Konkurencia a transakcie

Concurency control
Transaction management

Databázový systém predpokladá súčasnú prácu viacerých uživateľov klientov.

Definícia:
Transakcia je program obsahujúci operácie s bázou dát. Operácie s db sú: zápis (*w*), čítanie (*r*), vyhodnotenie dotazu (*q*), commit (c) a abort (a).

Tento model je zjednodušený *w* znamená insert, update alebo delete.

Vlastnosti transakcií - ACID

Očakávame, že transakcie sa budú chovať „rozumne“. Zatial predpokladáme, že transakcia nemá záškodnícke ciele, môže však z nejakého dôvodu „zlyhať“.

- Atomičnosť (všetko alebo nič).
- Consistency transformuje databázu z konzistentného stavu do konzistentného stavu.
- Nezávislosť (Isolation) činnosť transakcie je neovplyvniteľná činnosťou iných transakcií.
- Trvanlivosť (Durability) výsledky transakcie po jej „úspešnom skončení“ pretrvajú v databáze.

Transakcie - príklad

Transakcia = databázový program, ktorý sa musí buď celý vykonať, alebo sa z neho nesmie vykonať nič.

set transaction

…

commit;

Program každého užívateľa je postupnosť transakcií. Pre databázy je charakteristické, že sa viac programov vykonáva súčasne.

Príklad: Rezervácia letenky (Bratislava – Singapore).

(Bratislava - Viedeň - Rím - Kalkata - Singapúr)

Syntax: [Set / Start] transaction /* začiatok transakcie */
… rollback … /* transakcia „sa rozhodla“, že neurobí nič */

commit[work]; /* práve jeden; transakcia „si myslí“, že úspešne skončila. Systém môže mať iný názor. */

Paralelné vykonanie – rozvrh

Či už pôjde o skutočný paralelizmus alebo simulovaný paralelizmus, budeme predpokladať, že operácie s bázou dát sa neprekrývajú.

Definície:
Rozvrh S = Sched({T_1, … , T_n}) je postupnosť operácii taká, že obsahuje práve všetky operácie uvedených transakcií, pričom postupnosť operácii každej transakcie je podpostupnosť S.

Formálne rozvrh je postupnosť trojíc tvaru <T, o, data>. V rozvrhu každá transakcia končí commit alebo abort.

Rozvrh S je sériový, ak je rovný postupnosťí operácii v nejakej z permutácii <T_1, … , T_n>.

Rozvrh S je sériovateľný, ak je ekvivalentný (vedie k rovnakému stavu databázy a dáva rovnaké výsledky jednotlivých transakcií) niektorému z n! sériových rozvrhov.

Príklad:

A B

transaction T_1:{a:= a+2; b:=3×b;}

transaction T_2:{a:= 3×a; b:=b+2;}

T_1T_2 a=3a+6, b=3b+2

T_2T_1 a=3a+2, b=3b+6

Výsledok *S:* a=3a+6, b=3b+6

Rozvrh: *S*

krok	T_1	T_2
1	A	-
2	-	A
3	-	B
4	B	-

Jemnejšie členenie akcií: **read,** compute, **write.**
Kritické operácie sú read a write.

Rozbitie kritických akcií:
operačný systém: **lock** a; read a; **unlock** a;
lock a; write a; **unlock** a;

„Sémantická“ ekvivalentnosť transakcií

Vo všeobecnosti nie sme schopní analyzovať, čo transakcie počítajú (robia). Predpokladáme, že pri každom zápise premennej *a* sa realizuje priradenie:

a := f(a, všetky premenné prečítané pred write a);

kde f_i je pri každom zápise nová funkcia. Znalosť, že niektoré časti transakcií sa opakujú alebo sú rovnaké, nevyužívame.

Príklad:

T_1		T_2		T_3	
read a		read b		read a	
read b		read c		read c	
write a	$f_1(a, b)$	write b	$f_3(b, c)$	write c	$f_6(a, c)$
write b	$f_2(a, b)$	read a		write a	$f_7(a, c)$
		write c	$f_4(a, b, c)$		
		write a	$f_5(a, b, c)$		

Krok	akcia	a	b	c
1	T_1:read a			
2	T_2:read b			
3	T_2:read c			
4	T_2:write b		$f_3(b, c)$	
5	T_1:read b	φ		
6	T_1:write a	$f_1(a, f_3(b, c))$		ψ
7	T_2:read a			
8	T_2:write c			$f_4(f_1(a, f_3(b, c)), b, c)$
9	T_2:write a	$f_5(\varphi, b, c)$		
10	T_3:read a			
11	T_3:read c			
12	T_1:write b		$f_2(a, f_3(b, c))$	
13	T_3:write c			$f_6(f_5(\varphi, b, c), \psi)$ FW
14	T_3:write a	$f_7(f_5(\varphi, b, c), \psi)$		

Výsledky transakcií

Obvykle každá transakcia okrem toho, že mení stav databázy, produkuje správu o tom, či úspešne skončila a výsledok. Ak pridáme túto informáciu k stavu databázy. Stačí výpočet rozvrhu.

Definícia: Hovoríme, že v rozvrhu *S* transakcia T_j číta hodnotu *x* z transakcie T_i *(i≠j)*, ak mezi operáciami T_i*:write x* a T_j*:read x* neexistuje v rozvrhu *S* žiadna operácia *write x*.

Tajne predpokladáme, že každá transakcia jednu hodnotu, číta a piše najviac raz (inak by sme museli čitánie a zápis v rámci transakcie číslovať). Navyše, keď hovoríme o sériovateľnosti, nemôže sa stať, že T_i podľa jednej premennej číta z T_j a podľa inej premennej je tomu naopak.

Definícia: Hovoríme, že rozvrhy *S* a *S'* sú view ekvivalentné, ak:

- Pre každé *i≠j* T_j číta z T_i v *S* práve vtedy, keď je to tak v *S'*.
- *FW(S) = FW(S')*

Sériovateľnosť

Definícia: Rozvrh je sémanticky sériovateľný, ak je sémanticky ekvivalentný nejakému sériovému rozvrhu.

Definícia: Rozvrh je view sériovateľný, ak je view ekvivalentný nejakému sériovému rozvrhu.

Hoci testovanie view ekvivalentnosti je ľahké (lineárne). View sériovateľnosť je ťažká (NP – úplný problém).

Definícia: Akcie T_i: o_1 x a T_j: o_2 x sú v konflikte, ak $i \neq j$ a aspoň jedna s operácií o_1, o_2 je write.

Označme *Con(S)* množinu dvojíc $<a_1, a_2>$ konfliktov rozvrhu *S*.

Definícia: Rozvrhy S a S' také, sú konflikt ekvivalentné, ak Con(S) = Con(S').

Definícia: Rozvrh S je konflikt sériovateľný, ak je konflikt ekvivalentný niektorámu sériovému rozvrhu.

Polygrafy

Definícia: Polygraf $G = \langle N, E, P \rangle$, kde N je možina vrcholov, E množina orientovaných hrán a P množina polyhrán – dvojíc orientovaných hrán nepatriacich do E.

Definícia: Polygraf G je acyklický, ak existuje taký výber P' hrán, z každej dvojice v P jednej, že graf $G' = \langle N, E \cup P' \rangle$ je acyklický.

Hoci testovanie na acykličnosť polygrafu vyzerá jednoducho, existuje $2^{|P|}$ rôznych výberov pre P'. Oveľa múdrejšie sa to ani nedá urobiť. Problém je NP-úplný.

Konštrukcia polygrafu k rozvrhu.

1. Pridaj umelé transakcie: T_0, ktorá nič nečíta a zapíše všetky, dáta vyskytujúce sa v rozvrhu. T_f, ktorá prečíta všetky dáta v rozvrhu a výsledky transakcií. Každej transakcii priraď uzol polygrafu.
2. Pridaj hranu $T_i \xrightarrow{a} T_j$, ak pre nejaký údaj a v rozvrhu S je T_j:*read a* prvé čítanie a po T_i:*write a*.
3. Najdi zbytočné uzly (z ktorých nevedie cesta do T_f) a odstráň hrany do nich vchádzajúce.
4. Pre každú hranu $T_i \xrightarrow{a} T_j$, a každú inú transakciu T, ktorá zapisuje *a* pridaj
   1 if $(T_i \neq T_0 \wedge T_j \neq T_f)$ *polyhranu* $(T \rightarrow T_i, T_j \rightarrow T)$
   2 else if $(T_i = T_0 \wedge T_j \neq T_f)$ *hranu* $T_j \rightarrow T$
   3 else if $(T_i \neq T_0 \wedge T_j = T_f)$ *hranu* $T \rightarrow T_i$

Test view sériovateľnosti

Veta 1: Rozvrh *S* je view sériovateľný práve vtedy, ak jeho polygraf je acyklický.

Dôkaz: Ak polygraf je acyklický v procese testovania acykličnosti, správnou orientáciou hrán, dostaneme acyklický graf. Topologickým utriedením tohto graf získame ekvivalentný sériový rozvrh. Má tie isté „*čítania z*“ a tie isté množiny finálnych zápisov. Naopak, ak *S* je sériovateľný, potom existuje ekvivaletný sériový rozvrh *S*'. „Čítania z“ sú určené t.j., ak $w_i(x)$ predchádza $r_j(x)$ v *S*', musí to tak byť aj v *S*. Zápis $w_k(x)$ z transakcie T_k nesmie narušiť poradie $w_i(x)$, $r_j(x)$ musí teda v *S*', T_k predchádzať T_i alebo nasledovať za T_j . Výber poradia T_k z *S*′ je teda konzistetný s polygrafom rozvrhu *S*. Rozvrh *S*′ je sériový a jeho graf následnosti transakcií neobsahuje cyklus. □

Graf následnosti transakcií, test konflikt sériovateľnosti

Konštrukcia grafu následnosti transakcií:

1. Uzly sú transakcie
2. Ak konflikt (T_i: o_1 x, T_j: o_2 x) a akcia T_i: o_1 x v rozvrhu S predchádza akciu T_j: o_2 x, potom hrana $T_i \rightarrow T_j$, inak hrana $T_j \rightarrow T_i$. (Konflikty sú neusporiadané dvojice, poradie v rozvrhu je úplné usporiadanie)

Veta 2: Rozvrh je sériovateľný práve vtedy, ak graf následnosti transakcií je acyklický. Ekvivaletný sériový rozvrh dostaneme topologickým utriedením tohto grafu.

Vzájomný vzťah medzi rôznymi typmi sériovateľnosti

Označíme:

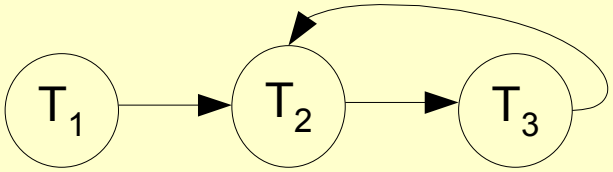
- KS množinu konflikt sériovateľných rozvrhov
- VS množinu view sériovateľných rozvrhov
- SS množinu sémanicky sériovateľných rozvrhov
- MS množinu sériovateľných rozvrhov

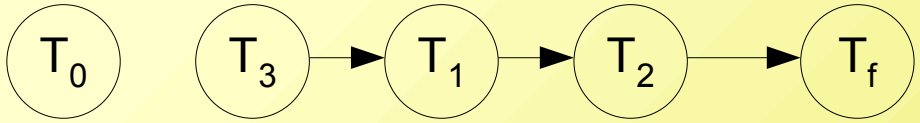
Veta: KS $\subset$ VS $\subset$ SS $\subset$ MS, všetky inklúzie sú vlastné.

Dôkaz: MS požaduje len rovnaký výsledný stav databázy a rovnaké výsledky. SS to isté pre všetky možné výpočty. VS navyše požaduje, aby transakcie aj čítali tie isté hodnoty ako v sériových rozvrhoch. KS žiada zachovať aj poradie akcií pri možných konfliktov. Presnejšie, keď sa zachová poradie transakcií pri konfliktoch zachovajú sa aj „čítania z“ aj finálne zápisy.

Pokračovanie vlastné inklúzie

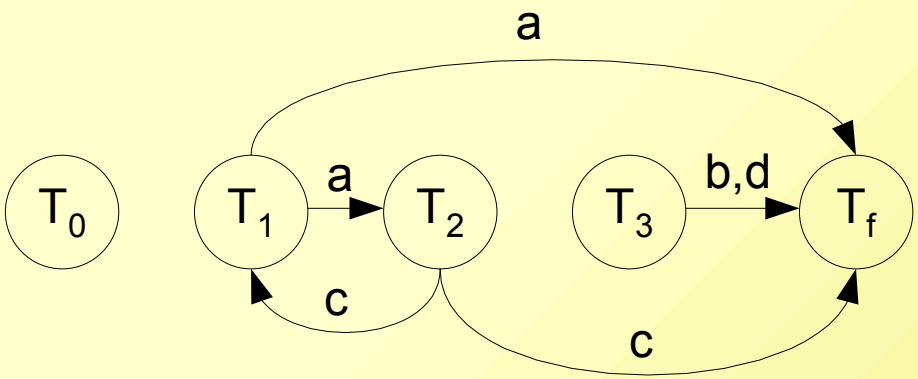
S_1: $w_1x\ r_2x\ w_3x\ w_2x$ je view seriovateľný, ale nie konflikt seriovateľný





S_2: $w_1a\ w_2c\ r_2a\ w_2a\ w_2b\ r_1c\ w_1d\ w_3b\ w_3d$

rozvrh S je sémantický sériovateľný ale nie view sériovateľný



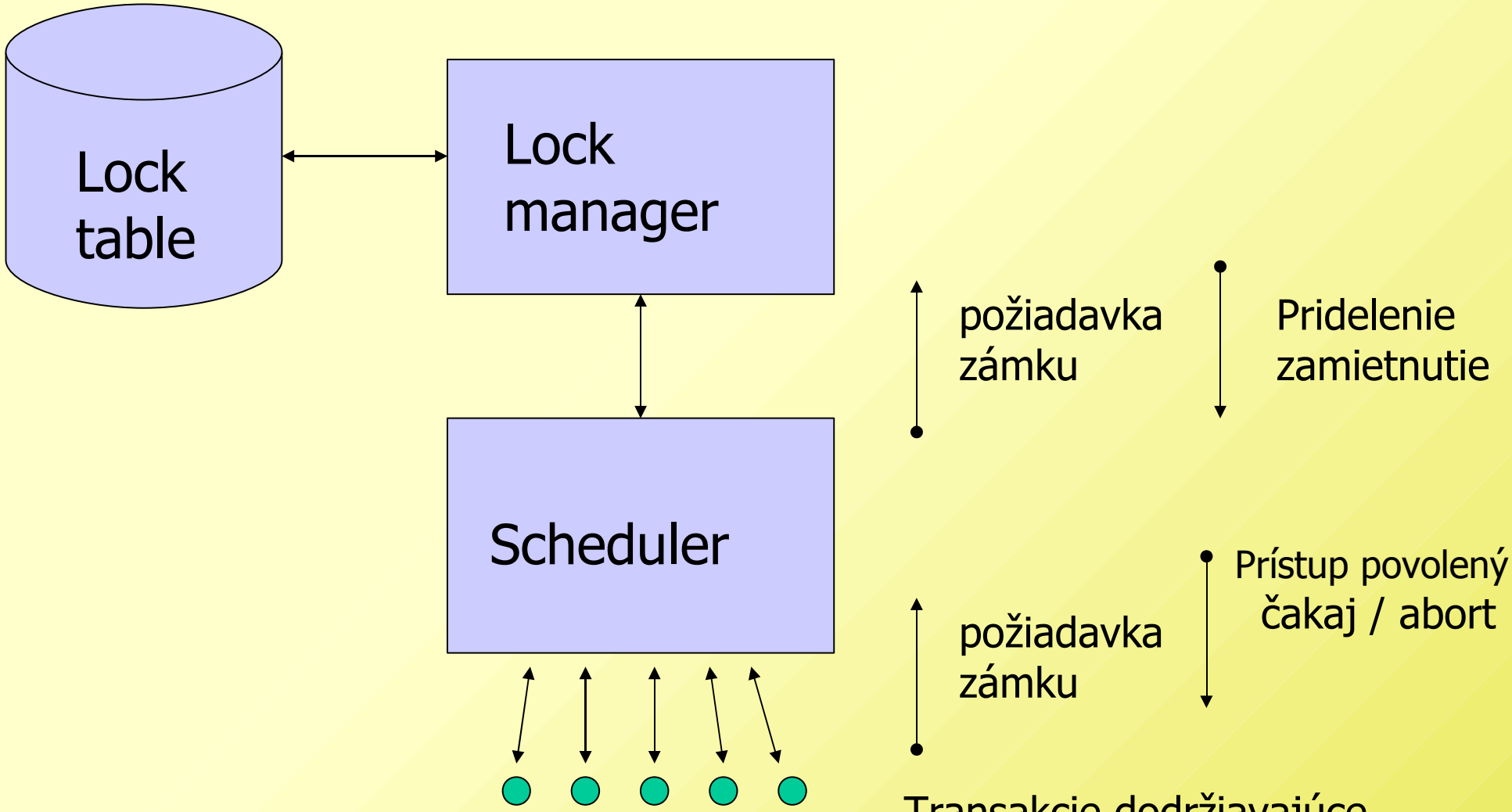
Dôvod je, že transakcia T_3 „odtienií“ nesériovateľný efekt pri výpočte b,d v transakciách T_1 a T_2.

Ad absurdum, ak rozvrh obsahuje transakciu, ktorá bez čítania napíše všetky údaje a jej zápisy sú finálne je sémanticky sériovateľný.

Implementácia

- Zámky – transakcie musia pred kritickými operáciami si zamknúť dáta, aby iné transakcie nemohli s nimi interferovať nežiadúcim spôsobom.
- Časové razítka – transakcie a dáta sú opatrené časovými razítkami. Na základe porovnania časových razítiek sa rozhodne, ako má transakcia pokračovať.
- Validácia – pridelíme transakciám pamäť (buffers) a necháme veciám voľný priebeh a vo vhodný moment rozhodneme, či transakcia má zapísať a commitovať alebo abortovať.
- MVCC sledujeme viac verzií dát. Každá transakcia pracuje s nejakou verziou pred zápisom testujeme, či zápis neporuší seriovateľnosť,

Nástroje - plánovače (schedulers), zámky, protokoly



Realistický model - rlock / wlock

Read (shared) lock: transakcia bude len čítať zamknutú premennú (prípadne ju použije k výpočtu inej premennej). **Rlock** bráni iným transakciám zmeniť zamknutú premennú, ale nebráni jej čítaniu.

Write (exclusive) lock: Zámky v predošlom zmysle. Len jedna transakcia môže mať **wlock** na danú premennú v danom okamihu.

Kompatibilita zámkov:

požadovaný zámok \ existujúci zámok	rlock	wlock
rlock	Yes	No
wlock	No	No

Sériovateľnosť zámkami

Definícia hrán grafu sériovateľnosti:

- Nech T_i má rlock alebo wlock na premennú *a*. Nech T_j je nasledujúca transakcia požadujúca wlock na *a.* Potom hrana $T_i \rightarrow T_j$.
- Nech T_i má wlok na premennú *a*. Nech T_m je nasledujúca transakcia požadujúca rlock na *a* potom, čo ho T_i odomkne. Potom hrana $T_i \rightarrow T_m$.

Veta 1 Acyklický graf je sériovateľný. Topologické triedenie dáva evivalentný sériový rozvrh.

Veta 2 Sériovateľnosť zámkami $\Rightarrow$ konflikt sériovateľnosť

Dvojfázový protokol

Definícia: Protokol sa nazýva ***dvojfázový*** ak v každej jeho transakcii všetky operácie zamykania (lock) predchádzajú prvú operáciu odomykania (unlock) v danej transakcii.

Veta 3 Dvojfázový protokol zaručuje sériovateľnosť**.**

Dôkaz: Nepriamo. Predpokladajme, že rozvrh S je dvojfázový a nie je sérioveľný. Teda graf následnosti transakcií z S obsahuje cyklus $T_{i1} \rightarrow T_{i2} \rightarrow \ldots T_{ik} \rightarrow T_{i1}$. To ale znamená, že nejaké zamykanie T_{ij+1} nasleduje odomknutie v T_{ij}. Podľa toho však Ti1 niečo zamkla potom, čo nejaký zámok uvolnila a to je spor s definíciou dvojfázovosti. □

Ďalšie zámky – ilock

Zámok pre zvýšenie alebo zníženie hodnoty

a += i; resp. a -=d; (Použitie napríklad v bankomatoch.)

Takéto zámky môžeme zaviesť pre komutatívne a asociatívne operácie.

Kompatibilita zámkov:

požadovaný zámok \ existujúci zámok	rlock	wlock	ilock
rlock	Yes	No	No
wlock	No	No	No
ilock	No	No	Yes

Ďalšie zámky – updatelock

Update lock je read lock, ktorý môže byť v priebehu transakcie upgradovaný na write lock.

Použitie pri interaktívnych transakciách, kde uživateľ sa v priebehu transakcie môže rozhodnúť, že chce updatovať hodnoty.

Kompatibilita zámkov:

existujúci zámok

požadovaný zámok	rlock	wlock	Ulock
rlock	Yes	No	No
wlock	No	No	No*
Ulock	Yes	No	No

* okrem transakcie, ktorá drží update lock

Neúspešné transakcie

Dôvody neúspechu (failure) transakcií:

- Prerušenie užívateľom, zlyhanie aritmetickej operácie. Nedostatok práv k prístupu alebo nedostatok prostriedkov.
- Plánovač odhalí deadlock a rozhodne sa transakciu zrušiť.
- Plánovač odvolá transakciu potom, čo detekoval nesério-vateľnosť.
- Zlyhanie software alebo hardware.

commit - posledný príkaz úspešnej transakcie

Neúspešná (aborting) transakcia má za následok **rollback.**

Nečisté (dirty) dáta - dáta zapisané transakciou, ktorá nebola ešte potvrdená (committed).

Journal (log)

Žurnál

- UNDO log – čas, transakcia, údaj, stará hodnota
- REDO log – čas, transakcia, údaj, nová hodnota
- REDO/UNDO log

Pravidlo: poradie zápisu do žurnálu a do databázy.
Najprv do žurnálu!

Do žurnálu okrem toho zapisujeme, začiatok a koniec transakcie: čas, transakcia, start
čas, transakcia, commit
čas, transakcia, abort

Výskyt checkpointu

Kaskádovitý rollback

	T_1	T_2
1	lock a	
2	read a	
3	a:=a-1	
4	write a	
5	unlock a	
6		lock a
7		read a
8		a:=a+2
9		write a
10		unlock a
11		commit
12	lock b	
13	read b	
14	b:=b/a	

Hoci T_2 je committed. Fail T_1 vyvolal neplatnosť premennej a, tým aj nutnosť zrušenia transakcie T_2 .

Striktná dvojfázovosť:

- Transakcia nesmie písať do databázy pokiaľ nedosiahla commit point.
- Transakcia nesmie uvoľniť žiaden zámok pokiaľ nezapísala do databázy.

Časové razítka

Základná myšlienka: každá transakcia dostane časové razítko - okamih začatia.

Sériový rozvrh = rozvrh v poradí časových razítok.

Pravidlá sériovateľnosti:

- Transakcia nemôže čítať hodnoty, ktoré boli napísané neskôr začatou transakciou
- Transakcia nemôže písať hodnoty, ktoré boli prečítané neskôr začatou transakciou

Implementácia namiesto zámku dvojica časov $<t_r, t_w>$; transakcia s časovým razítkom t:

- *read* and $t > t_w$: {*read*; **if** $t_r < t$ **then** $t_r = t$; }
- *write* and $t \geq t_r$ and $t \geq t_w$: {*write*; $t_w := t$; }
- *write* and $t_r \leq t < t_w$: { do nothing }
- otherwise abort;

Neporovnateľnosť sériovateľnosti časovými razítkami a zámkami

S

	T_1	T_2
1		read b
2	read a	
3	write c	
4		write c

T

	T_1	T_2	T_3
1	read a		
2		read a	
3			read d
4			write d
5			write a
6		read c	
7	write b		
8		write b	

Rozvrh *S* je sériovateľný zámkami, ale nie je sériovateľný časovými razítkami. Rozvrh *T* naopak.

Validácia

Transakcie sledujú tri časy: T_{start}, T_{val}, T_{fin} a dve množiny RS(T) read set , WS(T) write set. Transakcie počítajú vo svojom pamäťovom priestore. Tesne pred prvým zápisom do databázy sa uskutoční validácia T_{val}. Ak je validácia úspešná, transakcia zapíše do databázy inak abortuje.

Validácia transakcie T je neúspešná, ak existuje transakcia U splňujúca aspoň jednu z nasledujúcich dvoch podmienok:

- $U_{fin} > T_{start} \wedge U_{val} < T_{val} \wedge (RS(T) \cap WS(U) \neq \emptyset)$
- $U_{fin} > T_{val} \wedge U_{val} < T_{val} \wedge (WS(T) \cap WS(U) \neq \emptyset)$

O časoch udalosti, ktoré ešte nenastali predpokladáme, že sú ∞. Prvá podmienka hovorí, že transakcia T mohla prečítať nejakú hodnotu predtým ako ju U zapísala. Druhá zasa, že U môže prepísať hodnotu, ktorú má zapísať T.

MVCC – multiversion concurency control

Transakcia T dostane časové razítko t.
Vždy keď transakcia, zapíše údaj vznike nová verzia s časom t_w= t.

Ak transakcia T číta nejaký údaj dostane verziu s najväčším časom $t_w < t$.

Jediný konflikt vzniká, keď transakcia T s časovým razítkom t, chce zapísať údaj A, a už existuje jeho verzia A_i s časom $t_w < t$ a časom $t_r > t$. (Transakcia, ktorá čítala, mala prečítať až verziu od T.)

Transakcia T musí abortovať.

Problém: S: r_1a, r_2a, w_2a, w_1a

Postgres aj oracle používajú MVCC !

Výpis z dokumentácie Postgresu

In fact PostgreSQL's Serializable mode does not guarantee serializable execution in mathematical sense. As an example, consider a table mytab, initially containing

```
class | value
-------+-------
    1 |    10
    1 |    20
    2 |   100
    2 |   200
```

Suppose that serializable transaction A computes
SELECT SUM(value) FROM mytab WHERE class = 1;
and then inserts the result (30) as the value in a new row with class = 2.
Concurrently, serializable transaction B computes
SELECT SUM(value) FROM mytab WHERE class = 2;
and obtains the result 300, which it inserts in a new row with class = 1. Then both transactions commit. None of the listed undesirable behaviors have occurred, yet we have a result that could not have occurred in either order serially. If A had executed before B, B would have computed the sum 330, not 300, and similarly the other order would have resulted in a different sum computed by A.