

<http://www.dcs.fmph.uniba.sk/~plachetk/TEACHING/DB1>

Tomáš Plachetka

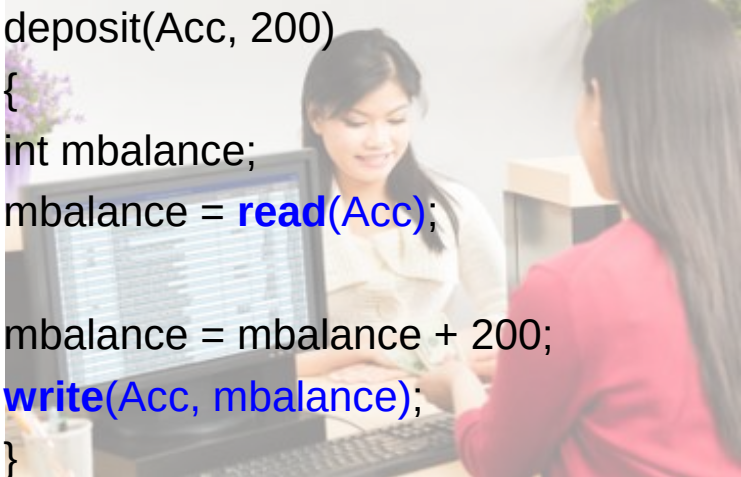
Fakulta matematiky, fyziky a informatiky,
Univerzita Komenského, Bratislava

Zima 2025–2026

Transakcie: súčasný prístup klientov k zdieľaným dátam

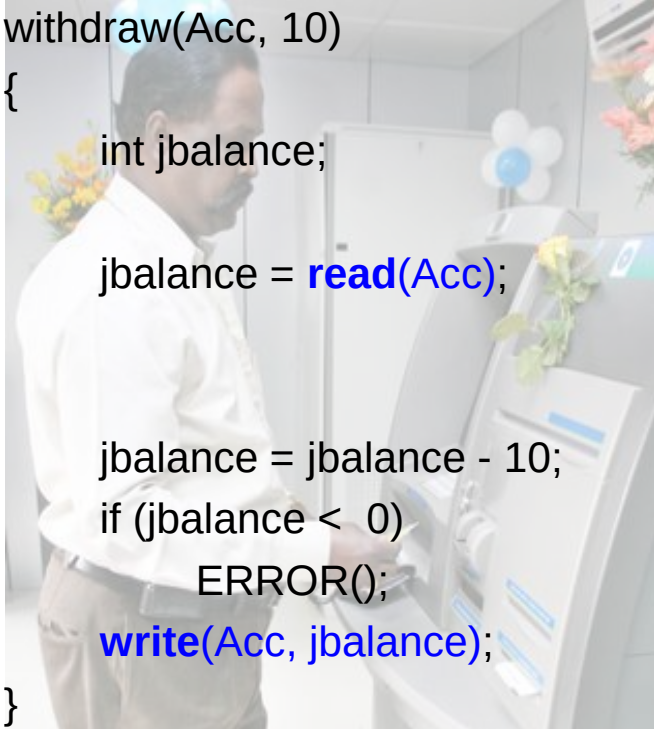
Od databázových systémov sa vyžaduje nielen riešenie manipulácie s dátami (dotazy, vkladanie, mazanie a aktualizácie dát). **Musia počítať so súčasným prístupom viacerých používateľov k databáze**

Príklad: Mary a John majú spoločný účet v banke, Acc, na ktorom je teraz **£1000**. Mary je práve v banke a v pokladni ukladá na Acc £200. John v tej chvíli vyberá z Acc £10



```
deposit(Acc, 200)
{
  int mbalance;
  mbalance = read(Acc);
  mbalance = mbalance + 200;
  write(Acc, mbalance);
}
```

1000
1200
1200



```
withdraw(Acc, 10)
{
  int jbalance;
  jbalance = read(Acc);
  jbalance = jbalance - 10;
  if (jbalance < 0)
    ERROR();
  write(Acc, jbalance);
}
```

1000
990
990

Na účte Acc má byť **£1190**, lenže je len **£990**. Mary ani John za to nemôžu. Kto teda?

Transakcie: zlé časovanie

Problém spôsobila banka (ktorej databázový systém nerátal so „zlým časovaním“)

Z pohľadu systému sa stalo toto:

```
mary.read(Acc); john.read(Acc); mary.write(Acc); john.write(Acc)
```

Ale napríklad takéto poradie by bolo v poriadku:

```
mary.read(Acc); mary.write(Acc); john.read(Acc); john.write(Acc)
```

Aj takéto poradie by bolo v poriadku:

```
john.read(Acc); john.write(Acc); mary.read(Acc); mary.write(Acc)
```

Dokonca aj toto by bolo v poriadku:

```
john.read(Acc); mary.read(Acc); john.write(Acc); a systém povie Mary, že vklad zlyhal
```

Od databázových systémov sa vyžaduje, aby dovolili **len „dobré“ postupnosti** (každému klientovi vytvorila ilúziu, ako keby s databázou pracoval sám). Okrem toho sa vyžaduje **odolnosť voči výpadkom v ľubovoľnom momente**. Napríklad, kedykoľvek môže vypadnúť elektrický prúd v banke či v bankomate. Kedykoľvek sa môže prerušiť spojenie medzi klientom a bankou. Kedykoľvek sa môže pokaziť hard-disk či počítač. Atd'. Systém sa musí z takýchto havárií zotaviť, **databáza musí zostať konzistentná**

Odporúčaná literatúra k tejto prednáške (link na elektronickú verziu je na web stránke prednášky):

P.A. Bernstein, V. Hadzilacos, N. Goodman: Concurrency Control and Recovery in Database Systems, Addison-Wesley, 1987

Transakcie (z pohľadu používateľa)

Definícia (neformálna). **Transakcia** je program, ktorý pristupuje k databáze

Transakcií môže v tom istom čase bežať viac, pričom jedna o druhej nevedia. Napríklad k bankovým kontám (dokonca aj k rovnakému kontu) môže pristupovať naraz operátor v banke aj klient pri bankomate. Alebo viacero klientov či operátorov môže naraz narábať s rezervačným systémom leteniek. Alebo viacero klientov môže narábať s knižničnou databázou

Pre jednoduchosť v tejto prednáške **budeme predpokladať, že databáza je centralizovaná.** (Z pohľadu transakcií je jedno, či je databáza centralizovaná alebo distribuovaná, avšak z hľadiska systému to jedno nie je. Implementácia distribuovaného systému prináša kvalitatívne nové problémy.)

Transakcie (z pohľadu databázového systému)

Definícia. **Transakcia** je postupnosť nasledujúcich operácií. Pritom *START musí* byť v tej postupnosti práve raz, a to na začiatku (ak to tak nie je, systém transakciu abortuje). COMMIT resp. ABORT *môže* byť v tej postupnosti *najviac* raz, a to na konci

READ: číta objekt (napr. záznam) z databázy

WRITE: zapisuje objekt do databázy

INSERT: vkladá objekt do databázy

DELETE: odstraňuje objekt z databázy

START: začína transakciu

COMMIT: úspešne končí transakciu

ABORT: neúspešne končí transakciu

Príklad: bankový prevod sumy S z účtu A na účet B

$\text{transfer}(S, A, B)$

```
{  
  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:

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

- Súčasne môže bežať viacero inštancií transakcie *transfer*. Z pohľadu systému sú rôzne, takže systém im prideluje rôzne IDs (táto transakcia má ID 1)
- V prípade READ systém nevidí lokálnu premennú, do ktorej sa ukladá hodnota prečítaná z databázy, vidí len tú hodnotu
- V prípade WRITE systém nevidí lokálnu premennú, ktorej hodnota sa ukladá do databázy, vidí len tú hodnotu

Požiadavky na transakčný databázový systém: ACID

Atomicity: transakcia sa vykoná buď celá alebo vôbec

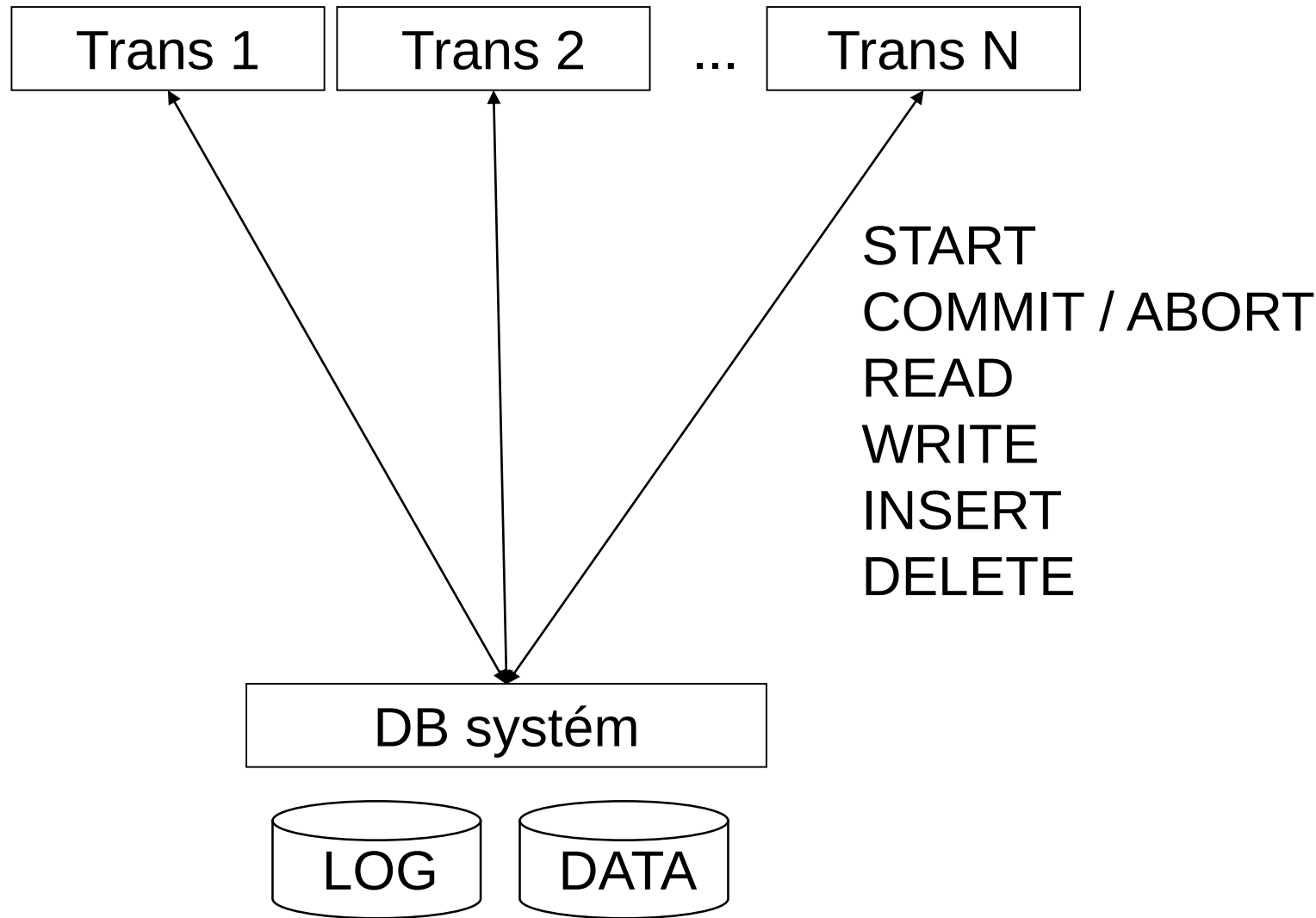
Consistency: vykonanie transakcie znamená prechod od konzistentného stavu databázy opäť ku konzistentnému stavu
(toto nie je požiadavka na systém, ale na implementorov transakcií)

Isolation: hoci systém môže vykonávať viacero transakcií „paralelne“, výsledný efekt musí byť taký, ako keby sa vykonávali celé transakcie sériovo (jedna po druhej)

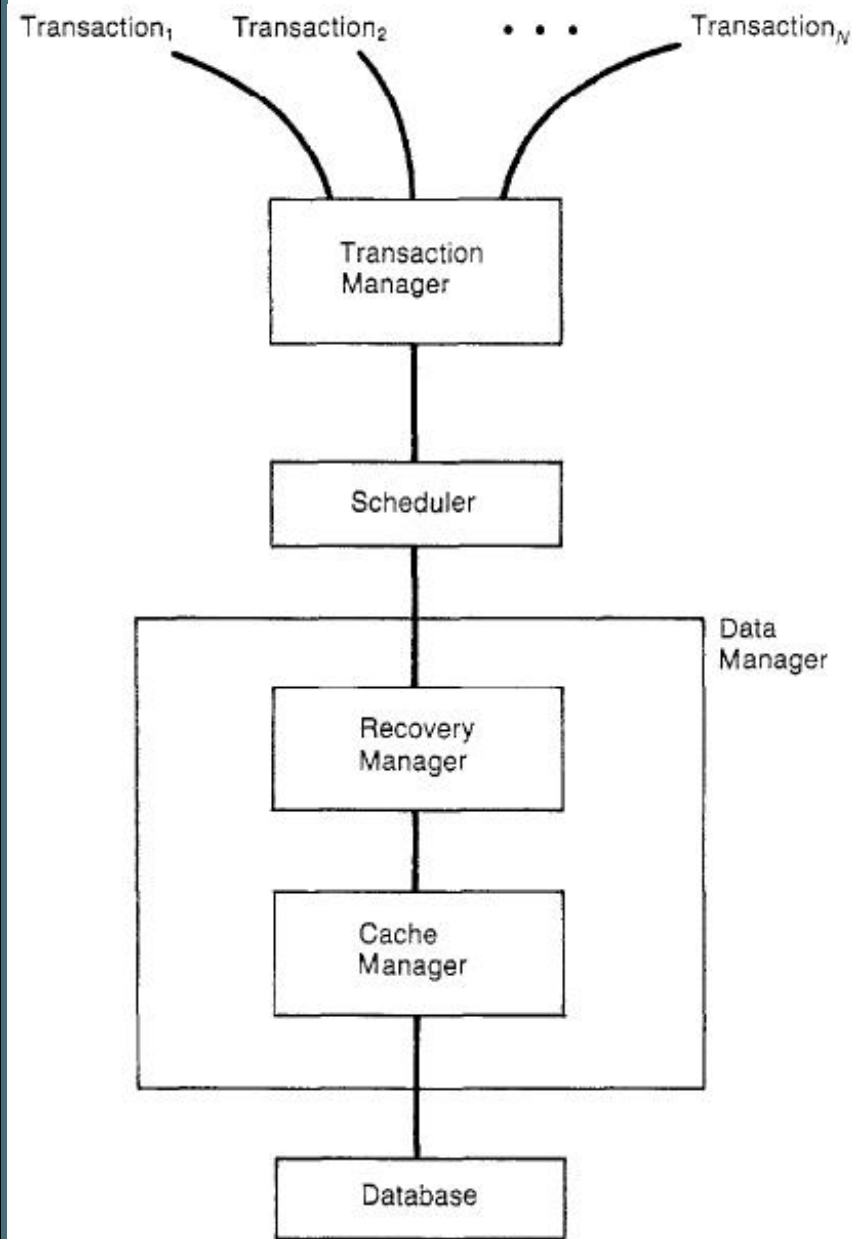
Durability: ak transakcia úspešne skončí (t.j. ak systém vykoná COMMIT), tak všetky zmeny, ktoré vykonala v databáze, budú navždy zachované

Toto musí byť garantované aj v prípade havárií, ako je napríklad nečakaný výpadok v ľubovoľnom momente (na strane klienta alebo servera alebo oboch súčasne)

Transakčný databázový systém: 2-tier architektúra



Transakčný databázový systém



start, read, write, commit, abort

autorizácia transakcií, čítanie a
bufferovanie operácií

poradie vykonávania operácií

log-file, riešenie výpadkov

synchronizácia RAM a disku

Dôvody pre ABORT transakcie

- **Výpadok na strane klienta** spôsobí ABORT transakcie
- **Výpadok spojenia medzi klientom a serverom** spôsobí ABORT transakcie
- **Výpadok na strane servera** spôsobí ABORT všetkých transakcií, ktoré sú v tom momente aktívne
- **Transakcia sa sama rozhodne pre ABORT**, napr. keď užívateľ na strane klienta stlačí tlačítko „cancel“
- **Systém (presnejšie, scheduler) sa sám rozhodne pre ABORT transakcie** (napr. keď zistí, že COMMIT nebude môcť nikdy vykonať, lebo by tým porušil niektorú z ACID požiadaviek)
- ...

Samozrejme, cieľom systému je ABORTovať „čo najmenej“ transakcií

Rozvrhy (plány, histórie)

Definícia. **Rozvrh (plán, história)** je postupnosť, ktorá vznikne premiešaním operácií niekoľkých transakcií, vo všeobecnosti nekompletných. Toto premiešanie nie je ľubovoľné—zachováva poradie operácií jednotlivých transakcií (**projekcia rozvrhu** na individuálnu transakciu je postupnosť operácií tej transakcie) **Rozvrh je kompletný**, ak v ňom všetky transakcie končia buď operáciou COMMIT alebo ABORT

Transakcia je v danom čase aktívna, ak v tom čase už začala (t.j. bola vykonaná jej operácia START) a zároveň v tom čase ešte neskončila (t.j. nebola vykonaná jej operácia COMMIT či ABORT)

Scheduler transformuje (on-the-fly, online) **vstupný rozvrh** na **výstupný rozvrh**. Na to používa operácie **execute**, **delay** a **reject**

„Dobré“ vs. „zlé“ rozvrhy

Príklad (Garcia-Molina): 2 transakcie

T1:	Read(A)	T2:Read(A)
	$A \leftarrow A+100$	$A \leftarrow A \times 2$
	Write(A)	Write(A)
	Read(B)	Read(B)
	$B \leftarrow B+100$	$B \leftarrow B \times 2$
	Write(B)	Write(B)

Constraint: $A=B$

Constraint: $A=B$

„Dobré“ vs. „zlé“ rozvrhy

Rozvrh A: **dobrý**

		A	B
T1	T2	25	25
Read(A); A ← A+100			
Write(A);		125	
Read(B); B ← B+100;			
Write(B);			125
Commit;			
	Read(A);A ← A×2;		
	Write(A);	250	
	Read(B);B ← B×2;		
	Write(B);		250
	Commit;	250	250

„Dobré“ vs. „zlé“ rozvrhy

Rozvrh B: **dobrý**

		A	B
T1	T2	25	25
	Read(A); A ← A×2;		
	Write(A);	50	
	Read(B); B ← B×2;		
	Write(B);		50
	Commit;		
Read(A); A ← A+100			
Write(A);		150	
Read(B); B ← B+100;			
Write(B);			150
Commit;		150	150

„Dobré“ vs. „zlé“ rozvrhy

Rozvrh C: **dobrý**

		A	B
T1	T2	25	25
Read(A); A ← A+100			
Write(A);		125	
	Read(A); A ← A×2;		
	Write(A);	250	
Read(B); B ← B+100;			
Write(B);			125
Commit;			
	Read(B); B ← B×2;		
	Write(B);		250
	Commit;	250	250

„Dobré“ vs. „zlé“ rozvrhy

Rozvrh D: **zlý**

		A	B
T1	T2	25	25
Read(A); A ← A+100			
Write(A);		125	
	Read(A); A ← A×2;		
	Write(A);	250	
	Read(B); B ← B×2;		
	Write(B);		50
	Commit;		
Read(B); B ← B+100;			
Write(B);			150
Commit;		250	150

„Dobré“ vs. „zlé“ rozvrhy

Rozvrh E (rovnaký ako D, s mierne zmenenou T2): **zlý, hoci...**

		A	B
T1	T2'	25	25
Read(A); A ← A+100			
Write(A);		125	
	Read(A);A ← A×1;		
	Write(A);	125	
	Read(B);B ← B×1;		
	Write(B);		25
	Commit;		
Read(B); B ← B+100;			
Write(B);			125
Commit;		125	125

„Dobré“ vs. „zlé“ rozvrhy

Intuitívne, „dobré“ rozvrhy sú A, B, C

Rozvrh D nezachováva izoláciu

Rozvrh E zachováva izoláciu, ale len „náhodou“

Rozvrhy A, B sú **sériové**

Rozvrh C je **sériovateľný** (presnejšie, **konflikt-sériovateľný**)

Rozvrhy D a E sú „zlé“, lebo nie sú sériovateľné (to, že rozvrh E sa správa ako sériový rozvrh, závisí od vnútornej štruktúry transakcií resp. obsahu dát)

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

Konfliktné operácie a konflikt-ekvivalencia rozvrhov

Definícia. V histórii sú dve **operácie konfliktné** ak patria rôznym transakciám, ich operandom je rovnaký objekt a aspoň jedna z tých operácií je *write*

Idea generovania „dobrých“ rozvrhov: postupnosť nekonfliktných operácií môžeme ľubovoľne premiešať a rozvrh ostane „dobrý“; ale postupnosť konfliktných operácií musíme zachovať (zachovať treba samozrejme aj projekcie na jednotlivé transakcie)

Definícia. Dve **histórie sú konflikt-ekvivalentné** ak

- pozostávajú z rovnakých operácií a
- relatívne poradie každých dvoch konfliktných operácií je rovnaké v oboch históriách

Sériové a sériovateľné rozvrhy

Definícia. **Rozvrh je sériový** ak je kompletný (t.j. každá transakcia v tom rozvrhu končí commitom alebo abortom) a pre každú dvojicu transakcií T1, T2 platí, že buď všetky operácie T1 v tom rozvrhu predchádzajú operáciám T2 alebo naopak

Definícia. **Rozvrh je konflikt-sériovateľný** ak jeho projekcia na commitované transakcie je konflikt-ekvivalentná niektorému sériovému rozvrhu tých commitovaných transakcií

Prečo projekcia na commitované transakcie? Lebo len pre commitované transakcie dáva systém nejaké garancie!

Testovanie sériovateľnosti rozvrhov: precedenčný graf

Definícia. Nech S je rozvrh, ktorý obsahuje commitované transakcie $T_1, \dots, T_n$.

Precedenčný graf je orientovaný graf s vrcholmi $T_1, \dots, T_n$, v ktorom je hrana $T_i \rightarrow T_j$ práve vtedy, ak pre niektorú konfliktnú dvojicu operácií $O_m \in T_i$, $O_n \in T_j$ platí, že O_m je v rozvrhu skôr ako O_n .

Príklad (rozvrh E): $r1(A)$, $w1(A)$, $r2(A)$, $w2(A)$, $r2(B)$, $w2(B)$, $c2$, $r1(B)$, $w1(B)$, $c1$

$\text{Read}(A); A \leftarrow A+100$

$\text{Write}(A);$

$\text{Read}(A); A \leftarrow A \times 1;$

$\text{Write}(A);$

$\text{Read}(B); B \leftarrow B \times 1;$

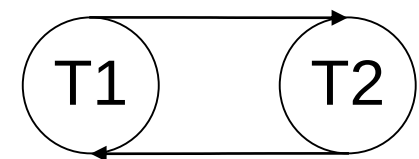
$\text{Write}(B);$

$\text{Commit};$

$\text{Read}(B); B \leftarrow B+100;$

$\text{Write}(B);$

$\text{Commit};$



Testovanie sériovateľnosti rozvrhov: precedenčný graf

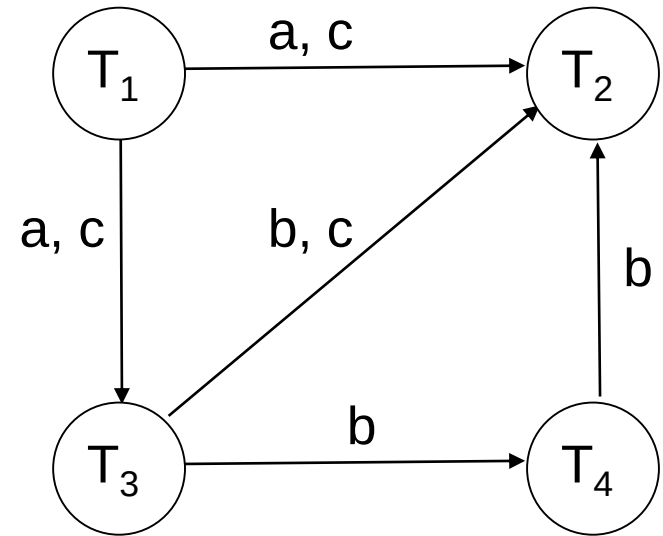
Veta. Rozvrh je konflikt-sériovateľný práve vtedy, ak jeho precedenčný graf je acyklický

Veta. Ak je rozvrh konflikt-sériovateľný, tak topologické usporiadanie jeho precedenčného grafu hovorí **ktorému** sériovému rozvrhu je ten rozvrh ekvivalentný

Testovanie sériovateľnosti rozvrhov: precedenčný graf

Príklad (kde T_1, T_2, T_3, T_4 sú commitované transakcie)

T_1	T_2	T_3	T_4
read(a) read(c) write(a) write(c)	read(b)	read(b) write(b)	write(b)
	read(a) write(c)	read(a) write(c)	

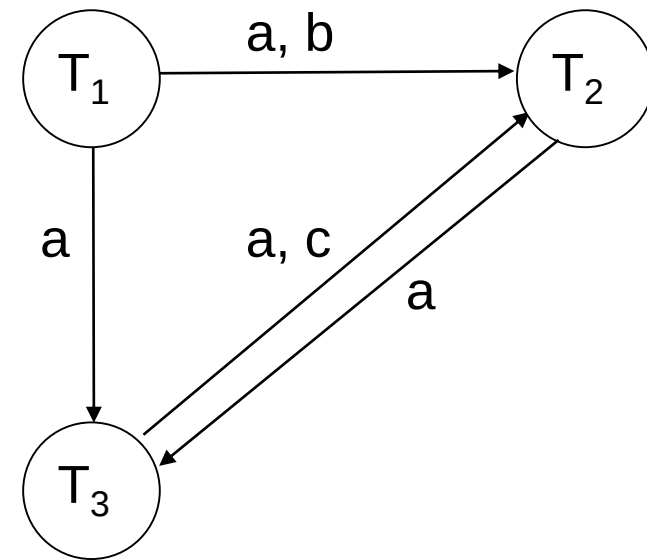


Precedenčný graf
neobsahuje cyklus, takže
rozvrh je konflikt-
séριοvateľný. Rozvrh je
konflikt-ekvivalentný
séριοvému rozvrhu
 $T_1 \rightarrow T_3 \rightarrow T_4 \rightarrow T_2$

Testovanie sériovateľnosti rozvrhov: precedenčný graf

Príklad (kde T_1 , T_2 , T_3 sú commitované transakcie)

T_1	T_2	T_3
read(a) read(b) write(a)		
	read(b)	read(a)
	read(c) write(b) read(a)	write(c)
	write(c) write(a)	write(a)



Precedenčný graf obsahuje
cyklus $T_2 \rightarrow T_3 \rightarrow T_2$, takže rozvrh
nie je konflikt-sériovateľný

View-sériovateľnosť

Ekvivalenciu rozvrhov je možné definovať aj inými spôsobmi.

Definícia. 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 $r_2(X)$

Definícia. **Dve histórie H a H' sú view-ekvivalentné**, ak (sú definované nad tými istými transakciami a zároveň)

- pre každú dvojicu operácií v H , kde nejaká transakcia T_2 číta X od T_1 existuje rovnaká dvojica operácií v H' , 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 H , tak aj v H' je T_i je posledná transakcia ktorá píše do X (final write)

(a zároveň to všetko platí aj s prehodenými H' a H)

Intuitívne, rozvrhy sú view-ekvivalentné, ak majú rovnaký efekt na transakcie (čítania) aj na databázu (zápisy)

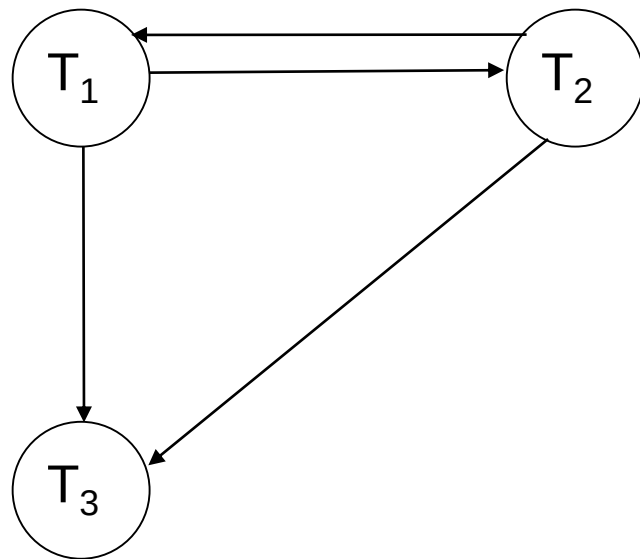
View-sériovateľnosť

Definícia: rozvrh je **view-sériovateľný**, ak commitovaná projekcia **každého jeho prefixu** je view-ekvivalentná niektorému sériovému rozvrhu

Príklad:

$w1(X)$, $r2(Y)$, $w1(Y)$, $r3(Y)$, $w2(X)$, $w1(X)$, $w3(X)$, $c1$, $c2$, $c3$

Tento rozvrh nie je konflikt-sériovateľný, ale je view-sériovateľný
(je view-ekvivalentný sériovému rozvrhu $T_2 \rightarrow T_1 \rightarrow T_3$)



Platí: ak je rozvrh konflikt-sériovateľný, tak je aj **view-sériovateľný** (ale nie vždy to platí naopak)

Možno sa to nezdá, ale **testovanie, či je rozvrh view-sériovateľný, je NP-ťažký problém** (hoci testovanie view-ekvivalencie rozvrhu **dvoch** transakcií sa dá urobiť v polynomiálnom čase)

Izolácia: Generovanie konflikt-sériovateľných rozvrhov

Vieme **testovať**, či je rozvrh sériovateľný (konflikt-sériovateľnosť vieme testovať efektívne). Ale vieme zostrojiť online algoritmus, ktorý **efektívne generuje** iba sériovateľné rozvrhy?

Triviálna možnosť je generovať *len sériové rozvrhy*, t.j. vykonávať len operácie jednej transakcie, kým tá transakcia neskončí (scheduler operácie ostatných transakcií dovtedy zdržiava a ukladá niekam do buffera). Táto možnosť je však zjavne nepraktická, lebo príliš redukuje priepustnosť systému. Generovať view-sériovateľné rozvrhy (opačný extrém) je ťažké. Existuje niekoľko metód, ktoré generujú **nejaké podtriedy (view- resp. konflikt) sériovateľných rozvrhov** medzi týmito dvomi extrémami

Izolácia: Generovanie konflikt-sériovateľných rozvrhov

- **Algoritmus zámkov**: transakcie zamykajú dátové objekty pred každým čítaním či zápisom, aby iné transakcie s nimi nemohli ľubovoľne interferovať
- **Algoritmus časových pečiatok**: na základe časových pečiatok sa pri každej read/write operácii systém rozhoduje, či transakcia bude pokračovať alebo abortuje
- **Multiversion algoritmus**: každá transakcia zapisuje svoje zmeny do svojej lokálnej kópie databázy namiesto do ostrej databázy. Systém rozhodne až pri žiadosti o COMMIT, či transakcia commituje a lokálne zmeny sa prenesú do ostrej databázy, alebo či transakcia abortuje. Systém udržiava pre jeden dátový objekt viacero lokálnych verzií a pri operácii READ rozhoduje, ktorá verzia sa bude čítať
- **Validačný algoritmus**: systém **optimisticky** vykonáva operácie tak ako idú, ale prísne kontroluje, či dovolí COMMIT

Izolácia: Zamykanie (locking)

Idea: do rozvrhu sa pridajú operácie lock a unlock, ktorými sa zamykajú resp. odomykajú dátové objekty. Transakcia smie dátový objekt čítať/písať len v momente, keď vlastní potrebný zámok na ten objekt. Commit je triviálny, netreba nič kontrolovať

Základné typy zámkov:

- Read-lock (RL): dovoľuje iba čítať
- Write-lock (WL), exclusive-lock: dovoľuje **čítať aj písať**

Granularita zámkov:

- (na jeden atribút jedného záznamu)
- na jeden záznam
- na celú tabuľku
- (na diskový blok)

Izolácia: Zamykanie (locking)

Kompatibilita zámkov:

	RL	WL
RL	true	false
WL	false	false

Inými slovami, transakcia smie získať RL na X aj vtedy, ak iná transakcia v tej chvíli drží RL na X. Akákoľvek iná kombinácia zámkov v jednom momente je neprípustná a transakcia žiadajúca o zámok musí v takom prípade **počkať**, kým tá druhá transakcia svoj zámok uvoľní (operáciou unlock). To čakanie zabezpečí scheduler tak, že operáciu žiadosti o zámok (t.j. operáciu RL, resp. WL) nevykoná hneď, ale odloží jej vykonanie na neskôr (delay)

Dvojfázové zamykanie (two-phase locking)

Pravidlá:

1. Ak transakcia už niekedy urobila unlock, nesmie už nikdy žiadať o ďalší lock. (T.j. v prvej fáze každá transakcia získava zámky, v druhej fáze ich odovzdáva späť systému.) Scheduler si udržiava stav zámkov, ktoré prideliť a vie ľahko otestovať, či niektorá transakcia porušuje toto dvojfázové pravidlo
2. Transakcia musí vlastniť potrebný zámok keď sa pokúša o čítanie/písanie dátového objektu. Aj toto vie scheduler jednoducho overiť

Ak niektorá transakcia poruší pravidlá hry, scheduler ju hneď abortuje (reject)

Po ukončení transakcie uvoľní scheduler zámky tej transakcie

Dvojfázové zamykanie (two-phase locking)

Veta: Two-phase locking generuje len konflikt-sériovateľné rozvrhy

Dôkaz: nepriamo. Nech je rozvrh dvojfázový a nech nie je konflikt-sériovateľný. To druhé znamená, že precedenčný graf obsahuje cyklus $T_i \rightarrow \dots \rightarrow T_i$, kde T_i je prvá transakcia v rozvrhu, ktorá je v nejakom cykle. Lenže to znamená, že transakcia T_i musela niečo zamykať po tom, čo nejaký zámok uvoľnila, inak by sa nedostala do toho druhého konfliktu, ktorý spôsobil ten cyklus. To je spor s dvojfázovosťou rozvrhu. QED

Naopak to neplatí. Nie každý konflikt-sériovateľný rozvrh sa dá generovať two-phase locking metódou. Napríklad:
 $r1(X), r2(X), r1(X), w2(X), c1, c2$

Dvojfázové zamykanie (two-phase locking)

Trieda rozvrhov generovaných 2PL sa dá rozšíriť napr. pridaním operácie upgrade-lock (UPGL), ktorá transakcii umožňuje konvertovať upgrade-lock na write-lock. Pravidlá pre pridelenie zámkov sa trochu skomplikujú, ale zvýši sa priepustnosť systému (samozrejme, generované rozvrhy ostanú konflikt-sériovateľné). S prideleným UPGL zámkom smie transakcia len čítať, no kedykoľvek smie požiadať o jeho konverziu na WL (alebo RL)

Kompatibilita zámkov RL, WL, UPGL:

	RL	UPGL	WL
RL	true	false	false
UPGL	true	false	false
WL	false	false	false

Asymetria UPGL a RL: systém smie prideliť UPGL, aj keď iné transakcie držia RL. Ale systém nesmie prideliť RL, keď už nejaká iná transakcia drží UPGL

Iná (tiež pesimistická) metóda izolácie: časové pečiatky

Každá transakcia T_i dostane pri svojom štarte timestamp $TS(T_i)$.

Každý **objekt** X má dve pečiatky: jedna sa aktualizuje keď sa X číta, druhá sa aktualizuje keď sa do X zapisuje

Keď transakcia číta resp. píše do X , tak nechá v X svoju časovú pečiatku, ak je pečiatka transakcie novšia ako príslušná pečiatka pri X

Pravidlá (Bernstein a iní ich trochu zoslabujú) :

1. Transakcia nesmie čítať hodnoty, ktoré písala neskôr začatá transakcia
2. Transakcia nesmie písať hodnoty, ktoré čítala alebo písala neskôr začatá transakcia

Zámky vs. časové pečiatky

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.

Optimistická metóda izolácie: validácia

Systém vykonáva operácie tak, ako prichádzajú. Pre každú transakciu T sleduje časy $TS(T)$, $TVAL(T)$, $TF(T)$ (pečiatka udalosti, ktorá ešte nenastala, je ∞), a dve množiny dát: read-set $RS(T)$ a write-set $WS(T)$. Transakcie nepíšu priamo do databázy, **píšu len do svojej lokálnej kópie** dát. Vo chvíli, keď T požiadala o commit, dostane pečiatku $TVAL(T)$ a začne sa jej validácia. Ak je validácia neúspešná, T sa abortuje. Ak validácia bola úspešná, T začne finálnu fázu, v ktorej sa lokálna kópia dát zapisuje do databázy. Po ukončení kopírovania dát dostane T pečiatku $TF(T)$.

Validácia T je **neúspešná**, ak existuje transakcia T_c , pre ktorú platí jedna z nasledujúcich podmienok:

$$\begin{aligned} & (RS(T) \cap WS(T_c) \neq \emptyset) \wedge TVAL(T_c) < TVAL(T) \wedge TF(T_c) > TS(T), \\ & (WS(T) \cap WS(T_c) \neq \emptyset) \wedge TVAL(T_c) < TVAL(T) \wedge TF(T_c) > TVAL(T) \end{aligned}$$

Idea: commitovaná projekcia výstupného rozvrhu bude konflikt-ekvivalentná sériovému rozvrhu transakcií v poradí, v ktorom sa transakcie validujú

Validácia: ilustrácia prípadov

Príklad:

Úspešná validácia T vzhľadom na T_c

$TS(T_c)$ ***TVAL(T_c)*** $TF(T_c)$

work	validate	update
------	----------	--------

$TS(T)$ ***TVAL(T)***

work

Validácia: ilustrácia prípadov

Príklad:

Ak $RS(T) \cap WS(T_c) \neq \emptyset$, tak validácia T vzhľadom na T_c je neúspešná
(lebo T mohla čítať iné hodnoty než tie od T_c)

$TS(T_c)$ $TVAL(T_c)$ $TF(T_c)$

work	validate	update
------	----------	--------

$TS(T)$ $TVAL(T)$

work

$TS(T)$ $TVAL(T)$

work

$TS(T)$ $TVAL(T)$

work

Validácia: ilustrácia prípadov

Príklad:

Ak $WS(T) \cap WS(T_c) \neq \emptyset$, tak validácia T vzhľadom na T_c je neúspešná
(lebo inak by konfliktné zápisy T a T_c mohli byť urobené v nesprávnom poradí)

$TS(T_c)$ $TVAL(T_c)$ $TF(T_c)$

work	validate	update
------	----------	--------

$TS(T)$ $TVAL(T)$

work	
------	--

$TS(T)$ $TVAL(T)$

work	
------	--

Multiversion concurrency control (MVCC)

Každá transakcia T dostane pri svojom štarte timestamp $TS(T)$

Každý **objekt** X (presnejšie, každá verzia X) má pečiatky X_r a X_w

Ked' transakcia T píše do X , tak vytvorí novú verziu objektu X , s $X_w = TS(T)$. Systém pamätá aspoň 2 posledné verzie každého objektu

Ked' transakcia T číta z X , scheduler „podhodí“ na čítanie poslednú takú verziu X , pre ktorú platí $X_w < TS(T)$. Ďalej, ak pre túto verziu platí $TS(T) > X_r$, tak do X_r sa priradí $TS(T)$

Pri zápise objektu X od transakcie T systém kontroluje, či existuje nejaká verzia X s $X_w < TS(T)$ a zároveň $X_r > TS(T)$. Ak existuje, tak systém abortuje transakciu T (lebo čítajúca transakcia mala prečítať až verziu od T)

Multiversion concurrency control (MVCC)

Postgres a Oracle používajú MVCC, ktorý sa dostal do módy (lebo systém musí tak či tak pamätať aspoň 2 posledné verzie objektu). Pozor na úskalia! Citát z manuálu PostgreSQL:

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.

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 programátorom obetovať ACID garancie v záujme urýchlenia aplikácií

H.Berenson, P.Bernstein, J.Gray, J.Melton, E.O'Neil, P.O'Neil, A Critique of ANSI SQL Isolation Levels, ACM SIGMOD'95

<http://research.microsoft.com/apps/pubs/default.aspx?id=69541>

Módny trend v IT reprezentuje rôznorodá skupina **NoSQL** (Not-Only-SQL) systémov, ktoré sa spravidla vyhýbajú použitiu relačného dátového modelu, resp. jazyka SQL. Majú skôr charakter distribuovaných file-systémov, špecializovaných na prácu s dátami typu „key-value“. Dôraz sa obvykle kladie na „agilitu“, konzistencia dát je druhoradá

<http://nosql-database.org/>

<http://www.igvita.com/2010/03/01/schema-free-mysql-vs-nosql/>

Sériovateľnosť nestačí. Príklad:

$r_1(A)$, $w_1(A)$, $r_2(A)$, $w_2(C)$, c_2 , c_1



Tento rozvrh je sériovateľný. Smie však scheduler vskutku vygenerovať takýto rozvrh? Predpokladajme, že nastane výpadok (abort) T_1 v momente, na ktorý ukazuje šípka. Transakcia T_2 už bola commitovaná, takže jej efekt bol už navždy zapísaný do databázy (efekt operácie $w_2(C)$ ostane v databáze aj po výpadku). Lenže transakcia T_1 , od ktorej T_2 čítala, bola abortovaná a efekt $w_1(A)$ bol zo systému navždy odstránený. Tým pádom transakcia T_2 mohla prečítať neplatnú hodnotu objektu A , od ktorej mohla závisieť zapísaná hodnota objektu C . Hrozila nekonzistencia! Len šťastnou náhodou aj T_1 commituje...

Cieľom recovery algoritmov je garantovať ACID aj v prípade neočakávaných výpadkov

Predpoklady:

- non-volatile fyzické médiá (disky, RAID) sú spoľahlivé—ak nie, treba siahnuť k záložnej kópii (backup)
- zápis do non-volatile médií je atomický aspoň na úrovni blokov (bloky majú konštantnú veľkosť, ale sú dostatočne veľké)
- transakcie sú vnútorne konzistentné
- výpadky je možné detekovať

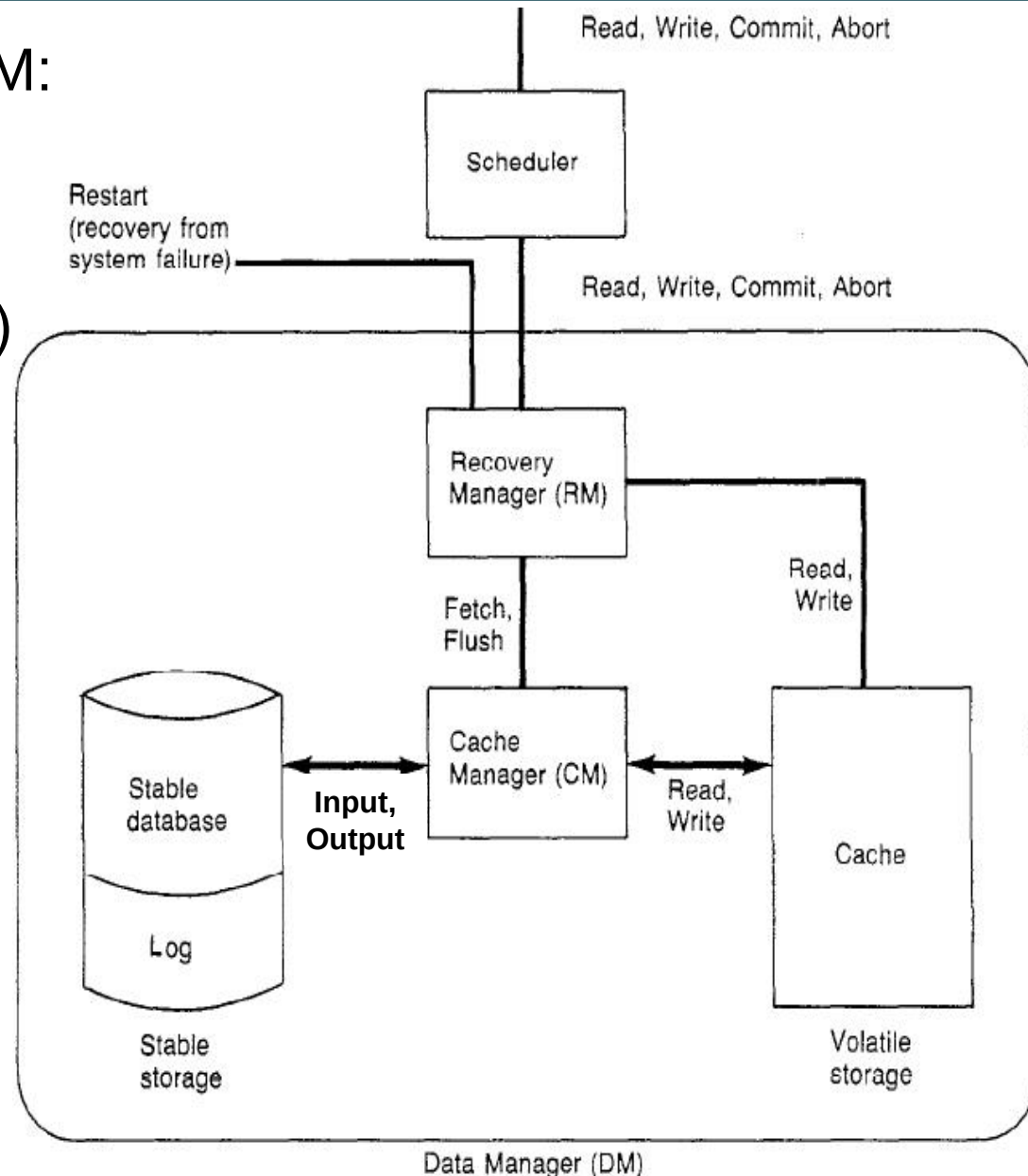
Recovery algoritmy (ARIES) sa skladajú z 2 častí:

1. Zbieranie informácií počas normálneho behu: **log**
2. Reakcia na výpadok: **idempotentné operácie UNDO (pre abortované transakcie) a REDO (pre commitované transakcie)**

Buffer management: Cache Manager

Synchronizácia disku a RAM:

- Prvý read(X) spôsobí čítanie X z disku (fetch→input), ďalší read(X) spôsobí čítanie z RAM (fetch→read)
- write(X) nemusí spôsobiť okamžité písanie na disk (flush→output), ale iba do RAM (flush→write)



Dôležitý technický detail: Blok v cache sa nesmie aktualizovať (a niekedy ani čítať) v momente, keď je zapisovaný na disk (output). Je potrebná synchronizácia medzi hardwarom (DMA) a softwarom (cache manager). Riešenie: **pred prístupom k bloku je potrebný exkluzívny zámok, ktorý stačí držať na veľmi krátku chvíľu.** Moderný hardware ponúka takéto zámky, hovorí sa im **latch**

Niektoré operácie vyžadujú okamžitý zápis bloku na disk. Ostatné bloky ostávajú v pamäti až kým sa pamäť nenaplní. Potom sa pre výmenu blokov používajú techniky známe z operačných systémov (napr. LRU, FIFO)

„Transakcia prebehne buď úplne, alebo vôbec“

Stačí sa sústrediť **iba na operácie write** (read nemení stav dát).

Dokonca (teoreticky), pre každý objekt sa stačí sústrediť na posledný write poslednej commitovanej transakcie

Dva prístupy k implementácii atomicity:

1. **Log**

2. Shadow-paging

Log-file je sekvenčný súbor uložený na disku, ktorý odráža zmeny spôsobené transakciami. Zmeny v log-file sa okamžite zapíšu na disk (output). Garancia atomicity diskových prístupov je dôležitá najmä pre log-file

Filozofia „**write-ahead**“: skôr ako transakcia urobí zmenu v dátach, zamýšľaná zmena sa zapíše do log-file

Obsah log-file:

- Začiatok transakcie T_i : $\langle T_i \text{ start} \rangle$
- Write-operácie transakcie T_i : $\langle T_i, X, \text{old_value}, \text{new_value} \rangle$
- Commit transakcie T_i : $\langle T_i \text{ commit} \rangle$
- atď. (napr. CHECKPOINT, DUMP)

Všeobecná, flexibilná metóda obnovy bez predpokladov na cache (possibly immediate data modification), normálny beh:

- Pred vykonaním operácie write(X) sa do log-file zapíšu stará aj nová hodnota objektu X
- Pri vykonávaní write(X) cache manager rozhodne o tom, kedy sa objekt X *skutočne* zapíše na disk, smie (ale nemusí) tak urobiť hoci aj hneď
- Po ABORTE niektorej transakcie sa hneď urobí UNDO tej transakcie (a REDO súbežných transakcií atď.)

Všeobecná metóda obnovy

- V prípade obnovy treba najprv robiť UNDO pre tie transakcie T_i , pre ktoré existuje záznam $\langle T_i \text{ start} \rangle$, ale neexistuje $\langle T_i \text{ commit} \rangle$. UNDO sa robí pri **zostupnom** čítaní log-file (od konca po začiatok). Počas UNDO prechodu systém zapisuje do databázy staré hodnoty objektov uložené v logu pri operáciách write; a vytvára 2 zoznamy identifikátorov transakcií: `undo_list` a `redo_list` (v `redo_list` sú commitované transakcie, v `undo_list` necommitované)
- Potom treba urobiť REDO pre tie transakcie v `redo_list`. REDO sa robí pri **vzostupnom** čítaní log-file (od začiatku po koniec)

Aj počas obnovy môže znovu nastať výpadok, ale to nevadí (vďaka idempotentnosti REDO / UNDO). Obnova sa jednoducho zopakuje pri opätovnom štarte systému

Príklad (Silberchatz et al.):

T0	T1
read (A)	read (C)
A = A – 50	C = C - 100
write (A, 950)	write (C)
read (B)	
B = B + 50	
write (B)	

<T ₀ start>	<T ₀ start>	<T ₀ start>
<T ₀ , A, 1000, 950>	<T ₀ , A, 1000, 950>	<T ₀ , A, 1000, 950>
<T ₀ , B, 2000, 2050>	<T ₀ , B, 2000, 2050>	<T ₀ , B, 2000, 2050>
	<T ₀ commit>	<T ₀ commit>
	<T ₁ start>	<T ₁ start>
	<T ₁ , C, 700, 600>	<T ₁ , C, 700, 600>
		<T ₁ commit>
(a)	(b)	(c)

- (a) UNDO(T₀): B:=2000, A:=1000
- (b) UNDO(T₁): C:=700,
REDO(T₀): A:=950, B:=2050
- (c) REDO(T₀): A:=950, B:=2050
REDO(T₁): C:=600

Oneskorený zápis dát (deferred data modification)

Oneskorený zápis dát (efektívna metóda, ale s predpokladmi na cache):

- write(X) sa síce zapíše do log-file aj do databázy, ale **nie na disk**
- do databázy na disku sa zapisuje **vždy** až niekedy po commit transakcie
- Do log-file netreba písať staré hodnoty dát (before-images)
- Operáciu UNDO netreba vôbec implementovať

Operáciu REDO treba vykonať len pre tie transakcie, pre ktoré existuje v log-file záznam $\langle T_i, \text{commit} \rangle$. Algoritmus REDO prechádza log-file raz, **zostupne** (toto je rozdiel oproti všeobecnej metóde). Pri tomto jedinom prechode sa vytvárajú dva zoznamy: redo_list a redone_list (s identifikátormi **objektov**). Systém pre každý záznam $\langle \text{commit } T_i \rangle$ pridá T_i do redo_list. Pre každý záznam $\langle T_i, X, \text{new_value} \rangle$ urobí systém toto:

```
if ( $T_i \in \text{redo\_list}$ ) AND ( $X \notin \text{redone\_list}$ ) then {  
  write(X, new_value);  
  redone_list = redone_list  $\cup$  X;  
}
```

Oneskorený zápis dát (deferred data modification)

Príklad (Silberchatz et al.):

T0	T1			
read (A)	read (C)	$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$
A = A – 50	C = C – 100	$\langle T_0, A, 950 \rangle$	$\langle T_0, A, 950 \rangle$	$\langle T_0, A, 950 \rangle$
write (A, 950)	write (C)	$\langle T_0, B, 2050 \rangle$	$\langle T_0, B, 2050 \rangle$	$\langle T_0, B, 2050 \rangle$
read (B)		$\langle T_0 \text{ commit} \rangle$	$\langle T_0 \text{ commit} \rangle$	$\langle T_0 \text{ commit} \rangle$
B := B + 50		$\langle T_1 \text{ start} \rangle$	$\langle T_1 \text{ start} \rangle$	$\langle T_1 \text{ start} \rangle$
write (B)		$\langle T_1, C, 600 \rangle$	$\langle T_1, C, 600 \rangle$	$\langle T_1 \text{ commit} \rangle$
		(a)	(b)	(c)

- (a) Žiadne REDO akcie
- (b) REDO(T_0) je potrebné, lebo log obsahuje $\langle T_0 \text{ commit} \rangle$: B:=2050, A:=950
- (c) REDO(T_1): C:=600
REDO(T_0): B:=2050, A:=950

Checkpointing

Checkpointing je technika, ktorá redukuje dĺžku log-files a skracuje čas potrebný na obnovu systému. Nezávisle od spracovania transakčných operácií sa dá kedykoľvek spustiť nasledujúca **atomická** procedúra:

1. Zapiš všetky log-záznamy na disk
2. Zapiš všetky dáta na disk
3. Zapiš <CHECKPOINT> do log-file (tento záznam sa niekedy dopĺňa o zoznam aktívnych transakcií a ďalšie informácie)
4. Skrát' log-file (zahod' časť pred záznamom <start T> najstaršej aktívnej transakcie T)

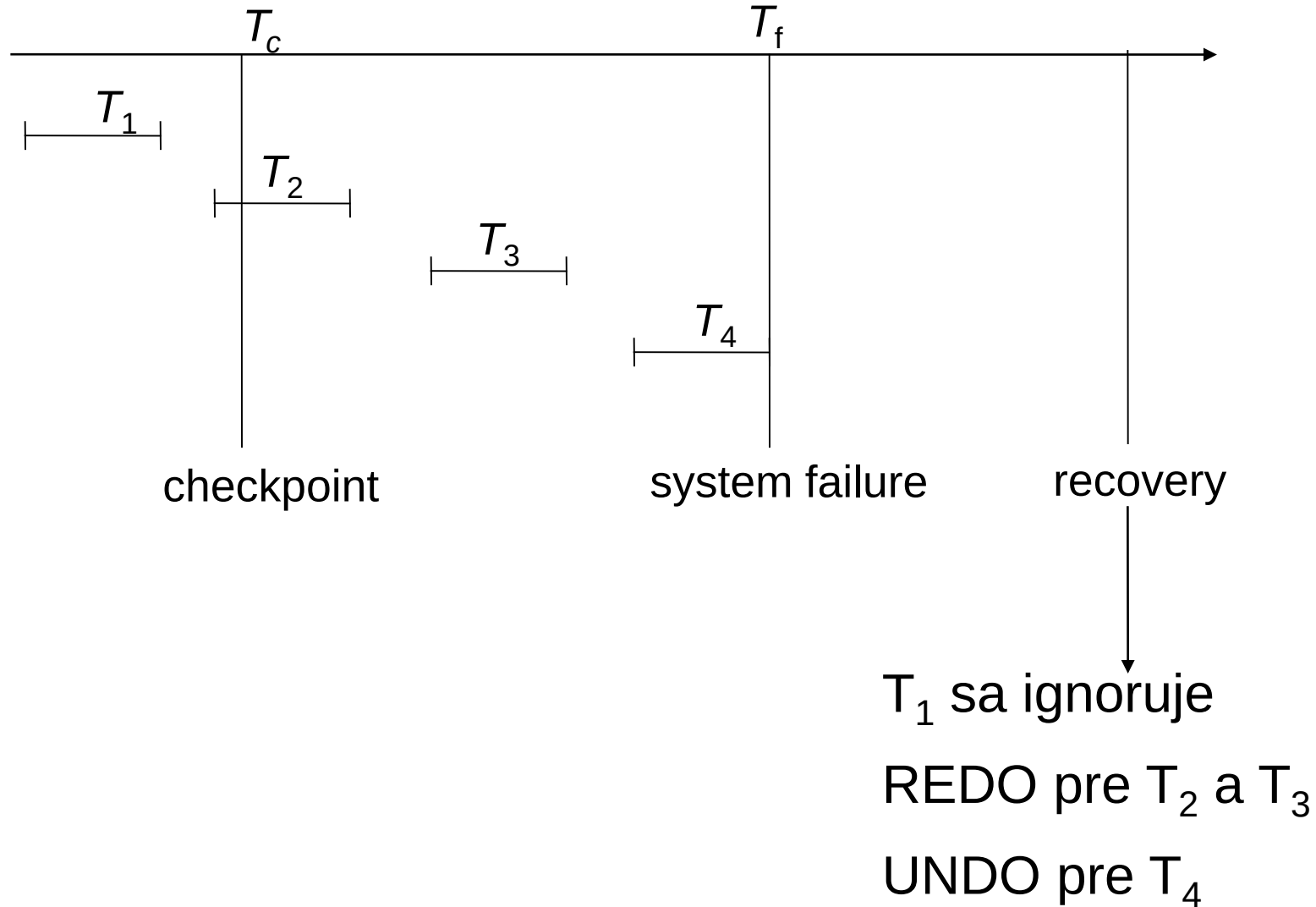
Existujú aj online verzie (on-the-fly checkpoint, incremental checkpoint), ktoré nevyžadujú prerušenie normálnej prevádzky. Efekt je taký, ako keby tá procedúra bola atomická

Checkpointing: recovery (všeobecný algoritmus)

0. Prechádzaj log-file **zostupne**, vytváraj pritom undo_list a redo_list a vykonávajú UNDO akcie
1. Keď narazíš na záznam <CHECKPOINT>, **pokračuj v zostupe** po najstarší záznam < T_i start> pre transakcie, ktoré boli aktívne v čase CHECKPOINT (t.j. zastav sa keď vyprázdniš UNDO zoznam resp. nájdeš začiatok logu), pritom vykonávajú UNDO akcie (ignoruj ďalšie záznamy <CHECKPOINT>)
2. Spusti **vzostupne** REDO-prechod, počínajúc posledným záznamom nájdeným v kroku 1

Časť log-file pred záznamom nájdeným v kroku 1 nie je pre obnovu dôležitá a možno ju zahodiť (garbage collection)

Checkpointing: príklad (Silberschatz et al.)



Backup je procedúra, ktorá zvyšuje odolnosť voči výpadkom médií. Nezávisle od spracovania transakčných operácií sa dá kedykoľvek spustiť nasledujúca **atomická** procedúra:

1. (Spusti procedúru CHECKPOINT, skráť log-file)
2. Vytvor kópiu log-file a kópiu databázy na novom médiu
3. Zapiš <DUMP> do log-file a zapiš log na disk

Existujú aj online verzie (on-the-fly backup, incremental backup), ktoré nevyžadujú prerušenie normálnej prevádzky. Efekt je taký, ako keby tá procedúra bola atomická

Definícia: Rozvrh je obnoviteľný (recoverable, RC) práve vtedy, keď pre každú commitovanú transakciu T2 čítajúcu necommitovanú hodnotu od T1 (dirty read) platí, že T1 commituje tiež—a zároveň commit T1 je v tom rozvrhu skôr ako commit T2

Motivácia je zrejmá. Napríklad R1: $w1(X) r2(X) w2(Y) c2 c1$ nie je obnoviteľný rozvrh, lebo po $c2$ mohol nasledovať abort transakcie T1. Ani toto nie je obnoviteľný rozvrh: R2: $w1(X) r2(X) w2(Y) c2 a1$

Rozvrh R3: $w1(X) r2(X) w2(Y) c1 c2$ je obnoviteľný. V rozvrhu R3 však hrozí iná nepríjemnosť: kaskádový abort (cascading abort). Ak sa v rozvrhu R2 transakcia T1 rozhodne pre abort, tak to spôsobí následný vynútený abort T2 (transakcia T2 nemôže už nikdy skončiť commitom, ak má byť ten rozvrh obnoviteľný)

Kaskádový abort môže byť vskutku „reťazová reakcia“, je lepšie sa mu vyhnúť:

T1 $r(X)$ $w(X)$ abort

T2 $r(X)$ $w(Y)$

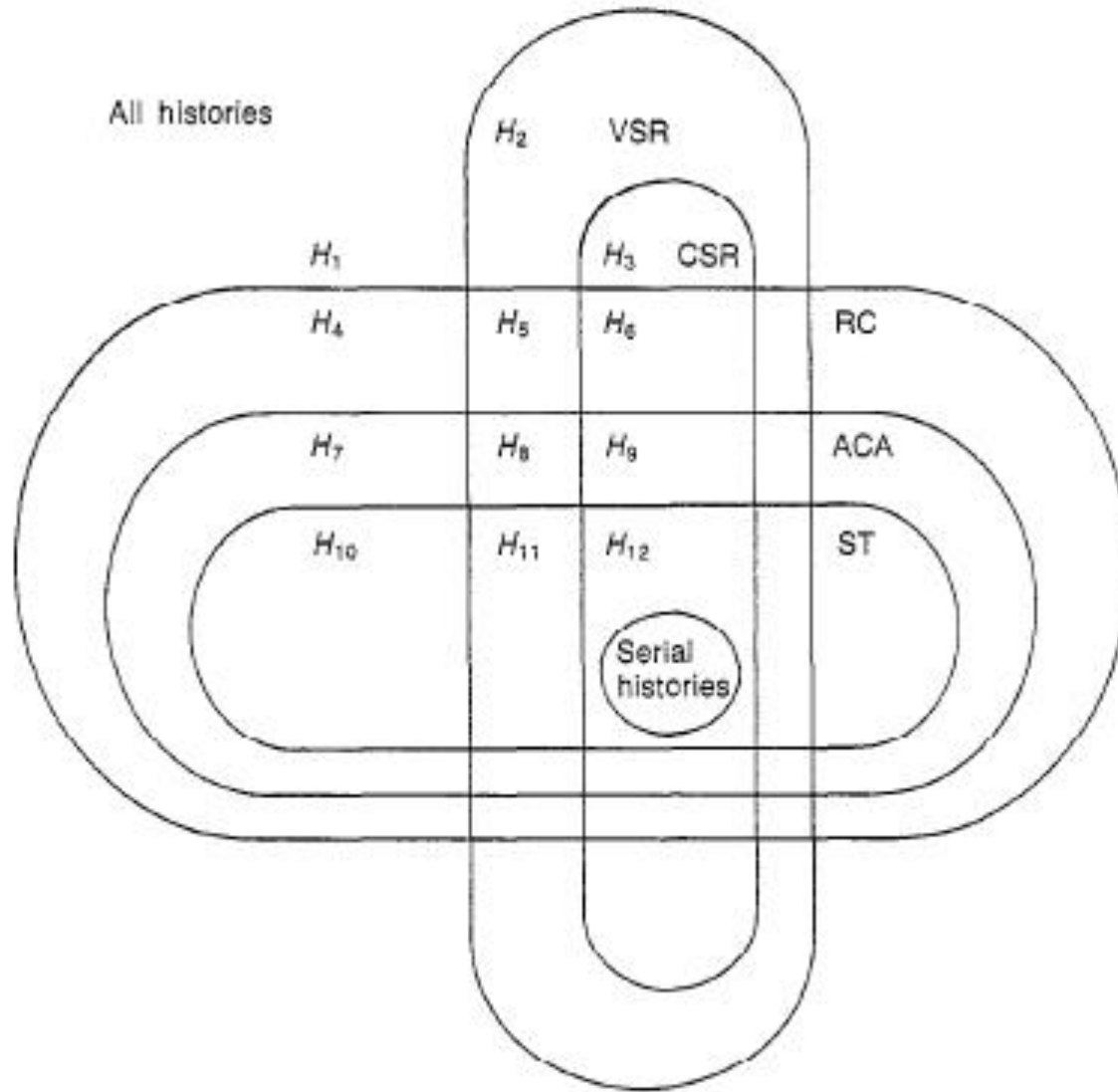
T3 $r(Y)$ $w(Z)$

Definícia: Rozvrh sa vyhýba kaskádovým abortom (avoids cascading aborts, ACA) práve vtedy, ak neobsahuje dirty read (dirty read je čítanie hodnoty zapísanej inou transakciou, ktorá v momente čítania nebola commitovaná)

Rozvrh $r_2(Y)$ $w_1(X)$ $w_2(X)$ a2 c1 sa vyhýba kaskádovým abortom, ale **v prípade výpadku vyžaduje recovery algoritmus dva prechody cez log-file** (ten druhý je kvôli REDO). Aj tomu sa možno oplatí vyhnúť. **Ak je garantované, že scheduler generuje len striktné rozvrhy, dá sa algoritmus obnovy urýchliť, resp. zjednodušiť**

Definícia: **Rozvrh je striktný (strict)** práve vtedy, ak neobsahuje dirty read ani dirty write
(dirty write je prepisovanie necommitovanej hodnoty zapísanej inou transakciou, ktorá v momente zápisu nebola commitovaná)

Dve ortogonálne hierarchie (Bernstein, Hadzilacos)



Minimálna praktická požiadavka:
 $VSR \cap RC$ (view-sériovateľné a obnoviteľné rozvrhy)

Rozumná praktická požiadavka:
 $CSR \cap ACA$, resp.
 $CSR \cap ST$ (konflikt-sériovateľné rozvrhy vyhýbajúce sa kaskádovým abortom, resp. striktné)

Ako generovať rozumné rozvrhy: Strict 2PL

Dvojfázový zamykací protokol garantuje sériovateľnosť, ale nie obnoviteľnosť

Príklad: $wl1(X)$, $w1(X)$, $ul1(X)$, $rl2(X)$, $r2(X)$, $ul2(X)$, $c2$, $c1$

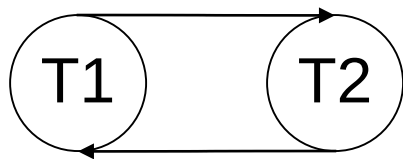
Strict 2PL je dvojfázový zamykací protokol, v ktorom je dovolené zámky uvoľňovať iba spolu s commitom (resp. s abortom). Striktný 2PL garantuje, že generovať sa budú len sériovateľné rozvrhy, ktoré sú zároveň striktné: $CSR \cap ST$

Deadlock

2PL aj striktný 2PL majú ešte jeden problém ktorý treba vyriešiť:
deadlock (uviaznutie)

Príklad: $wl1(X)$, $w1(X)$, $rl2(Y)$, $rl2(X)$, $wl1(Y)$

posledné dve operácie spôsobia, že T1 ani T2 nemôže ďalej pokračovať, lebo čakajú na zámky, ktoré nikdy nedostanú



Wait-for graph: T1 čaká na uvoľnenie zámku ktorý drží T2 a naopak

Veta. Transakcie (niektoré) sú v deadlocku práve vtedy, ak **wait-for graf** obsahuje cyklus. V deadlocku sú tie transakcie, ktoré sú v cykle

Optimistická stratégia: nevyhýbať sa deadlocku. Pravidelne spustiť detekciu cyklov vo WFG a rozbiť cykly abortovaním niektorých transakcií, resp. použiť timeout v žiadostiach o zámok (dobrovoľný abort transakcie, ktorá detekuje timeout pri žiadosti o zámok)

Pesimistická stratégia: systém udržiava wait-for graf (WFG). Ak práve vykonaná operácia pridala do WFG hranu, ktorá vyrobila cyklus, tak tú transakciu abortuje (alebo nejakú inú v tom cykle). (Dajú sa použiť tiež rôzne modifikácie bankárskeho algoritmu známeho z operačných systémov, napr. prostriedky sú očíslované a každá transakcia ich musí zamykať vo vzostupnom poradí.)

Riešenie deadlock (pesimistické stratégie)

Wait-die stratégia: každá transakcia T_i dostane pri svojom štarte timestamp $TS(T_i)$. Ak transakcia T_1 žiada o zámok, ktorý drží T_2 :

If $TS(T_1) < TS(T_2)$
then T_1 čaká kým T_2 neskončí
else abort T_1

Wound-wait (kill-wait) stratégia: každá transakcia T_i dostane pri svojom štarte timestamp $TS(T_i)$. Ak transakcia T_1 žiada o zámok, ktorý drží T_2 :

If $TS(T_1) < TS(T_2)$
then kill T_2 /* ak T_2 práve necommituje, tak bude abortovaná */
else T_1 čaká kým T_2 neskončí