

Prednáška 10

Transakcie

Obsah prednášky

- Súbežné spracovanie
 - Transakcie

Organizácia

- Linky na git z prednášky 8 ORM:
 - Django: <https://github.com/Sibyx/fiit-orm-django-example>
 - SQL Alchemy: <https://github.com/Sibyx/fiit-orm-sqlalchemy-example>

Súbežné spracovanie

Prístup k dátam v databáze

- Z hľadiska pristupovania k dátam uložených v DB môžeme uvažovať o dvoch spôsoboch:
 - K dátam pristupuje iba jeden používateľ v danom čase – hovoríme o **single-user** systéme (single-user DBMS)
 - K dátam pristupujú viacerí používatelia v danom čase – hovoríme o **multi-user** systéme (multi-user DBMS)
 - Tu nastáva problém so súbežnosťou prístupu k pamäti – keďže používatelia pristupujú k zdieľanej pamäti

Spracovanie viacerých procesov

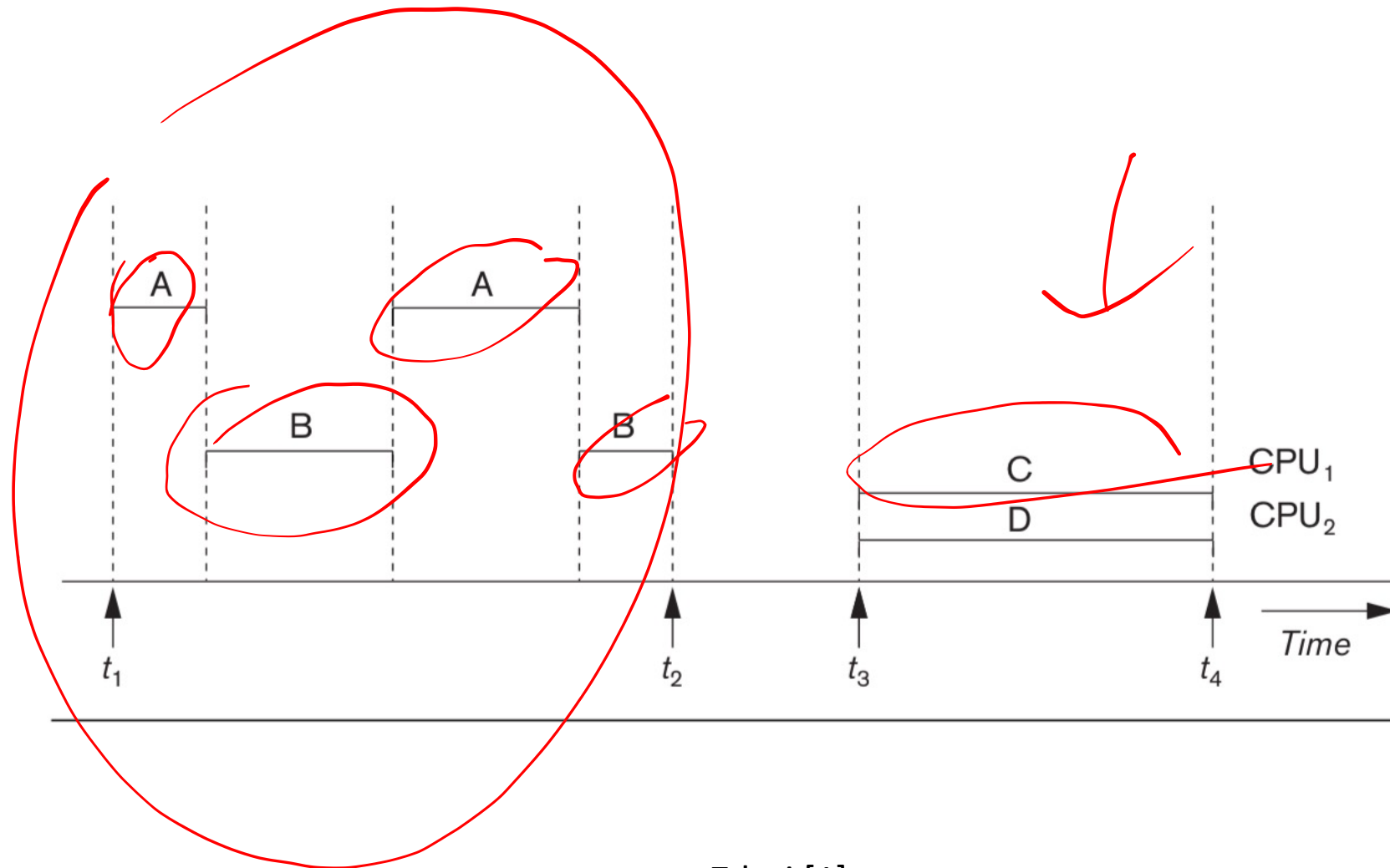


Figure 21.1

Interleaved processing versus parallel processing of concurrent transactions.

Súbežné operácie (II)

- Mary a John majú spoločný účet v banke (ACC), na ktorom je teraz **1000EUR**. Mary je práve v banke a vkladá na účet 200 EUR. John v tej istej chvíli vyberá z ACC 10EUR.

```
vklad(acc,200) {  
    int mbalance  
    mbalance = read(ACC); //1000  
    mbalance = mbalance + 200; //1200  
    write(ACC,mbalance);  
}
```

```
vyber(acc,10) {  
    int jbalance  
    mbalance = read(ACC); //1000  
    jbalance = jbalance - 10; //990  
    if (jbalance < 0)  
        error();  
    write(ACC,jbalance); //990  
}
```


čas

Na účte ACC je 990 EUR, ale **má byť 1190 EUR**. Kto môže za danú chybu?

Súbežné operácie (2)

- Problém na **strane banky (systému)**
 - Zlé časovanie jednotlivých operácií, s ktorým systém neuvažoval
- Z pohľadu systému sa stalo:
 - mary.read(acc), johny.read(acc), mary.write(acc), john.write(acc)
- Existujú ale iné poradia operácií, ktoré sú však v poriadku:
 - ~~mary.read(acc), mary.write(acc), johny.read(acc), john.write(acc)~~
 - ~~johny.read(acc), john.write(acc), mary.read(acc), mary.write(acc),~~
 - ~~mary.read(acc), johny.read(acc), mary.write(acc),~~ a systém johnovi povie, že výber zlyhal john.write(acc)

Problémy so súbežným spracovaním

- Problémy, ktoré môžu nastať v rámci súbežného spracovania:
 - **Lost Update problem**
 - Zmenenie hodnôt v rovnakom čase
 - **Dirty read problem**
 - Čítanie hodnoty, ktorá nebola ešte commitnuta a teda môže stále zlyhať
 - **Incorrect summary problem**
 - Výpočet agregáčnej funkcie a medzi tým iná transakcia zmenila niektorú z hodnôt takže je vypočítaná nesprávna hodnota
- 

Transakcia

- Je vykonanie postupnosti nasledujúcich operácií napr. SQL dopytov (queries) za účelom poskytnutia high-level funkcionality (napr. prevod prostriedkov medzi účtami)
- V DBMS **nie sú povolené** čiastočné transakcie, ktoré by vykonali iba časť daných operácií a ostatné by nemuseli byť vykonané
- Transakcia začína START, ktorý v rámci postupnosti sa nachádza práve raz a to na začiatku transakcie
 - syntax v rámci SQL závisí aj od použitého DBMS
- COMMIT alebo ABORT sa nachádza na konci transakcie a nachádza sa v transakcií práve raz

Transakcia - príklad

- Príklad – bankový prevod sumy S z účtu A na účet B

transfer(S,A,B)

{

float SA,SB;

start(transfer);

SA=READ(A);

SB=READ(B);

SA=SA-S;

SB=SB+S;

WRITE(A,SA);

WRITE(B,SB);

COMMIT(transfer);

}

Zjednodušený zápis:

s1, r1(A), r1(b), w1(A), w1(B), c1

- Zároveň môže byť vykonávaných viacero inštancií transakcie *transfer*. Z pohľadu systému sú rôzne, takže musia byť identifikovateľné na čo sa využíva ID transakcie.
- V prípade READ alebo WRITE nevidí systém lokálne premenné, vidí len konkrétnu hodnotu.

Transakcia - syntax

- Postgresql

```
BEGIN;  
SELECT, INSERT, UPDATE ....  
COMMIT;
```

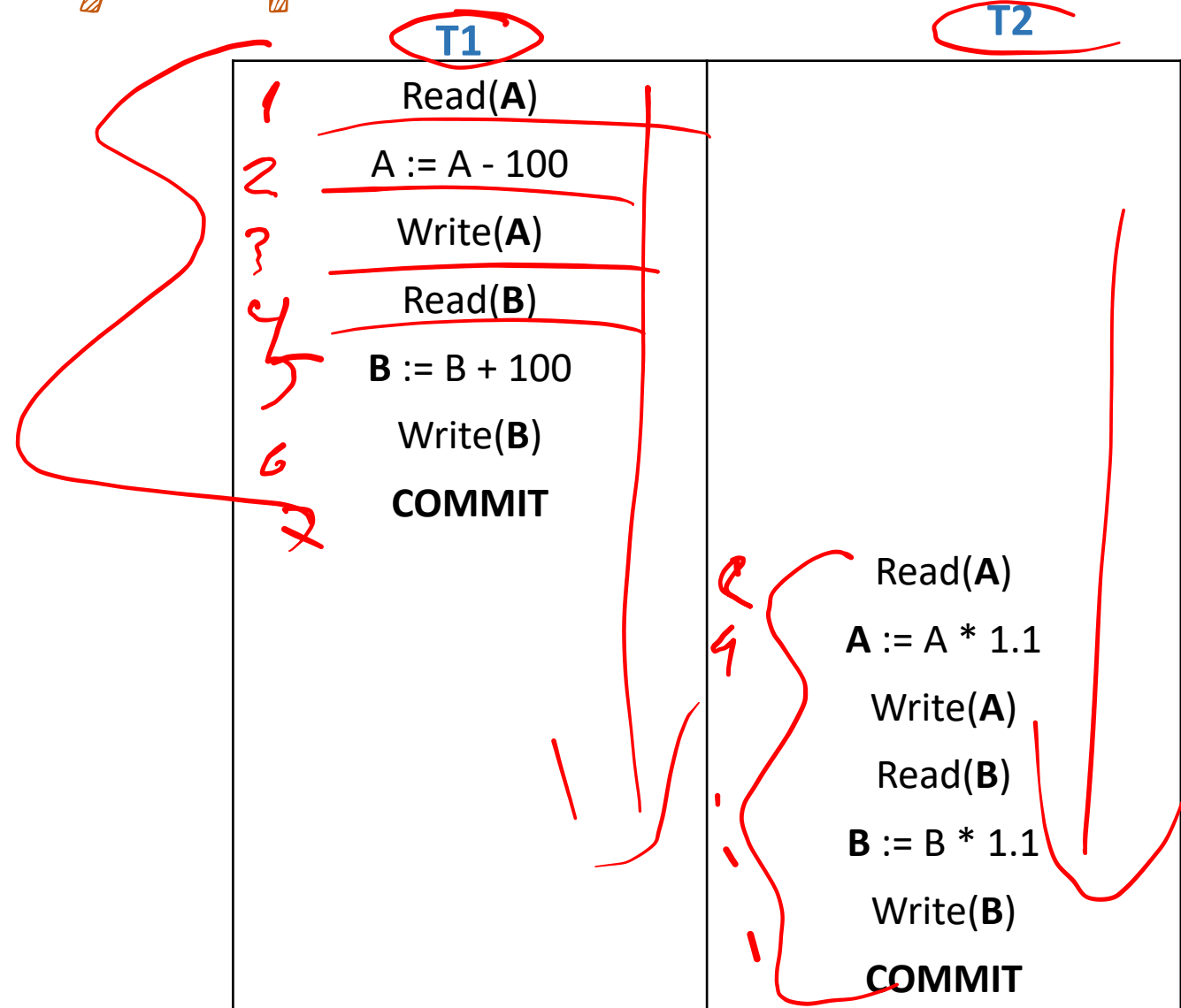
- MySQL

```
START TRANSACTION;  
SELECT, INSERT, UPDATE ....  
COMMIT;
```

Závisle od DBMS

Prístup k zdieľaným prostriedkom

- Dve transakcie T1, T2 pristúpajú k rovnakým prostriedkom súčasne
 - Prevod prostriedkov
 - A pripísanie úrokov
- Uvažujeme o
 - $A = 1000$
 - $B = 1000$
- Finálny výsledok
 - $A = 990$
 - $B = 1210$
- Pri výmene poradia vykonávania
 - $A = 1000$
 - $B = 1200$
- Spolu v oboch prípadoch **2200**



Sériové vykonávanie operácií

- Vykonávanie transakcií po jednej ako prišli do DBMS
 - Iba jedna transakcia môže byť vykonávaná v danom čase
- V porovnaní so súbežným vykonávaním transakcií
 - Horšia priepustnosť
 - Nutnosť čakania na I/O operácie
- Súbežné vykonávanie
 - Zlepšenie prepustnosti systému a tiež odovzdy systému pre používateľov

Nesériové poradie

- $A = 990$
- $B = 1200$
- Spolu = 2190

T1

Read(A)
 $A := A - 100$
Write(A)

~~Read(B)~~
 $B := B + 100$
~~Write(B)~~
COMMIT

T2

Read(A)
 $A := A * 1.1$
~~Write(A)~~
Read(B)
 $B := B * 1.1$
Write(B)
COMMIT

Nesériové poradie

- $A = 990$
- $B = 1200$
- Spolu = 2190

Niečo nám chyba!
Kde je 10 Eur?

T1

Read(A)
 $A := A - 100$
Write(A)

Read(B)
 $B := B + 100$
Write(B)
COMMIT

T2

Read(A)
 $A := A * 1.1$
Write(A)
Read(B)
 $B := B * 1.1$
Write(B)
COMMIT

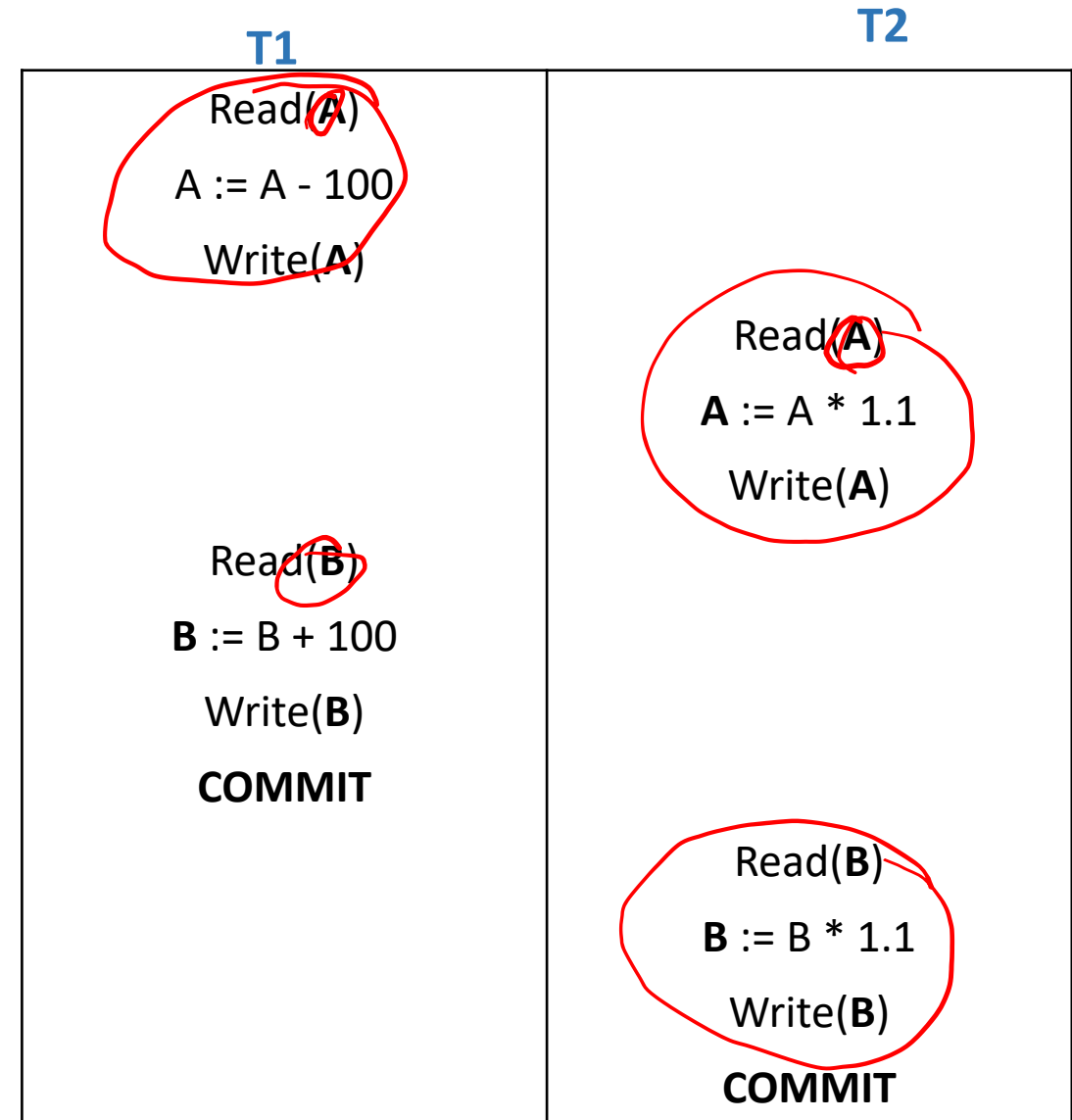
DBMS

- Ako teda dosiahnuť správnosť a férovosť v rámci systému?
 - Správnym poradím vykonávania operácií jednotlivých transakcií
 - Označujeme tiež rozvrh transakcií (Schedule)

Správne poradie

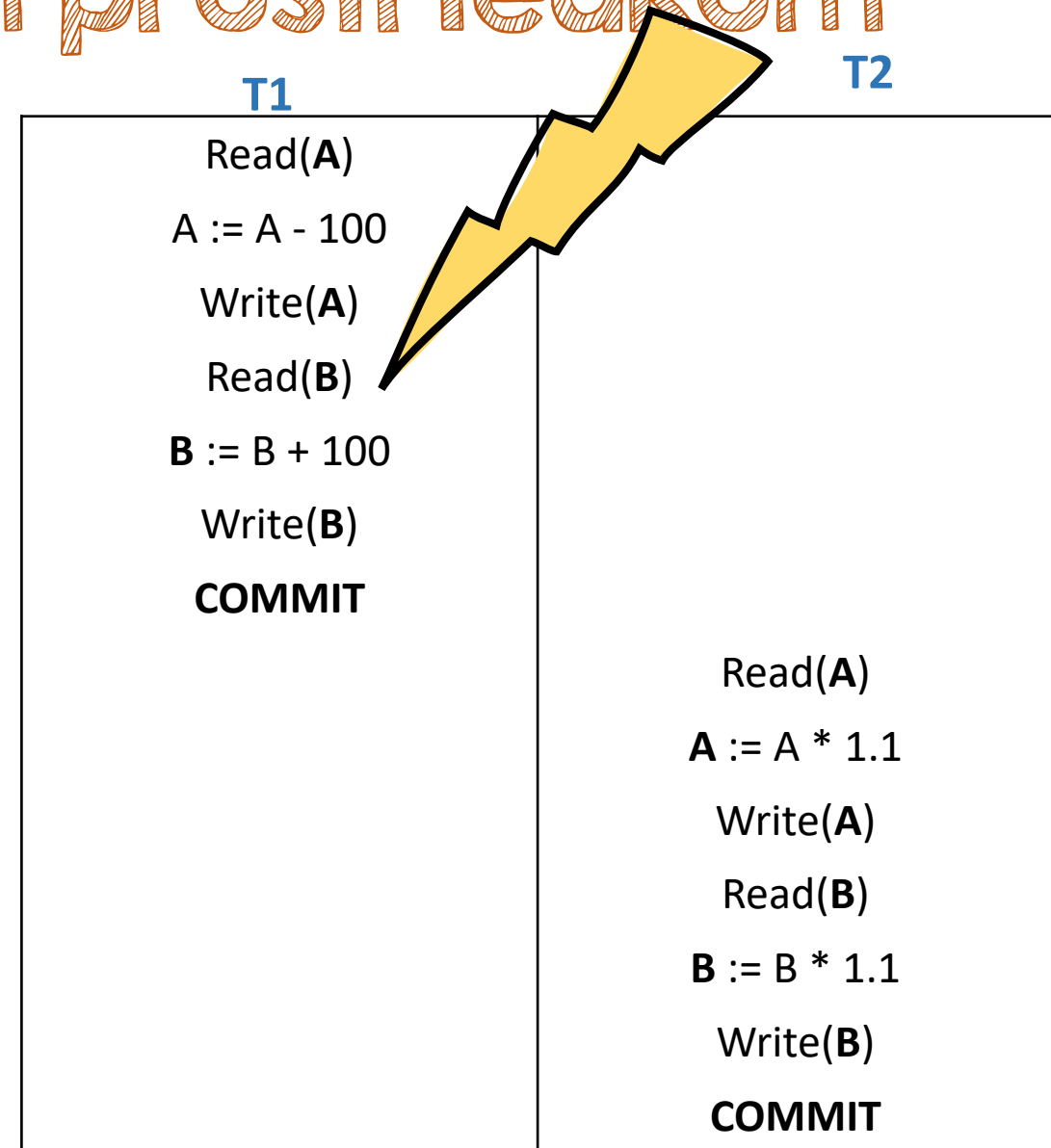
- Tentokrát správny výsledok
- Ako sa dopracovať k takémuto rozvrhu ?

• **Concurrency control
protokoly**



Prístup k zdieľaným prostriedkom

- Dve transakcie T1, T2 prístupujú k rovnakým prostriedkom súčasne
 - Prevod prostriedkov
 - A pripísanie úrokov



Prístup k zdieľaným prostriedkom

- Dve transakcie T1, T2 prístupujú k rovnakým prostriedkom súčasne
 - Prevod prostriedkov
 - A pripísanie úrokov

T1

Read(A)
A := A - 100
Write(A)
Read(B)
B := B + 100
Write(B)
COMMIT

T2

Read(A)
A := A * 1.1
Write(A)
Read(B)
B := B * 1.1
Write(B)
COMMIT

Čo ak dôjde k výpadku počas pripočítavania?

Prístup k zdieľaným prostriedkom

- Dve transakcie T1, T2 pristúpajú k rovnakým prostriedkom súčasne
 - Prevod prostriedkov
 - A pripísanie úrokov
- Ako dať databázu do korektného stavu ?

- **Logovanie + obnova dát**

Čo ak dôjde k výpadku počas pripočítavania?

T1

Read(A)
A := A - 100
Write(A)
Read(B)
B := B + 100
Write(B)
COMMIT

T2

Read(A)
A := A * 1.1
Write(A)
Read(B)
B := B * 1.1
Write(B)
COMMIT

Požiadavky na transakčný databázový systém

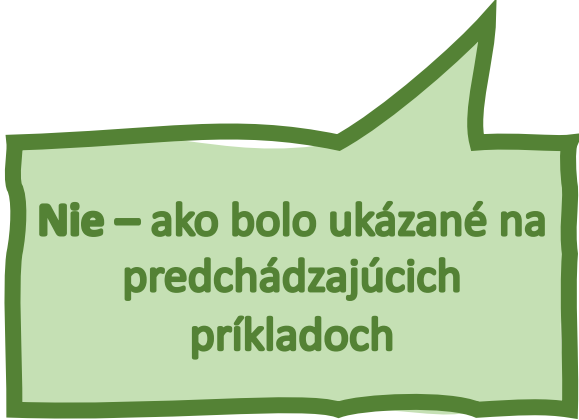
- Požiadavky pod skratkou ACID:
 - **Atomicity** – transakcia je buď vykonaná celá alebo nie je vykonaná vôbec
 - **nemôže** byť vykonaná iba časť operácií
 - **Consistency** - vykonanie transakcie znamená prechod od konzistentného stavu DB opäť do konzistentného stavu DB – túto požiadavku zabezpečuje implementátor transakcie/aplikácie alebo DBMS modul, ktorý je zodpovedný za dodržiavanie integritných obmedzení.
 - **Isolation** – napriek tomu, že systém môže vykonávať viacero transakcií "paralelne", výsledný efekt musí byť rovnaký ako keby boli dané transakcie vykonávané sériovo (jedna po druhej)
 - **Durability** – ak transakcia úspešne skončí (systém vykoná **COMMIT**), tak všetky zmeny, ktoré boli vykonané v DB budú navždy zachované
- Tieto požiadavky musia byť garantované aj v prípade havárií v ľubovoľnom momente (na strane klienta, servera)

Schedule - rozvrh

- Niekedy označovaný aj ako história (z angl. History)
 - označujeme ho ako S
- Definícia rozvrhu (história) z angl. Schedule (history):
 - Je postupnosť, ktorá vznikne premiešaním operácií niekoľkých transakcií – vo všeobecnosti nekompletných. Toto premiešanie je ľubovoľné - zachováva poradie operácií jednotlivých transakcií (projekcia rozvrhu na individuálnu transakciu je postupnosť operácií tej transakcie)
 - Operácie samotnej transakcie nemôžu byť premiešané
- príklad dvoch rozvrhov pre tie iste transakcie:
 - T_1 T_2
 - $S_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$
 - $S_b: r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); a_1;$
- Rozvrh je kompletný – ak v ňom všetky transakcie končia buď operáciou COMMIT a ABORT
- Transakcia je v danom čase aktívna – ak v danom čase už začala (bol vykonaný START transakcie) a zároveň v danom čase ešte neskončila (nebol vykonaný COMMIT alebo ABORT)

Shedule - rozvrh

- Môžeme jednotlivé rozvrhy vytvoriť ako nám vyhovuje ?



**Nie – ako bolo ukázané na
predchádzajúcich
príkladoch**

Sériový vs seriovateľný rozvrh

- **Sériový rozvrh**

- je kompletný – každá transakcia v ňom obsahuje všetky svoje operácie a končí buď COMMIT-om alebo ABORT-om.
- Musí tiež platiť, že pre dve transakcie T1 a T2 sa buď vykonajú všetky operácie T1 pred T2 alebo naopak.

- **Seriovateľný rozvrh**

- má na transakcie a stav databázy rovnaký efekt ako niektorý zo sériových rozvrhov, nezávisle od stavu DB a štruktúry transakcií

Konfliktné operácie a konflikt-ekvivalencia rozvrhov

- **Konfliktné operácie** – dve operácie v rozvrhu **S** su konfliktné ak patria rôznym transakciám a ich operandom je rovnaký objekt. Zároveň musí platiť, že jedna z týchto operácií je **WRITE**
 - napr. $W1(X)$, $R2(X)$ – sú konfliktné
 - napr. $R1(X)$, $R2(X)$ – **nie sú** konfliktné
- Myšlienka za generovaním dobrých rozvrhov je
 - postupnosť nekonfliktných operácií je možné ľubovoľne premiešať a rozvrh ostane **dobrý** (je dosiahnutý správny výsledok).
 - postupnosť konfliktných operácií je potrebné zachovať
 - projekciu na jednotlivé transakcie je potrebné tiež zachovať – nie je možné premiešať postupnosť operácií v samotných transakciách.
- Definícia: Dva Rozvrhy (sú konflikt-ekvivalentné) ak
 - Pozostávajú z rovnakých operácií a
 - Relatívne poradie každých dvoch konfliktných operácií je rovnaké v oboch prípadoch

Konflikt-seriovateľný rozvrh

- Konflikt-seriovateľný rozvrh – jeho projekcia na commitované transakcie je konflikt-ekvivalentná niektorému sériovému rozvrhu týchto commitovaných transakcií
 - commitované transakcie z toho dôvodu, že commitované transakcie nám dávajú nejaké garancie v rámci systému.

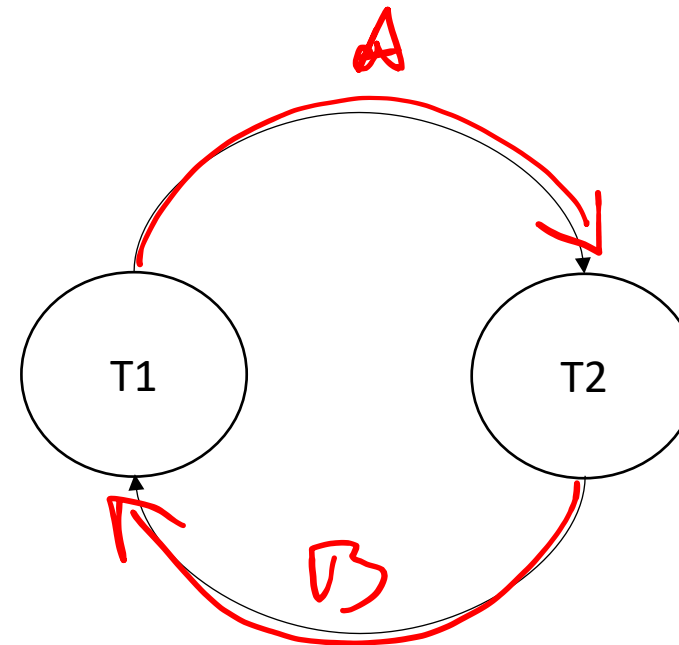
Testovanie sériovateľnosti rozvrhov - precedenčný graf

- Nech S je rozvrh, ktorý obsahuje commitované transakcie T_1, T_2, \dots, T_n .
Precedenčný graf (precedence graph alebo aj serialization graph)
 - je orientovaný graf s vrcholmi T_1, T_2, \dots, T_n ,
 - jednotlivé vrcholy reprezentujú jednotlivé transakcie
 - každá hrana e_i v grafe je $T_j \rightarrow T_k$, kde $1 \leq j \leq n$, $1 \leq k \leq n$, kde T_j je začiatkový uzol hrany e_i a T_k je koncový bod
 - hraná z $T_j \rightarrow T_k$ je vytvorená vtedy, ak pre niektorú operáciu v T_j sa objaví skôr ako **niektorá konfliktná operácia** v T_k

Príklad na precedenčný graf

Máme rozvrh S : r1(A), w1(A), r2(A), w2(A), r2(B), w2(B), c2, r1(B), w1(B), c1

T1	T2
READ(A); A = A + 100	
WRITE(A);	
	READ(A); A = A * 1.1;
	WRITE(A);
	READ(B); B = B * 1.1;
	WRITE(B);
	COMMIT;
READ(B); B = B + 100;	
WRITE(B);	
COMMIT;	



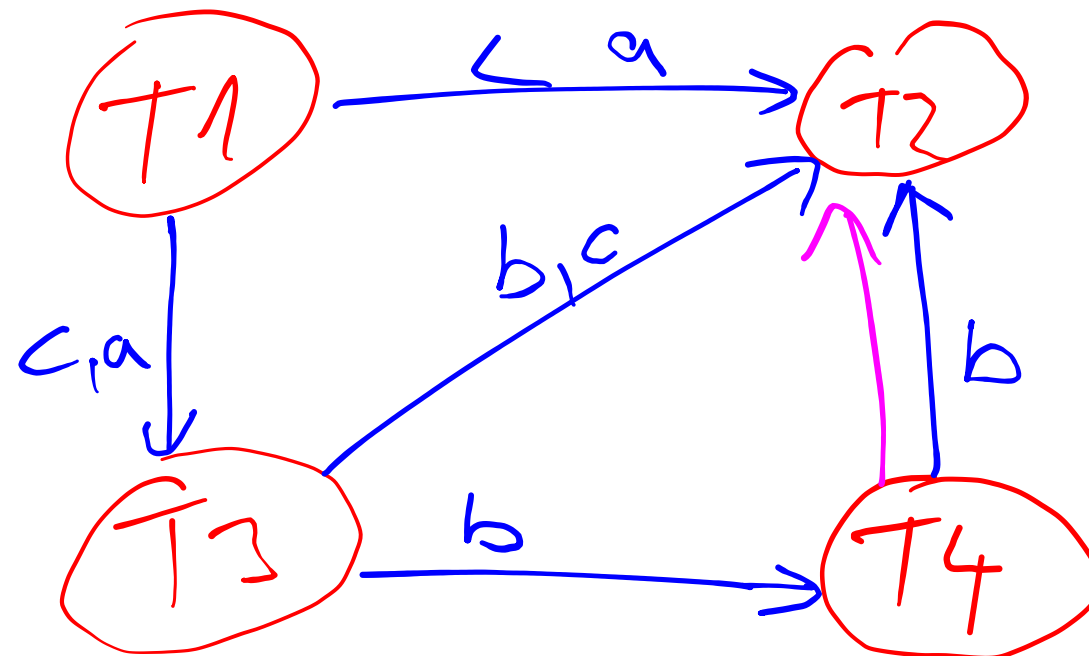
Precedenčný graf

- Rozvrh je konflikt-seriovateľný práve vtedy, keď graf je acyklický
- V prípade, že rozvrh je konflikt-seriovateľný, tak usporiadanie v rámci precedenčného grafu vieme určiť akému sériovému rozvrhu je daný rozvrh ekvivalentný

Príklad precedenčný graf(II)

- Transakcie T1,T2,T3,T4 predstavujú commitované transakcie

T1	T2	T3	T4
		read(b)	
		write(b)	
			write(b)
	read(b)		
read(a)			
read(c)			
write(a)			
write(c)			
		read(a)	
		write(c)	
	read(a)		
	write(c)		



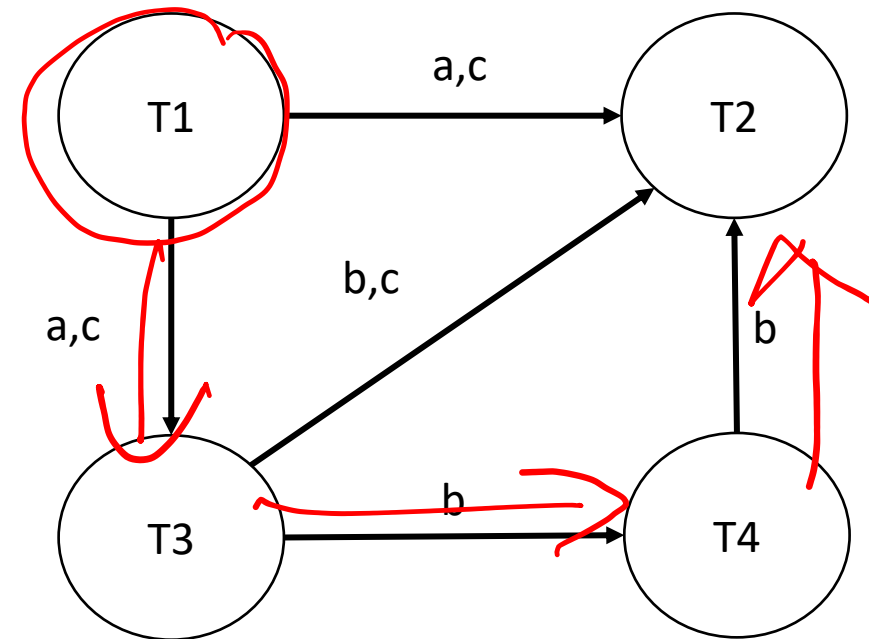
r	r
w	w
w	r
w	w

Konflikty

Príklad precedenčný graf(II)

- Transakcie T1,T2,T3,T4 predstavujú commitované transakcie

T1	T2	T3	T4
		read(b)	
		write(b)	
			write(b)
	read(b)		
read(a)			
read(c)			
write(a)			
write(c)			
		read(a)	
		write(c)	
	read(a)		
	write(c)		



Precedenčný graf **neobsahuje cyklus** a teda je rozvrh konflikt-seriovateľný a zodpovedá sériovému rozvrhu **T1 -> T3 -> T4 -> T2**

Príklad precedenčný graf(2)

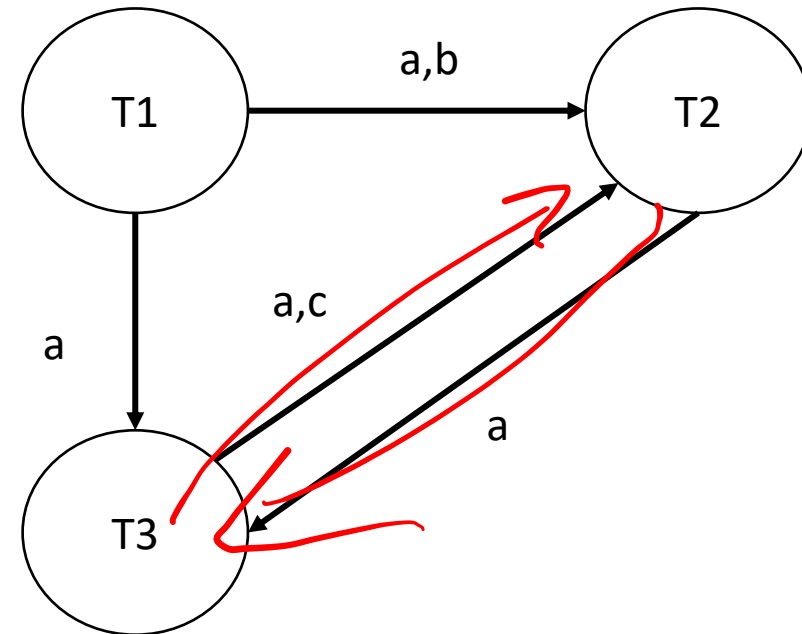
- Máme T1,T2,T3

T1	T2	T3
read(a)		
read(b)		
write(a)		
		read(a)
	read(b)	
		write(c)
	read(c)	
	write(b)	
	read(a)	
		write(a)
	write(c)	
	write(a)	

Príklad precedenčný graf(2)

- Máme T1,T2,T3

T1	T2	T3
read(a)		
read(b)		
write(a)		
		read(a)
	read(b)	
		write(c)
	read(c)	
	write(b)	
	read(a)	
		write(a)
	write(c)	
	write(a)	



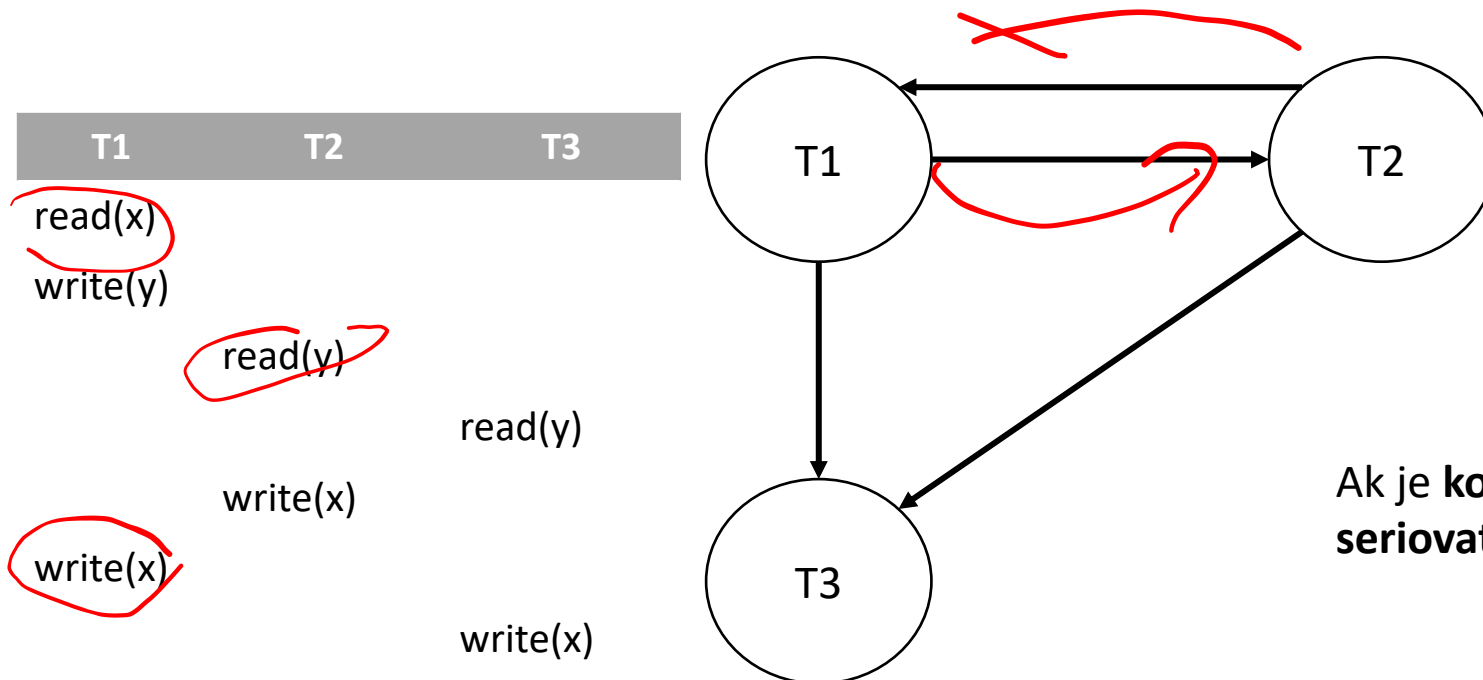
Precedenčný graf **obsahuje cyklus T3 -> T2 -> T3** a teda rozvrh **nie je** konflikt-seriovateľný.

View-sériovateľnosť (II)

- Vyjadrenie ekvivalentnosti dvoch rozvrhov je možné pristupovať aj iným spôsobom ako je konflikt-ekvivalentnosť.
- Ďalší spôsob je View-ekvivalentné
 - Hovoríme, že v rozvrhu transakcia T_2 číta X od transakcie T_1 , ak v tom rozvrhu existujú operácie $w_1(X)$ a $r_2(X)$, pričom $w_1(X)$ je posledný zápis do X pred $r_2(X)$
- Dva rozvrhy S a S' sú view-ekvivalentné, ak
 - sú definované nad tými istými transakciami
 - Zároveň pre každú dvojicu operácií v S , kde nejaká transakcia T_2 číta X od T_1 existuje rovnaká dvojica operácií v S' , kde T_2 tiež číta X od T_1 , a zároveň
 - Pre každý dátový objekt X , ak transakcia T_i je posledná transakcia, ktorá píše do X v S , tak aj v S' je T_i posledná transakcia, ktorá píše do X (final write)
 - A zároveň to platí aj s prehodenými rozvrhmi S' a S
- Je možné povedať, že rozvrhy sú view-ekvivalentné ak majú rovnaký efekt na transakcie (čítanie) a na databázu (zápisy)

View seriovateľnosť (2)

- Rozvrh je view-seriovateľný ak je view ekvivalentný niektorému sériovému rozvrhu.
- Príklad: $r1(x), w1(y), r2(y), r3(y), w2(x), w1(x), w3(x), c1, c2, c3$
- Sériovy rozvrh $T1 \rightarrow T2 \rightarrow T3$
 - $r1(x), w1(y), w1(x), c1, r2(y), w2(x), c2, r3(y), w3(x), c3$



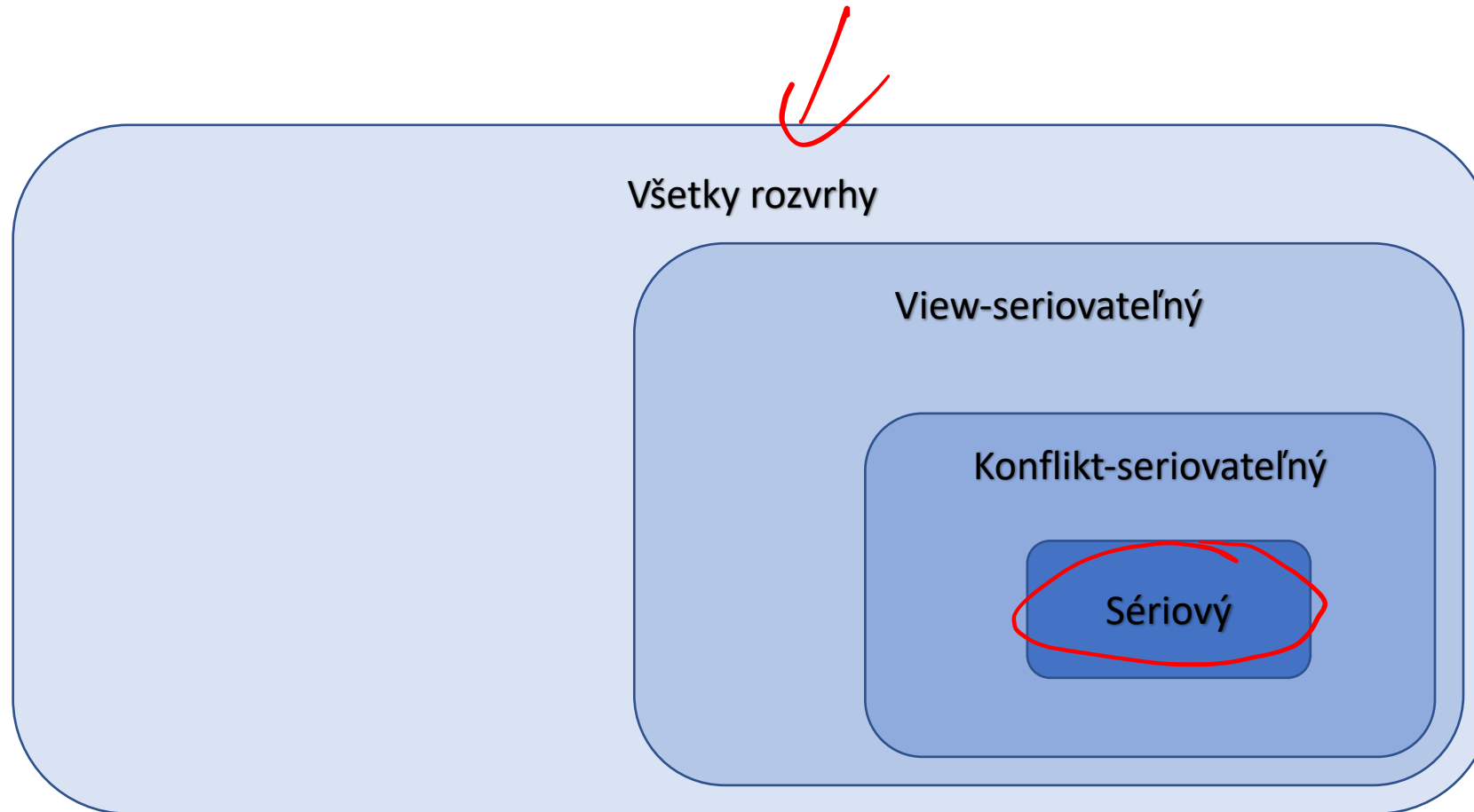
- Tento rozvrh nie je konflikt-seriovateľný, čo môžeme vidieť z grafu.
- Je View-seriovateľný
 - je view seriovateľný rozvrhu $T1 \rightarrow T2 \rightarrow T3$

Ak je **konflikt-seriovateľný**, tak je aj **view seriovateľný** – naopak to však nie vždy platí

View-seriovateľnosť (3)

- Testovanie rozvrhu, či je view-seriovateľný je NP-ťažký problém
 - testovanie view-ekvivalentnosti dvoch rozvrhov je možné urobiť v polynomiálnom čase.

Rozvrhy



ACID - izolácia

- Vieme testovať, či je rozvrh konflikt seriovateľný
 - je možné zostrojiť algoritmus, ktorý bude efektívne v reálnom čase generovať konflikt-seriovateľné rozvrhy ?
- Vieme generovať časť seriovateľných rozvrhov – nie všetky
 - jednoduché riešenie – generovanie iba sériových rozvrhov
 - veľká redukcia výkonnosti systému
 - generovanie View-seriovateľné rozvrhy – náročné
 - generovanie konflikt-seriovateľných – je možné viacerými spôsobmi – nejakých podtried v závislosti od spôsobu

Izolácia - Isolation

- Rozlišujeme dva typy protokolov pre riadenie súbežnosti
 - **Optimistické** – predpokladajú, že konflikty medzi transakciami dochádzajú zriedka a je preto lepšie ich riešiť až v prípade, že nastanú
 - **Pesimistické** – predpokladajú, že konflikty vznikajú často, a preto sa snaží zamedziť vzniku konfliktov.

Protokoly

- **Lock-Based Protocols** - označované aj ako Algoritmus zámkov
 - transakcie zamykajú dátové objekty pred čítaním a zápisom aby iné transakcie s nimi nemohli ľubovoľne zaobchádzať
- **Timestamp-Based Protocols** – označovaný ako algoritmus časových pečiatok
 - na základe časových pečiatok sa systém rozhoduje pri operáciách read/write či transakcia bude pokračovať alebo nie
- **Optimistic concurrency-control** – označovaný ako validačný algoritmus
 - optimistický prístup, ktorý vykonáva operácie tak ako idú. Prísne kontroluje, či dovolí COMMIT
- **Multiversion Schemes** – označovaný Multiversion algoritmus
 - každá transakcia si zapisuje svoj zmeny do svojej lokálnej kópie databázy (nezapisuje sa do hlavnej/ostrej databázy). Systém sa až pri žiadosti o COMMIT rozhodne, či sa dané zmeny zapíšu do DB alebo nie.
 - Systém musí udržiavať viacero kópií dátového objektu a tiež musí rozhodovať, ktorá verzia sa bude čítať pri operácií READ

Izolácia: Zamykanie(Locking)

- ako prvé môžeme uvažovať o binárnom zamknutí dátového objektu LOCK a UNLOCK
 - takéto striktné obmedzenia nie sú vhodné pre databázové objekty
- Z tohto dôvodu boli definované 2 typy zámkov
 - Read-Lock(RL) – shared lock pre čítanie
 - Write-Lock(WL) – exclusive lock – dovoľuje čítanie aj zapisovanie
- Tieto operácie sú v prípade obsiahnutia zamkynania pridané do rozvrhu. Transakcia musí najprv požiadať o priradenie daného zámku a až následne môže vykonať danú činnosť nad dátovým objektom.
- Pri Committe nie je potreba žiadna dodatočná kontrola
- Granularita zámkov:
 - Na jeden atribút jedného záznamu
 - Na jeden záznam
 - Na celú tabuľku
 - (na diskový blok)

Zamykanie (locking)

- Transakcia môže získať RL zámok na objekt X v prípade, že na objekt X nie je žiaden zámok alebo je zámok typu RL. Vtedy transakcia môže získať zámok typu RL
- V prípade, že transakcia T žiada o zámok typu WL nad objektom X, tak je to možné iba v prípade, že nad objektom nie je žiaden zámok.
- V prípade, že nemôže v daný okamžik získať požadovaný typ zámku tak je transakcia priradená na čakaciu listinu.
- Čo ak transakcia T vlastní ako jedina zámok typu RL nad objektom X a žiada o zámok WL nad objektom X ?

	RL	WL
RL	True	False
WL	False	False

Two-phase locking - dvojfázove zamykanie

- generuje len konflikt-sériovateľné rozvrhy
- nie každý konflikt-sériovateľný rozvrh je generovaný two-phase locking
- Ma dve fázy:
 - expandig – v tejto fáze transakcia získava zámky
 - shrinking – v tejto fáze transakcia už len odomýká jednotlivé dátové objekty
- Ak už transakcia urobí jeden unlock nie je možné urobiť akýkoľvek ďalší lock



Two-phase locking - verzie

- **basic** – základná verzia ako bolo popísane vyššie
- **conservative** – vyžaduje zamknutie všetkým objektov pred tým ako sa začne vykonávať transakcia
 - musí čakať kým budú všetky zámky voľné –nezískava ich postupne
 - je deadlock-free
- **strict** – uvoľní exkluzívne zámky WL až pokiaľ neprebehne COMMIT alebo ABORT
 - nie je deadlock-free
- **rigorous** – uvoľní RL a WL až po vykonaní COMMIT a ABORT

Algoritmus časových pečiatok

- Každá transakcia dostáva priradenú unikátnu časovú pečiatku
 - Môže byť realizovaná pomocou systémových hodín alebo logického počítadla
- Časové pečiatky určujú poradie sierovateľnosti
 - Teda aj akému sériovému rozvrhu sa bude výsledok podobat'

Algoritmus časových pečiatok - pravidla

1. Suppose that transaction T_i issues $\text{read}(Q)$.
 - a. If $\text{TS}(T_i) < \text{W-timestamp}(Q)$, then T_i needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and T_i is rolled back.
 - b. If $\text{TS}(T_i) \geq \text{W-timestamp}(Q)$, then the read operation is executed, and $\text{R-timestamp}(Q)$ is set to the maximum of $\text{R-timestamp}(Q)$ and $\text{TS}(T_i)$.
2. Suppose that transaction T_i issues $\text{write}(Q)$.
 - a. If $\text{TS}(T_i) < \text{R-timestamp}(Q)$, then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls T_i back.
 - b. If $\text{TS}(T_i) < \text{W-timestamp}(Q)$, then T_i is attempting to write an obsolete value of Q . Hence, the system rejects this write operation and rolls T_i back.
 - c. Otherwise, the system executes the write operation and sets $\text{W-timestamp}(Q)$ to $\text{TS}(T_i)$.

Algoritmus časových pečiatok

- Thomas' write rule
 - Dovoľuje vznik view-seriovateľného rozvrhu

T_{27}	T_{28}
read(Q)	write(Q)
write(Q)	

Čo je možné urobiť s touto operáciou ?

Algoritmus časových pečiatok

- Thomas' write rule
 - Dovoľuje vznik view-seriovateľného rozvrhu

T_{27}	T_{28}
read(Q)	write(Q)
write(Q)	

$T_{27} \rightarrow T_{28}$

Môžeme od ignorovať –
nijako nám nezmení
výsledok

Validačný algoritmus

- Obsahuje tri fázy
 - Fáza čítania (Read phase)
 - Vykonáva sa transakcia T – načítavanie dát do lokálnych premenných a vykonanie všetkých write operácií nad lokálnymi premennými
 - Fáza validácie (Validation phase)
 - Fáza kedy sa validuje transakcia T, či neporušila seriovateľnosť – v prípade porušenia je transakcia zrušená
 - Zapisovacia fáza (Write phase)
 - Zapisovanie lokálnych premenných do databázy. Transakcie, ktoré iba čítajú nevykonávajú túto fázu

Multiversion concurrency control

- V prípade operácie Write sa vytvorí nová verzia zapisovaného objektu
- Keď dochádza k čítaniu, tak Concurrency control manažér vyberie vhodnú verziu daného objektu
 - Musí byť zabezpečená seriovateľnosť
- **Writers do not block readers.**
- **Readers do not block writers.**
- **Read-only transakcie čítajú konzistentný snapshot**
- Podpora **time travel queries**

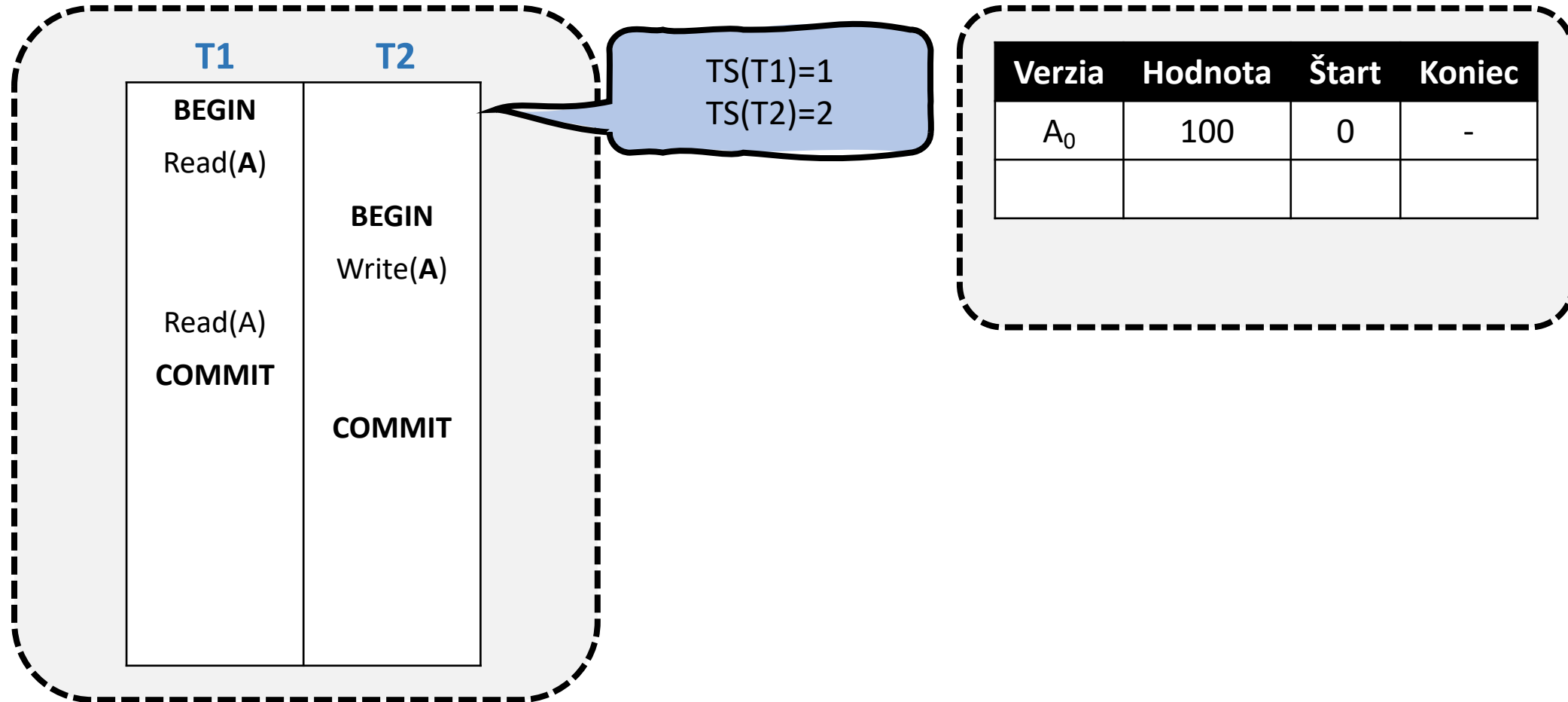
MVCC

- Súvisiace činnosti DBMS s MVCC
 - **Ukladanie verzii**
 - **Garbage Collection**
 - **Index management**

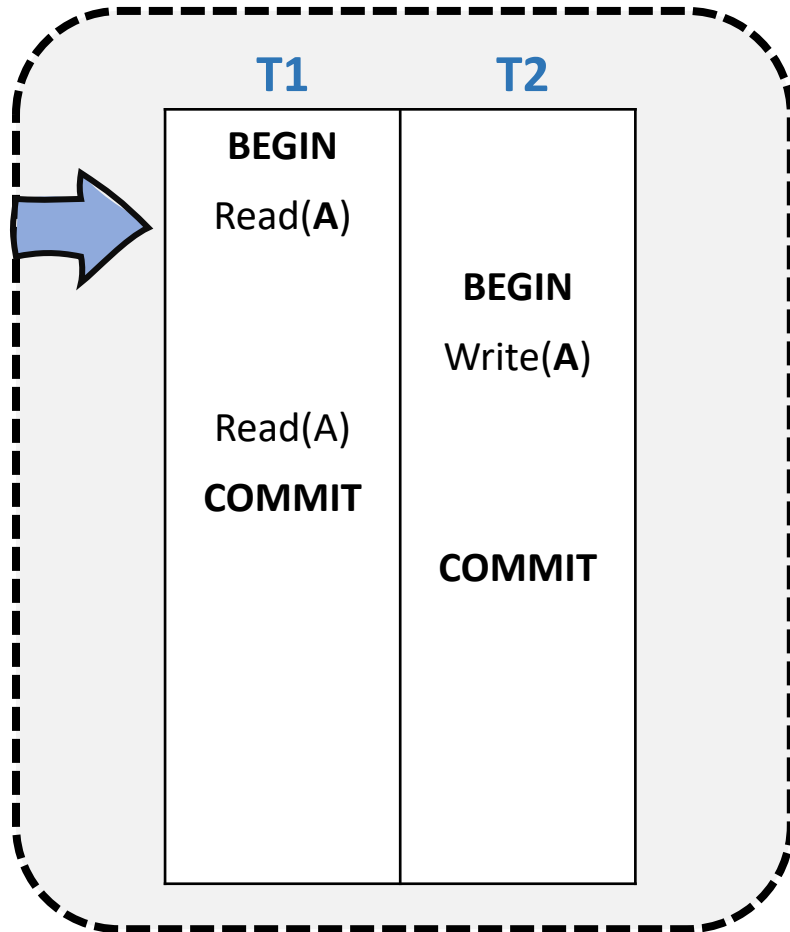
MVCC

- Existencia rôznych verzií
 - **Timestamp Ordering**
 - **Two-Phase Locking**
 - **Snapshot Isolation**

Snapshot isolation

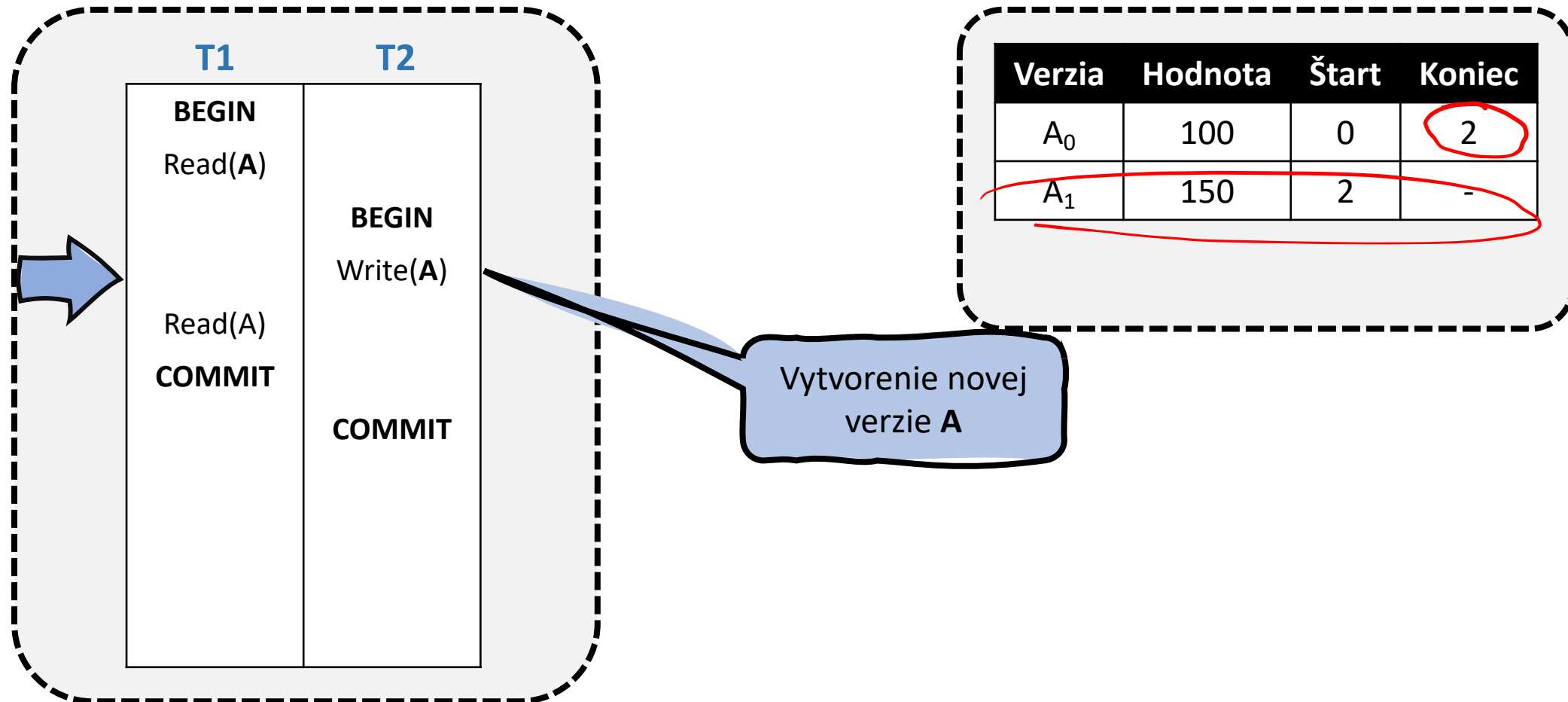


Snapshot isolation

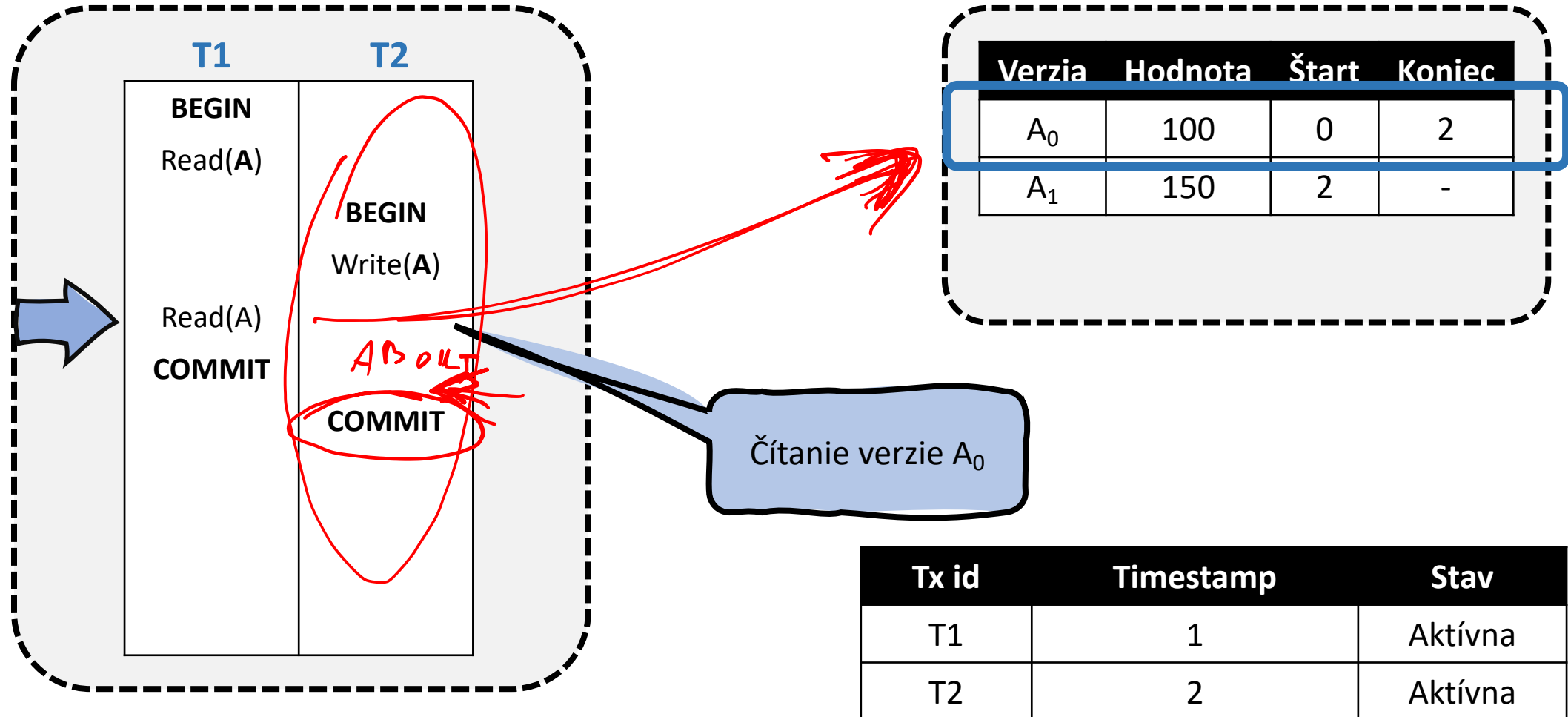


Verzia	Hodnota	Štart	Koniec
A ₀	100	0	-

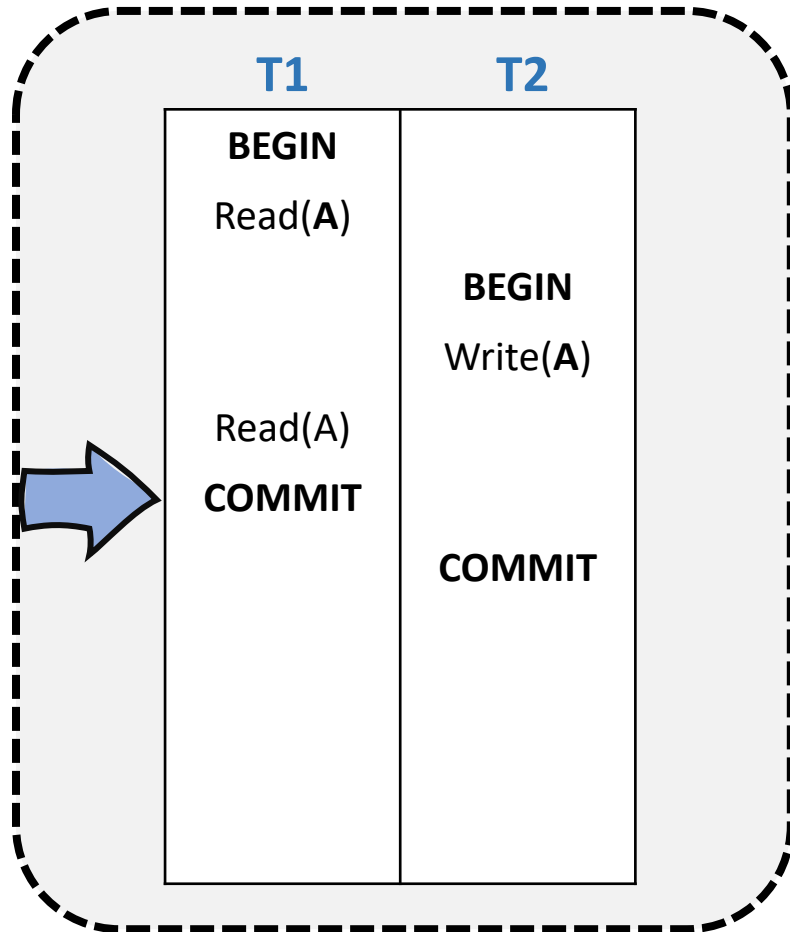
Snapshot isolation



Snapshot isolation



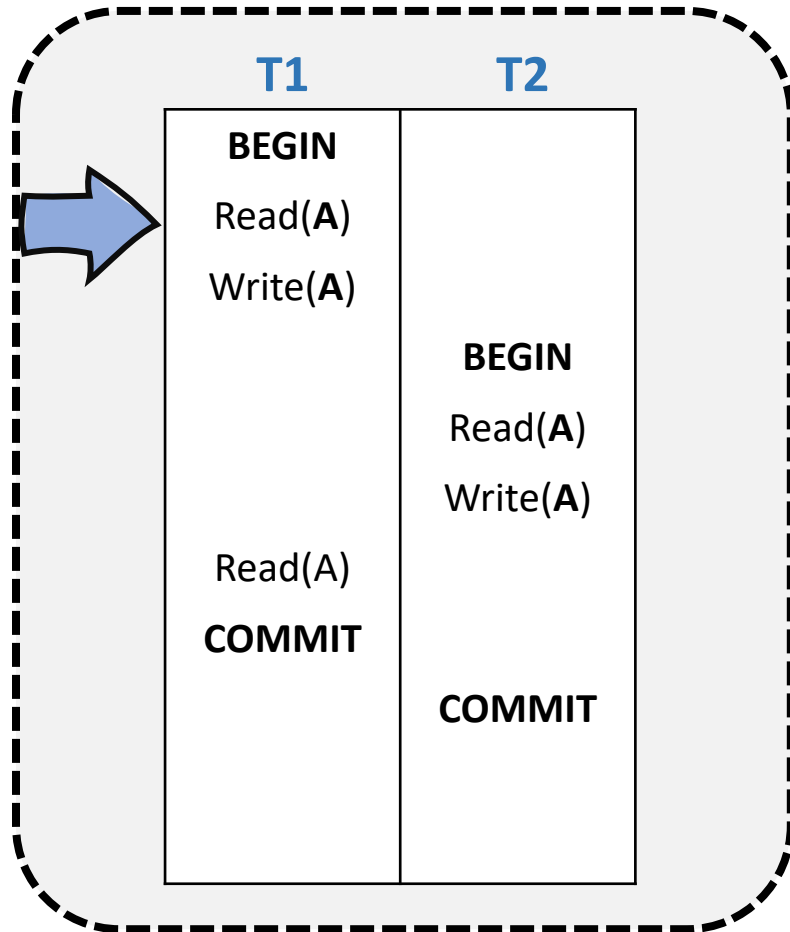
Snapshot isolation



Verzia	Hodnota	Štart	Koniec
A ₀	100	0	2
A ₁	150	2	-

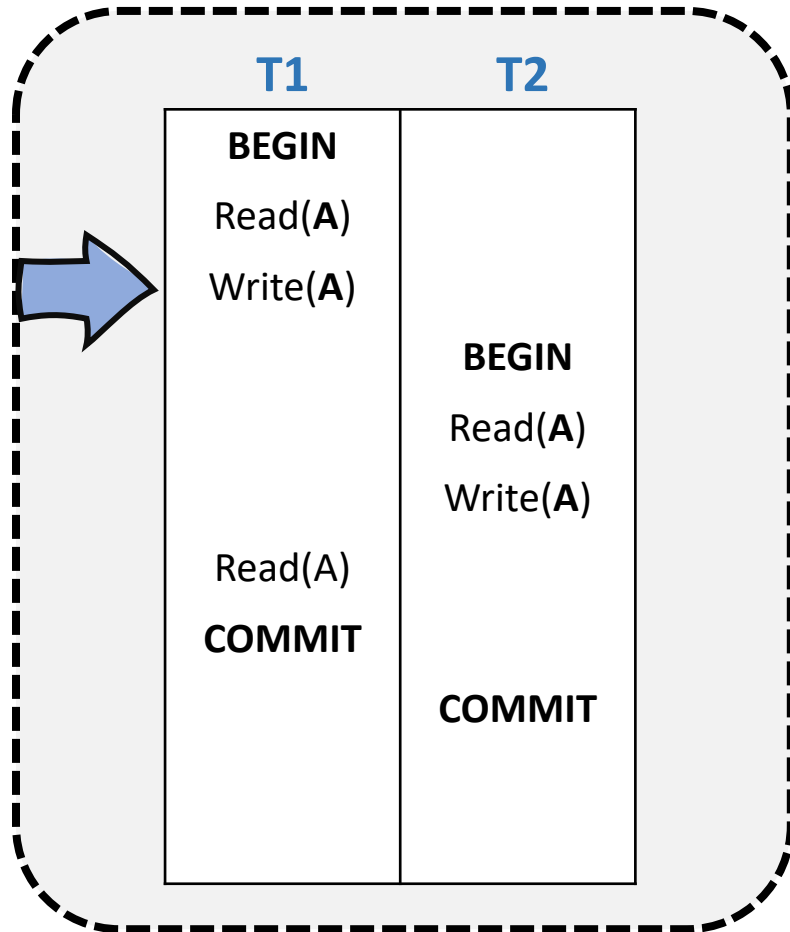
Tx id	Timestamp	Stav
T1	1	Committed
T2	2	Aktívna

Snapshot isolation - príklad 2



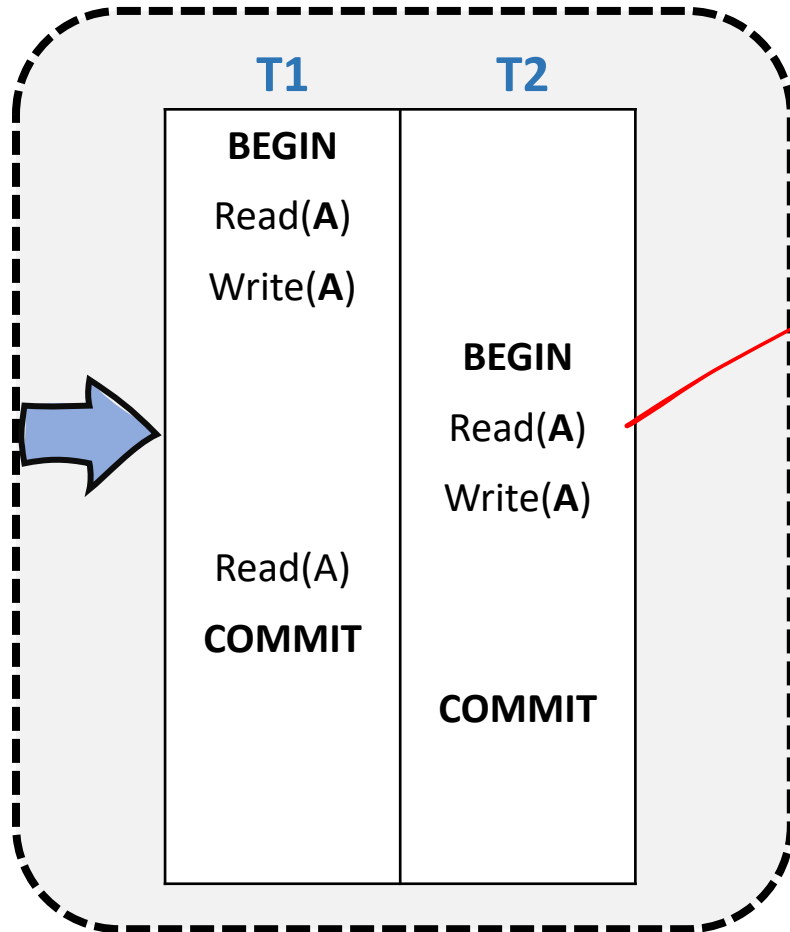
Verzia	Hodnota	Štart	Koniec
A ₀	100	0	-

Snapshot isolation - príklad 2



Verzia	Hodnota	Štart	Koniec
A ₀	100	0	1
A ₁	120	1	-

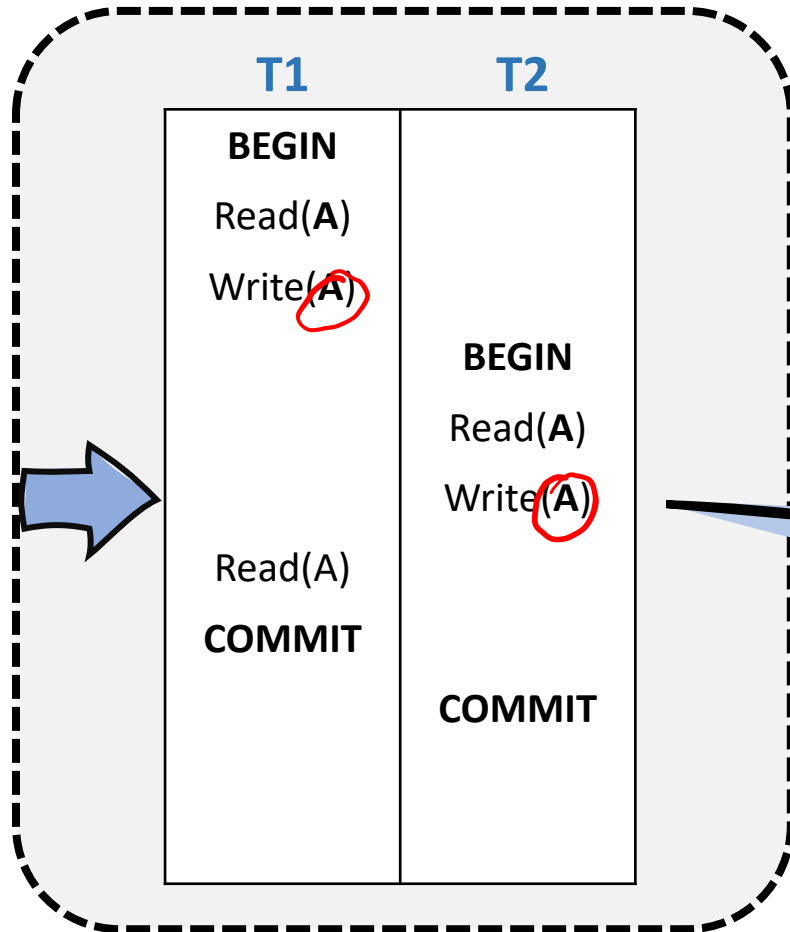
Snapshot isolation - príklad 2



Verzia	Hodnota	Štart	Koniec
A ₀	100	0	1
A ₁	120	1	-

Tx id	Timestamp	Stav
T1	1	Aktívna
T2	2	Aktívna

Snapshot isolation - príklad 2



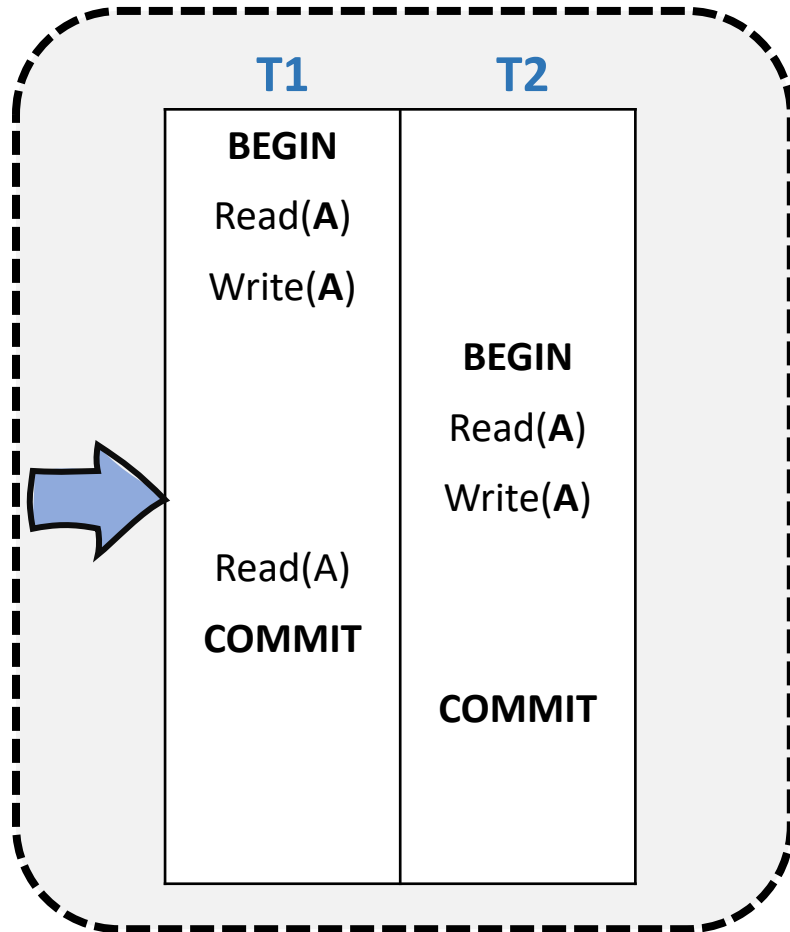
Verzia	Hodnota	Štart	Koniec
A ₀	100	0	1
A ₁	120	1	2

- Dva možné prístupy:
1. First committer wins
 2. first updater wins

Snapshot isolation - stratégie

- Dve stratégie
 - **first committer wins**
 - **first updater wins**

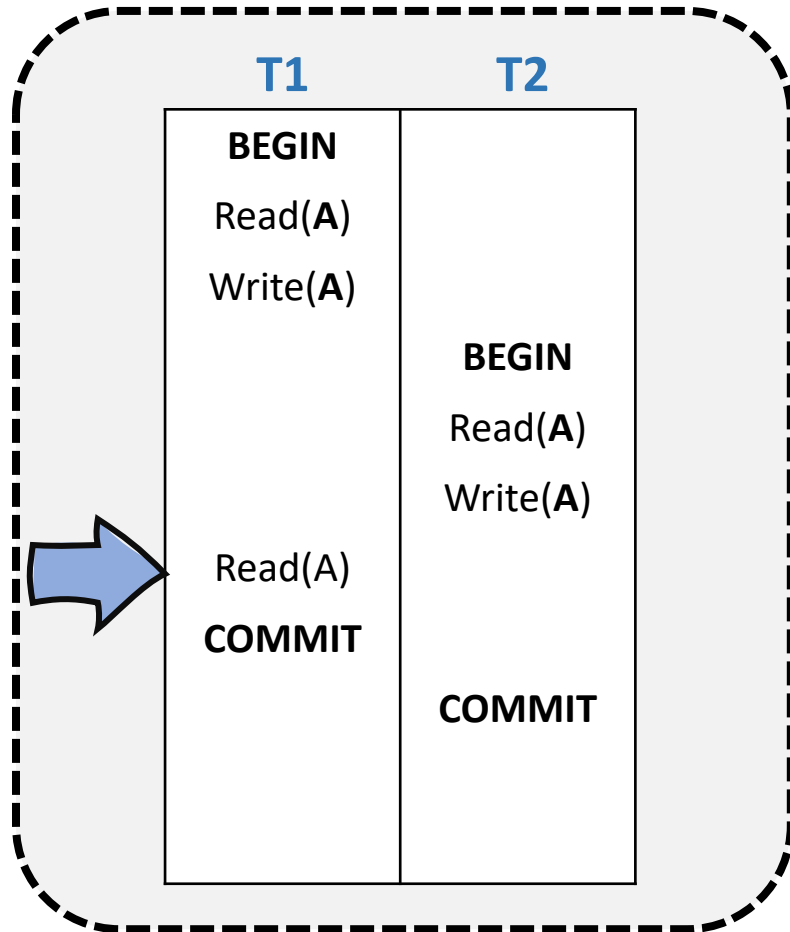
SI - first committer wins



Verzia	Hodnota	Štart	Koniec
A ₀	100	0	1
A ₁	120	1	2
A ₂	200	2	-

Tx id	Timestamp	Stav
T1	1	Aktívna
T2	2	Aktívna

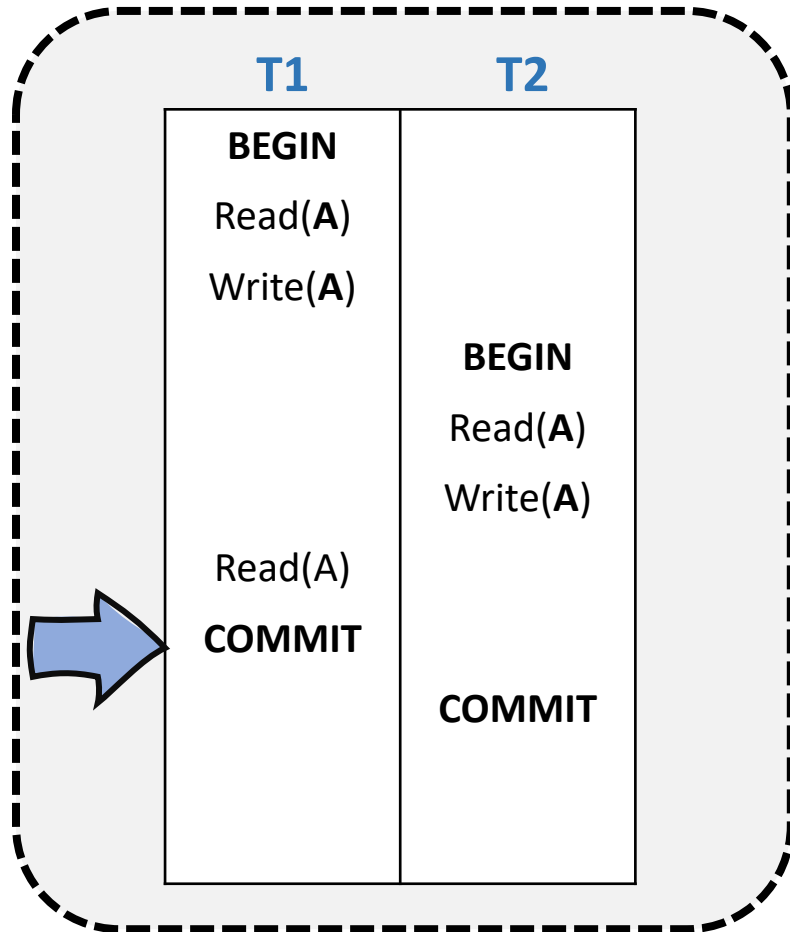
SI - first committer wins



Verzia	Hodnota	Štart	Koniec
A ₀	100	0	1
A ₁	120	1	2
A ₂	200	2	-

Tx id	Timestamp	Stav
T1	1	Aktívna
T2	2	Aktívna

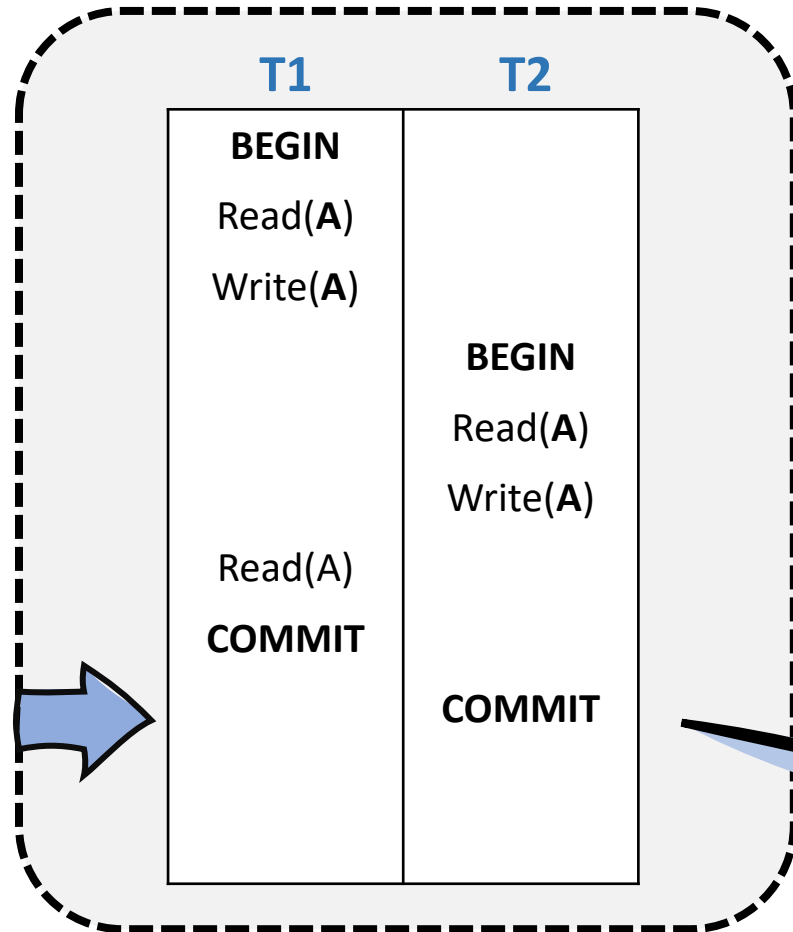
SI - first committer wins



Verzia	Hodnota	Štart	Koniec
A ₀	100	0	1
A ₁	120	1	2
A ₂	200	2	-

Tx id	Timestamp	Stav
T1	1	Committed
T2	2	Aktívna

SI - first committer wins



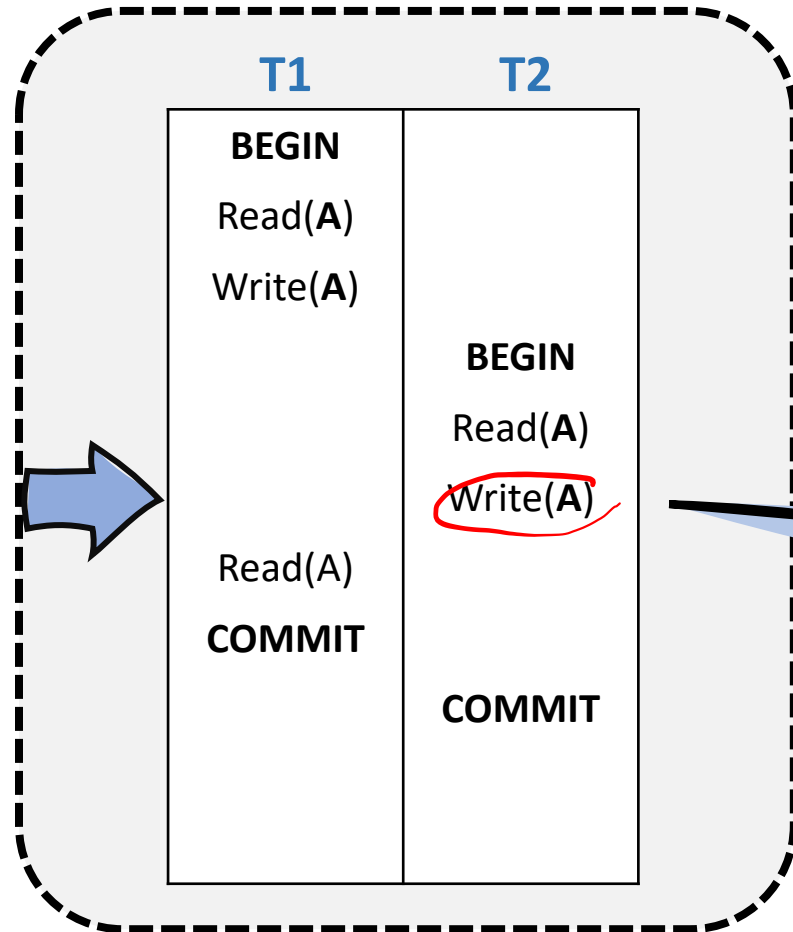
Verzia	Hodnota	Štart	Koniec
A ₀	100	0	1
A ₁	120	1	2
A ₂	200	2	-

Tx id	Timestamp	Stav
T1	1	Committed
T2	2	aktívna

Commit alebo Abort ?

SI - first updater wins

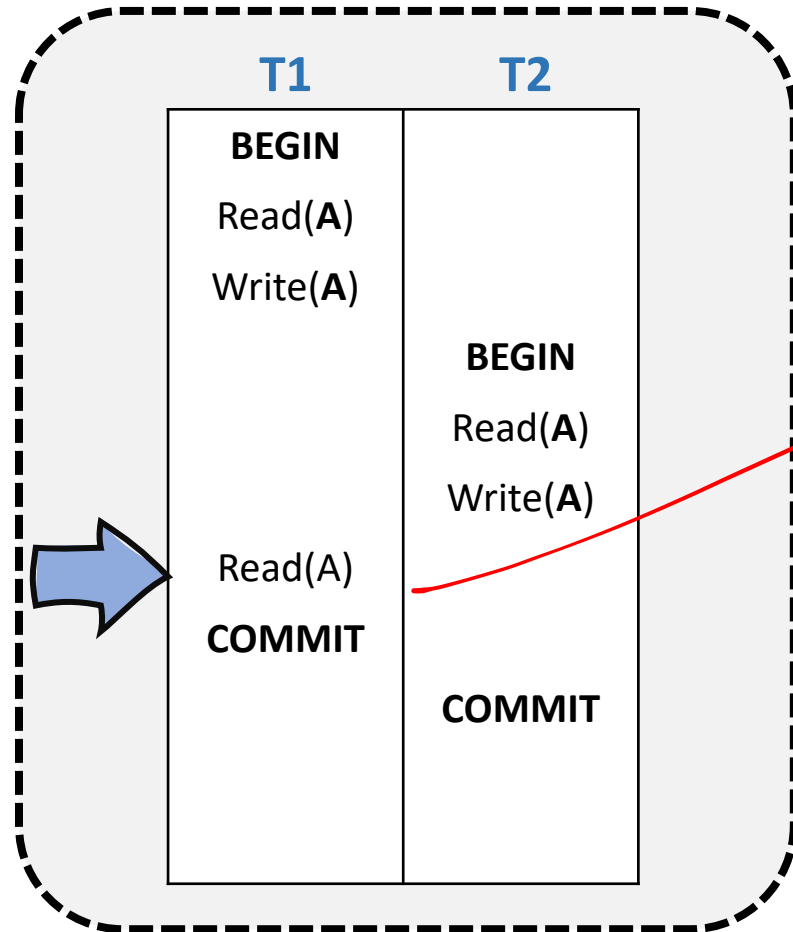
SI - first updater wins



Verzia	Hodnota	Štart	Koniec
A ₀	100	0	1
A ₁	120	1	-

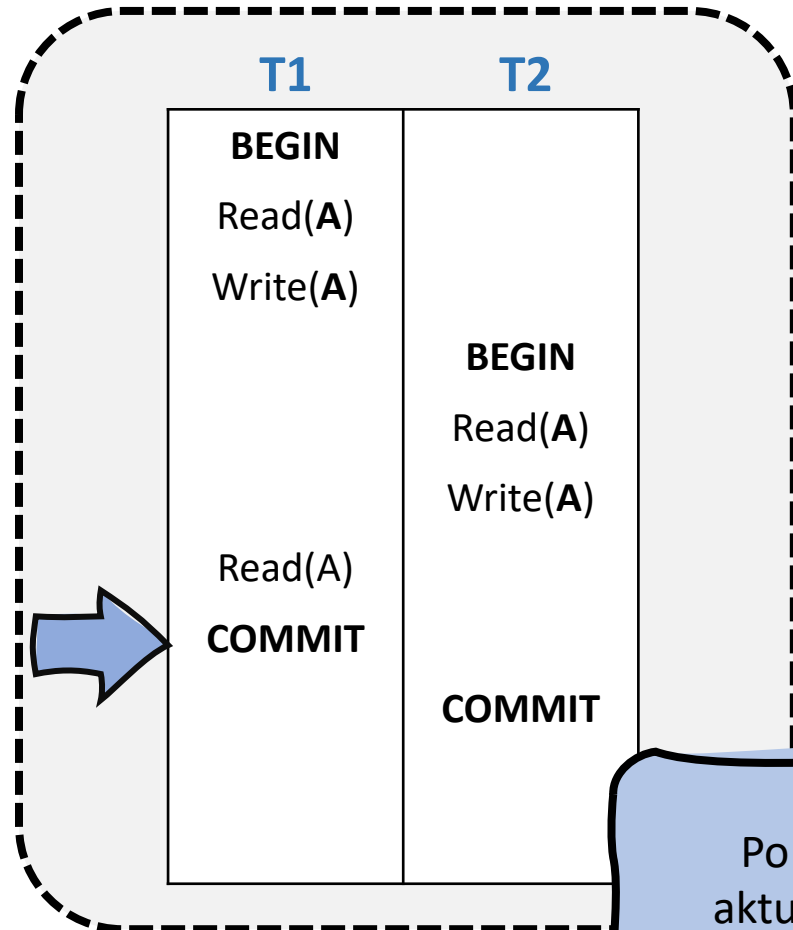
Musí počkať kým T1 urobí
COMMIT alebo ABORT

SI - first updater wins



Verzia	Hodnota	Štart	Koniec
A ₀	100	0	1
A ₁	120	1	-

SI - first updater wins

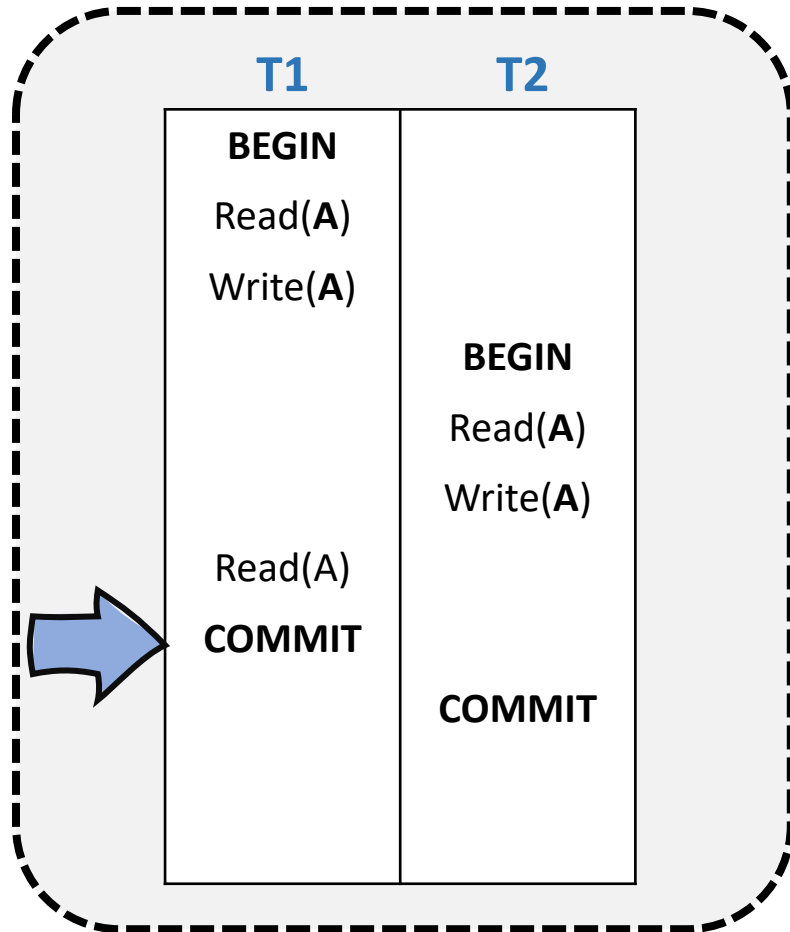


Verzia	Hodnota	Štart	Koniec
A ₀	100	0	1
A ₁	120	1	-

Tx id	Timestamp	Stav
T1	1	Committed
	2	Aktívna

Po commite môže T2
aktualizovať hodnotu A

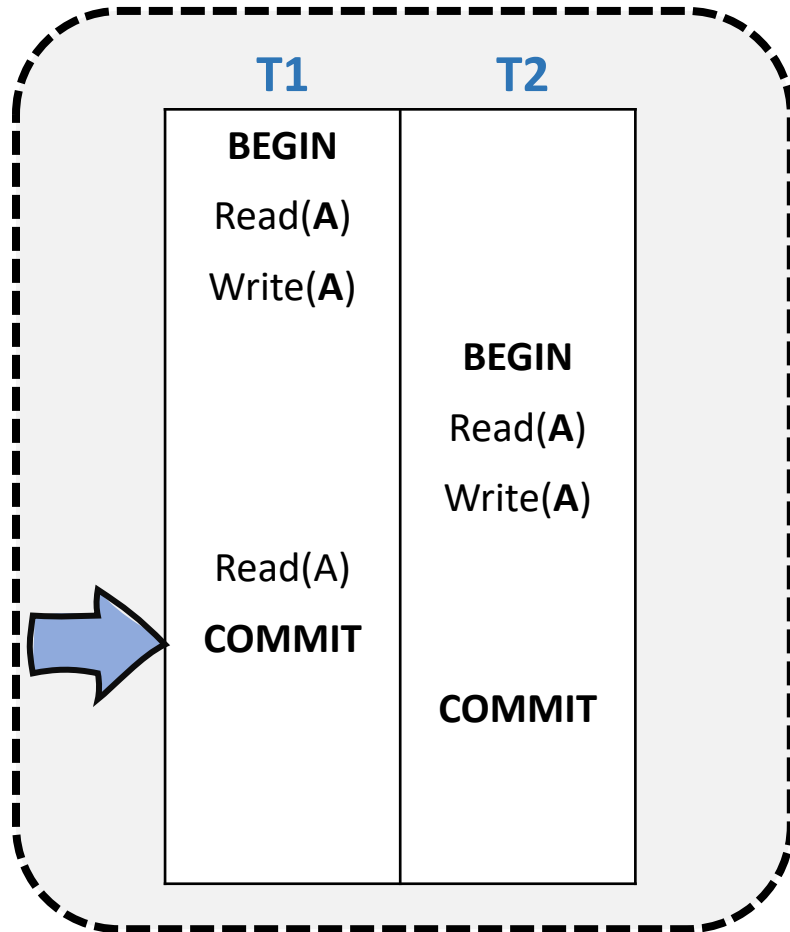
SI - first updater wins



Verzia	Hodnota	Štart	Koniec
A ₀	100	0	1
A ₁	120	1	2
A ₂	150	2	-

Tx id	Timestamp	Stav
T1	1	Committed
T2	2	Aktívna

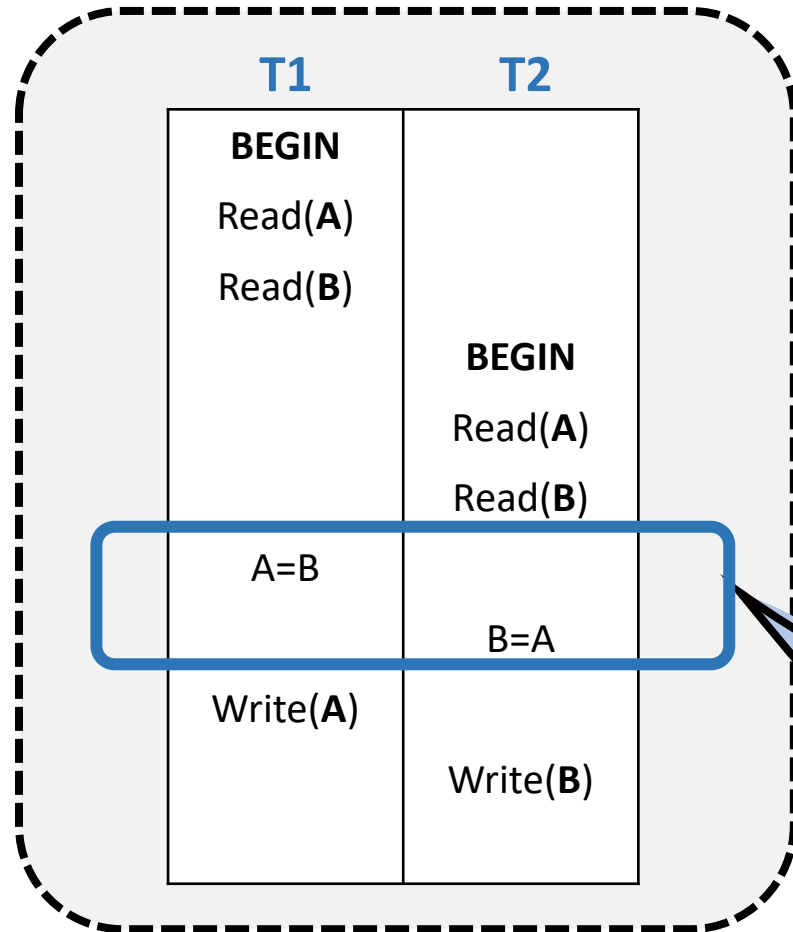
SI - first updater wins



Verzia	Hodnota	Štart	Koniec
A ₀	100	0	1
A ₁	120	1	2
A ₂	150	2	-

Tx id	Timestamp	Stav
T1	1	Committed
T2	2	Aktívna

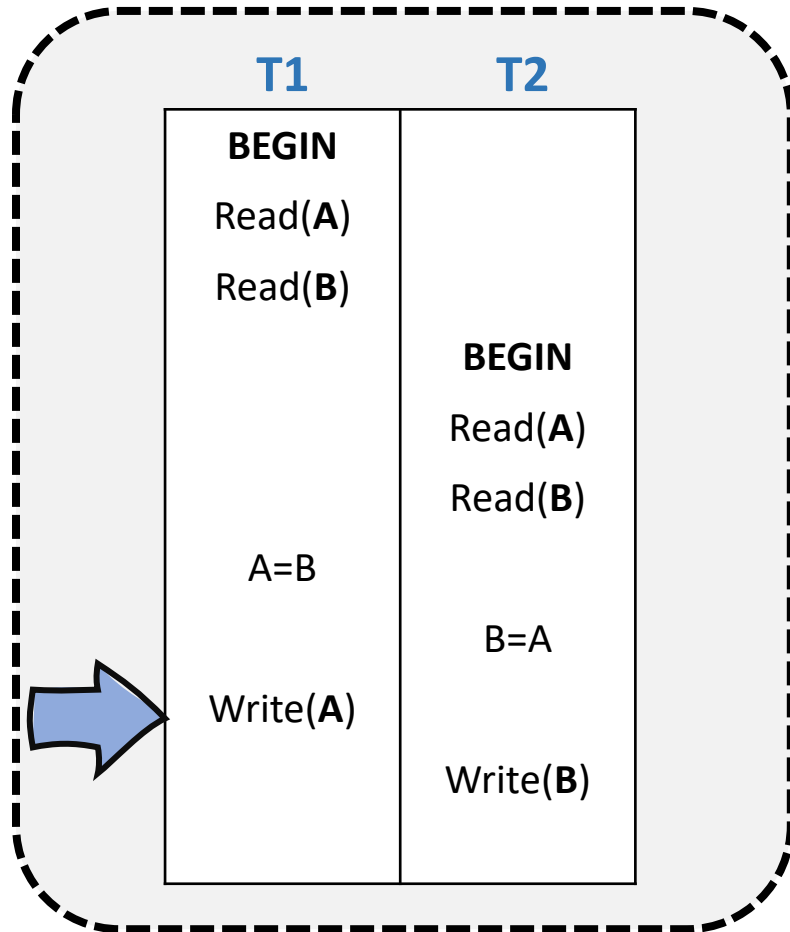
Snapshot Isolation



Verzia	Hodnota	Štart	Koniec

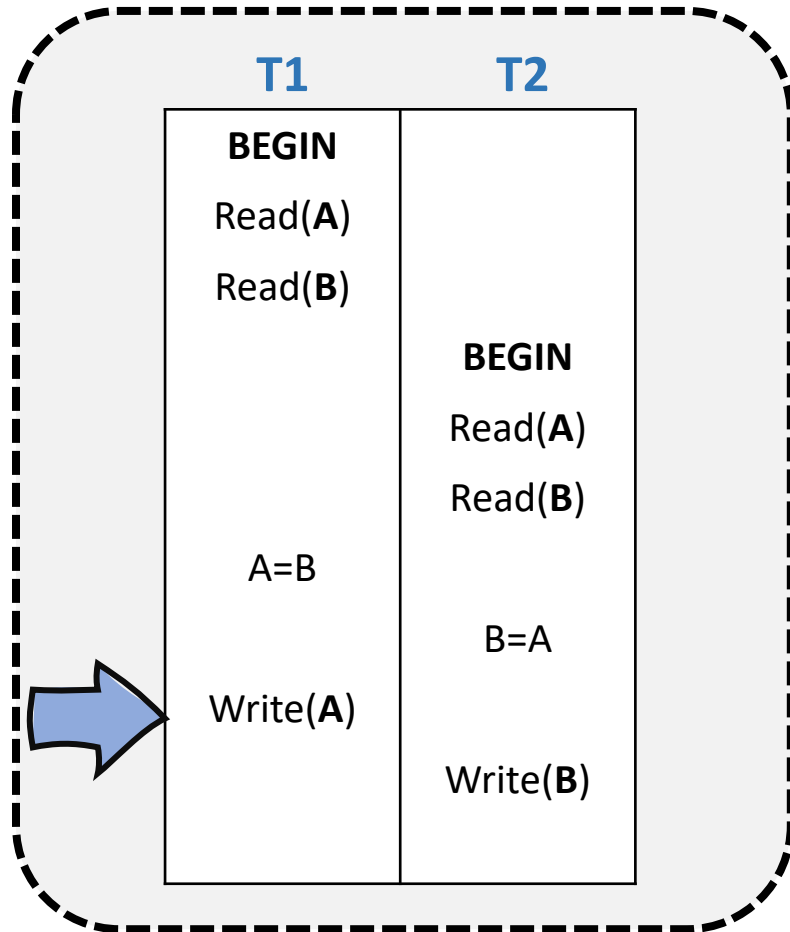
Operácie, ktoré sa uskutočňujú niekde na backende

Snapshot Isolation



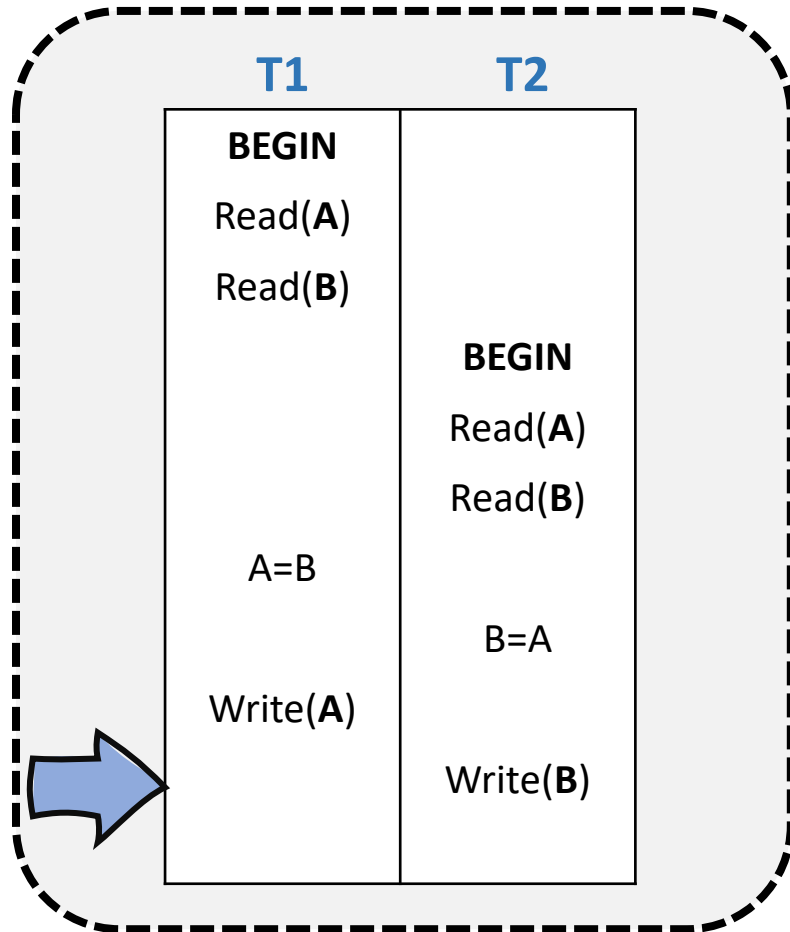
Verzia	Hodnota	Štart	Koniec
A ₀	100	0	-
B ₀	200	0	-

Snapshot Isolation



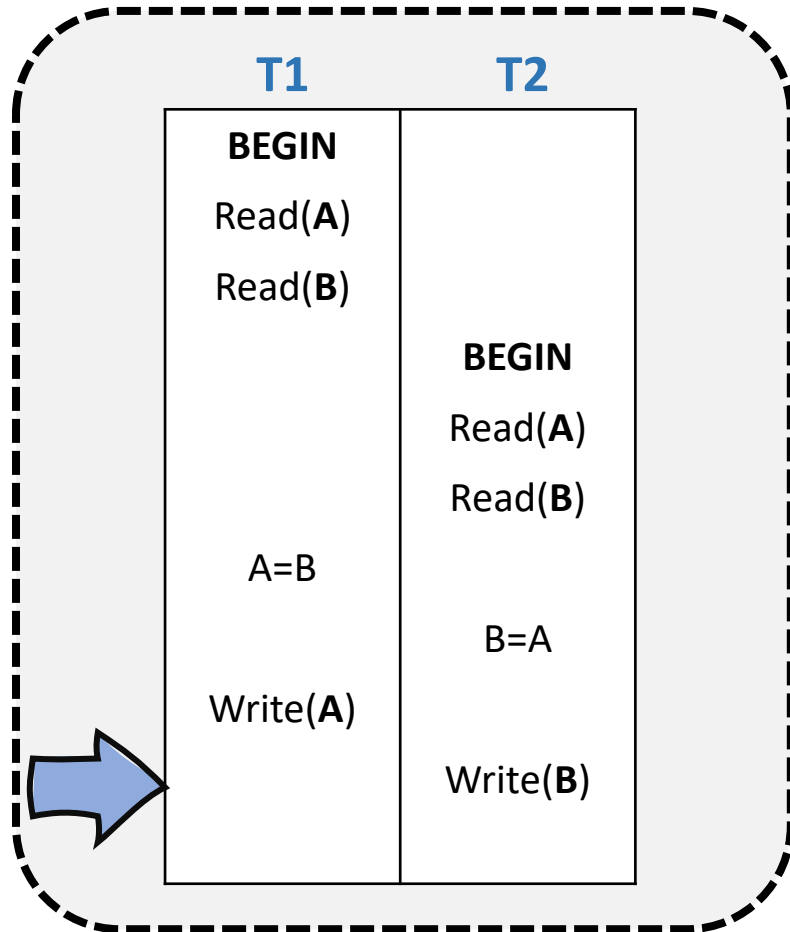
Verzia	Hodnota	Štart	Koniec
A ₀	100	0	1
B ₀	200	0	-
A ₁	200	1	-

Snapshot Isolation



Verzia	Hodnota	Štart	Koniec
A ₀	100	0	1
B ₀	200	0	2
A ₁	200	1	-
B ₁	100	2	-

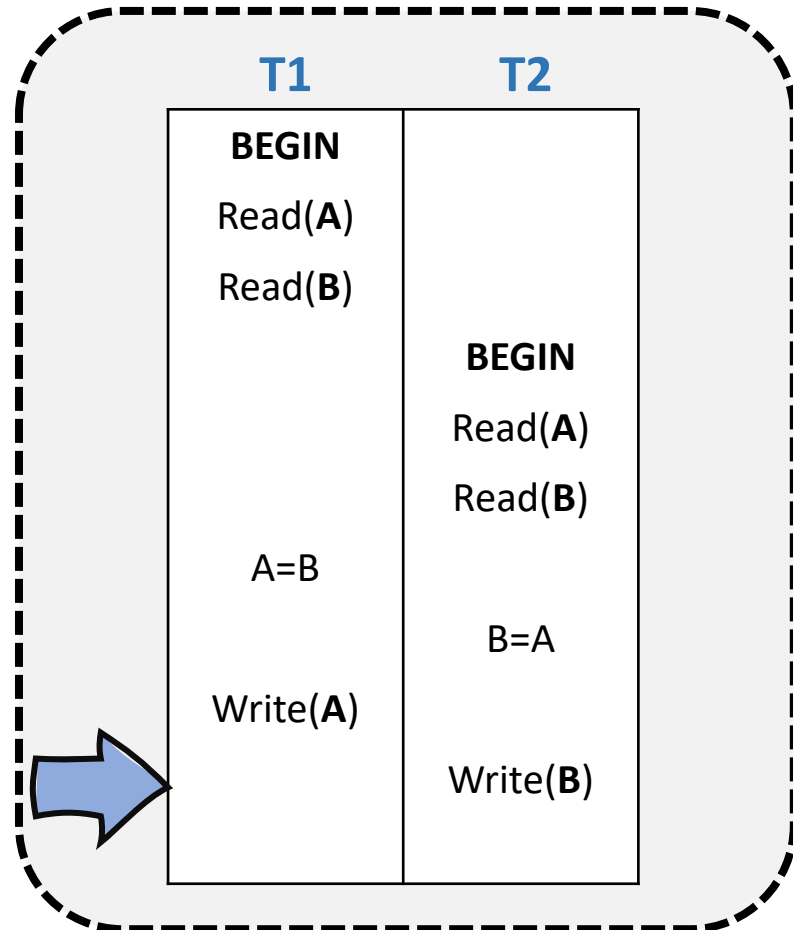
Snapshot Isolation



Verzia	Hodnota	Štart	Koniec
A ₀	100	0	1
B ₀	200	0	2
A ₁	200	1	-
B ₁	100	2	-

Aký problém vznikol ?

Snapshot Isolation



Verzia	Hodnota	Štart	Koniec
A ₀	100	0	1
B ₀	200	0	2
A ₁	200	1	-
B ₁	100	2	-

Problém nazývaný Write Skew

Write Skew - riešenie I

```
SELECT *  
  FROM instructor  
 WHERE id = 22222 FOR UPDATE;
```

Z pohľadu DBMS bude brané
ako "write"

Write Skew - riešenie 2

- **Serializable snapshot isolation**

- berie do úvahy konfliktnú hranu medzi read-write operáciami

Izolácia v praktických systémoch

- Štandard ANSI/ISO SQL používa rétoriku odlišnú od teórie, na ktorej buduje. Definuje 4 stupne izolácie transakcií:
 - READ UNCOMMITTED,
 - READ COMMITTED,
 - REPEATABLE READ,
 - SERIALIZABLE (Najvyšší stupeň izolácie)
- Ideou je umožniť aplikačným programom obetovať ACID garancie v záujme urýchlenia aplikácií

Izolácia v praktických systémoch

Isolation Level	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	May occur	May occur	May occur
Read Committed	Don't occur	May occur	May occur
Repeatable Read	Don't occur	Don't occur	May occur
Serializable	Don't occur	Don't occur	Don't occur

- <https://pgdash.io/blog/postgres-transactions.html>
- <https://www.interdb.jp/pg/pgsql05.html>

Zdroje

- Abraham Silberschatz; Henry F. Korth; S. Sudarshan; Database System Concepts
- Intro to Database Systems <https://db.cs.cmu.edu>