Predavanje XII.

Kontrola istodobnosti

BAZE PODATAKA II

doc. dr. sc. Goran Oreški
Fakultet informatike,
Sveučilište Jurja Dobrile, Pula

Sadržaj

- ponavljanje prethodnih predavanja
  - transakcije
  - rasporedi
  - serijalizacija
  - kontrola istodobnosti
- kontrola istodobnosti
- lock-based protokol
- dvo-fazni protokol zaključavanja
- zastoji
  - prevencija zastoja
  - detekcija zastoja
- višestruka granularnost
- Timestamp-Based protokoli
- Validation-Based protokoli
- Snapshot Isolation

Ponavljanje

- transakcije se definiraju kao **jedinice** posla koje pristupaju i (ponekad) mijenjaju podatke u bazi podataka
- transakcije se sastoje od skupa operacija koje čine logičku cjelinu posla
- prilikom izvođenja transakcija pojavljuju se dva izazova:
  1. različiti razlozi prekida transakcije
  2. istodobno izvršavanje više transakcija
- rezultat može biti **nekonzistentno stanje** (*engl. inconsistent state*), tj. stanje koje ne odgovara stvarnom stanju svijeta kojeg baza predstavlja

Ponavljanje

- da bi se zaštitio integritet podataka sustav baze podataka mora osigurati:
  - nedjeljivost transakcije: ili su sve operacije transakcije uspješno izvršene u bazi podataka ili nije niti jedna
  - konzistentnost: izvršavanje transakcije u izolaciji održava konzistentnost baze podataka
  - izolacija: iako se mnoge transakcije mogu izvršiti istodobno, transakcija ne smije biti svjesna drugih transakcija koje se izvršavaju u istom vremenu
    - međurezultati moraju biti skriveni od drugih istodobnih transakcija
    - tj. za svaki par transakcija T_i i T_j, transakciji T_i se čini ili da je transakcija T_j završila prije nego što je T_i počela ili je T_j počela izvršavanje nakon što je T_i završila
  - trajnost: nakon što transakcija uspješno završi, promjene koje su u bazi učinjene su trajne čak i ako se sustav sruši

Ponavljanje

- **raspored** (izvođenja) – sekvence instrukcija koje definiraju kronološki redoslijed izvođenja instrukcija unutar istodobnih transakcija
- osnovna pretpostavka – svaka transakcija čuva konzistentnost baze podataka
- stoga, serijsko izvođenje transakcija čuva konzistentnost baze podataka
- sustav baze podataka mora pružiti mehanizam koji osigurava da su mogući rasporedi:
  - conflict serializable
  - recoverable i cascadeless

Kontrola istodobnosti

- na prošlom predavanju smo vidjeli da je osnovno svojstvo transakcije izolacija
- kada se više transakcija izvodi istodobno u bazi podataka, izolacija više ne mora biti osigurana
- sustav mora kontrolirati interakciju između istodobnih transakcija
  - za to su zadužene sheme za kontrolu istodobnosti (engl. *concurrency-control schemes*)
- postoji više različitih shema koje se koriste, svaka s prednostima i nedostatcima
  - najčešće se koriste dvo-fazno zaključavanje i korištenje snimki

Lock-based protokol

- zaključavanje (engl. lock) je mehanizam za kontrolu istodobnog pristupa podatkovnim elementima
- podatkovni elementi mogu biti zaključani na dva načina:
  - **exclusive (X) mode** – elementi se mogu čitati i pisati
    - X-lock ćemo zahtijevati koristeći lock-X instrukciju
  - **shared (S) mode** – elementi se mogu samo čitati
    - S-lock ćemo zahtijevati koristeći lock-S instrukciju
- lock zahtjevi se šalju upravljaču za kontrolu istodobnosti (*engl. concurrency-control manager*), zahtjeve inicira programer
  - transakcija može krenuti s izvođenjem samo nakon što je zahtjev odobren

Lock-based protokol

- matrica kompatibilnosti lock-ova

	S	X
S	true	false
X	false	false

- transakciji se može odobriti lock samo kada je traženi lock kompatibilan s već dodijeljenim lock-ovima na podatkovnom elementu
- neograničen broj transakcija može držati lock nekog elementa
- ukoliko lock ne može biti odobren, zahtjev se stavlja na čekanje dok svi nekompatibilni lock-ovi nisu otpušteni
  - u nastavku predavanja starvation of transactions

Lock-based protokol

- primjer transakcije s izvođenjem lock-a:

T_1:
```
lock-X(B);
read(B);
B := B – 50;
write(B);
unlock(B);
lock-X(A);
read(A);
A := A + 50;
write(A);
unlock(A)
```

T_2:
```
lock-S(A);
read (A);
unlock(A);
lock-S(B);
read (B);
unlock(B);
display(A+B)
```

- zaključavanje prikazano u primjeru nije dovoljno da bi se osigurala serijabilnost; ukoliko neka transakcija promijeni vrijednost A ili B poslije čitanja a prije ispisa, rezultat neće biti točan
- **protokol zaključavanja** (*engl. locking protocol*) je skup pravila koje moraju slijediti sve transakcije prilikom zahtijevanja ili otpuštanja lock-a
- protokoli zaključavanja smanjuju skup mogućih rasporeda

Dvo-fazni protokol zaključavanja

- ovaj protokol osigurava konflikt serijabilne rasporede (*engl. The Two-Phase Locking Protocol*)
- *faza 1: rastuća faza*
  - transakcija može pribaviti lock
  - transakcija ne može otpustiti lock
- *faza 2: silazna faza*
  - transakcija ne može pribaviti lock
  - transakcija može otpustiti lock
- može se dokazati da transakcije mogu biti serijalizirane u rasporedu njihovih lock točaka (*engl. lock point*); trenutak u kojem je transakcija dobila svoj posljednji lock

Dvo-fazni protokol zaključavanja

- dio rasporeda s dvo-faznim protokolom zaključavanja

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

kaskadni rollback problem

strict i rigorous inačice protokola!

Dvo-fazni protokol zaključavanja

- faze protokola s konverzijama (*engl. lock conversions*):
  - prva faza:
    - može pribaviti lock-S za element
    - može pribaviti lock-X za element
    - može napraviti konverziju lock-S za lock-X na nekom elementu (upgrade)
  - druga faza:
    - može otpustiti lock-S za element
    - može otpustiti lock-X za element
    - može napraviti konverziju lock-X za lock-S na nekom elementu (downgrade)
- ovaj protokol osigurava serijabilnost, ali se temelji na izdavanju različitih lock instrukcija od strane programera
  - strict i rigorous inačice protokola s konverzijama se intenzivno koriste u komercijalnim DB sustavima

Automatski dohvat lock-ova

- transakcija T_i izdaje standardne *read/write* instrukcije, bez eksplicitnog lock poziva
- operacija *read(D)* se obrađuje kao:

```
if T_i has a lock on D
    then
        read(D)
    else begin
        if necessary wait until no other
            transaction has a lock-X on D
        grant T_i a  lock-S on D;
        read(D)
    end
```

Automatski dohvat lock-ova

- operacija *write(D)* se obrađuje kao:

```
if T_i has a  lock-X on D
      then
        write(D)
      else begin
          if necessary wait until no other transaction has any lock on D,
          if T_i has a lock-S on D
              then
                 upgrade lock on D  to lock-X
              else
                 grant T_i a lock-X on D
              write(D)
       end;
```

- svi lock-ovi se otpuštaju nakon commita-a ili abort-a

Deadlocks (zastoji)

- dan je isječak rasporeda:

T_3	T_4
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

- niti jedna transakcija ne može nastaviti izvođenje; *lock-s(B)* čeka otpuštanje lock-a transakcije T_3, a *lock-x(A)* otpuštanje transakcije T_4
- takva situacija se naziva deadlock (zastoj)
  - da bi situacija riješila jedna transakcija mora napraviti roll back i otpustiti svoje lock-ove

Deadlocks (zastoji)

- korištenje dvo-faznog zaključavanja ne znači izbjegavanje deadlock-ova
- štoviše, postoji mogućnost nastanka situacije koja se naziva ***starvation***
- starvation se događa u slučaju kada je *concurrency control manager* loše dizajniran
  - transakcija čeka na *X-lock*, dok se *S-lock* odobrava nizu transakcija na istom podatkovnom elementu
  - ista transakcija uzastopno radi roll back da bi se izašlo iz deadlock-a
- *ccm* može biti dizajniran da se starvation izbjegne

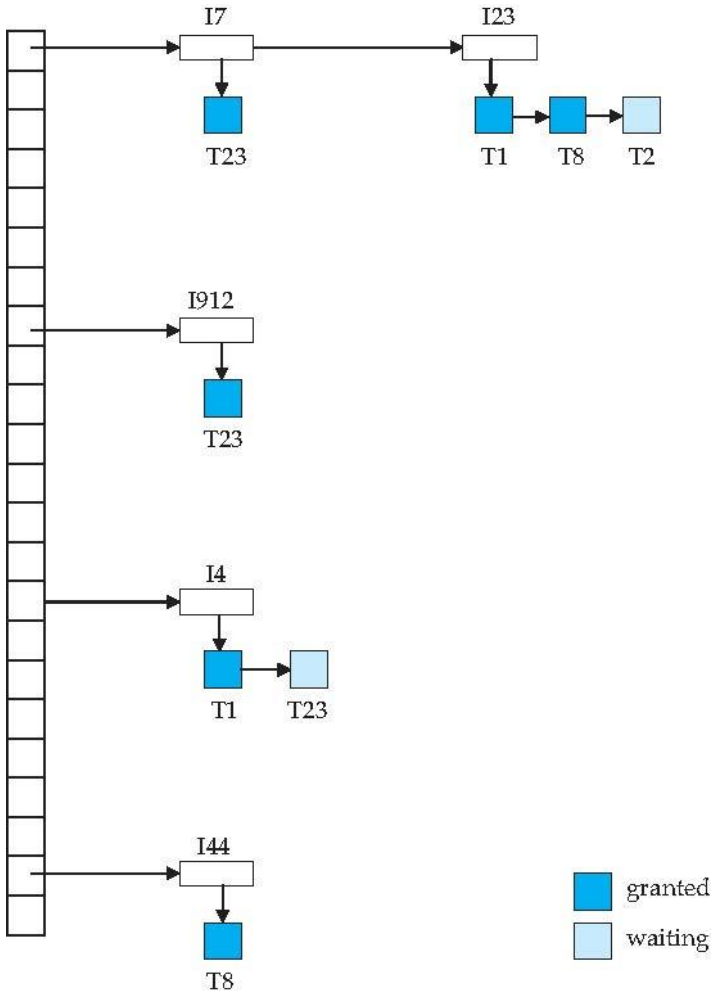
Deadlocks (zastoji)

- mogućnost događanja deadlock-a postoji u gotovo svim protokolima zaključavanja
- kada se deadlock dogodi postoji mogućnost nastanka cascading rollback-a
- cascading rollback je moguć u okviru dvo-faznog zaključavanja, da bi se to izbjeglo potrebno je modificirati protokol:
  - ***strict two-phase locking***
    - transakcija mora držati sve X-lock-ove do trenutka commit/abort
  - ***rigorous two-phase locking***
    - transakcija mora držati sve lock-ove do trenutka commit/abort
    - prema ovom protokolu transakcije mogu biti serijalizirane prema redoslijedu commita

Implementacija zaključavanja

- *lock manager* može biti implementiran kao zaseban proces kojem transakcije šalju zahtjeve za odobravanjem lock-ova
- *lock manager* odgovara na zahtjeve tako da vraća poruke kojima odobrava lock (*grant*) ili poruku kojom traži od transakcije rollback zbog nastanka deadlock-a
- transakcija koja traži zahtjev mora čekati do odgovora manangera
- lock manager održava strukturu podataka pod nazivom lock tablica u kojoj vodi evidenciju odobrenih zahtjeva i onih koji čekaju
- lock tablica se u pravilu implementira kao in-memory hash tablica koja je indeksirana s nazivima podatkovnih elemenata za koje se traži lock

Lock tablica



- tamno plavi kvadrati označavaju odobrene lock-ove; svjetlo plavi zahtjeve na čekanju
- lock tablica bilježi tipove lock-ova koji su odobreni i traženi
- novi zahtjevi se dodaju na kraj reda za željeni podatkovni element, i odobravaju kada su kompatibilni s ranijim lock-ovima
- otpuštanje zahtjeva rezultira s brisanjem zahtjeva, te provjerom slijedećih zahtjeva za odobravanje
- ukoliko transakcija prekine izvođenje svi odobreni zahtjevi, te oni na čekanju, se brišu

Upravljanje zastojima

- sustav je u zastoju ukoliko postoji skup transakcija takvih da svaka transakcija u skupu čeka drugu transakciju
- **prevencija zastoja** (*engl. deadlock prevention*) protokoli osiguravaju da sustav nikada ne uđe u stanje zastoja
- neke od strategija su:
  - zahtijevanje da svaka transakcija zaključa sve podatkovne elemente prije nego što počne izvođenje (predeclaration)
  - stvaranje redoslijeda svih podatkovnih elemenata i zahtijevanje da transakcija može zaključati podatkovne elemente samo po stvorenom redoslijedu

Prevencija zastoja

- postoje i druge strategije prevencije zastoja, od kojih neke koriste vrijeme nastanka (timestamp) transakcije za prevenciju
- ***wait-die shema*** - non-preemptive
  - starija transakcija smije čekati mlađu za otpuštanje podatkovnog elementa (starija znači manji timestamp)
  - mlađa transakcija nikada ne čeka stariju već radi rollback
  - transakcija može biti otkazana nekoliko puta prije nego što dohvati željeni podatkovni element
- ***wound-wait shema*** – preemptive
  - starija transakcija prekida (prisilni rollback) mlađu transakciju umjesto da ju čeka, mlađa transakcija može čekati stariju
  - može rezultirati s manje rollback operacija od wait-die sheme

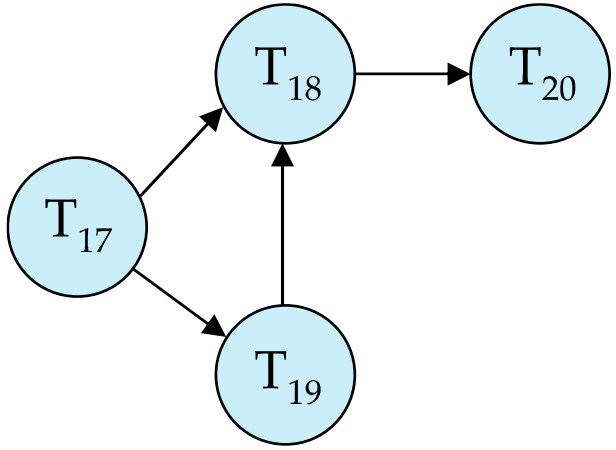
Prevencija zastoja

- kod obje sheme *wait-die* i *wound-wait*, transakcije koje naprave rollback se ponovno pokreću s originalnim timestamp-om
  - stoga starije transakcije imaju prednost ispred novih čime se izbjegava starvation
- ***timeout-based schemes***
  - transakcija čeka lock samo određeno vrijeme
    - ukoliko lock nije odobren u tom vremenu, transakcija radi rollback te se ponovno pokreće
    - stoga, deadlock nije moguć
    - jednostavno za implementaciju; pojava starvationa je moguća
    - teško je odrediti najbolje vrijeme za čekanje

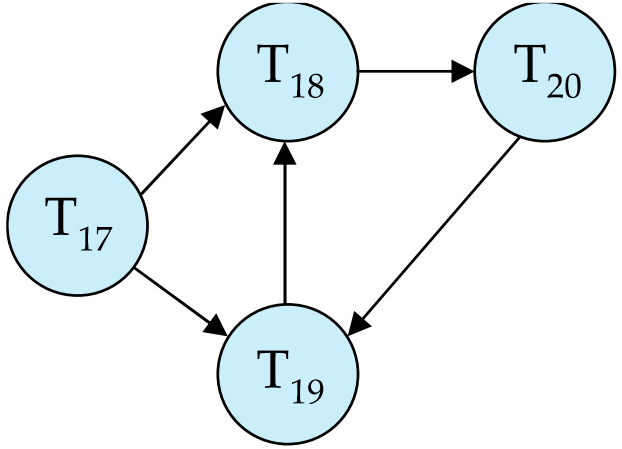
Detekcija zastoja

- zastoj se može opisati *wait-for* grafom, $G = (V,E)$,
  - V je skup čvorova (engl. vertices) – sve transakcije u sustavu
  - E je skup bridova (engl. edges) – svaki element je uređeni par $T_i \rightarrow T_j$
- ako je $T_i \rightarrow T_j$ u E, tada postoji usmjereni brid iz T_i u T_j koji implicira da T_i čeka T_j na otpuštanje elementa
- kada T_i pošalje zahtjev za podacima koji su trenutno zaključani od T_j tada se brid $T_i \rightarrow T_j$ umeće u graf
  - brid se uklanja samo kada T_j više ne drži podatke koje traži T_i
- sustav je u zastoju ako i samo ako *wait-for* graf ima ciklus
  - za traženje ciklusa periodično se mora koristiti algoritam za detekciju zastoja

Detekcija zastoja



wait-for graf bez ciklusa



wait-for graf sa ciklusom

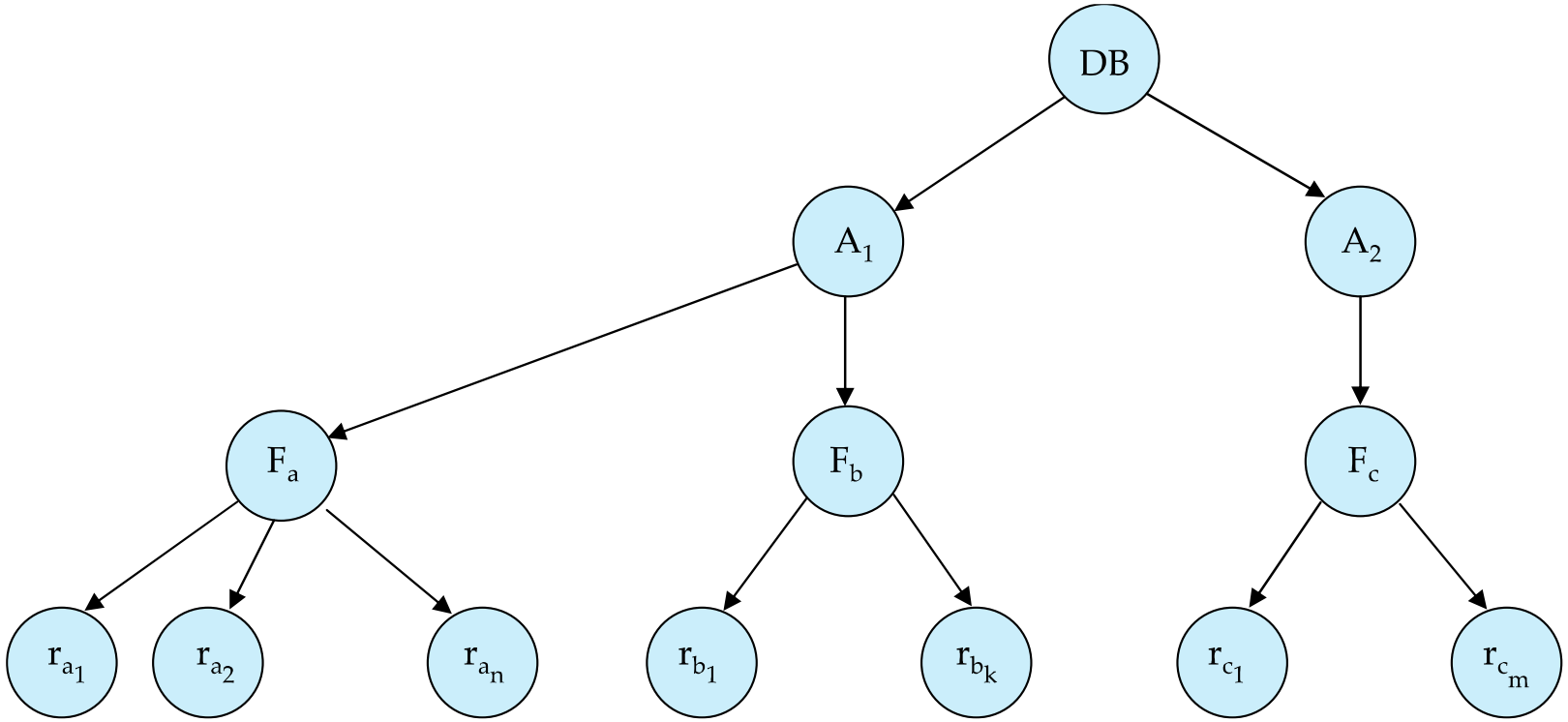
Oporavak iz zastoja

- kada se otkrije zastoj:
  - neka od transakcija će morati napraviti rollback (postati žrtva) da bi se zastoj razbio
    - potrebno je odabrati transakciju koja će stvoriti najmanji trošak
  - rollback – potrebno je odrediti koliko daleko će se transakcija vratiti
    - potpuni rollback (engl. total rollback) – prekini transakciju te ju ponovno pokreni iz početka
    - učinkovitije je vratiti transakciju samo toliko daleko da bi se zastoj razbio
  - starvation se događa ukoliko se uvjek ista transakcija odabire za žrtvu
  - potrebno je voditi evidenciju tj. uključiti broj rollback akcija u računanje faktora troška da bi se izbjegla starvation situacija

Višestruka granularnost

- *engl. multiple granularity*
- dopuštanje podatkovnim elementima da budu različitih veličina i definiranje hijerarhija između njih na način da elementi manje granularnosti budu ugniježđeni unutar većih
- grafički se može prikazati kao stablo
- kada transakcija eksplicitno zaključa neki čvor, implicitno zaključa i sve čvorove potomke tog čvora
  - zaključati se može bilo koji čvor
  - implicitno zaključavanje se radi u istom modu

Primjer višestruke granularnosti



- <u>razine:</u>
  - baza podataka
  - područje
  - datoteka
  - zapis

Intention lock modovi

- kao dodatak *S* i *X* modovima, kod višestruke modularnosti postoje dodatna tri lock moda:
  - **intention-shared** (IS): označava eksplicitno zaključavanje na nižim razinama stabla u S modu
  - **intention-exclusive** (IX): označava eksplicitno zaključavanje na nižim razinama stabla u S ili X modu
  - **shared and intention-exclusive** (SIX): podstablo koje ima korijen u promatranom čvoru je eksplicitno zaključano u *S* modu, i eksplicitno zaključavanje je učinjeno na nižim razinama pomoću exclusive-mode lock-a
- dodatni modovi omogućavaju da čvorovi u višim razinama mogu biti zaključani u *S* ili *X* modu bez da se moraju provjeravati svi čvorovi potomci

Matrica kompatibilnosti sa svim modovima

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Višestruka granularnost

- protokol zaključavanja temeljen na višestrukoj granularnosti zahtjeva od transakcije ukoliko želi zaključati element *Q* da:
  - primjenjuje matricu kompatibilnosti
  - korijen stabla mora biti zaključan prvo, te može biti zaključan u bilo koje modu
  - čvor *Q* može biti zaključan od T_i u *S* ili *IS* modu samo ukoliko je roditelj od *Q* zaključan od T_i u *IX* ili *IS* modu
  - čvor *Q* može biti zaključan od T_i u *X*, *SIX*, *IX* modu samo ukoliko je roditelj od *Q* zaključan od T_i u *IX* ili *SIX* modu
  - T_i može zaključati čvor samo ukoliko prethodno nije otključala niti jedan čvor (tj. T_i je dvo-fazna)
  - T_i može otključati čvor *Q* samo ukoliko nitko od *Q* djece nije trenutno zaključan od T_i
- zaključavanje se radi od korijena prema listu, a otključavanje od lista prema korijenu
- zastoji su mogući u ovom protokolu

Timestamp-Based Protocols

- svakoj transakciji se dodjeljuje timestamp kada ulazi u sustav
  - ukoliko starija transakcija T_i ima timestamp TS(T_i), novoj transakciji T_j se dodjeljuje timestamp TS(T_j) takav da TS(T_i)<TS(T_j)
- protokol upravlja istodobnim izvršavanjem tako da timestamp određuje redoslijed serijabilnosti
- s ciljem da osigura takvo ponašanje, protokol bilježi dvije timestamp vrijednosti za svaki podatak Q:
  - **W-timestamp**(Q) ja najveći timestamp bilo koje transakcije koja je uspješno izvršila write(Q)
  - **R-timestamp**(Q) ja najveći timestamp bilo koje transakcije koja je uspješno izvršila read(Q)

Timestamp-Based Protocols

- timestamp protokol (*engl. timestamp-ordering protocol*) osigurava da se bilo koja konfliktna *read* i *write* operacija izvršava po timestamp redoslijedu
- pretpostavimo da transakcija T_i zatraži *read(Q)*
  - ako TS(T_i) <= W-timestamp(Q) tada T_i treba čitati vrijednost Q koja je već promijenjena
    - stoga se *read* operacija odbija, a T_i radi *rollback*
  - ako TS(T_i) >= W-timestamp(Q) tada se izvršava *read* operacija, a R-timestamp(Q) se postavlja na *max(R-timestamp(Q), TS(T_i))*

Timestamp-Based Protocols

- pretpostavimo da transakcija T_i zatraži *write(Q)*
  - ako TS(T_i) < R-timestamp(Q) tada vrijednost od Q koju T_i tek treba stvoriti je bila prethodno potrebna, stoga sustav pretpostavlja da vrijednost niti ne treba mijenjati
    - *write* operacija se odbija, a T_i radi rollback
  - ako TS(T_i) < W-timestamp(Q) tada T_i pokušava zapisati zastarjelu vrijednost za Q
    - *write* operacija se odbija, a T_i radi rollback
  - u ostalim slučajevima, *write* operacija se izvršava, a W-timestamp(Q) se postavlja na TS(T_i)
- ukoliko je transakcija odbijena od *ccs*, sustav joj dodjeljuje novi timestamp i ponovno je pokreće

Ispravnost protokola

- rasporedi koji su mogući s timestamp-ordering protokolom ne moraju biti mogući s two-faze locking protokolom, i obrnuto
- timestamp-ordering protokol osigurava konflikt serijabilnost
- protokol također osigurava proces od pojave zastoja, obzirom da niti jedna transakcija nikada ne čeka
  - starvation problem je moguć
- ali raspored ne mora biti cascadeless, a čak ne mora biti ni recoverable

Ispravnost protokola

- problem s timestamp-ordering protokolom:
  - pretpostavimo da T_i prekine izvođenje, ali je T_j pročitala podatke izmijenjene od T_i
  - tada se i T_j mora prekinuti, u slučaju da je transakcija T_j već napravila commit, raspored ne osigurava povratak (*engl. not recoverable*)
  - nadalje, svaka transakcija koja je čitala podatke zapisane od T_j se mora prekinuti
  - što dovodi do kaskadnog rollback-a
- postoji više rješenja za definirani problem, koja se zasnivaju na različitim pristupima (od pažljivog definiranja strukture transakcije do ograničenog oblika zaključavanja)

Thomas Write pravilo

- je modificirana verzija timestamp-ordering protokola u kojem se zastarjele operacije pisanja mogu ignorirati ukoliko su zadovoljeni određeni uvjeti
- ukoliko pokuša zapisati vrijednost u *Q*, a $TS(T_i) < W\text{-}timestamp(Q)$ tada T_i pokušava zapisati zastarjelu vrijednost za *Q*
  - umjesto da se radi *rollback* transakcije T_i, kao što se to radi po timestamp-ordering protokolu, ta ista *write* operacija se može ignorirati
- u ostalim scenarijima dva protokola su ista
- omogućava veći potencijal istodobnosti
  - više rasporeda

Validation-Based Protocol

- izvršavanje transakcije se odvija u tri faze:
  - čitanje i izvršavanje: transakcija T_i zapisuje samo u privremene lokalne varijable
  - validacija: transakcija T_i izvodi *test* da bi provjerila da li njeno izvršavanje, tj. zapisivanje lokalnih varijabli u bazu podataka, krši pravila serijabilnosti
  - pisanje: ukoliko je validacija uspješna, transakcija može unijeti promjene u bazu podataka, u suprotnom se radi rollback
- svaka transakcija mora proći kroz faze u navedenom redoslijedu
- faze se mogu naizmjenično izvršavati u transakcijama koje se izvršavaju istodobno
- ponekad se naziva i optimistična kontrola istodobnosti je se operacije u potpunosti izvršavaju u nadi da se transakcija proći test u fazi validacije

Validation-Based Protocol

- svaka transakcija evidentira tri vremena:
  - **Start**(T_i) – vrijeme kada je transakcija T_i počela izvođenje
  - **Validation**(T_i) – vrijeme kada je transakcija T_i ušla u fazu validacije
  - **Finish**(T_i) – vrijeme kad je transakcija završila T_i fazu pisanja
- redoslijed serijalizacije je određen vremenom ulaska u fazu validacije, s ciljem povećanja istodobnosti
- ovaj protokol je koristan i daje veći stupanj istodobnosti ukoliko je vjerojatnost konflikata mala:
  - redoslijed serijalizacije nije unaprijed određen
  - vrlo malo transakcija će morati raditi rollback

Validacija transakcije T_j

- za sve T_i za koje vrijedi TS(T_i) < TS(T_j) jedan od slijedećih uvjeta vrijedi:
  - finish(T_i) < start(T_j)
  - start(T_j) < finish(T_i) < validation(T_j) i skup podataka zapisanih od T_i se ne preklapa sa skupom podataka koji je pročitala T_j
- tada se T_j transakcija potvrđuje, u protivnom validacija neće proći te se transakcija prekida
- ukoliko se ostvari prvi uvjet tada ne postoji preklapanja u izvršavanju, u slučaju drugog uvjeta dvije su mogućnosti:
  - zapisivanja iz transakcije T_j ne utječu na čitanja iz transakcije T_i obzirom da se događaju kada je transakcija T_i završila čitanje
  - zapisivanja iz transakcije T_i ne utječu na čitanja iz transakcije T_j jer T_j ne čita niti jedan podatak koji je zapisala T_i

Raspored stvoren validacijom

- primjer rasporeda stvorenog validacijom

T_{25}	T_{26}
read (B)	
	read (B)
	$B := B$ 50
	read (A)
	$A := A + 50$
read (A)	
⟨ *validate* ⟩	
display ($A + B$)	
	⟨ *validate* ⟩
	write (B)
	write (A)

Multiversion sheme

- čuvaju starije verzije podataka da bi povećale istodobnost
- postoje dvije verzije:
  - *Multiversion Timestamp Ordering*
  - *Multiversion Two-Phase Locking*
- svaki uspješan *write* stvara novu verziju podataka koji su zapisani
- timestamp se koristi da se verzije imenuju
- kada se izdaje *read(Q)* naredba, odabire se ispravna verzija podataka temeljem timestamp vrijednosti transakcije
- operacija *read* nikada ne mora čekati jer joj je ispravna verzija podataka odmah proslijeđena

Multiversion Timestamp Ordering

- svaki podatkovni element *Q* ime niz verzija <Q_1, Q_2,...., Q_m>
- svaka verzija Q_k sadrži tri podatkovna polja:
  - **content** – vrijednost verzije Q_k
  - **W-timestamp**(Q_k) - timestamp transakcije koja je stvorila (wrote) verziju Q_k
  - **R-timestamp**(Q_k) - najveći timestamp transakcije koja je uspješno pročitala verziju Q_k
- kada transakcija T_i stvori Q_k verziju *Q*, tada se podaci *W-timestamp* and *R-timestamp* nove verzije inicijaliziraju na TS(T_i)
- R-timestamp od Q_k se ažurira kada transakcija T_j pročita Q_k, i TS(T_j) > R-timestamp(Q_k).

Multiversion Timestamp Ordering

- pretpostavimo da T_i izda read(Q) ili write(Q) operaciju; neka Q_k označava verziju Q čiji W-timestamp je najveći W-timestamp koji je manji ili jednak TS(T_i)
  1. ukoliko transakcija T_i izda **read**(Q), tada će joj biti vraćen sadržaj Q_k
  2. ukoliko transakcija T_i izda **write**(Q)
     1. ako TS(T_i) < R-timestamp(Q_k), tada transakcija T_i radi rollback
     2. ako TS(T_i) = W-timestamp(Q_k), sadržaj Q_k se prepisuje novim sadržajem
     3. u suprotnom se stvara nova verzija Q
- read operacija je uspješna u svakom slučaju
- protokol garantira serijabilnost

Multiversion Two-Phase Locking

- razlikuje transakcije koje samo čitaju (*read-only*) i one koje mijenjaju (*update*) sadržaj
- update transakcije dohvaćaju *read* i *write* lock-ove, i sve ih zadržavaju do kraja transakcije, tj. update transakcije prate rigorous two-phase locking
  - svaki uspješan write rezultira stvaranjem nove verzije podatkovnog elementa koji se zapisuje
  - svaka verzija podatkovnog elementa ima jedinstveni timestamp čija vrijednost se dohvaća iz brojača ts-counter koji se povećava tijekom obrade commit-a
- read-only transakcijama se dodjeljuje timestamp trenutne vrijednosti ts-counter-a prije nego što transakcija krene s izvršavanjem
  - prate multiversion timestamp-ordering protocol za izvođenje čitanja

Multiversion Two-Phase Locking

- kada update transakcija želi čitati podatkovni element:
  - dohvati shared lock na elementu, i čita posljednju verziju
- kada želi zapisati podatkovni element
  - dohvaća *X* lock on; stvara novu verziju elementa i postavlja timestamp nove verzije ∞
- kada update transakcija T_i završi, izvodi se *commit*:
  - T_i postavlja timestamp na verzijama koje je kreirao na *ts-counter* + 1
  - T_i povećava brojač *ts-counter* za 1
- read-only transakcije koje počinju nakon T_i poveća *ts-counter* i vidjeti će promjene koje je napravila T_i
- read-only transactions koje počinju prije T_i poveća *ts-counter* vidjeti će vrijednosti koje su bile prije izmjene T_i.
- stvaraju se samo serijalizirani rasporedi

MVCC problemi implementacije

- stvaranje više verzija opterećuje sustav pohrane
- za starije verzije se može koristiti garbage collection
  - npr. ako *Q* ima dvije verzije *Q5* i *Q9*, a najstarija aktivna transakcija ima timestamp > 9, tada *Q5* nikada više neće biti potrebna i može biti pobrisana

Snapshot Isolation

- sustavi za potporu odlučivanju koji čitaju velike količine podataka dolaze u konflikt s OLTP transakcijama koje mijenjanju samo nekoliko redova
  - rezultat su loše performanse
- rješenje 1: pružanje logičke snimke baze podataka read-only transakcijama, read-write transakcije koriste normalno zaključavanje
  - multiversion 2-phase locking
  - dobro rješenje, ali kako sustav može prepoznati read-only transakciju?
- rješenje 2: pružanje snimke stanja baze podataka svim transakcijama, samo update transakcije koriste 2-phase locking kao zaštitu od istodobnih transakcija
  - problem: različite anomalije kao što je izgubljeni update
  - djelomično rješenje: snapshot isolation level
    - predloženo od Berenson et al, SIGMOD 1995
    - varijante implementirane u mnogim sustavima baze podataka
      - Oracle, PostgreSQL, SQL Server 2005

Snapshot Isolation

- transakcije se izvode koristeći snapshot isolation:
  - prilikom pokretanja transakcija uzima snimku potvrđenih podataka
  - uvjek čita/modificira podatke u svojoj vlastitoj snimci
  - izmjene nisu vidljive u istodobnim transakcijama
  - zapisivanje se završava s potvrdom
- first-committer-wins pravilo:
  - potvrđuje se transakcija samo ako niti jedna istodobna transakcija nije zapisivala podatke koji se žele izmijeniti

T1	T2	T3
W(Y := 1) Commit		
	Start R(X) → 0 R(Y)→ 1	
		W(X:=2) W(Z:=3) Commit
	R(Z) → 0 R(Y) → 1 W(X:=3) Commit-Req Abort	

Snapshot read

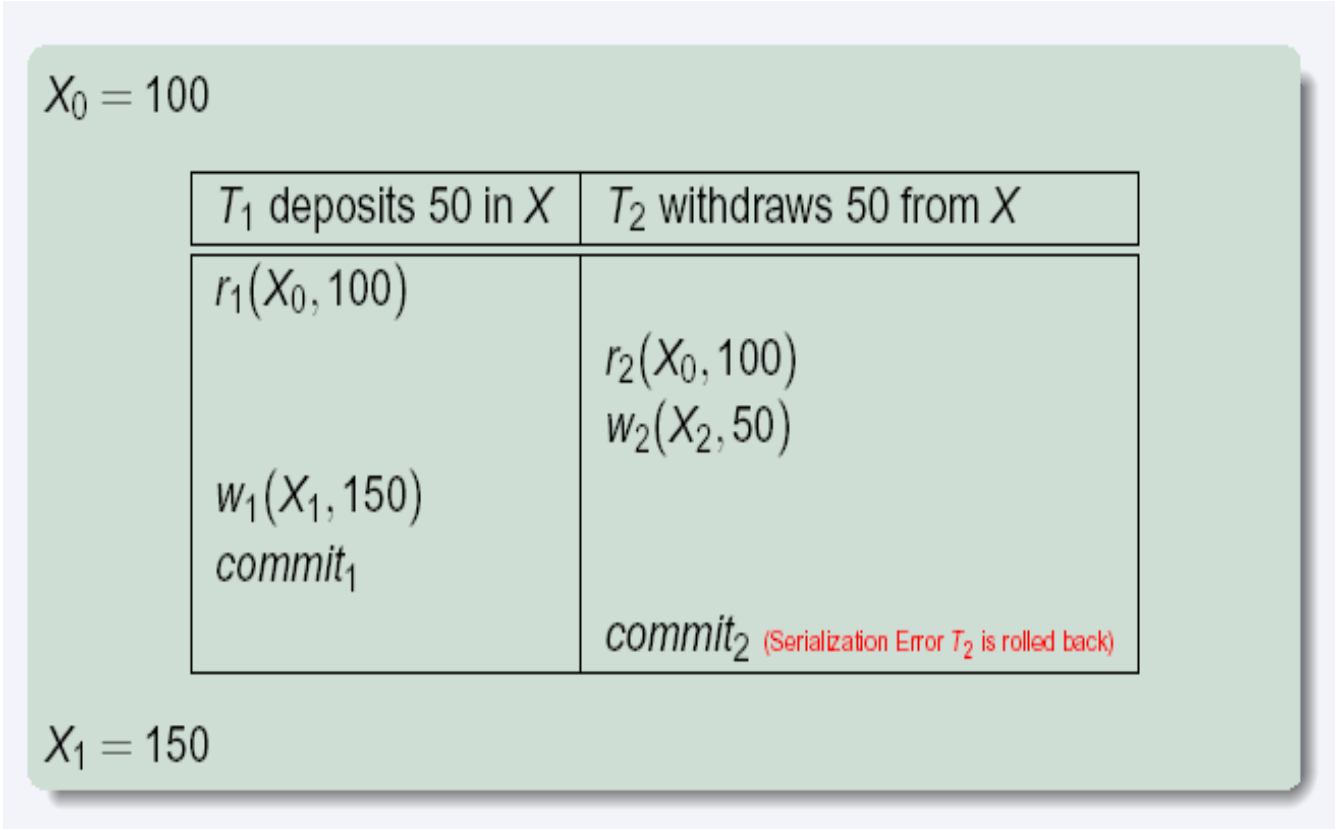
- istodobna izmjena je nevidljiva za snapshot read

$X_0 = 100,\ Y_0 = 0$

T_1 deposits 50 in Y	T_2 withdraws 50 from X
$r_1(X_0, 100)$	
$r_1(Y_0, 0)$	
	$r_2(Y_0, 0)$
	$r_2(X_0, 100)$
	$w_2(X_2, 50)$
$w_1(Y_1, 50)$	
$r_1(X_0, 100)$ (update by T_2 not seen)	
$r_1(Y_1, 50)$ (can see its own updates)	
	$r_2(Y_0, 0)$ (update by T_1 not seen)

$X_2 = 50,\ Y_1 = 50$

Snapshot Write: First Committer Wins



- varijanta: First-updater-wins

Prednosti Snapshot izolacije

- čitanje nikada nije blokirano
  - niti ne blokira aktivnosti drugih transakcija
- performanse su slične Read Committed
- ne dopušta tipične anomalije:
  - dirty read
  - no lost update
  - no non-repeatable read
  - no phantoms
- problem sa SI:
  - SI ne daje uvjek serijabilno izvršavanje

Snapshot Isolation

- primjer problema sa SI:
  - T1: x := y
  - T2: y := x
  - inicijalne vrijednosti x = 3 i y = 17
    - serijsko izvođenje: x = ?, y = ?
    - ako obje transakcije započnu u isto vrijeme, izvođenje Snapshot izolacijom: x = ?, y = ?
- pojam se naziva *skew write*
- također se može dogoditi i kod umetanja:
  - pronađi najveći broj računa
  - stvori novi račun s brojem = max prijašnji + 1

Anomalije Snapshot izolacije

- SI narušava serijabilnost kada transakcije modificiraju različite elemente, od kojih se svaki temelji na prijašnjem stanju elementa kojeg je druga transakcija modificirala
  - ne događa se često u praksi
  - kada su transakcije u konfliktu zbog modificiranja različitih podataka, često postoji i zajednički element kojeg obje modificiraju stoga će SI prekinuti jednu od njih
  - serializable snapshot isolation (SSI) – nadogradnja SI koja podržava serijabilno izvršavanje
- SI može uzrokovati i anomaliju read-only transakcije gdje ista vidi nekonzistentno stanje
- korištenje snimaka za provjeru primarnog/stranog ključa može dovesti do nekonzistentnosti
  - provjeravaju se prilikom commit-a na stvarnoj bazi podataka ne na snimci

SI u Oracle-u i PostgreSQL

- SI se koristi kada je razina izolacije postavljena na *serializable*
- vrijedi za PostgreSQL do verzije 9.1
- Oracle implementira *first-updater-wins* tj. varijantu od *first-committer wins*
  - provjera istodobnosti se vrši prilikom operacije write, a ne na commit
  - omogućava transakciji da prije napravi rollback
  - Oracle i PostgreSQL < 9.1 ne podržavaju pravo serijalizirano izvršavanje!?
- PostgreSQL 9.1 je predstavio novi protokol “Serializable Snapshot Isolation” (SSI)
  - novi protokol može osigurati pravu serijalizaciju

Literatura

- Pročitati
  - [DSC] poglavlje 15.
  - prezentacija Poglavlje 15. – Database System Concepts; Silberschatz, Korth i Sudarshan
  - detalji o SSI [DSC] 15.8
- Slijedeće predavanje
  - [DSC] poglavlje 16.