Téma 11 – Transakce a řízení souběhu

Obsah

1. Transakce a jejich stavy
2. Souběh transakcí
3. Sériovost, serializovatelnost, obnovitelnost
4. Řízení souběhu
5. Úrovně konzistence
6. Řídicí protokoly se zámky
7. Dvoufázové zamykání
8. Grafové protokoly
9. Uváznutí transakcí, detekce, prevence a zotavení

SQL a vyhodnoceni SQL

- Prof. RNDr. J. Pokorný, CSc. – MFF UK
  http://www.ksi.mff.cuni.cz/~pokorny/vyuka/dj1/DJ1-SQL.pdf
  http://www.ksi.mff.cuni.cz/~pokorny/vyuka/dj1/DJ1-optimalizace.pdf
  http://www.ksi.mff.cuni.cz/~pokorny/vyuka/dj1/DJ1-vyhodnoceni.pdf
-

Pojem transakce

- Transakce je posloupnost akcí a operací, které se buď provedou všechny, nebo se neprovedou vůbec
  - Z abstraktního pohledu uživatelského programu na SŘBD, transakce je logicky svázaná série čtecích a zápisových operací
  - Každá transakce musí "vidět" databázi v konzistentním stavu
- Během provádění transakce může být databáze dočasně nekonzistentní
  - Musí být zaručeno, že sekvence celé řady akcí, z nichž se transakce skládá, proběhne atomicky jako celistvý krok práce programu
- Když transakce skončí úspěšně (tj. výsledky jsou zapamatovány), databáze musí být opět konzistentní
- Více transakcí může probíhat souběžně (paralelně)
- Hlavní dva problémy, jež je třeba řešit:
  - Selhání různých typů, např. hardwarové chyby, havárie OS
  - Souběh transakcí při vícenásobném přístupu

Vlastnosti "ACID"

- K zachování konzistence a integrity databáze transakční mechanismus musí zajistit:
  - Atomicitu (***A****tomicity*): Buď se správně zobrazí do databáze výsledky všech dílčích operací nebo žádné
  - Konzistence (***C****onsistency*): Transakce samotná neporuší konzistenci databáze
  - Izolovanost (***I****solation*): Přestože může běžet více transakcí paralelně, žádná z transakcí nesmí ovlivňovat jinou souběžně prováděnou transakci. Mezivýsledky transakce musí být skryté a ostatní transakce nesmí tyto mezivýsledky vidět
    - Tzn., pro každý pár transakcí T_i a T_j musí platit, že transakci T_i je jeví databáze jakoby transakce T_j již skončila nebo ještě nezačala
  - Trvalost (***D****urability*): Všechny změny dat, které transakce provedla, se po jejích úspěšném ukončení promítnou do databáze a již nemohou být ztraceny
    - ani při havárii systému

Příklad převodu peněz

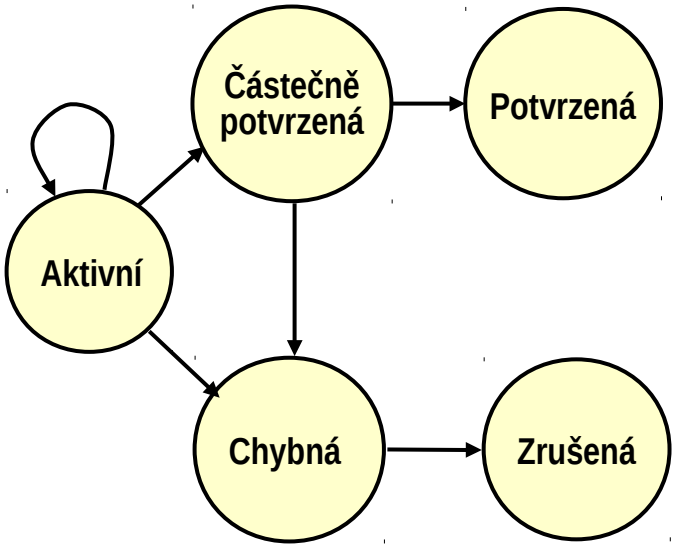
- Transakce k převodu 500 Kč z účtu A na B:

1. **read**(*A*)
2. *A* := *A* – 500
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 500
6. **write**(*B*)

  - Atomicita
    - Pokud transakce selže mezi kroky 3 a 6, systém MUSÍ zaručit, že v do dat se nepromítne žádná změna (jinak by se narušila konzistence)
  - Konzistence
    - Součet A a B se transakcí zachová.
  - Izolovanost
    - Jestliže mezi kroky 3 a 6 přistoupí jiná transakce k těmže účtům, nalezne je v nekonzistentním stavu (suma *A* + *B* bude menší, než by měla být)
      - Izolace lze snadno dosáhnout tak, že transakcím bude dovoleno běžet výhradně **sériově**, tj., jedna po druhé
      - Sériové provádění transakcí však výrazně snižuje průchodnost a efektivitu DBMS
  - Trvalost
    - Jakmile je uživatel (program volající databázový stroj) informován o tom, že 500 Kč bylo převedeno, tento fakt zůstane zachycen v databázi natrvalo

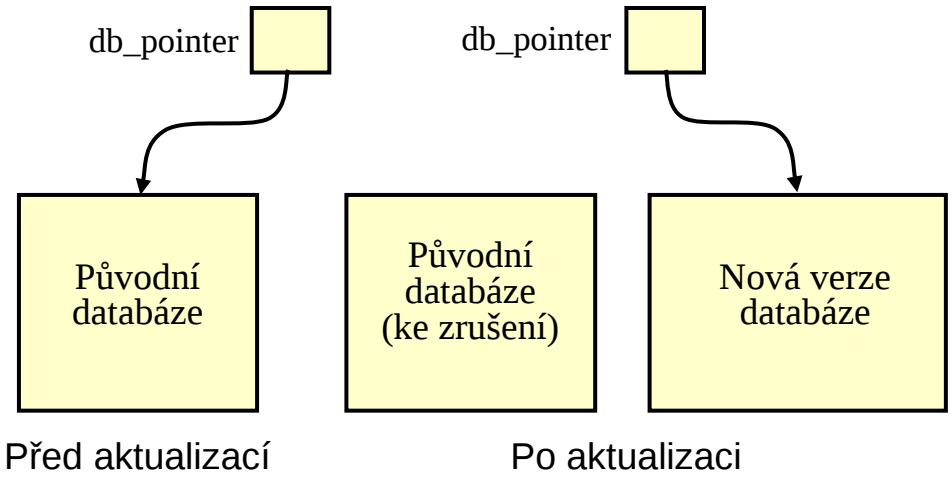
Stavy transakcí

- Aktivní
  - základní a počáteční stav; transakce se nachází v tomto stavu během svého vykonávání (běhu)
- Částečně potvrzená (*Partially committed*)
  - poté, kdy proběhla poslední dílčí operace
- Chybná (*Failed*)
  - poté, co se zjistí, že nelze pokračovat v normálním provádění
- Zrušená (*Aborted*)
  - stav, kdy transakce odstraní všechny uskutečněné změny a databáze je uvedena do stavu před zahájením transakce (rollback)
- Potvrzená (*Committed*)
  - po úspěšném ukončení, kdy všechny změny jsou natrvalo zapsány v databázi



Implementace atomicity a trvalosti

- Komponenta DBMS zvaná správce zotavení implementuje podporu atomicity a trvalosti
- Metoda stínové databáze:
  - Předpokládejme, že běží vždy jen jedna transakce
  - Ukazatel označovaný ***db_pointer*** vždy odkazuje na momentální konzistentní databázi
  - Všechny aktualizace se provádějí na *stínové kopii* databáze a *db_pointer* bude změněn teprve poté, co všechny akce se stínovou kopií úspěšně skončily a byly zobrazeny na disku
    - Pokud transakce zhavaruje, bude se nadále používat původní konzistentní kopie odkazovaná ukazatelem *db_pointer* a stínová kopie se smaže
  - Předpoklad: disk nezhavaruje
  - Extrémně neefektivní pro velké databáze (proč?)
  - Neřeší souběh transakcí
- Existují lepší schémata
  - Transakce pracují s lokálními kopiemi dat
  - Změna se do databáze zapíše až při úspěšném dokončení transakce



Zotavení z chyb

Klasifikace chybových stavů

- Chyby transakcí
  - Logické chyby: vlivem nějaké vnitřní logické podmínky nemůže transakce skončit úspěšně
  - Systémové chyby: DBMS musí přerušit aktivní transakci kvůli vzniku detekovaného chybového stavu (např. uváznutí)
- Havárie systému
  - Výpadek napájení nebo jiná chyba hardware či selhání podloženého operačního systému způsobí havárii
  - Předpoklad: obsah persistentní paměti nebude havárií narušen
    - DBMS dělají většinou celou řadu kontrolních operací s cílem zabránit poškození dat na disku
- Havárie disku:
  - Chyba či havárie diskové jednotky způsobí zničení části nebo dokonce celého obsahu disku
  - Předpoklad: Destrukce je detekovatelná
    - diskové jednotky používají kontrolní součty k detekci chyb

Algoritmy obnovy

- Metoda stínové databáze je velmi neefektivní
  - Potřeba lepších řešení
- Algoritmy obnovy (zotavení)
  - Způsoby a techniky, jak zajistit konzistenci databáze podmíněnou atomicitou transakcí a trvalostí jejich výsledků i v případě existence chyb a havárií
- Algoritmy obnovy mají dvě komponenty
  1. Akce prováděné během běžného zpracování transakcí, které poskytnou dostatek informací nutných k zotavení v případě selhání
  2. Akce, které po vzniku chybové situace spolehlivě dovedou databázi do stavu, který zaručí konzistenci, atomicitu transakcí a trvalost konzistentního stavu

Typy pamětí

- Volatilní paměť
  - obsah nepřežije havárii systému
  - Příklady: cache, operační (primární) paměť
- Persistentní paměť
  - obsah zůstane zachován, i když systém zhavaruje
  - Příklady: disk, mg. páska, flash paměť, CD-ROM
    baterií zálohovaná RAM
- Stabilní paměť
  - ideální (a neexistující) paměť, jejíž obsah přežije všechny chyby a havárie
  - Aproximuje se pomocí vícenásobných kopií dat ukládaných na různá persistentní media

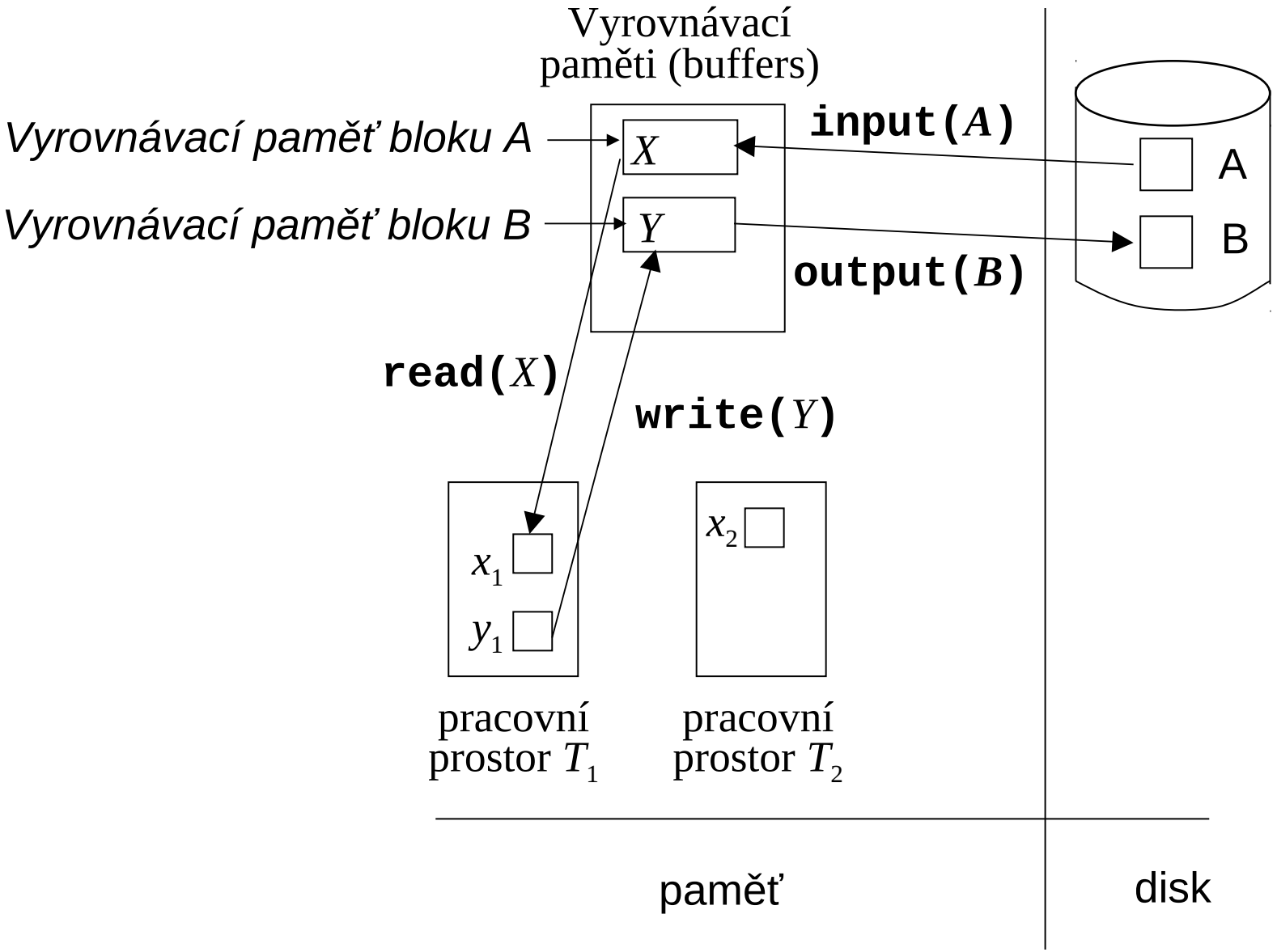
Přístup k datům

- Fyzické bloky jsou uloženy na persistentním disku
- Vyrovnávací paměti bloků jsou dočasné kopie fyzických bloků v hlavní (primární) paměti
- Přesuny bloků mezi diskem a pamětí jsou vyvolány operacemi
  - **`input(`***B***`)`** přenáší obsah fyzického bloku *B* do hlavní paměti
  - **`output(`***B***`)`** přenáší obsah vyrovnávací paměti bloku *B* na disk a přepíše obsah příslušného fyzického bloku
- Každá transakce T_i má svůj privátní pracovní prostor, v němž jsou lokální kopie datových položek, s nimiž transakce pracuje
  - Lokální kopii datové položky *X* patřící transakci T_i budeme značit x_i.
  - Pro jednoduchost (a lepší čitelnost) budeme předpokládat, že každá datová položka je uložena uvnitř jednoho bloku
    - Nepřechází tedy přes "hranici" bloku

Přístup k datům (pokr.)

- Transakce přenáší datové položky mezi systémovými vyrovnávacími pamětmi bloků a svým privátním pracovním prostorem pomocí operací
  - **read(*X*)** – přiřadí hodnotu položky *X* lokání proměnné x_i.
  - **write(*X*)** – zapíše hodnotu lokální proměnné x_i do datové položky {*X*} nacházející se ve vyrovnávací paměti bloku
  - obě tyto operace vyvolají potřebu operace **input(B_X)** před tím, než lze provést přiřazení, pokud blok B_X obsahující položku *X* není již v hlavní paměti
- Transakce
  - musí provést **read(*X*)** při prvním přístupu k *X*
  - Všechny další přístupy se pak dějí s lokální kopií
  - Po poslední modifikační manipulaci s položkou je nutno provést **write(*X*)**
- Ne každá operace **write(*X*)** vyvolá okamžitě **output(B_X)**.
  - Z důvodů efektivity odkládá systém fyzické zápisy **output** až do doby, kdy to považuje za vhodné

Příklad přístupu k datům



Zotavení a atomicita

- Každá transakce musí proběhnout buď úspěšně celá (**`commit`**) nebo se databáze musí vrátit zpět do konzistentního stavu
  - Transakce T_i převádějící peníze z účtu *A* na účet *B* musí převést celou částku (tj. uskutečnit všechny modifikace) nebo nic
  - K završení T_i bude nutno uskutečnit několik operací **`output`** (zde pro *A* a *B*). Chyba může nastat poté, kdy se první modifikace úspěšně zapsala na disk, avšak druhou už kvůli chybě nelze dokončit
  - K zaručení atomicity i v případě chyb se nejprve ukládají informace popisující předpokládané modifikace na "stabilní paměť" a teprve pak se provádějí vlastní modifikace databáze
- Ukážeme základní přístup zotavení s protokolem (*log-based recovery*)
  - Zpočátku předpokládejme, že transakce poběží sériově

Zotavení založené na protokolu

- Na "stabilní paměti" se vytváří a udržuje protokol (*log*)
  - Je to sekvence protokolových záznamů, která eviduje aktuální akce prováděné s databází (zejména zápisové operace)
- Protokol se buduje následovně:
  - Při startu se transakce T_i registruje záznamem $\langle T_i \text{ start} \rangle$
  - ***Před*** jakýmkoliv **`write`**(X) transakce T_i uloží záznam $\langle T_i, X, V_1, V_2 \rangle$, kde V_1 je hodnota X před **`write`** a V_2 je zapisovaná hodnota položky X
    - Je tedy zaprotokolována jak původní tak i nová hodnota položky X spolu s informací, že jde o akci T_i
  - Když T_i končí úspěšně, zaprotokoluje se záznam $\langle T_i \text{ commit} \rangle$
- Předpokládáme zatím, že transakce běží sériově a záznamy protokolu se zapisují přímo na stabilní paměť
  - nepřecházejí přes vyrovnávací paměť (*unbuffered*)
- Používají se dva přístupy založené na protokolování
  - Odložená modifikace databáze
  - Okamžitá (průběžná) modifikace databáze

Odložená modifikace databáze

- Princip metody odložené modifikace databáze je v tom, že se modifikuje jen protokol a všechny operace **`write`** jsou odloženy
  - Transakce začíná zápisem záznamu $\langle T_i \text{ start} \rangle$ do protokolu
  - Operace **`write`**(X) zapíše záznam $\langle T_i, X, V \rangle$, kde V je nová hodnota X
    - Původní hodnotu při odložené modifikaci nepotřebujeme
    - Vlastní modifikace dat se zatím nedělá
  - Když T_i úspěšně ukončí svoji poslední datovou operaci, dostane se do stavu částečně potvrzená (*partially commited*) ➔ a zapíše do protokolu $\langle T_i \text{ commit} \rangle$
- Na závěr se prochází protokol a všechny zaznamenané akce týkající se T_i jsou interpretovány
  - proběhnou všechny odložené operace **`write`**

Odložená modifikace databáze (pokr.)

- Při zotavení po havárii je nutno modifikace vytvořené transakcí opakovat právě tehdy, jsou-li v protokolu oba záznamy $\langle T_i \text{ start} \rangle$ a $\langle T_i \text{ commit} \rangle$
  - Opakování transakce T_i (**redo** T_i) nastavuje hodnoty všech datových položek na hodnoty nové
- Havárie mohou nastat
  - při původním zapisování nových hodnot, nebo
  - při opakování těchto zápisů

- Příklad:
  - Nechť T_0 běží před T_1

T_0:	T_1:
read(A)	**read**(C)
$A := A - 50$	$C := C - 100$
write(A)	**write**(C)
read(B)	
$B := B + 50$	
write(B)	

Odložená modifikace databáze (pokr. 2)

- Stav protokolu ve třech okamžicích

(a)	(b)	(c)
$\langle T_0$ start$\rangle$	$\langle T_0$ start$\rangle$	$\langle T_0$ start$\rangle$
$\langle T_0, A, 950\rangle$	$\langle T_0, A, 950\rangle$	$\langle T_0, A, 950\rangle$
$\langle T_0, B, 2050\rangle$	$\langle T_0, B, 2050\rangle$	$\langle T_0, B, 2050\rangle$
	$\langle T_0$ commit$\rangle$	$\langle T_0$ commit$\rangle$
	$\langle T_1$ start$\rangle$	$\langle T_1$ start$\rangle$
	$\langle T_1, C, 600\rangle$	$\langle T_1, C, 600\rangle$
		$\langle T_1$ commit$\rangle$

- Dojde-li k havárii v čase

(a) Nemusí se dělat nic (chybí $\langle T_0$ commit$\rangle$ – nic není ani částečně potvrzeno, žádná změna dat se neuskutečnila)

(b) Je nutno udělat **`redo`**(T_0), neboť je zaznamenáno $\langle T_0$ commit$\rangle$

(c) Je nutno udělat **`redo`**(T_0) následované **`redo`**(T_1), neboť v protokolu je jak $\langle T_0$ commit$\rangle$ tak i $\langle T_1$ commit$\rangle$

Okamžitá modifikace databáze

- Metoda okamžité modifikace databáze umožňuje, aby aktualizace databáze probíhala souběžně s operacemi **`write`**
  - Protože přichází v úvahu anulování (*undo*) všech provedených zápisů, protokol musí obsahovat jak nové tak i původní hodnoty
  - Záznam o aktualizaci položky musí být zapsán do protokolu *před* zápisem do databáze
    - Opět předpokládáme, že záznamy protokolu se zapisují přímo na stabilní paměť
  - Zápisy to protokolu lze dokonce odsunout až do okamžiku, kdy se provádí **`output`**(*B*), kde *B* datový blok se zapisovanou položkou. Před **`output`**(*B*) však musí být protokol se všemi změnami hodnot v *B* na stabilní paměti
  - Skutečný zápis aktualizovaných bloků lze však provést kdykoliv po bezpečném uložení protokolu, a to i před úspěšným ukončením transakce
  - Pořadí, v jakém se bloky vypisují na disk, se může lišit od pořadí operací **`write`**
- Okamžitá modifikace databáze může být rychlejší

Okamžitá modifikace databáze - příklad

T_0: **read**(A)
$A := A - 50$
write(A)
read(B)
$B := B + 50$
write(B)

T_1: **read**(C)
$C := C - 100$
write(C)

Protokol	**Write**	**Output**
⟨ T_0 start ⟩		
⟨ T_0, A, 1000, 950 ⟩		
⟨ T_0, B, 2000, 2050 ⟩		
	A = 950	
	B = 2050	
⟨ T_0 commit ⟩		
⟨ T_1 start ⟩		
⟨ T_1, C, 700, 600 ⟩		
	C = 600	
		B_B, B_C
⟨ T_1 commit ⟩		
		B_A

– B_X značí blok obsahující X

Okamžitá modifikace databáze (pokr.)

- Procedura zotavení je složitější a potřebuje dvě operace
  - **undo**(T_i) – anulování výsledků transakce T_i obnoví původní hodnoty všech datových položek modifikovaných T_i
    - zpětným průchodem protokolu lze zjistit, co kdy bylo modifikováno a jaké byly původní hodnoty
  - **`redo`**(T_i) nastaví nové hodnoty všech položek modifikovaných T_i podle protokolu počínaje prvním záznamem pro transakci T_i
- Obě operace musí být idempotentní
  - Tj., i když se operace provede několikrát, výsledek musí být stejný, jakoby proběhla právě jednou
    - Nutné, protože operace mohou být během obnovy restartovány
- Obnova
  - Transakce T_i musí být anulována (**undo**), pokud protokol obsahuje záznam $\langle T_i \text{ start}\rangle$, avšak neobsahuje $\langle T_i \text{ commit}\rangle$
  - Transakce T_i musí být opakována (**`redo`**), pokud protokol obsahuje jak záznam $\langle T_i \text{ start}\rangle$ tak i $\langle T_i \text{ commit}\rangle$
  - Napřed se dělají všechny operace **undo** a pak všechny **`redo`**

Obnova při okamžité modifikaci - příklad

- Stav protokolu ve třech okamžicích

(a)	(b)	(c)
$\langle T_0$ start$\rangle$	$\langle T_0$ start$\rangle$	$\langle T_0$ start$\rangle$
$\langle T_0, A, 1000, 950\rangle$	$\langle T_0, A, 1000, 950\rangle$	$\langle T_0, A, 1000, 950\rangle$
$\langle T_0, B, 2000, 2050\rangle$	$\langle T_0, B, 2000, 2050\rangle$	$\langle T_0, B, 2000, 2050\rangle$
	$\langle T_0$ commit$\rangle$	$\langle T_0$ commit$\rangle$
	$\langle T_1$ start$\rangle$	$\langle T_1$ start$\rangle$
	$\langle T_1, C, 500, 600\rangle$	$\langle T_1, C, 700, 600\rangle$
		$\langle T_1$ commit$\rangle$

- Dojde-li k havárii v čase
  (a) **undo**(T_0) vrátí A na 1000 a B na 2000
  (b) **undo**(T_1) obnoví C na 700 a **redo**(T_0) nastaví A na 950 a B na 2050
  (c) **redo**(T_0) nastaví A na 950 a B na 2050 a **redo**(T_1) nastaví C na 600

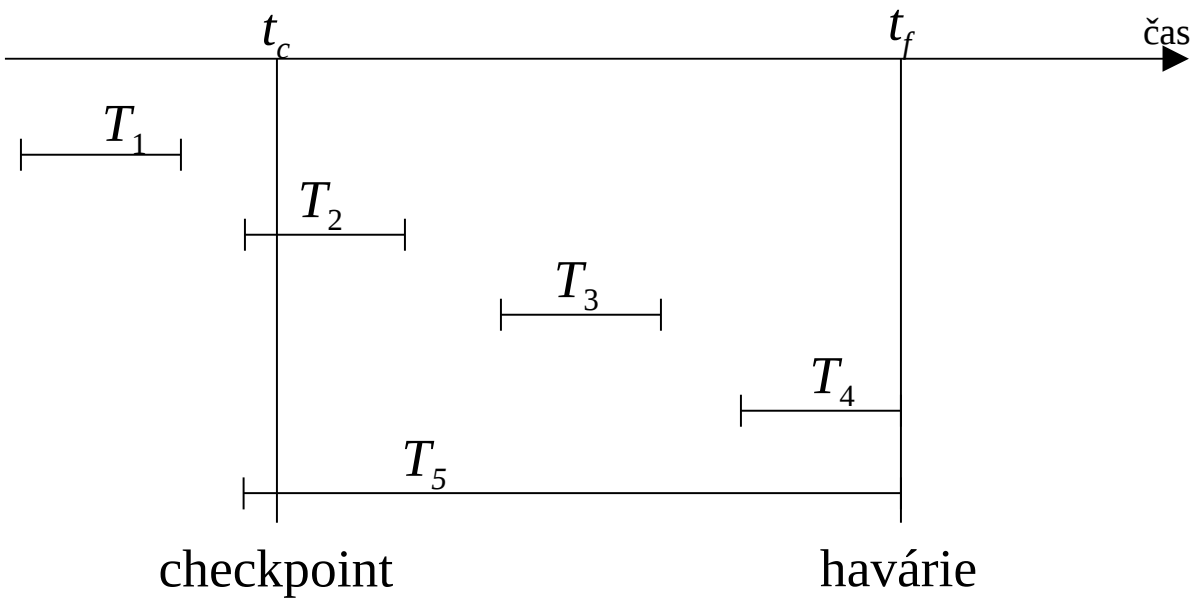
Kontrolní body - *checkpointing*

- Problémy s obnovou
  1. Prohledávání celého protokolu je časově náročné
  2. Zbytečné opakování transakcí (**`redo`**), jejichž výsledky jsou již uloženy v databázi
- Efektivnějšího zotavování lze dosáhnout periodickým voláním metody zvané checkpointing
  1. Vypiš (*flush*) všechny protokolové záznamy uložené v operační paměti na stabilní paměť
  2. Vypiš všechny vyrovnávací paměti bloků na disk
  3. Když se vše podaří, ulož do protokolu, jako kontrolní bod, záznam ⟨checkpoint⟩ a vypiš protokol na stabilní paměť
- Při zotavení se lze opírat o polohu záznamu ⟨checkpoint⟩ a starší akce lze ignorovat

Kontrolní body – *checkpointing* (pokr.)

- Stačí uvažovat transakci T_i, která začala jako poslední před kontrolním bodem, a všechny transakce, které začaly po T_i
  1. Prohlížej protokol zpětně od konce až do nalezení posledního záznamu ⟨checkpoint⟩
  2. Pokračuj ve zpětném hledání do nalezení záznamu ⟨T_k start⟩
  3. Nyní stačí uvažovat jen část protokolu, která následuje po ⟨T_k start⟩. Starší záznamy lze ignorovat či dokonce vymazat
  4. Pro transakci T_k a všechny transakce, které začaly po T_k a které nemají v protokolu záznam ⟨T_i commit⟩, proveď **undo**(T_i)
     - Má smysl jen při okamžité modifikaci databáze
  5. Prohledávej protokol od T_k směrem vpřed a pro transakci T_k a všechny transakce, které začaly po ní a mají v protokolu záznam ⟨T_i commit⟩, proveď **redo**(T_i).

Checkpointing – příklad



- T_1 bude ignorována
  - aktualizace jsou již uloženy na disku, o čemž svědčí exitence kontrolního bodu
- S T_2 a T_3 se musí provést **`redo`**
- Transakce T_4 a T_5 musí být anulována (**undo**)

Zotavení při souběžných transakcích

- Zotavení podle protokolu lze použít i pro případ souběžných transakcí
  - Všechny transakce sdílí společnou vyrovnávací paměť diskových bloků a jediný protokol
  - Obsah vyrovnávací paměti jednoho bloku může být modifikován jednou či více transakcemi
- Uvažujme řízení souběhu striktním dvoufázovým zamykáním
  - tj. ostatní transakce nesmí vidět aktualizace provedené dosud nepotvrzenými transakcemi
    - Kdyby tomu tak nebylo, pak situace, kdy T_1 změní A, pak T_2 změní A a potvrdí úspěch, a poté T_1 zhavaruje, by byla neřešitelná
- Protokol
  - záznamy různých transakcí budou v protokolu promíchány
- Technika kontrolních bodů se ale musí upravit
  - několik transakcí může být v běhu v okamžiku, kdy vzniká kontrolní bod

Zotavení při souběžných transakcích (pokr.)

- Kontrolní body
  - Ukládání kontrolních bodů do protokolu je shodné s předchozím případem, avšak záznam má tvar ⟨checkpoint L⟩, kde L seznam transakcí, které byly aktivní v okamžiku, kdy se kontrolní bod vytvářel
    - Předpokládáme, že při vypisování protokolu na stabilní paměť a vyrovnávacích pamětí bloků na disk jsou aktualizace pozastaveny (později tento předpoklad oslabíme)
- Když se systém zotavuje z havárie
  1. Utvoř prázdné seznamy *undo-list* a *redo-list*
  2. Procházej protokol od konce zpětně až po první záznam ⟨checkpoint L⟩. Pro každý záznam nalezený během zpětného průchodu:
     - je-li to ⟨T_i commit⟩, přidej T_i k seznamu *redo-list*
     - je-li to ⟨T_i start⟩ a přitom T_i není v seznamu *redo-list*, přidej T_i do seznamu *undo-list*
  3. Pro všechny transakce T_i v L testuj, zda T_i již je v seznamu *redo-list*, pokud ne, tak přidej T_i do *undo-list*

Zotavení při souběžných transakcích (pokr.)

- Nyní *undo-list* obsahuje seznam všech nekompletních transakcí, které musí být anulovány (**undo**), a *redo-list* všechny hotové transakce, které je nutno zopakovat (**redo**)
- Zotavení pokračuje následovně:
  1. Pokračuj dále ve zpětném procházení protokolu, dokud nenajdeš záznam $\langle T_i$ start$\rangle$ a to pro všechny T_i v *undo-list*u.
     Při procházení proveď **undo** transakcí nacházejících se v *undo-list*u
  2. Přesuň se na poslední záznam $\langle$checkpoint $L\rangle$
  3. Procházej protokol směrem ke konci
     Při procházení hledej záznamy transakcí nacházejících se v *redo-list*u a pro každou z nich proveď **redo**

Vyrovnávací paměti protokolu

- Pro zefektivnění tvorby protokolu potřebujeme vyrovnávací paměť
  - Protokolovací záznamy se tvoří v bloku v hlavní paměti místo přímého zápisu na stabilní paměť. Záznamy se vypíší, až když je blok zaplněn nebo při akci **`log force`**. **`Log force`** se provádí jako součást **`commit`** transakce, aby se všechny informace o transakci uložily na stabilní paměť
  - Takto se přenese najednou několik záznamů protokolu a tím se redukuje počet I/O operací
- K zabezpečení protokolu s vyrovnávací pamětí je nutno dodržet několik pravidel
  - Protokolovací záznamy jsou přenášeny na stabilní paměť v pořadí, jak byly vytvářeny
  - Transakce T_i přejde do stavu "Potvrzená" (*commited*) pouze když záznam $\langle T_i \text{commit} \rangle$ je bezpečně zapsán na stabilní paměť
  - Před tím, než začne být vyrovnávací paměť datového bloku zapisována na disk, všechny záznamy v protokolu, týkající se tohoto bloku, musí být uloženy na stabilní paměti
    - Pravidlo ***write-ahead logging***

Vyrovnávací paměti při okamžité modifikaci

- DBMS udržuje v primární paměti sadu vyrovnávacích pamětí na datové bloky (*database buffer*)
  - Když je třeba nový blok a sada je plná, je nutno uvolnit paměť dosud obsazenou jiným blokem (oběť)
  - Byl-li obsah bloku modifikován, je nutno ho uložit na disk
  - Obsahuje-li takový blok nepotvrzené aktualizace, pak všechny záznamy v protokolu týkající se tohoto bloku musí být na stabilní paměti dříve, než se začne příslušný blok vypisovat na disk – *write-ahead logging*
  - Když je blok ukládán na disk, nelze připustit žádné aktualizační operace s tímto blokem. To lze zajistit následovně:
    - Transakce musí získat výlučný zámek (*X*-lock) k bloku obsahujícímu modifikovanou datovou položku ještě před zápisem datové položky
    - Zámek lze uvolnit, jakmile je modifikace provedena
      - Blok je uzamčen velmi krátkou dobu
    - Systém musí získat výlučný zámek vyrovnávacího bloku před pokusem o jeho zápis na disk

Vyrovnávací paměti při odložené modifikaci

- Oproti předchozímu případu, není možné modifikovaný blok uložit do databáze, to se provede až na závěr transakce
- Používá se metoda stínového stránkování
  - Na začátku se vytvoří dvě pomocné tabulky stránek (bloků) dat obsahující polohu dat na disku – aktuální a stínová
  - Byl-li obsah bloku modifikován a je nutno ho uložit na disk (není volná paměť), zapíše se blok dat na volné místo na disku a v aktuální tabulce se změní jeho poloha na nové místo (stínová tabulka obsahuje polohu originálních dat)
  - Skončí-li transakce bezchybně, pak platí aktuální tabulka a místo ze stínové tabulky, které bylo přesunuto se označí za volné
  - Skončí-li transakce poruchou, pak platí stínová tabulka, uvolní se nově alokované bloky z aktuální tabulky.

Správa vyrovnávacích pamětí

- Sada vyrovnávacích pamětí v reálné paměti v oblasti vyhrazené databázi
  - Paměť je fixně rozdělena předem na vyrovnávací paměť a prostor pro ostatní aplikace (včetně databázového stroje), což omezuje pružnou práci. Požadavky se v čase mění, ale zde je vše rozděleno dopředu.
- Běžně se vyrovnávací paměti umisťují do virtuální paměti
  - Nevýhoda:
    - Když operační systém potřebuje obětovat stránku a ta byla modifikována, musí ji napřed vykopírovat do odkládacího prostoru na disku
    - Když se DBMS rozhodne, že vyrovnávací paměť bloku je třeba vypsat na disk a stránka s tímto blokem je odložena, musí ji OS napřed zavést do paměti, aby ji DBMS mohl opět uložit na disk (ale jinam). To vede ke zbytečným I/O přenosům.
  - V ideálním případě, potřebuje-li OS obětovat stránku s vyrovnávací pamětí, měl by předat řízení databázi, která by měla
    1. Vypsat stránku do databáze místo do odkládacího prostoru (samozřejmě za dodržení pravidla ***write-ahead logging***)
    2. Uvolnit stránku pro potřeby OS

    Bohužel běžné OS takovou funkcionalitu nepodporují

Paralelní zpracování transakcí

Souběh transakcí, rozvrhování akcí

- Souběžné transakce
  - zvyšují průchodnost systému
  - redukují průměrný čas odpovědi
    - krátké transakce čekají na dokončení dlouhých (viz konvojový efekt 🡐)
- Transakce se obvykle implementují jako samostatné procesy nebo vlákna
- Rozvrh (plán) operací v transakcích je posloupnost udávající chronologické pořadí operací v souběžně prováděných transakcích
  - rozvrh musí zahrnout všechny operace všech souběžných transakcí
  - musí zachovat pořadí, v němž jsou prováděny operace každé jednotlivé transakce
- Transakce, která skončila bezchybně, provede na závěr operaci **`commit`**, kterou potvrdí svůj úspěch
- Transakce, která zhavaruje, bude volat operaci **`abort`**, která značí potřebu obnovy původního stavu databáze

Příklady korektních rozvrhů

- Nechť T_1 převádí 50 z účtu A na B, a T_2 převádí 10% zůstatku z A na B
  - S_1 a S_2 jsou sériové rozvrhy
  - S_3 není sériový, ale je *ekvivalentní* sériovému S_1 či S_2

Rozvrh S_3

T_1	T_2
read(*A*)	
A := *A* – 50	
write(*A*)	
	read(*A*)
	tmp := A*0.1
	A := *A* – *tmp*
	write(*A*)
read(*B*)	
B := *B* + 50	
write(*B*)	
	read(*B*)
	B := *B* + *tmp*
	write(*B*)

Rozvrh S_1

T_1	T_2
read(*A*)	
A := *A* – 50	
write(*A*)	
read(*B*)	
B := *B* + 50	
write(*B*)	
	read(*A*)
	tmp := A * 0.1
	A := *A* – *tmp*
	write(*A*)
	read(*B*)
	B := *B* + *tmp*
	write(*B*)

$T_1 \prec T_2$

Rozvrh S_2

T_1	T_2
	read(*A*)
	tmp := A * 0.1
	A := *A* – *tmp*
	write(*A*)
	read(*B*)
	B := *B* + *tmp*
	write(*B*)
read(*A*)	
A := *A* – 50	
write(*A*)	
read(*B*)	
B := *B* + 50	
write(*B*)	

$T_2 \prec T_1$

Všechny uvedené rozvrhy zachovávají ($A + B$)

Chybný rozvrh

- Rozvrh S_4 nezachovává hodnotu (A + B) a tím porušuje požadavek konzistence

Rozvrh S_4

T_1	T_2
read(A)	
A := A – 50	
	read(A)
	tmp := A * 0.1
	A := A – tmp
	write(A)
	read(B)
write(A)	
read(B)	
B := B + 50	
write(B)	
	B := B + tmp
	write(B)

Serializovatelnost (uspořádatelnost)

- Základní předpoklad – každá jednotlivá transakce zachovává konzistenci databáze
  - Sériové vykonávání transakcí tedy zachovává konzistenci též
- Rozvrh (uvažující možný souběh) je serializovatelný, je-li ekvivalentní některému sériovému rozvrhu
  - Budeme uvažovat pouze operace **read** a **write** (služby OS) a budeme ignorovat všechny ostatní
    - Transakce mohou s daty provádět libovolné další operace ve svých lokálních proměnných a vyrovnávacích pamětech mezi read a write
  - "Naše" rozvrhy budou tvořeny jen operacemi **read** a **write**

Vzájemně neslučitelné operace

- Operace (instrukce) I_i a I_j patřící k transakcím T_i a T_j jsou vzájemně neslučitelné (konfliktní) právě tehdy, když existuje položka (záznam) Q, k níž přistupují I_i i I_j, a aspoň jedna z těchto operací do Q zapisuje

1.	I_i = **read**(Q),	I_j = **read**(Q)	nekonfliktní
2.	I_i = **read**(Q),	I_j = **write**(Q)	konfliktní
3.	I_i = **write**(Q),	I_j = **read**(Q)	konfliktní
4.	I_i = **write**(Q),	I_j = **write**(Q)	konfliktní

- Konflikt mezi I_i a I_j si vynucuje časové uspořádání operací I_i a I_j.
  - Pokud I_i a I_j následují v nějakém rozvrhu a nejsou ve vzájemném konfliktu, pak výsledek bude stejný, i když budou vzájemně prohozeny

Serializovatelný rozvrh

- Jestliže rozvrh S lze transformovat na rozvrh $S´$ sérií záměn nekonfliktních instrukcí, pak se S a $S´$ označují za ekvivalentní vůči konfliktu
- Rozvrh pak je serializovatelný, je-li ekvivalentní vůči konfliktu se sériovým rozvrhem
  - Rozvrh S_3 lze převést na sériový S_6, kde $T_1 \prec T_2$ sérií vzájemných prohození nekonfliktních operací
    - S_3 je tedy serializovatelný rozvrh
  - Rozvrh S_5 serializovatelný není
    - Nelze najít posloupnost vhodných prohození nekonfliktních operací

Rozvrh S_3

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Rozvrh S_6

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

aby vznikl sériový rozvrh $T_3 \prec T_4$ nebo $T_3 \prec T_4$

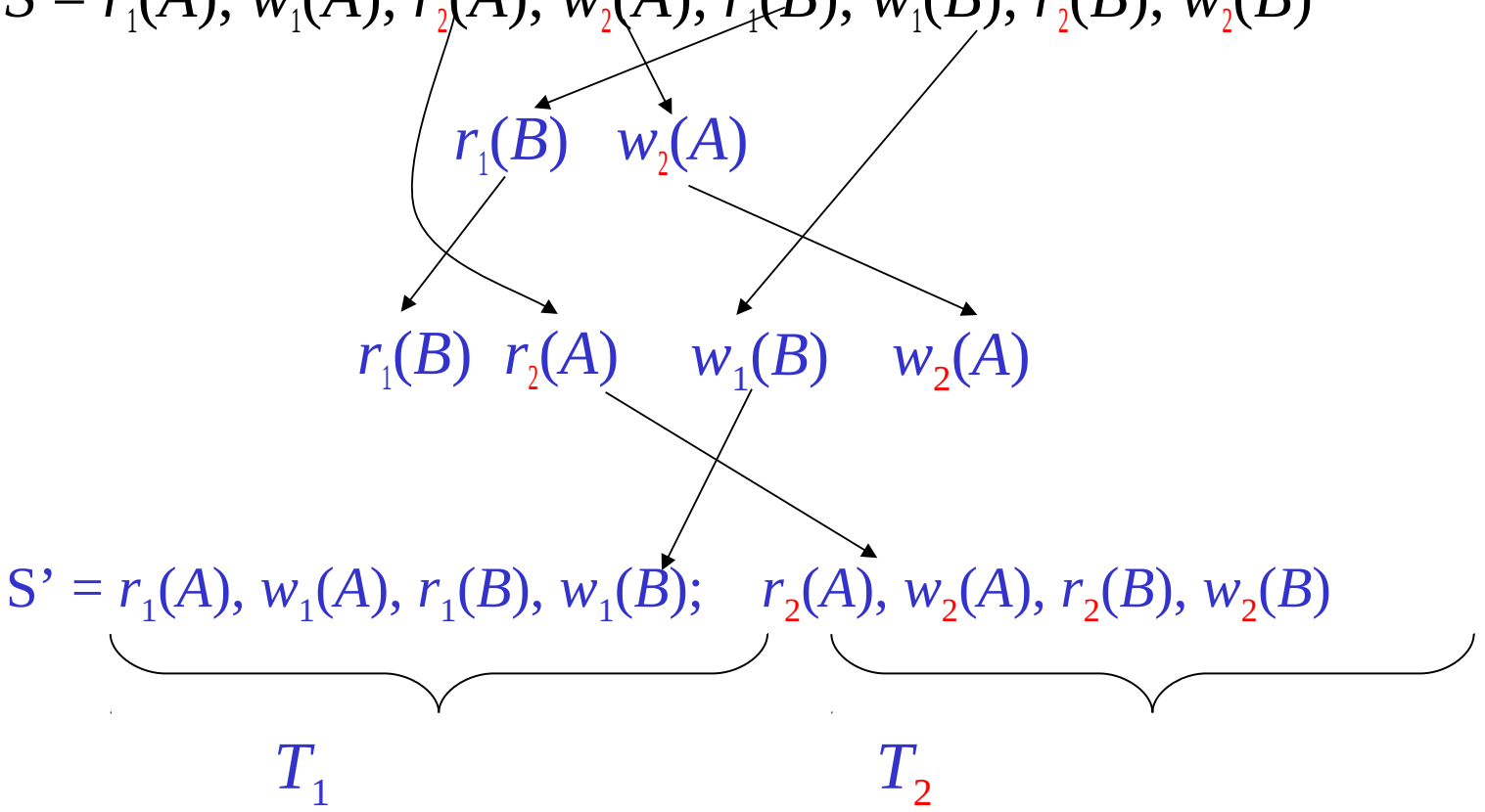
Rozvrh S_5

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

Příklad serializace

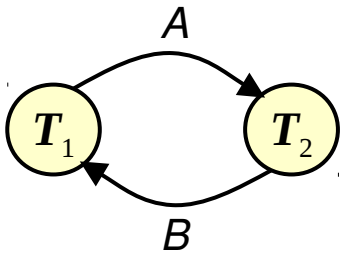
- Příklad:
  - r_1, w_1 – operace transakce 1, r_2, w_2 – operace transakce 2

$S = r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)$

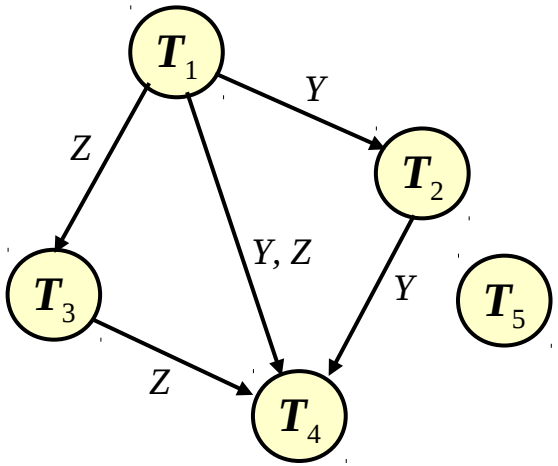


Test serializovatelnosti

- Uvažme rozvrh jisté množiny transakcí T_1, T_2, ..., T_n
- Precedenční graf je orientovaný graf, kde
  - vrcholy jsou transakce a
  - hrana vede z T_i do T_j, když T_i a T_j jsou v konfliktu a transakce T_i přistoupila ke konfliktním datům dříve
  - Hrany se obvykle označují datovou položkou, která konflikt způsobila
- Příklad:

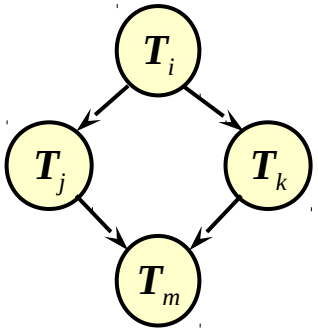


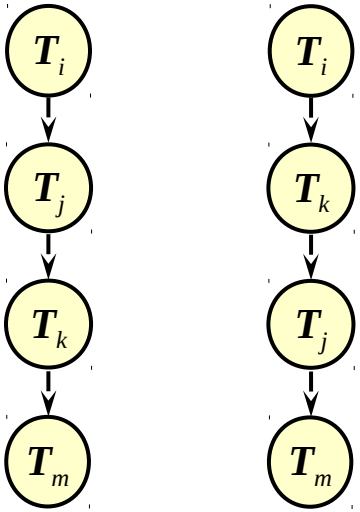
T_1	T_2	T_3	T_4	T_5
	$r(X)$			
$r(Y)$				
$r(Z)$				
				$r(V)$
				$r(W)$
	$r(Y)$			
	$w(Y)$			
		$w(Z)$		
$r(U)$				
			$r(Y)$	
			$w(Y)$	
			$r(Z)$	
			$w(Z)$	
$r(U)$				

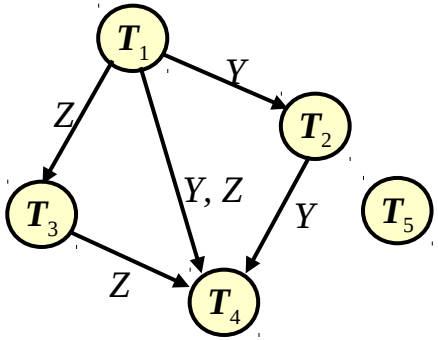


Test serializovatelnosti (pokr.)

- Rozvrh je serializovatelný právě tehdy, je-li precedenční graf acyklický
- Algoritmy na detekci cyklů ← (Téma 5)
  - mají časovou složitost n^2, kde n je počet vrcholů grafu
    - Existují i algoritmy se složitostí $n + e$, kde e počet hran
- Je-li precedenční graf acyklický, pak sériové pořadí transakcí lze určit tzv. topologickým řazením grafu
  - Je to lineární řazení, v němž každý uzel předchází všechny uzly, k nimž vedou výstupní hrany. Topologické řazení není jednoznačné
    - Existuje více možností
  - Např., topologické řazení pro rozvrh z předchozí stránky může být $T_5 \prec T_1 \prec T_3 \prec T_2 \prec T_4$
    - Existují nějaká jiná řazení?







Obnovitelné rozvrhy

- Vzhledem k tomu, že obvykle nelze zabránit chybovým stavům souběžných transakcí, je třeba zajistit obnovu výchozího stavu
- Obnovitelný rozvrh
  - Jestliže transakce T_j čte data dříve modifikovaná transakcí T_i, pak potvrzení (**`commit`**) operace T_i ***musí*** předcházet potvrzení (**`commit`**) operace T_j.

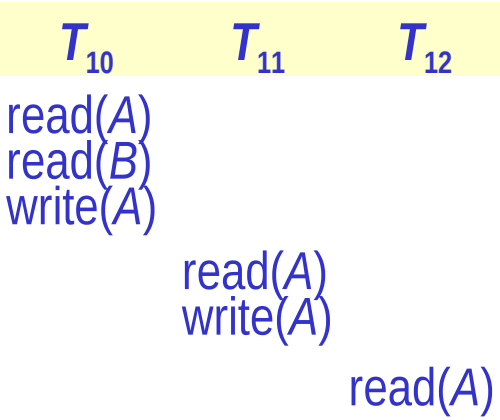
Rozvrh S_{11}

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

- Rozvrh S_{11} není obnovitelný, pokud transakce T_9 potvrdí svůj úspěch bezprostředně po čtení
  - Kdyby transakce T_8 zhavarovala a byla nucena provést **`abort`**, T_9 by četla (a možná i prezentovala uživateli) databázi v nekonzistentním stavu
- DBMS musí zajistit, že rozvrhy budou obnovitelné

Kaskáda návratů (*rollbacks*)

- Havárie jedné transakce může způsobit nutnost návratu databáze do výchozího stavu pro celou sérii transakcí. Taková situace se nazývá kaskáda návratů (*cascaded rollback*)
  - Rozvrh, kde žádná z transakcí dosud nepotvrdila úspěšné ukončení
    Pokud T_{10} zhavaruje, musí dojít k návratu i pro T_{11} a T_{12}



- Dochází k velkým ztrátám
- Nekaskádové rozvrhy jsou rozvrhy, kde nedochází ke kaskádovým návratům
  - Pro každý pár transakcí T_i a T_j takových, že T_j čte data modifikovaná dříve transakcí T_i, musí operace potvrzení v T_i (**`commit`**) předcházet operaci **`read`** v transakci T_j.
- Každý nekaskádový rozvrh je též obnovitelný
- Je žádoucí tvořit, pokud možno, jen nekaskádové rozvrhy

Řízení souběhu

- DBMS musí poskytovat mechanismy, které zajistí, že všechny rozvrhy při paralelním běhu transakcí budou
  - serializovatelné, obnovitelné a (pokud možno) nekaskádové
  - Strategie, kdy běží transakce jedna po druhé sice splňuje tyto požadavky, ale systém má špatnou průchodnost
- Cíl: vyvinout protokoly pro řízení souběhu, které zajistí serializovatelnost
  - Protokoly musí dovolit souběh za současné tvorby serializovatelných, obnovitelných a nekaskádových rozvrhů
  - Protokoly obvykle nezkoumají precedenční graf. Místo toho formulují závazná pravidla vylučující rozvrhy, které nejsou serializovatelné
- Protokoly pro řízení souběhu se navzájem liší stupněm paralelismu a velikostí působených režijních ztrát
  - Testy serializovatelnosti se explicitně neaplikují, pouze pomáhají pochopit, proč je ten který protokol korektní

Úrovně konzistence

- Některé aplikace se spokojí s nepřesnými výsledky, což dovoluje, aby rozvrhy nemusely být serializovatelné
  - Např. transakce, která chce zjistit přibližný součet zůstatků na všech účtech
    - Takové transakce nemusí být serializovatelné vůči transakcím ostatním
  - Kompromis mezi přesností a výkonností
- Úrovně konzistence podle standardu SQL-92
  - Serializable – implicitní úroveň
  - Repeatable read
    - smí se číst jen potvrzené záznamy (**`commit`**), opakované čtení téhož záznamu musí dodat tutéž hodnotu. Avšak transakce nemusí být plně serializovatelná – může např. najít jen některé záznamy vložené jinou transakcí, ale ne nutně všechny
  - Read committed
    - smí se číst jen potvrzené záznamy, ale opakované čtení nemusí vždy vrátit totéž
  - Read uncommitted
    - lze číst i nepotvrzené záznamy
  - Nižší úrovně konzistence lze použít pro získávání přibližných informací

Protokoly využívající zamykání

- Zamykací mechanismy k řízení souběhu jsou tytéž jako u kritických sekcí🡐
  - Položky dat lze zamykat ve dvou režimech:
  1. *výlučný – exkluzivní* (*X*) *režim*: Položka je plně uzamčena a může být čtena i zapisována.
     - X-zámek nad datovou položkou se zakládá pomocí "instrukce" **`lock-X`**
  2. *sdílený* (*S*) *režim*: Položka se smí pouze číst
     - S-zámek se zakládá pomocí "instrukce" **`lock-S`**
  - Matice kompatibility zámků

	S	*X*
S	true	false
X	false	false

     - Transakce smí přistoupit k položce (zamknout ji), jen pokud požadovaný režim zámku je kompatibilní s existujícími zámky vlastněnými jinými transakcemi nad stejnou položkou
       - Libovolný počet transakcí může současně získat S-zámek, avšak transakce "držící" X-zámek brání všem ostatním v přístupu k položce
  - Pokud zámek nelze získat, transakce je nucena čekat na uvolnění položky. Poté je přístup umožněn
    - Je to tatáž situace jako u semaforů a dalších synchronizačních mechanismů 🡐

Protokoly využívající zamykání (pokr.)

- Příklad transakce se zamykáním:

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

  - Uvedená posloupnost zamykání není dostatečná k zaručení serializovatelnosti
    - pokud by *A* a/nebo *B* bylo změněno, zobrazený součet by byl chybný
- Zamykací protokol je množina pravidel, která se aplikují při operacích **`lock-S`**, **`lock-X`** a **`unlock`**
  - Zamykací protokoly omezují množinu přípustných rozvrhů
  - Zamykání může být nebezpečné
    - Nebezpečí uváznutí (*deadlock*)
      - Nelze kompletně vyřešit – transakce budou musí být občas rušeny ("zabíjeny") a stav databáze obnovován
    - Nebezpečí stárnutí (*starvation*)
      - Transakce je opakovaně rušena a je vynucováno obnovování kvůli uváznutí
      - Manažer souběhu je schopen vhodnými prostředky zabránit stárnutí ➔

Dvoufázový zamykací protokol

- Jde o protokol, který zajišťuje rozvrhy ekvivalentní vůči konfliktu se sériovými
  - Fáze 1: Růst
    - transakce smějí pouze zamykat a nemohou zámky uvolňovat
  - Fáze 2: Smršťování
    - transakce smějí zámky pouze uvolňovat a nemohou nic dalšího zamykat
  - Protokol zajišťuje serializovatelnost.
    - Dá se dokázat, že transakce poběží v sérii podle pořadí dosahování svých zamykacích bodů (tj. bodů, kdy transakce získaly svůj poslední potřebný zámek)

Dvoufázový zamykací protokol (pokr.)

- Dvoufázový zamykací protokol *nevylučuje* uváznutí a neobsahuje ani prevenci vzniku kaskádových návratů
  - K tomu účelu byl vyvinut protokol zvaný striktní dvoufázové zamykání
    - Transakce musí držet všechny své *exkluzivní zámky*, dokud nepotvrdí své úspěšné ukončení (**`commit`**) nebo nezhavaruje (**`abort`**)
  - Rigorózní dvoufázové zamykání je ještě přísnější:
    - *Všechny* zámky musí transakce držet do svého úspěšného konce nebo havárie (**`commit`**/**`abort`**). V tomto protokolu jsou transakce serializovány v pořadí, jak končí

Automatické získávání zámků

- Transakce T_i užívá jen standardní operace read/write
  - bez explicitního zamykání, které je součástí těchto operací
  - Operace **`read`**(D) je implementována jako
    - `read_OS` a `write_OS` značí základní operace poskytované OS

```
if Ti vlastní zámek D then
    read_OS(D)
else begin
    if některá jiná transakce drží lock-X na D then wait;
    přiřaď transakci Ti lock-S na D;
    read_OS(D)
end
```

  - Operace **`write`**(D) se implementuje jako

```
if Ti vlastní lock-X na D then
    write_OS(D)
else begin
    if některá jiná transakce drží jakýkoliv zámek na D then wait;
    if  Ti vlastní lock-S na D then
        změň ("povyš") zámek z lock-S na D na lock-X
else
    přiřaď transakci Ti lock-X na D;
    write_OS(D)
end;
```

  - Všechny zámky jsou uvolněny jako součást **`commit`**/**`abort`**

Implementace zamykání

- Správu zámků má zpravidla ve své kompetenci samostatný výpočetní proces (či vlákno) – správce zámků (*lock manager*), jemuž všechny transakce zasílají žádosti o přidělení a uvolňování zámků
  - Žádající transakce čeká na odpověď od správce zámků
  - Správce zámků odpovídá na žádost o přidělení zámku
    - zasláním zprávy o přidělení zámku v případě úspěchu,
    - nebo zprávy se žádostí o návrat (obnovení) dat v případě, že je detekováno uváznutí
- Správce zámků se opírá o tabulku zámků
  - Obsahuje přidělené zámky a čekající nevyřízené žádosti
  - U přiděleného zámku či žádosti se pamatuje i typ zámku (*X*/*S*)
  - Tabulka zámků je zpravidla implementována jako hašovaná tabulka v operační paměti, indexovaná jménem datové položky podléhající zamykání

Tabulka zámků

- Nová žádost o přidělení zámku je zařazena na konec fronty žádostí a je ihned vyřízena, pokud je kompatibilní s existujícími zámky k příslušné datové položce
- Žádost o odemčení způsobí smazání zámku. Pak je prohledán zbytek fronty, kdy se zjišťuje, zda lze přidělit zámky čekajícím žádostem
- Když se transakce ruší (**`abort`**), všechny přidělené zámky i čekající žádosti se smažou. Dále se postupuje jako při odemykání zámků
  - k zefektivnění této operace může správce zámků udržovat i seznam zámků přidělených dané transakci

Grafově orientované protokoly

- Grafové protokoly jsou alternativou k dvoufázovému zamykání
- Zavedeme částečné uspořádání $\rightarrow$ na množině $\mathbf{D} = \{d_1, d_2, ..., d_h\}$ datových položek, s nimiž transakce mají pracovat
  - Jestliže $d_i \rightarrow d_j$, pak transakce používající jak d_i tak i d_j musí přistoupit k d_i před přístupem k d_j.
  - Na množinu **D** lze nyní pohlížet jako na orientovaný acyklický graf, zvaný graf databáze
    - Připomeňme dříve vysvětlený princip "číslování sdílených prostředků" u kritických sekcí, který zabraňuje uváznutí ←

- Jednoduchým grafovým protokolem je stromový protokol (*tree-protocol*)

Stromový protokol

1. Uvažujme jen *X*-zámky
2. Jako první může transakce T_i zamknout kteroukoliv položku dat. Následně položka *Q* smí být transakcí T_i uzamčena pouze když některý předchůdce položky *Q* ve stromu je též uzamčen touto transakcí
3. Odmykat lze kdykoliv
4. Položka, která byla transakcí T_i jednou zamčena a odemčena nesmí následně být stejnou transakcí zamčena znovu

Vlastnosti stromového protokolu

- Výhody
  - Stromový protokol zabezpečuje serializovatelnost a odstraňuje riziko uváznutí
  - Odmykání záznamů může proběhnout dříve než u dvoufázového zamykání
    - kratší čekání, lepší průchodnost, vyšší stupeň paralelismu
  - Díky neexistenci rizika uváznutí není potřeba návrat (*rollback*)
- Nevýhody
  - Protokol nezaručuje obnovitelnost ani odolnost vůči kaskádním návratům
    - Je tedy nutno zavést vzájemné závislosti mezi operacemi **`commit`**, aby se zajistila obnovitelnost
    - Vlivem toho, transakce budou muset držet zámky i na záznamech, které již nepotřebují, což vede k částečné degradaci uvedených výhod
- Některé rozvrhy nerealizovatelné při dvoufázovém zamykání jsou pro stromový protokol přípustné, ale také naopak

Granularita zamykání

- Data mají různou důležitost a velikost
  - celá databáze, relace a s ní spojený index, jen index, jeden záznam, ...
  - Má tedy smysl definovat hierarchii působnosti zámků, kde "jemnější zámky" jsou zanořeny (podřízeny) hrubším
  - Hierarchii lze znázornit stromem
    - POZOR: nejde o stromový protokol

  - Když transakce *explicitně* zamkne některý uzel stromu, pak se *implicitně* zamknou všechny podřízené položky a to ve stejném režimu zamykání

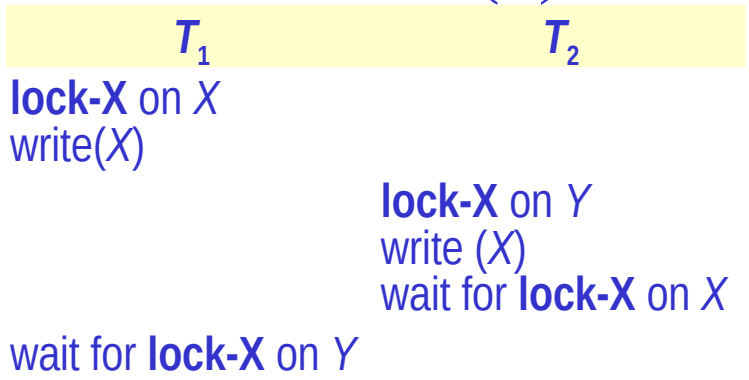
Uváznutí

- Mějme 2 transakce:

  T_1: write(X)
  write(Y)

  T_2: write(Y)
  write(X)

  – Rozvrh s uváznutím



- Protokoly s prevencí uváznutí zajistí, že k uváznutí nikdy nedojde
  - Základní strategie prevence jsou:
    - Požaduj, aby transakce předem zamkla vše, co bude potřebovat pro svoji budoucí práci.
    - Částečné uspořádání datových položek – grafový protokol
  - Ne vždy je lze použít
    - Vždy limitují průchodnost

Pokročilé strategie prevence uváznutí

- Strategie s časovými značkami
  - Nepreemptivní strategie wait-die (čekej-zahyň)
    - Starší transakce smí čekat, než mladší strategie uvolní drženou datovou položku. Mladší transakce nikdy nečeká, místo toho volá **`abort`** a obnoví data do stavu před svým spuštěním (zahyne)
    - Transakce může zahynout mnohokrát, než se jí podaří získat data
  - Preemptivní strategie wound-wait (poraň-čekej)
    - Starší transakce *poraní* (= vynutí obnovu stavu - *rollback*) mladší transakci, místo čekání na ni. Mladší transakce však mohou na starší čekat
    - Obecně způsobuje méně návratů než strategie *wait-die*
  - Pro obě strategie platí
    - Transakce, které obnovily stav dat, jsou restartovány se svými původními časovými značkami. Tím je zaručeno, že starší (tj. dříve zahájené) transakce budou mít přednost před mladšími, čímž je odstraněno riziko stárnutí.
- Strategie založené na časových prodlevách (*timeout*)
  - Transakce čeká na zámek nejvýše určitou dobu
    - Když se nedočká, volá **`abort`** (a tím obnovu dat)
    - Časové omezení brání vzniku uváznutí
    - Snadná implementace, avšak hrozí stárnutí.
    - Obtížné je stanovení dobré hodnoty časového omezení

Detekce uváznutí a zotavení

- Uváznutí se detekuje pomocí čekacího grafu (🡸 Téma 5)
  - Vrcholy jsou transakce T_i
  - Orientovaná hrana $T_i \rightarrow T_j$ značí, že T_i čeká, až T_j odemkne datovou položku
  - Je-li v čekacím grafu cyklus, došlo k uváznutí
  - Algoritmus detekce byl popsán v dříve (Téma 5)
- Když se zjistí uváznutí
  - Je nutno nalézt obětní transakci a vnutit jí **`abort`** (a tím i obnovu dat). Obětuje se obvykle nejmladší transakce, tj. ta, která ještě neudělala mnoho změn
  - Transakce mohou stárnout, bude-li za oběť vybírána vždy nejmladší transakce. Proto je vhodné do kriteria výběru oběti zahrnout i počet transakcí provedených návratů.
  - Která data se ale mají obnovovat?
    - Totální obnova transakci úplně zruší, data se vrátí do počátečního stavu, a transakce se restartuje. To může být velmi nákladné
    - Efektivnější je, když se transakce "vrací postupně" do stavu, kdy uváznutí zmizí. Tento postup je ale náročný na evidenci kroků a změn transakcí provedených
    - Používá se např. tzv. metoda kontrolních bodů (*checkpointing*) 🡺

Dotazy