moguće kada se desila neka hardverska ili softverska greška koja nije nastala zbog unutrašnje logike transakcija. Restartovana transakcija se smatra novom transakcijom.
2. Ubijanje transakcije – ovo se najčešće dešava zbog neke unutrašnje logičke greške koja može biti otklonjena jedino ispravkama u programu.

Transakcija ulazi u *COMMITED* stanje ako se nalazi u *PARTIALLY COMMITED* stanju i ako je garantovano da nikad neće doći do otkaza. Postoje razni mehanizmi za sprovodjenje ovog zahtjeva, najčešće se koriste sljedeća 3:

1. INCREMENTALNI LOG sa odloženim upisom,
2. INCREMENTALNI LOG sa direktnim upisom,
3. SHADOW PAGING.

Napomena: Moramo biti oprezni kada imamo "vidljive" spoljašne upise, tj. ona pisanja koja ne mogu biti izbrisana, na ekranu ili na štampaču. Neki sistemi ne dozvoljavaju izvršenje takvih pisanja prije COMITTED stanja.

Inkrementalni log sa odloženim upisom

Podsjetimo se primjera sa bankovnim računima. Prebacujemo 50$ sa računa A (1000$) na račun B (2000$). Pretpostavimo da se pad sistema desio posle operacije *output(A)*, a prije operacije *output(B)*. Kako je sadržaj memorije izgubljen, moguće je ponovo izvršiti transakciju (što je loše) ili ne izvršiti transakciju ponovo (i ovo je loše). U oba slučaja dobijamo nekonzistentno stanje. Problem je u tome što je dozvoljen upis iako nismo bili sigurni da će se transakcija izvršiti. Da bi se ovaj problem otklonio mora postojati odredjeni mehanizam koji će održati konzistentno stanje baze.

Tokom izvršenja transakcije svi upisi se odlažu dok transakcija ne udje u parcijalno završeno stanje i sve izmjene se upisuju u specijalni fajl koji održava SUBP i koji se naziva *log fajl* (žurnal). Kada transakcija udje u stanje parcijalno završeno, informacija koja je povezana sa ovom transakcijom iz log fajla se koristi da bi se izvršio upis. Ako se sistem sruši prije nego što transakcija završi svoje izvršavanje onda se informacija u log fajlu ignoriše. Izvršavanje transakcije se izvodi na sljedeći način. Na početku transakcije T_i zapis < T_i, *starts* > se upisuje u log fajl. Nakon toga, svaka naredba oblika $write(X,X_j)$ u transakciji T_i rezultuje upisom novog zapisa u log fajl i taj zapis sadrži ime transakcije, naziv podatka i novu vrijednost. Na kraju, kad T_i udje u pacijalno završeno stanje zapis < T_i, *commit* > se upisuje u log. Kako se kvar može desiti i tokom izvršavanja odloženih upisa, tj. prilikom čitanja log fajla, prije početka ovih upisa moramo obezbijediti da svi log zapisi budu upisani na stabilnu memoriju. Tek kada se to izvrši, stvarno upisivanje može da se izvrši i tada transakcija ulazi u commited stanje.

T_0:
read (A,a_i)
$a_i = a_i - 50$
WRITE (A,a_i)
read (B,b_i)
$b_i = b_i + 50$
WRITE (B,b_i)

T_1:
read (C,c_1)
$c_1 = c_1 - 100$
WRITE (C,c_1)

Pretpostavimo da su stanja na računima A, B i C na početku bila 1000$, 2000$ i 700$.

Postoje različiti redosledi po kojima stvarni upisi mogu da se dese i u bazi podataka i u log fajlu. Pod pretpostvakom da je T_0 izvršeno prije T_1, dio log fajla koji se odnosi na ove transakcije je:

< T_0, starts >
< T_0, A, 950 >
< T_0, B, 2050 >
< T_0, commit > .
< T_1, start >
< T_1, C, 600 >
< T_1, commit >

Log	Baza
< T_0, start >	
< T_0, A, 950 >	
< T_0, B, 2050 >	
< T_0, commit >	
	A = 950
	B = 2050
< T_1, start >	
< T_1, C, 600 >	
< T_1, commit >	

Koristeći log, sistem može pravilno obraditi svaki kvar koji nije rezultirao gubitkom informacije na stabilnoj memoriji. Ova šema za oporavak sistema koristi jednu operaciju, *REDO*(T_i) koja postavlja vrijednosti svih podataka koje je unijela transakcija T_i na nove vrijednosti. Skup podataka koje je mijenjala transakcija T_i, kao i njihove vrijednosti, mogu se naći u logu. Izvršenje redo operacije mora biti idempotentno, tj. da više izvršavanja bude ekvivalentno jednom izvršenju. Ovo se zahtijeva da bi se garantovalo pravilno ponašanje i u slučaju da se kvar desi tokom procesa oporavka. Nakon što se desio kvar podsistem za oporavak konsultuje žurnal da odredi koje transakcije treba ponovo izvršiti. Transakciju treba ponovo izvršiti ako log sadržaj sadrži zapise < T_i, *start* > i < *Ti, commit* >.

Neka se transakcije T_0 i T_1 izvršavaju jedna iza druge i neka se kvar desio prije završetka transakcija. Pogledajmo kako funkcioniše mehanizam oporavka.

1° Pretpostavimo da se kvar desio tačno posle upisa u log zapisa za naredbu WRITE (B,b_1) u stabilnu memoriju. Log u ovom trenutku izgleda ovako:

< T_0, START >
< T_0, A, 950 >
< T_0, B, 2050 >

Kada se sistem ponovo podigne nikakva procedura oporavka nije potrebna jer ništa nije ni upisivano u bazu.

2° Pretpostavimo da se kvar desio odmah nakon što je log zapis za naredbu WRITE (C, c_1) upisan u log. Log fajl izgleda ovako:

< T_0, START >
< T_0, A, 950 >
< T_0, B, 2050 >
< T_0, COMMIT >
< T_1, START >
< T_1, C, 600 >

Kada se sistem podigne, kako transakcija T_0 ima i *START* i *COMMIT* u logu, ona treba opet da se izvrši *(REDO)*. Nakon ovoga, računi A i B imaju vrijednost 950 i 2050, a račun C ostaje nepromijenjen.

3° Pretpostavimo da se kvar desio nakon što je log zapis < T_1, COMMIT > upisan u stabilnu memoriju. Kada se sistem podigne i T_0 i T_1 imaju i *START* i *COMMIT* i zato se mora izvršiti i *REDO(T_0)* i *REDO(T_1)*. Nakon ovoga, vrijednosti na računima su 950, 2050 i 600.

Na kraju, pretpostavimo da se novi kvar desio tokom oporavka od prvog kvara. Neke promjene su se možda desile u bazi kao rezultat redo operacije, ali može biti da neke promjene nisu izvršene. Kada se sistem podigne, oporavak ide potpuno isto kao u prethodnim primjerima.

Znači, za svaki zapis <T_i , *COMMIT*> nadjen u logu operacija *redo (T_i)* se izvršava. Drugim riječima, proces oporavka je pokrenut iznova. Kako redo upisuje u bazu nezavisno od trenutnih vrijednosti u bazi, rezultat drugog izvršenja operacije redo je isti kao da je ova operacija uspjela i prvi put.

Inkrementalni log sa direktnim upisom

Kod ove tehnike sve izmjene se direktno upisuju u bazu, a u isto vrijeme vodi se tzv. inkrementalni log o svim promjenama u sistemu. Kada se desi kvar informacija u logu se koristi za restauriranje prethodno konzistentnog stanja baze. Prije nego što transakcija počne svoje izvršavanje zapis < T_i, *START* > se upisuje u log. Tokom izvršavanja, *prije* svake WRITE (X, X_j) operacije u transakciji T_i, upisuje se novi zapis u log. Svaki takav zapis se sastoji od sljedećih elemenata: imena transakcije, naziva podatka, stare i nove vrijednosti podatka. Kada se transakcija T_i parcijalno završi zapis < T_i, *COMMIT* > se upisuje u log. Kako se informacija u log fajlu koristi za rekonstruisanje stanja baze ne smijemo dozvoliti da se stvarni upisi izvrše prije nego što je log zapis upisan na stabilnu memoriju.

Pretpostavimo da su T_0 i T_1 izvršene jedna za drugom.

Log fajl izgleda ovako:

< T_0, START >
< T_0, A, 1000, 950 >
< T_0, B, 2000, 2050 >
< T_0, COMMIT >
< T_1, START >
< T_1, C, 700, 600 >
< T_1, COMMIT >

Koristeći ovaj log fajl sistem može obraditi bilo koji kvar koji ne rezultuje gubitkom informacije u stabilnoj memoriji. Ova šema za oporavak koristi dvije procedure:

1. *UNDO(T_i),* koja vraća vrijednosti svih podataka koje je izmijenila T_i na stare vrijednosti,
2. *REDO(T_i),* koja postavlja vrijednost svih podataka koje je mijenjala transakcija T_i na nove vrijednosti.

Skup podataka koje je T_i mijenjala, kao i nove i stare vrijednosti mogu se naći u logu. *UNDO* i *REDO* operacije moraju biti idempotentne da bi se garantovalo korektno ponašanje ako se kvar desi i u toku oporavka. Nakon što se kvar desio, sistem konsultuje log da odredi koje transakcije treba iznova izvršiti, a koje poništiti. Transakciju T_i treba poništiti ako log sadrži zapis < *T_i, START* >, a ne sadrži zapis < *T_i, COMMIT* >. Transakciju treba iznova izvršiti, ako log sadrži i < *T_i, START* > i < *T_i, COMMIT* >. Mehanizam oporavka funkcioniše na sljedeći način.

1° Pretpostavimo da se kvar desio odmah nakon što je log zapis za naredbu WRITE (B,b_1) upisan u stabilnu memoriju. U trenutku kvara log fajl izgleda ovako:

< T_0, START >
< T_0, A, 1000, 950 >
< T_0, B, 2000, 2050

Kada se sistem oporavi on nalazi da T_0 ima *START*, a nema *COMMIT*. Ovo znači da se T_0 mora poništiti, tj. mora se izvršiti operacija *UNDO(T_0)*. Kao rezultat ove operacije vrijednosti na računima A i B se vraćaju na stare vrijednosti.

2° Pretpostavimo da se kvar desio odmah nakon upisivanja u log zapisa za naredbu WRITE (C,c_1) u stabilnu memoriju. Log fajl u ovom slučaju izgleda:

< T_0, START >
< T_0, A, 1000, 950 >
< T_0, B, 2000, 2050 >
< T_0, COMMIT >
< T_1, START >
< T_1, C, 700, 600 >

Kada se sistem podigne dvije stvari moraju da se urade: operacija *UNDO(T_1)* mora da se izvrši zato ztšo u log fajlu nema zapisa < *T_i, COMMIT* > i *REDO(T_0)* zato što T_0 sadrži i *START* i *COMMIT*.
Na kraju postupka za oporavak, vrijednosti računa A, B, i C su 950, 2050 i 700.

3° Pretpostavimo da se kvar desio nakon što je < T_1, COMMIT > upisano. U ovom slučaju mora se izvršiti *REDO(T_0)* i *REDO(T_1)* i stanje na računima će biti 950, 2050 i 600.

CHECKPOINTS

Kada se desi kvar sistema neophodno je konsultovati log da bi se odredilo koje transakcije treba poništiti, koje izvršiti ponovo i cio log se mora pretražiti. Tu imamo dva problema:

1. proces pretraživanja zahtijeva puno vremena
2. većina transakcija za koje treba uraditi *REDO* su već stvarno upisale svoje izmjene u bazu i moraju biti ponovo izvršene. Njihovo ponovno izvršavanje neće izazvati nikakvu štetu, ali proces oporavka traje duže.

Da bi se smanjio suvišni posao uvodi se pojam checkpoint – a. Tokom izvršavanja sistem održava log fajl kao što je ranije rečeno, međutim, povremeno vrši sljedeći niz akcija:

1. upisuje sve log zapise koji se trenutno nalaze u glavnoj memoriji na stabilnu,
2. upisuje sve baferske blokove na disk,
3. upisuje u log zapis < checkpoint >.

Sa mehanizmom chekpoint-a mogu se modifikovati prethodne šeme za oporavak. Nakon što se desio kvar, sistem za oporavak istražuje svoj log fajl i nalazi poslednju transakciju koja je počela izvršavanje prije poslednjeg chekpoint-a. Takva transakcija se može naći pretraživanjem unazad po log fajlu do prvog chekpoint-a, a nakon toga prvi zapis <T_i, *START*> je traženi. Kada je takva transakcija identifikovana *REDO* i *UNDO* se trebaju izvršiti samo na toj transakciji T_i i svim transakcijama T_j koje su počele izvršavanje poslije nje. Za svaku transakciju T_K za koju je < T_K, *COMMIT* > u logu izvrši *REDO*, a za svaku koja nema < T_K, *COMMIT* > izvrši se *UNDO*. Očigledno je da undo ne mora biti izvršena kada se koristi log sa odloženim upisom. Ako imamo T_0,- - T_{100} po ovom redosledu i poslednji checkpoint se odigrao tokom T_{56}. Tada samo T_{56}, T_{57},- -, T_{100} trebamo razmatrati, ponovo izvršiti ili poništiti.

Upravljanje baferom

U modelu transakcija kojeg smo do sada koristili pretpostavljalo se da kad god se izvrši *input (X)* operacija ima prostora u memoriji za taj blok, što u realnim sistemima ovo ne mora biti slučaj. OS koji koriste virtuelnu memoriju ovu situaciju rešavaju na taj način što izvrše operaciju prisilnog upisa tog bloka na disk. Ova strategija je u konfliktu sa zahtjevima šeme za oporavak, a postoji i dodatni zahtjev da svi log zapisi vezani za neki blok budu upisani na disk prije samog bloka.

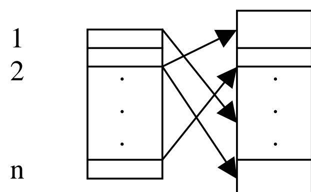
Pretpostavimo da je stanje loga < T_0, *START* >, < T_0, *A, 1000, 950* > i da transakcija T_0 izvršava naredbu READ (B,b_1). Pretpostavimo da fizički blok na kome se nalazi B nije u memoriji i da je ona puna. Takođe, pretpostavimo da blok na kome se nalazi A je izabran da bude prisilno upisan na disk. Ako se kvar desi nakon tog upisa, stanja računa A, B i C su 950, 2000 i 700, što je nekonzistentno. Log zapis < T_0, *A, 1000, 950* > mora biti upisan prije samog bloka na kome je A na disk.

Ovaj primjer dovodi do pravila koje mora da važi za upravljanje baferom u SUBP – u: prije izvršavanja output operacije na bloku u glavnoj memoriji svi log zapisi koji se odnose na podatke tog bloka moraju biti prisilno upisani na stabilnu memoriju, ukoliko već nisu tamo. Kada se log zapisi upisuju u stabilnu memoriju treba upisivati cijele blokove log zapisa zato što tipično jednako košta upisivanje cijelog bloka ili dijela bloka. Ovo smanjuje rad oko operacija upisivanja loga zato što upisivanje jednog log zapisa uzrokuje upisivanje drugih log zapisa bez dodatnih troškova.

SHADOW PAGING

To je alternativni mehanizam u odnosu na log fajlove i nekad može zahtijevati manje pristupa disku, ali ima i nedostatke.

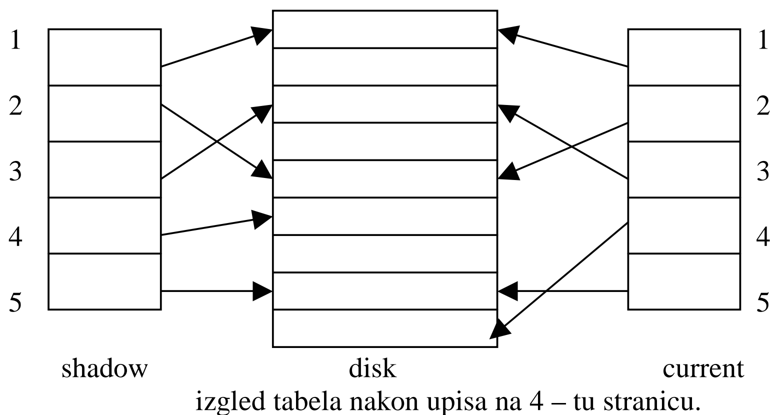
Baza je izdijeljena na blokove fiksne veličine koje nazivamo stranicama. Termin stranica je iz operativnih sistema zato što se ovdje koristi sličan mehanizam kao kod upravljanja memorijom. Neka imamo n stranica označenih 1...n i one ne moraju biti smještene ni u kakvom posebnom redosledu. Mora postojati mehanizam za nalaženje konkretne stranice i to se izvodi putem tabele stranica koja ima n redova. Svaki red sadrži pokazivač na stranicu na disku.



Osnovna ideja iza mehanizma shadow paginga je da se održavaju dvije tabele stranica tokom života transakcije – *current* i *shadow* (tekuća i u sjenci) tabela stranica. Kada transakcija počne da se izvršava obije tabele su identične. Tabela stranica u sjenci se nikada ne mijenja tokom trajanja transakcije, a tekuća tabela stranica se može promijeniti kada transakcija izvršava WRITE operaciju. Sve input i output operacije koriste tekuću tabelu stranica da lociraju stranice baze na disku. Pretpostavimo da transakcija

vrši *WRITE(X, X_j)* operaciju i da se X nalazi na i – toj stranici. Write operacija se izvršava na sljedeći način:

1. ako i – ta stranica nije u memoriji izvrši *input(X),*
2. ako je ovo prva WRITE operacija izvedena na i – toj stranici od strane ove transakcije onda modifikovati tekuću tabelu stranica ovako:
    a) nadji neiskorišćenu stranicu na disku,
    b) izbriši stranicu nadjenu u koraku 2°a) iz liste slobodnih stranica,
    c) modifikuj tekuću tabelu stranica tako da i – ti red ukazuje na stranicu pronadjenu u koraku 2°a).
3. Dodijeli vrijednost promjenljive X_j podatku X u baferskoj stranici.



izgled tabela nakon upisa na 4 – tu stranicu.

Kada se transakcija završi, tekuća tabela stranica se upisuje u stabilnu memoriju, tekuća tabela stranica postaje nova tabela stranica u sjenci i sljedeća T može da počne izvršavanje. Korišćenje tabele stranica u sjenci podrazumijeva njeno smještanje u stabilnoj memoriji tako da stanje baze poslije izvršenja transakcije može biti vraćeno na stanje prije početka izvršavanja transakcije. Važno je da tabela stranica u sjenci bude smještena u stabilnoj memoriji zato što je ona jedini način da lociramo stanje baze. Tekuća tabela se može držati u glavnoj memoriji i ako se ona izgubi nije važno, zato što se sistem oporavlja korišćenjem tabele stranica u sjenci. Uspješni oporavak podrazumjeva njeno pronalaženje na disku (najsigurnije je da ona bude na nekoj fiksnoj lokaciji). Kada se sistem podigne ona se kopira u glavnu memoriju i koristi se za dalji rad transakcije. Zbog naše definicije WRITE operacije garantovano je da će tabela stranica u sjenci ukazivati na one stranice baze koje odgovaraju stanju baze prije početka izvršavanja transakcija.

Za razliku od log šema nisu potrebne nikakve undo operacije, a da bi se završila transakcija moramo raditi sljedeće:

1° obezbijediti da su sve baferske stranice u glavnoj memoriji, koje su izmijenjene od strane ove transakcije, upisane na disk. Primijetimo da operacija upisa neće promijeniti stranice baze podataka na koje ukazuje tabela stranica u sjenci.

2° upisati tekuću tabelu stranica na disk (ne smije se prepisati tabela stranica u sjenci jer nam ona može trebati za oporavak)

3° upisati disk adresu tekuće tabele stranica na fiksnu lokaciju u stabilnoj memoriji koja sadrži adresu shadow tabele stranica. Ovim tekuća tabela postaje shadow tabela i transakcija je izvršena.

Ako se kvar desi prije završetka koraka 3° vraćamo se na stanje prije početka izvršavanja, a ako se kvar desi nakon izvršenja 3° efekti transakcije će biti sačuvani i nikakve redo operacije nisu potrebne.

Shadow paging ima nekoliko prednosti u odnosu na log tehnike: nema upisivanja log zapisa, a i oporavak je znatno brži (nema undo i redo operacija). Međutim postoje i problemi:

1° Fragmentacija podataka. Shadow paging zahtijeva da stranice baze mijenjaju mjesta kada se ažuriraju i kao rezultat gubimo svojstvo lokalnosti, tj. bliske stranice po brojevima neće biti bliske i na disku. U principu, ili gubimo svojstvo lokalnosti ili moramo uvoditi kompleksnije (sa većim troškovima) šeme za fizičku organizaciju baze podataka.

2° Sakupljanje otpada (garbage collection): Svaki put kada se transakcija završi, stranice baze koje sadrže staru verziju podataka koje je izmijenila transakcija postaju nedostupne (na primjer, stranica na koju ukazuje 4 – ti red tabele stranica u sjenci u prethodnom primjeru). Ovakve stranice se nazivaju otpadom (*garbage*) zato što ne sadrže ništa korisno, a nisu slobodne. Otpad može nastati kao posledica padova i periodično je potrebno naći sve otpad stranice i dodati ih listi slobodnih. Ovaj postupak se naziva sakupljanje otpada i donosi dodatni rad, troškove i kompleksnost u sistem.

Osim toga, shadow paging je znatno teže od log mehanizma adaptirati za sisteme koji omogućavaju više konkurentnih transakcija. U takvim sistemima tipično je potreban neki log mehanizam uz shadow paging. Za log šeme potrebne su samo minorne izmjene da bi efikasno funkcionisale u konkurentnom sistemu baza podataka.

Kvar sa gubitkom stabilne memorije

Iako su kvarovi u kojma se gubi sadržaj stabilne memorije rijetki, moramo biti pripremljeni i na ovaj tip kvara. Osnovna šema za oporavak u ovom slučaju je pravljenje rezervnih (*backup* ili *dump*) kopija čitave baze, periodično, jednom dnevno ili nedeljno. Ovo se obavlja na diskove ili na magnetne trake. U slučaju ozbiljnog oštećenja fizičkih blokova najsvježija kopija se koristi da se baza vrati u poslednje konzistentno stanje. Prilikom ovoga ne dozvoljavaju se nikakve transakcije i izvršava se procedura slična radu sa checkpoint – ima:

1° svi log rekordi iz glavne memorije upisuju se na stabilnu,

2° svi baferski blokovi se upisuju na disk,

3° log zapis < dump > se upisuje na stabilnu memoriju.

Kada se ovo završi procedura za kopiranje može da počne.

Da bi se sistem oporavio od gubitka stabilne memorije konsultuje se log i sve transakcije koje su se završile od poslednjeg kopiranja se ponovo izvršavaju. Nikakve *undo* operacije nisu potrebne.

Što se tiče implementacije stabilne memorije, ona se uglavnom zasniva na replikaciji potrebne informacije na nekoliko stabilnih medija (tipično diskovi) koji imaju nezavisne režime kvara, a ažuriranje informacija se izvodi na kontrolisani način tako da se obezbijedi da kvar prilikom transfera podataka ne ošteti potrebnu informaciju.

Kontrola konkurentnosti

Jedan od najvažnijih savremenih koncepata je multiprogramiranje. Kada imamo više transakcija koje se izvršavaju istovremeno veća je iskorišćenost procesora (gubi se vrijeme čekanja) i propusnost transakcija, tj. obim rada koji je izvršen u odredjenom vremenskom periodu.

Za multiprogramiranje postoje:

- neinteraktivni sistemi (*batch*), procesi se ažuriraju jedan za drugim
- interaktivni sistemi – podrazumijevaju veliki broj kratkih transakcija za koje korisnik čeka rezultat. Njihovo izvršavanje mora biti veoma brzo, reda nekoliko sekundi. Kako je transakcija kratka ona troši malo procesorskog vremena pa svaki korisnik ima utisak da ima svoj kompjuter.

Mehanizmi za kontrolu konkurentnosti se nazivaju šeme za kontrolu konkurentnosti.

Serijabilnost

Posmatrajmo naš bankarski sistem i dvije transakcije (T_0 - prebacivanje 50$ sa računa A na račun B i T_1 - koja prebacuje 10% sa računa A na račun B).

T_0:		T_1:	
	READ (A)		READ (A)
	A = A – 50		TEMP = A · 0,1
	WRITE (A)		A = A – TEMP
	READ (B)		WRITE (A)
	B = B + 50		READ (B)
	WRITE (B)		B = B + TEMP
			WRITE B

Neka su tekuće vrijednosti 1000$ i 2000$. Mogući su razni redosledi izvršavanja transakcija. Redosledi izvršavanja nazivaju se rasporedi (*schedules*).

Na primjer, transakcije se mogu izvršavati jedna za drugom - prvo T_0 pa T_1 (*Raspored 1*) ili prvo T_1 pa T_0 (*Raspored 2*). Ova dva rasporeda nazivaju se *serijski*. Oni se sastoje iz niza instrukcija raznih transakcija i instrukcije jedne transakcije izvršavaju se jedna za drugom.

T_0	T_1
READ (A)	
A = A – 50	
WRITE (A)	
READ (B)	
B = B + 50	
WRITE (B)	
	READ (A)
	TEMP = A · 0,1
	A = A – TEMP
	WRITE (A)
	READ (B)
	B = B + TEMP
	WRITE B

Raspored 1

T_0	T_1
	READ (A)
	TEMP = A · 0,1
	A = A – TEMP
	WRITE (A)
	READ (B)
	B = B + TEMP
	WRITE B
READ (A)	
A = A – 50	
WRITE (A)	
READ (B)	
B = B + 50	
WRITE (B)	

Raspored 2

Vrijednosti na računima u prvom slučaju biće 855$ i 2145$. Ukupna suma A + B je ista (3000$). U drugom slučaju, konačne vrijednosti računa A i B biće 850$ i 2150$ i u ovom slučaju suma A + B ostaje ista.

Za skup transakcija koje se izvršavaju serijski imamo n! rasporeda. Kada se transakcije izvršavaju paralelno odgovarajući raspored ne mora biti serijski i zato je njihov broj znatno veći od n!.

Ukoliko se transakcije ne izvršavaju serijski, mogući su razni rasporedi npr. *Raspored 3* ili *Raspored 4*. Nakon izvršetka Rasporeda 3 vrijednosti računa A i B su 950$ i 2100$ i suma nije ostala ista (pojavilo se novih 50$). Ovo je nekorektno stanje. Ne moraju sva paralelna izvršavanja da rezultuju nekorektnim stanjem. Na primjer, Raspored 4 dovodi do korektnog stanja.

Kod rasporeda 3 dobili smo nekonzistentno, nekorektno stanje pa se prirodno nameće uslov da transakcija mora da čuva konzistentnost stanja. Svaka transakcija kada se izvrši prevodi sistem iz korektnog u korektno stanje iako privremeno može doći do nekorektnog stanja.

T_0	T_1	T_0	T_1
READ (A)		READ (A)	
A = A – 50		A = A – 50	
	READ (A)	WRITE (A)	
	TEMP = A · 0,1		READ (A)
	A = A – TEMP		TEMP = A · 0,1
	WRITE (A)		A = A – TEMP
	READ (B)		WRITE (A)
WRITE (A)		READ (B)	
READ (B)		B = B + 50	
B = B + 50		WRITE (B)	
WRITE (B)			READ (B)
	B = B + TEMP		B = B + TEMP
	WRITE B		WRITE B

Raspored 3 *Raspored 4*

Prirodan način da se definiše korektnost u konkurentnoj obradi je da se zahtijeva da rezultat bude isti kao kada bi transakcije izvršavali serijski. Ovo svojstvo se naziva ***serijabilnost***. "Isti rezultat" zavisi od tipa operacija koje transakcija izvršava na podatku u periodu od čitanja do upisivanja. Ako je rezultat izvršavanja operacija isti nezavisno od njihovog redosleda, onda kažemo da operacije komutiraju. U opštem slučaju, teško je utvrditi da li operacije komutiraju ili ne. Iz ovog razloga, kada razmatramo transakcije ne razmatramo tip operacija koje transakcija izvodi na podatku Q. Zato pretpostavljamo da između naredbe *read(Q)* i *write(Q)* transakcija može izvršiti proizvoljan niz operacija na podatku Q. S naše tačke gledišta jedine značajne operacije su *read(Q)* i *write(Q)*.
Da bi se formalizovao koncept serijabilnosti uvodimo pojam ekvivalentnih rasporeda.

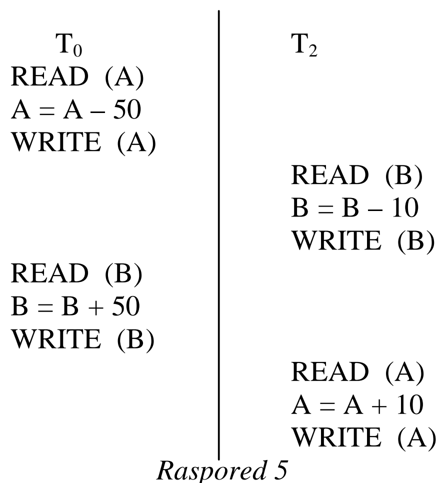
Kažemo da su dva rasporeda S_1 i S_2 *izračunljivo ekvivalentni*, u oznaci $S_1 \equiv S_2$, ako:

1. Skup transakcija koje učestvuju u S_1 i S_2 su isti.
2. Za svaki podatak Q, ako u rasporedu S_1 transakcija T_i izvršava operaciju *read(Q)* i ta vrijednost Q, koju ona čita, je bila upisana od strane transakcije T_j, tada isto važi i u rasporedu S_2.
3. Za svaki podatak Q ako u rasporedu S_1 transakcija T_i izvršava poslednje *write(Q)* tada isto važi i u rasporedu S_2.

Uslov 1 obezbjeđuje da isti skup transakcija učestvuje u oba rasporeda. Uslov 2 obezbjeđuje da svaka transakcija čita iste vrijednosti u oba rasporeda i zato vrši ista izračunavanja. Uslov 3 kombinovan sa uslovom 2 obezbjeđuje da oba rasporeda rezultuju istim stanjem sistema.

Raspored 1 i Raspored 2 nisu ekvivalentni jer u Rasporedu 1 vrijednost podatka A, koju čita transakcija T_1, je upisana od strane transakcije T_0, što nije slučaj u Rasporedu 2. S druge strane, Raspored 1 je ekvivalentan sa Rasporedom 4 zato što vrijednosti računa A i B, koje čita transakcija T_1, su proizvedene od strane transakcije T_0 u oba rasporeda.

Moguće je da imamo 2 rasporeda koja prave isti rezultat, ali koji nisu ekvivalentni po našoj definiciji. Posmatrajmo transakciju T_2 koja prebacuje 10$ sa računa B na račun A. Neka imamo sljedeći raspored:



T_0	T_2
READ (A)	
A = A – 50	
WRITE (A)	
	READ (B)
	B = B – 10
	WRITE (B)
READ (B)	
B = B + 50	
WRITE (B)	
	READ (A)
	A = A + 10
	WRITE (A)

Raspored 5

Neka je Raspored 6 serijski raspored T_0 i T_2. Raspored 5 nije ekvivalentan sa Rasporedom 6 zato što u Rasporedu 5 vrijednost računa B koju čita transakcija T_0 je proizvedena od strane transakcije T_2, što nije slučaj u Rasporedu 6. Međutim, konačne vrijednosti računa nakon izvršenja Rasporeda 5 ili Rasporeda 6 su iste. Ovo je zbog toga što mi ne razmatramo tip operacija koji se može izvršiti nad podatkom. Pod ovom pretpostavkom naša definicija izračunljive ekvivalencije je potreban i dovoljan uslov za ekvivalentnost rasporeda.

Kada imamo definiciju ekvivalentnosti rasporeda možemo i formalno definisati pojam serijabilnosti. Neka je $\{T_0,..., T_n\}$ skup transakcija koje učestvuju u rasporedu S. Kažemo da je raspored S ***serijabilan*** ukoliko postoji serijski raspored S', tako da važi $S \equiv S'$.

Testiranje serijabilnosti

Neka imamo odredjeni raspored S i želimo da ustanovimo da li je on serijabilan. Za ovo konstruišemo usmjereni graf koji se naziva grafom precedencije. To je graf G (V,E), gdje je V skup vrhova i E skup grana (ivica). Skup vrhova odgovara skupu svih transakcija koje učestvuju u rasporedu, a skup ivica sadrži sve ivice za koje važi jedan od sljedeća dva uslova:

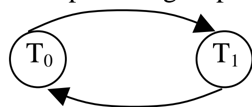
- T_i izvršava *WRITE (Q)* prije nego što T_j izvrši *READ(Q)*,
- T_i izvršava *READ (Q)* prije nego što T_j izvrši *WRITE (Q)*.

Ako postoji ivica $T_i \rightarrow T_j$ u grafu precedencije to implicira da u bilo kom serijskom rasporedu S' ekvivalentnom sa S, T_i se mora pojaviti prije T_j.

Grafovi precedencije - (a) za Raspored 1, (b) za Raspored 2.

Graf za Raspored 1 ima jednu ivicu između T_0 i T_j zato što se sve instrukcije u transakciji T_0 izvršavaju prije prve instrukcije transakcije T_1. Slično za Raspored 2.

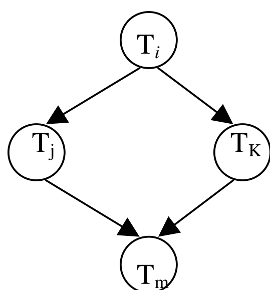
Za Raspored 3 graf precedencije izgleda ovako:



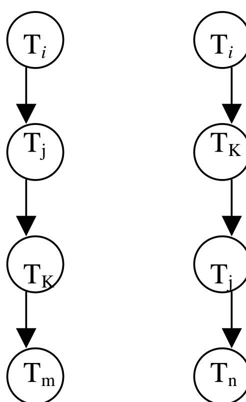
On sadrži ivicu $T_0 \rightarrow T_1$ zato što T_0 izvršava *read(A)* prije nego što T_1 izvrši *write(A)*. Sadrži ivicu $T_1 \rightarrow T_0$ zato što T_1 izvršava *read(B)* prije nego što T_0 izvrši *write(B)*.

Ako graf precedencije za raspored S ima ciklus tada raspored nije serijabilan, a ako ne sadrži ciklus onda jeste serijabilan.

Sam redosled serijabilnosti se dobija kroz topološko sortiranje koje određuje linearni poredak saglasan sa parcijalnim poretkom grafa precedencije. Zato je moguće više ekvivalentnih linearnih rasporeda.



Sam graf nema ciklusa.
Redosled koji mu odgovara je serijski.



Ilustracija topološkog sortiranja

Da bi se testirala serijabilnost potrebno je konstruisati graf precedencije i na njemu primijeniti algoritam za detekciju ciklusa. Kako nalaženje ciklusa u grafu zahtijeva $O(n^2)$ operacija gdje je $|V| = n$ (broj transakcija) ovo nije skup algoritam, tj. imamo efikasan algoritam za određivanje serijabilnosti.

Ovo razmatranje je bilo za transakcije koje prvo čitaju pa onda upišu neke vrijednosti (*read before write*) i za njih postoji efikasan algoritam. Ukoliko se ispusti ovaj uslov, tj. dozvolimo da transakcija piše bez prethodnog čitanja podatka tada ne postoji efikasan elgoritam za ispitivanje serijabilnosti. Isto se konstruišu grafovi, a pretraživanje po njemu spada u NP kompletne probleme.

Zaključavanje

Jedan od načina za obezbjeđivanje serijabilnosti je da se zahtijeva da pristup podacima bude na uzajamno isključiv način, tj. dok jedna transakcija ima pristup nekom podatku nijedna druga transakcija ne može da modifikuje taj podatak. Najuobičajeniji metod za implementaciju ovoga je da se omogući transakciji da pristupi nekom podatku samo ako ona trenutno drži katanac (*lock*) na njemu. Postoje razni načini na koje podatak može biti zaključan. Najprostiji je model sa dvije vrste katanca: dijeljeni (*SHARED*) i ekskluzivni (*EXCLUSIVE*) katanac. Ako je transakciji dodijeljen dijeljeni katanac S na podatak Q tada ona može čitati ovaj podatak, ali ne može upisivati. Ako je transakcija dobila ekskluzivni karanac X na podatak Q tada ona može i čitati i pisati podatak Q. Svaka transakcija mora da zahtijeva odgovarajući katanac na podatak Q, u zavisnosti od toga kakve operacije ona namjerava da izvrši na podatku Q.

Objekti koji se zaključavaju mogu biti razni: cijele relacije, pojedine torke, djelovi torki, grupe atributa, skupovi torki, itd. Ovo se naziva *granularnost* zaključavanja.

Ako imamo skup načina zaključavanja (katanaca) možemo definisati na njima *funkciju kompatibilnosti*. Neka su A i B načini zaključavanja. Pretpostavimo da transakcija T_i zahtijeva način zaključavanja A na podatku Q na kome transakcija T_j ($T_i \neq T_j$) trenutno drži način zaključavanja B. Tada, ako transakciji T_i može odmah biti dat način zaključavanja A na Q, nezavisno od prisustva načina zaključavanja B, mi kažemo da je način zaključavanja A kompatibilan sa načinom zaključavanja B, tj. vrijednost funkcije *comp (A,B) = true*. Ova funkcija se načešće predstavlja matricom:

	S	X
S	true	false
X	false	false

Jasno je da je dijeljeni katanac kompatibilan sam sa sobom, ali nije sa ekskluzivnim katancem. U bilo koje vrijeme, nekoliko dijeljenih katanaca mogu biti držani istovremeno od strane raznih transakcija na nekom podatku. Naknadni zahtjevi za ekskluzivni katanac moraju da čekaju sve dok tekući dijeljeni katanci ne budu pušteni. S druge strane, ako transakcija drži ekskluzivni katanac na nekom podatku, nijednoj transakciji ne može biti dodijeljen ni ekskluzivni ni dijeljeni katanac na tom podatku, sve dok taj ekskluzivni katanac ne bude pušten.

Transakcija zahtijeva dijeljeni katanac na podatku Q pomoću instrukcije *LS(Q)*, a ekskluzivni katanac sa *LX(Q)*. Katanac se uklanja sa podatka Q sa *UN(Q)* (od *unlock* - otključati). Kao što je ranije rečeno, da bi transakcija pristupila podatku ona ga mora prvo zaključati. Ako je podatak već zaključan od strane druge transakcije u nekompatibilnom katancu, tada transakcija mora čekati dok se ne otključaju svi nekompatibilni katanaci. Transakcija T_i može otključati katanac koji je ranije dobila.

Transakcija mora držati katanac na podatku sve dok mu pristupa. Nije uvijek poželjno da transakcija otključa podatak odmah nakon poslednjeg pristupa njemu zato što možda neće biti obezbijedjena serijabilnost.

Neka imamo transakcije T_6 (prebacuje 50 $ s jednog računa na drugi) i T_7 (štampa ukupan iznos novca na ova dva računa).

T_6:		T_7:	
	LX (B)		LS (A)
	READ (B)		READ (A)
	B = B – 50		UN (A)
	WRITE (B)		LS (B)
	UN (B)		READ (B)
	LX (A)		UN (B)
	READ (A)		DISPLAY (A + B)
	A = A + 50		
	WRITE (A)		
	UN (A)		

Pretpostavimo da su vrijednosti računa A i B 100$ i 200$. Ako se ove dvije transakcije izvršavaju serijski T_6 pa T_7 ili T_7 pa T_6, transakcija T_7 će štampati 300.

Međutim, razmotrimo sljedeći raspored:

T_6	T_7
LX (B)	
READ (B)	
B = B – 50	
WRITE (B)	
UN (B)	
	LS (A)
	READ (A)
	UN (A)
	LS (B)
	READ (B)
	UN (B)
	DISPLAY (A + B)
LX (A)	
READ (A)	
A = A + 50	
WRITE (A)	
UN (A)	

U ovom rasporedu transakcija T_7 štampa 250$, što je nekorektno. Razlog za ovo leži u tome što je transakcija T_6 pustila katanac na podatak B previše rano, zbog čega transakcija T_7 vidi nekorektno stanje.

Pretpostavimo sada da je otključavanje odloženo do neke kasnije tačke:

T_8:		T_9:	
	LX (B)		LS (A)
	READ (B)		READ (A)
	B = B – 50		LS (B)
	WRITE (B)		READ (B)
	LX (A)		DISPLAY (A + B)
	READ (A)		UN (A)
	A = A + 50		UN (B)
	WRITE (A)		
	UN (B)		
	UN (A)		

T_8:	T_9:
LX (B)	
READ (B)	
B = B – 50	
WRITE (B)	
	LS (A)
	READ (A)
	LS (B)
LX (A)	⋮
READ (A)	⋮
⋮	⋮

Raspored 8

U Rasporedu 8 transakcija T_8 drži ekskluzivni katanac na B, a T_9 traži dijeljeni na B, zato T_9 čeka T_8 da otključa B. Slično, kako T_9 drži dijeljeni katanac na A, a T_8 traži ekskluzivni katanac na A, T_8 čeka na T_9 da otključa A i došli smo u stanje gdje nijedna transakcija ne može nastaviti sa normalnim izvršavanjem. Ovo stanje se naziva *deadlock* ili ćorsokak. Kada se ovo desi, sistem mora izvršiti operaciju *rollback* na jednoj od transakcija. Kada se ta operacija izvrši, otključavaju se svi katanci koje je ta transakcija držala. Nakon toga, neka od ostalih transakcija će nastaviti da se izvršava.

Iz ovih primjera zaključujemo da se zaključavanje mora koristiti pažljivo. S jedne strane, ako želimo da maksimizujemo konkurentnost otključavanjem podataka što je moguće prije, možemo doći u nekonzistentno stanje, ali, s druge strane, ako ne otključavamo katanace na podacima prije zaključavanja drugih podataka, može se desiti ćorsokak.

Iz ovih razloga, zahtijevamo da svaka transakcija u sistemu slijedi određeni skup pravila koje ćemo nazvati *protokolom zaključavanja*, koji govori kada transakcija može da zaključava i otključava neki podatak. Ovi protokoli ograničavaju broj mogućih rasporeda i skup takvih rasporeda je pravi podskup skupa svih serijabilnih rasporeda.

Postoji nekoliko protokola zaključavanja koji dozvoljavaju samo serijabilne rasporede.

Dvofazni protokol zaključavanja

Dvofazni protokol zaključavanja, koji je čest u praksi, zahtijeva da svaka transakcija traži zaključavanje i otključavanje u dvije faze:

1. Faza rasta ili zaključavanja (*growing phase*). U ovoj fazi transakcija može dobijati katance ali ih ne može otključavati.
2. Faza opadanja ili otključavanja (*shrinking phase*). U ovoj fazi transakcija može otključavati ali ne može zahtijevati zaključavanje.

U početku se transakcija nalazi u prvoj fazi i dobija sve katance koje je tražila, a čim pusti jedan katanac ne može da traži novi. Na primjer, transakcije T_8 i T_9 su dvofazne, a transakcije T_6 i T_7 nisu.

Ovaj dvofazni protokol zaključavanja obezbjeđuje serijabilnost, ali ne obezbjeđuje nepojavljivanje ćorsokaka. Na primjer, transakcije T_8 i T_9 su dvofazne ali u Rasporedu 8 one su u ćorsokaku.

Ako je T_i nedvofazna transakcija, uvijek je moguće naći dvofaznu transakciju T_j tako da postoji neserijabilni raspored za T_i i T_j.

Posmatrajmo transakcije:

T_{10} :	READ (a_1)	T_{11} :	READ (a_1)
	READ (a_2)		READ (a_2)
	⋮		DISPLAY ($a_1 + a_2$)
	READ (a_n)		
	WRITE (a_1)		

Ukoliko ove transakcije poštuju dvofazni protokol zaključavanja, T_{10} mora zaključati a_1 sa ekskluzivnim katancem i zato svako konkurentno izvršavanje ove dvije transakcije je serijsko. Ovdje možemo primijetiti da je transakciji T_{10} potreban ekskluzivni katanac samo na samom kraju. Zato bi bilo dobro da ona prvo uzme dijeljeni katanac a da na kraju, kada joj to zatreba, uzme ekskluzivni katanac. Time bi se omogućilo paralelno izvršavanje transakcija.

Ovo nas vodi do poboljšanja dvofaznog protokola zaključavanja u kome su dozvoljene *konverzije katanaca*. U konverzijama katanaca se dozvoljava "povećanje" dijeljenog katanca na ekskluzivni katanac i "spuštanje" sa ekskluzivnog katanca na dijeljeni katanac. Konverziju sa dijeljenog katanca na ekskluzivni katanac označavamo sa UP, a konverziju sa ekskluzivnog na dijeljeni sa DN.

Konverzije katanaca ne mogu da se dešavaju u proizvoljno vrijeme. "Povećanje" može da se desi samo u prvoj fazi, a "spuštanje" samo u drugoj. Naš prethodni primjer sa konverzijama:

T_{10}	T_{11}
LS (a_1)	
	LS (a_1)
LS (a_2)	
	LS (a_2)
LS (a_3)	
LS (a_4)	
	UN (a_1)
	UN (a_2)
LS (a_n)	
UP (a_1)	

Kada transakcija pokuša da poveća katanac na podatku Q ona može biti prisiljena da čeka. Ovo se dešava ukoliko je podatak Q trenutno zaključan od strane neke druge transakcije u dijeljenom modu.

Sada ćemo opisati kako se generišu instrukcije zaključavanja u transakciji. Kada transakcijaT_i izvršava *read(Q)* naredbu sistem izdaje *LS(Q)* instrukciju pa nakon nje *read(Q)*. Kada T_i izdaje *write(Q)* sistem provjerava da li T_i već drži dijeljeni katanac na Q. Ako je odgovor da, tada sistem izdaje *UP(Q)* instrukciju iza kojeg slijedi *write(Q)*. U drugom slučaju, ako nije bilo nikakvog katanca, sistem izdaje *LX(Q)* i nakon njega *write(Q)*.

Postoje serijabilni rasporedi za neki skup transakcija koji se ne mogu dobiti pomoću dvofaznog protokola zaključavanja. Da bi smo postigli bolje od dvofaznog zaključavanja potrebna nam je ili dodatna informacija o transakcijama ili neka struktura, odnosno poredak, na skupu podataka u bazi podataka.

Protokoli zasnovani na grafovima

Postoje razni protokoli koji su koriste dodatne informacije o transakcijama ili podacima. Razlike među protokolima su u prirodi informacija koje imamo o transakciji ili podacima. Najprostiji zahtijevaju da imamo prethodno znanje o poretku u kome se može pristupati podacima. Ako imamo takvu informaciju, moguće je konstruisati protokole zaključavanja koji nisu dvofazni.

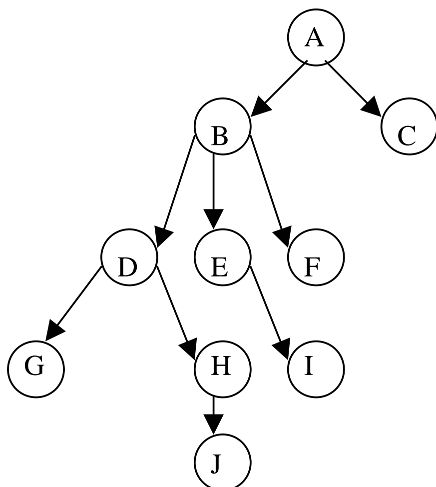
Uvodimo parcijalni poredak na skup $D = \{d_1,..., d_K\}$ svih podataka. Ako važi $d_i \rightarrow d_j$ svaka transakcija koja pristupa i podatku d_i i podatku d_j mora pristupiti podatku d_i prije d_j. Ovaj parcijalni poredak može biti rezultat bilo logičke, bilo fizičke organizacije podataka, ili može biti uveden zbog potreba za obezbjedjenjem kontrole konkurentnosti.

Parcijalni poredak obezbjeđuje da se skup podataka D može vidjeti kao usmjereni aciklični graf, koji se zove *graf baze podataka*. Zbog jednostavnosti, razmatraćemo samo grafove koji su stabla. Za takve grafove postoji jednostavni protokol, koji se naziva *protokol stabla*, i koji uključuje samo ekskluzivne katance. Jedina instrukcija zaključavanja je LX i svaka transakcija T_i može zaključati podatak najviše jednom i pri tom mora poštovati sljedeća pravila:

1. Prvi katanac od strane transakcije T_i može biti na bilo kom podatku.
2. Kasnije, podatak Q može biti zaključan od strane transakcije T_i samo ako je i otac od Q zaključan od strane T_i.
3. Podaci se mogu otključavati u bilo koje vrijeme.
4. Podatak koji je bio zaključan i otključan od strane T_i ne može odmah biti zaključan od strane T_i.

Svi rasporedi koji poštuju ovaj protokol su serijabilni.

Za ilustraciju ovog protokola posmatrajmo sljedeći graf:



Sljedeće 4 transakcije slijede ovaj graf protokol.

T_{12} : LX (B) , LX (E) , UN (E) , LX (D) , UN (B) ,
LX (G) , UN (D) , UN (G) .

T_{13} : LX (D) , LX (H) , UN (D) , LX (J) , UN (J) ,
UN (H) .

T_{12}	T_{13}	T_{14}	T_{15}
LX (B)			
	LX (D)		
	LX (H)		
	UN (D)		
LX (E)			
UN (E)			
LX (D)			
UN (B)		LX (B)	
		LX (C)	
LX (G)	UN (H)		
UN (D)	↓		
			LX (D)
			LX (H)
			UN (D)
			UN (H)
		UN (G)	
		UN (B)	
UN (G)			

Vidi se da transakcija može držati katance u raznim djelovima drveta.

Ovaj protokol obezbjeđuje serijabilnost i slobodu od ćorsokaka. Ovaj graf protokol ima prednost nad dvofaznim protokolom u tome što se otključavanje može desiti ranije, što rezultuje manjim vremenom čekanja, i zato je konkurentnost povećana. Takođe, kako je slobodan od ćorsokaka, nisu potrebne nikakve *ROLLBACK* operacije, odnosno otkazivanje transakcija. Nedostatak ovog protokola je što se, u nekim slučajevima, zaključava znatno više podataka nego što je potrebno, čime se potencijalno povećava vrijeme čekanja i smanjuje konkurentnost.

Oporavak od kvara

SUBP koji dozvoljava konkurentnost osim serijabilnosti mora obezbijediti neki mehanizam za oporavak u cilju obezbjeđivanja svojstva atomičnosti transakcije, tj. mora postojati šema za oporavak. Razne šeme za oporavak zasnovane na logovima (žurnalizaciji) mogu da se koriste u ovu svrhu. Glavna razlika je u tome što se ovdje može desiti da više transakcija, a ne samo jedna moraju biti poništene kao rezultat kvara. Ova pojava se zove *kaskadni otkaz* transakcija.

Posmatratrajmo sljedeći parcijalni raspored

T_{16}	T_{17}	T_{18}
LX (A)		
LX (B)		
UN (A)		
	LX (A)	
	UN (A)	
		LX (A)

Sve tri transakcije poštuju dvofazni protokol. Pretpostavimo da je izvršavanje T_{16} prekinuto zbog neke logičke greške. Zato se ona mora otkazati, ali kako je T_{17} zavisna od T_{16}, a T_{18} od T_{17} imamo situaciju kaskadnog otkaza.

Jasno je da je kaskadni otkaz nepoželjan zbog suvišnog posla koji se pojavljuje u sistemu. Da bi se izbjegao kaskadni otkaz sistem mora obezbijediti da transakcije mogu čitati samo potvrdjene vrijednosti, tj. vrijednosti koje su rezultat završenih transakcija. Ovo se može ostvariti putem zahtjeva da se otključavanje vrši tek nakon poslednje naredbe transakcije. Štaviše, otključavanje se može desiti tek pošto

su odgovarajući log zapisi upisani u stabilnu memoriju. Jasno je da ovo smanjuje konkurentnost i situacija postaje još gora ako imamo veći broj podataka koje treba zaključavati.

Postoje dva međusobno suprotstavljena kriterijuma za izbor šeme za konkurentnost.
1° izbjegavanje kaskadnih otkaza
2° povećanje paralelizma

Ako sistem može obezbijediti da se pad transakcija dešava relativno rijetko, tada se kaskadni otkazi mogu tolerisati, inače sistem se mora čuvati od njih.

Mehanizam vremenskih marki

U svakom od prethodnih protokola zaključavanja, za svaki par transakcija koje su u konfliktu, u vrijeme izvršavanja određuje se redosljed izvršavanja u trenutku kada ove dvije transakcije zahtijevaju zaključavanje prvog zajedničkog podatka u nekompatibilnim režimima zaključavanja. Drugačiji pristup za obezbjeđivanje serijabilnosti je da se unaprijed uvede neki poredak za svaki par transakcija. Najčešći metod za ovo je korišćenje mehanizma *vremenskih marki* (*timestamping*).

Svakoj transakciji T_i pridružuje se jedinstveni identifikator, tj. jedinstvena vremenska marka TS(T_i). Nju dodjeljuje SUBP prije početka izvršavanja transakcije T_i . Ako je T_i dobila vremensku marku TS (T_i), a transakcija T_j ulazi u sistem tada mora biti TS (T_i) < TS (T_j) . Postoje dvije jednostavne metode za realizaciju ovog mehanizma:
1° Korišćenje sistemskog sata kao vremenske marke.
2° Korišćenje logičkog brojača koji se povećava svaki put kada se dodijeli novoj transakciji.

Vremenske marke određuju serijabilni poredak. Ako je TS (T_i) < TS (T_j) sistem mora obezbijediti da je proizvedeni raspored ekvivalentan serijskom rasporedu u kojem se T_i pojavljuje prije T_j .

Ovo se realizuje tako što se za svaki podatak Q vezuju dvije vrijednosti vremenske marke:
- *W-marka*, koja označava najveću vrijednost marki bilo koje transakcije koja je uspješno izvršila write (Q) ;
- *R-marka,* označava najveću vremensku marku bilo koje T koja je uspješno izvršila read (Q) .

Ove vremenske marke se ažuriraju svaki put kada se izvrši *read(Q)* ili *write(Q)* operacija.

Protokol vremenskih marki obezbjeđuje da se sve konfliktujuće read i write operacije izvršavaju u poretku vremenskih marki. Definišimo protokol:

1° Ako T_i izdaje *read(Q)* naredbu:

a) Ako je *TS(T_i) < W-marka(Q)*, tada transakcija T_i želi da čita podatak Q koji je već prepisan, zato se read operacija mora odbiti, a transakcija T_i otkazati.

b) Ako je *TS (T_i) ≥ W-marka(Q)*, T_i hoće da čita nešto što je napisala neka transakcija koja i treba da bude prije T_i . Tada se read operacija izvršava i *R-marka(Q)* postavlja na maksimum od *R-marka(Q)* i TS (T_i).

2° Ako T_i izdaje *write(Q)* naredbu:

a) Ako je *TS (T_i) < R-marka(Q),* to znači da je vrijednost koju T_i treba da upiše bila potrebna nekome ranije i da taj neko pretpostavlja da taj podatak neće biti upisan posle. Zato se write mora odbiti i T_i se mora otkazati.

b) Ako je *TS(T_i) < W-marka(Q)* ovo znači da je neko već upisao, ali taj neko je već upisao (T_i pokušava da upiše zastarelu vrijednost podatka Q).

c) U ostalim slučajevima write se izvršava, a *W-marka(Q)* se podešava da bude maksimum od *W-marka(Q)* i TS (T_i).

Ako je T_i otkazana, ona dobija novu vremensku marku i izvršava se iznova.

Posmatrajmo sljedeće transakcije:

T_{19} :		T_{20} :
READ (B)		READ (B)
READ (A)		B = B – 50
DISPLAY (A + B)		WRITE (B)
		READ (A)
		A = A + 50
		WRITE (A)
		DISPLAY (A + B)

Neka je TS (T_{19}) < TS (20). Onda je po protokolu vremenskih marki moguć ovaj raspored:

T_{19}	T_{20}
READ (B)	
	READ (B)
	B = B – 50
	WRITE (B)
READ (A)	
	READ (A)
DISPLAY (A + B)	
	A = A + 50
	WRITE (A)

Ovaj raspored se ne može dobiti dvofaznim protokolom zaključavanja. Takođe, postoje rasporedi koji su mogući pod dvofaznim protokolom a nisu mogući pod protokolom vremenskih marki.

Protokol vremenskih marki obezbjeđuje serijabilnost zato što se konfliktne situacije obrađuju u poretku vremenskih marki.

Ovaj protokol garantuje slobodu od ćorsokaka zato što nema nikakvog čekanja. Ovako definisana šema vremenskih marki može rezultovati kaskadnim otkazima transakcija. Postoje mehanizmi za rješenje ovog problema.

Slabi nivoi konzistentnosti

Serijabilnost je koristan koncept zato što omogućuje programeru da ignoriše pitanja vezana za konkurentnost kada piše programe. Nažalost, protokoli koji obezbjeđuju serijabilnost omogućuju malu konkurentnost za određeni tip aplikacija. U ovim slučajevima koriste se slabi nivoi konzistentnosti. Oni, zbog povećanja konkurentosti, odustaju od potpune serijabilnosti i time ostavljaju programeru značajan teret obezbjeđivanja korektnosti baze.

Navešćemo slabe nivoe konzistentnosti (ili *nivoe izolacije transakcije*, transaction isolation level) u standardu SQL-92. Oni se postavljaju SQL komandom SET TRANSACTION ISOLATION LEVEL. To su:

- Serializable - serijabilnost
- Repeatable read (ponovljivo čitanje) omogućuje da samo potvrđene torke mogu biti čitane, i osigurava da između dva čitanja torke od strane jedne transakcije nijedna druga transakcija ne može ažurirati torku. Ipak, ovo nije dovoljno za serijabilnost. Na primjer, kada transakcija traži torke koje zadovoljavaju neki uslov, ona može naći neke torke koje su unijele potvrđene transakcije, dok neke ne može naći.

- Read commited (čitanje potvrđenih) dozvoljava da se čitaju samo potvrđene torke, ali ne i ponovljivo čitanje. Zato, na primjer, između dva čitanja torke od strane transakcije, torka može biti ažurirana od strane druge, potvrđene, transakcije.
- Read uncommited dozvoljava da se čak i nepotvrđene torke čitaju. Ovo je najniži nivo konkurentnosti u SQL-u. Ovo ima smisla za duge transakcije koje ne moraju biti apsolutno precizne, na primjer za prikupljanje statističkih informacija.