

Paralelní přístup k databázi

Motivační příklad:

Bankovní převod 100,- Kč z účtu "A" na účet "B" a současný výběr 200 Kč z účtu "B".

Transakce	Hodnota A	Hodnota B	Stav účtu A	Stav účtu B
			1000,-	1000,-
T1: čti A	T1: 1000			
T1: čti B		T1: 1000		
T1: uber 100 z A	T1: 900			
T1: přidej 100 k B		T1: 1100		
T1: zapiš stav A			900,-	
T2: čti B		T2: 1000		
T1: zapiš stav B				1100,-
T2: uber 200 z B		T2: 800		
T2: zapiš stav B				800,-
Výsledný stav			900,-	800,-
Správně			900,-	900,-

Při současném vykonávání transakcí by mohlo dojít k porušení konzistence databáze i když každá z transakcí by sama o sobě konzistenci zachovávala

Paralelní přístup k databázi

Transakce:

Vlastnost **ACID**:

<u>A</u>tomicity:	transakce atomická - buď se podaří a provede se celá nebo nic - nelze vykonat jen část transakce
<u>C</u>onsistency	transakce - korektní transformace stavu - zachování invariant - integritních omezení
<u>I</u>solation	(isolace = serializovatelnost). I když jsou transakce rozpracovány současně, výsledek je stejný, jako by byla vykonávána jedna po druhé
<u>D</u>urability	po úspěšném ukončení transakce (commit) jsou změny stavu databáze trvalé a to i v případě poruchy systému - zotavení z chyb.

Paralelní přístup k databázi

První zákon řízení paralelismu:

Současné vykonávání transakcí nesmí zapříčinit selhání programu.

Sériové provádění transakcí

Další transakce nezačne dříve, než předchozí skončila.

Bylo by postačujícím řešením v případě, že:

- všechny transakce krátké
- všechna data v paměti
- všechna data zpracována jediným procesorem

Paralelní přístup k databázi

Serializovatelné provádění transakcí:

- Více transakcí rozpracovaných současně (vyšší propustnost systému)
- Ale výsledek stejný jako při sériovém provádění.

Druhý zákon řízení paralelismu:

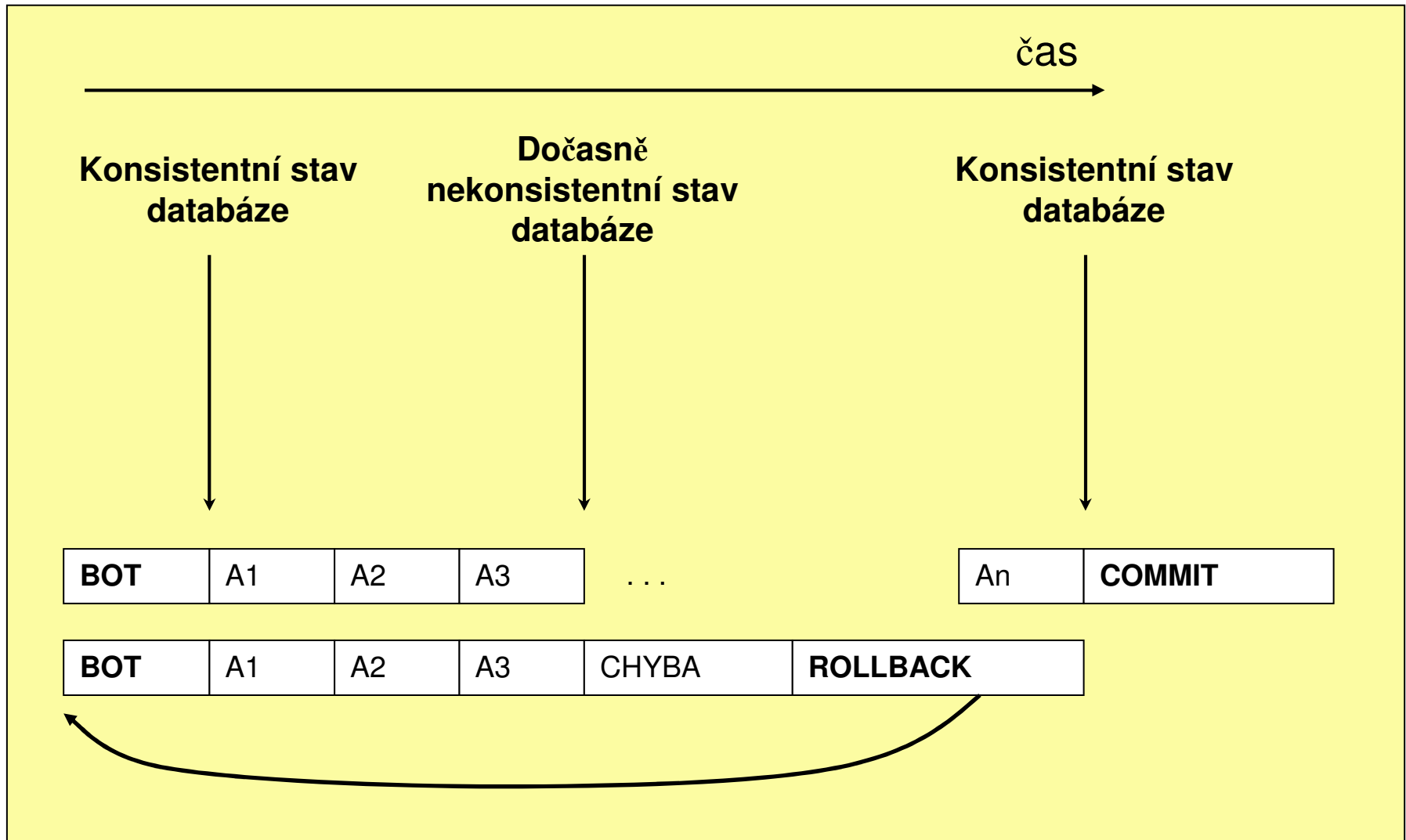
Metoda řízení paralelismu při současném vykonávání transakcí nesmí vést k nižší průchodnosti systému oproti sériovému vykonávání transakcí.

Řešení serializovatelnosti:

- zamykání (locking) na různé úrovni granularity:
 - zamykání celého systému (=> sériovost)
 - jednotlivých tabulek
 - Jednotlivých záznamů (vět)
- časové značky
- MVCC (multiversion concurrency control)
- predikátové zámky

Paralelní přístup k databázi

Transakce = sekvence akcí **read** a **write** na objektech (insert a delete zatím nejuvažujme).



Paralelní přístup k databázi

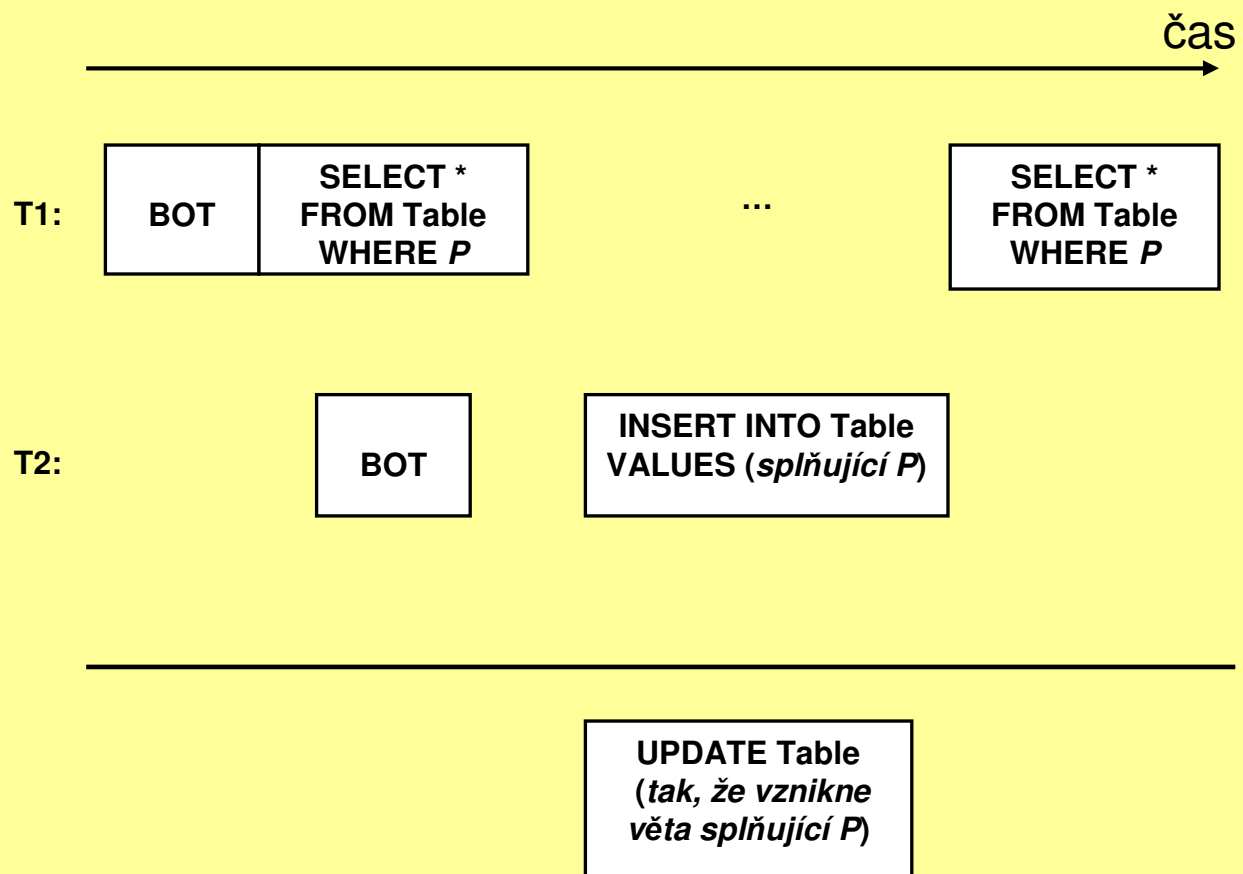
- Dvě akce **read** nad týmž objektem **nemohou** porušit konzistenci.
- Dvě akce **write** nad týmž objektem prováděné v **rámcí téže transakce** **netřeba uvažovat**. Pokud by porušily konzistenci, je transakce nekorektní bez ohledu na paralelismus (porušeno C v ACID).
- Pouze akce typu **read** a **write** v rámci různých transakcí **mohou** porušit konzistenci.

Paralelní přístup k databázi

Lost update	T1 WRITE	<0,1>	Nezachová se. Jakoby T1 vůbec neběžela.
	T2 WRITE	<0,2>	
	T1 READ	<0,2>	
Dirty read	T2 WRITE	<0,2>	T1 přečetla dočasnou (ne committed) hodnotu
	T1 READ	<0,2>	
	T2 ROLLBACK	<0,1>	
Unrepeatable read	T1 READ	<0,1>	nereprodukovatelné čtení
	T2 WRITE	<0,2>	
	T1 READ	<0,2>	
Phantom problem	Bude vysvětlen na následujícím slidu		

Paralelní přístup k databázi

Problém fantomu (Phantom problem)



Paralelní přístup k databázi

Příklad LOST UPDATE:

Transakce T_1 vybírá veškerý zůstatek z účtu A.

Transakce T_2 připisuje 3% úroků na účet A.

Příklad transakční historie neboli rozvrhu:

Krok	T_1	T_2
1.	BOT	
2.		BOT
3.	$a_1 := 0$	
4.		READ(A, a_2)
5.		$a_2 := a_2 * 1.03$
6.	WRITE(A, a_1)	
7.		WRITE(A, a_2)
8.	COMMIT	
9.		COMMIT

Paralelní přístup k databázi

Příklad DIRTY READ:

Transakce T_1 převádí 300,- Kč z účtu A na účet B.

Transakce T_2 připisuje 3% úroků na účet A, který není v daném časovém okamžiku v konsistentním stavu.

Příklad transakční historie neboli rozvrhu:

Krok	T_1	T_2
1.	READ(A, a_1)	
	$a_1 := a_1 - 300$	
	WRITE(A, a_1)	
		READ(A, a_2)
5.		$a_2 := a_2 * 1.03$
6.		WRITE(A, a_2)
7.	READ(B, b_1)	
	READ selhal, proto:	
8.	ROLLBACK	

ROLLBACK vrátí stav na začátek transakce T_1 , tím ale přijdeme o efekt transakce T_2 . To je problém transakce T_2 , že si přečetla údaj, který ještě nebyl potvrzen (committed)

Paralelní přístup k databázi

Příklad UNREPEATABLE READ:

Transakce T_1 převádí 300,- Kč z účtu A na účet B.

Transakce T_2 připisuje 3% úroků na účet A, který není v daném časovém okamžiku v konsistentním stavu.

Příklad transakční historie neboli rozvrhu:

Krok	T_1	T_2
1.	READ(A, a₁)	
2.		READ(A, a ₂)
3.		a ₂ := a ₂ * 1.03
4.		WRITE(A, a₂)
5.	a ₁ := a ₁ - 300	
6.	WRITE(A, a ₁)	
7.	READ(A, a ₁)	
8.	b ₁ := b ₁	
8.	WRITE(B, b ₁)	

T_2 přepsala údaj, který má transakce T_1 načtený a hodlá s ním nadále pracovat -> T_1 bude pracovat s nekonzistentními daty!

Proměnná a_1 neodpovídá stavu databáze. Kdybychom znovu provedli READ(A, a₁), byl by obsah proměnné a_1 jiný!

Paralelní přístup k databázi

Příklad PHANTOM PROBLEM:

V průběhu zpracování T_2 zavede T_1 do databáze nový údaj (větu), proto T_2 pro dva totožné dotazy poskytne dvě různé odpovědi.

Příklad transakční historie neboli rozvrhu:

Krok	T_1	T_2
1.		SELECT sum(<i>StavUctu</i>) FROM <i>Ucty</i>
2.	INSERT INTO <i>Ucty</i> VALUES (<i>StavUctu</i> , 1000)	
3.		SELECT sum(<i>StavUctu</i>) FROM <i>Ucty</i>

Paralelní přístup k databázi

Lost update	T1 WRITE	<0,1>	Nezachová se. Jakoby T1 vůbec neběžela.
	T2 WRITE	<0,2>	
	T1 READ	<0,2>	
Dirty read	T2 WRITE	<0,2>	T1 přečetla dočasnou (ne committed) hodnotu
	T1 READ	<0,2>	
	T2 ROLLBACK	<0,1>	
Unrepeatable read	T1 READ	<0,1>	nereprodukovatelné čtení
	T2 WRITE	<0,2>	
	T1 READ	<0,2>	
Phantom problem	T1 SELECT predikát	{ o1, o2}	
	T2 INSERT o3		
	T1 SELECT predikát	{ o1, o2}	

Transakční historie (rozvrh transakcí) - posloupnost akcí několika transakcí, jež zachovává pořadí akcí, v němž byly prováděny.

Historie (rozvrh) se nazývá **sériová**, pokud jsou všechny kroky jedné transakce provedeny před všemi kroky druhé transakce.

Serializovatelná historie		
Krok	T ₁	T ₂
1	BOT	
2	READ(A)	
3		BOT
4		READ(C)
5	WRITE(A)	
6		WRITE(C)
7	READ(B)	
8	WRITE(B)	
9	COMMIT	
10		READ(A)
11		WRITE(A)
12		COMMIT

Sériová historie	
T ₁	T ₂
BOT	
READ(A)	
WRITE(A)	
READ(B)	
WRITE(B)	
COMMIT	
	BOT
	READ(C)
	WRITE(C)
	READ(A)
	WRITE(A)
	COMMIT

Teorie SERIALIZOVATELNOSTI

Nechť se transakce T_i skládá z následujících elementárních operací:

- **READ_i(A)** - čtení objektu A v rámci transakce T_i
- **WRITE_i(A)** - zápis (přepis) objektu A v rámci transakce T_i
- **ROLLBACK_i** - přerušení transakce T_i
- **COMMIT_i** - potvrzení transakce T_i

Jsou možné následující 4 případy:

READ _i (A) - READ _j (A)	Není konflikt	Na pořadí nezávisí
READ _i (A) - WRITE _j (A)	Konflikt	Pořadí má význam
WRITE _i (A) - READ _j (A)	Konflikt	Pořadí má význam
WRITE _i (A) - WRITE _j (A)	Konflikt	Pořadí má význam

Zajímavé jsou navzájem konfliktní operace:

Dvě **historie H_1 a H_2** (na téže množině transakcí) jsou **ekvivalentní**, pokud jsou **všechny konfliktní operace** (nepřerušených) transakcí **provedeny v témže pořadí**.

To znamená, že pro dvě ekvivalentní historie a uspořádání $<_{H_1}$ indukované historií H_1 a $<_{H_2}$ indukované historií H_2 platí: pokud p_i a q_j jsou konfliktní operace takové, že $p_i <_{H_1} q_j$, musí platit také $p_i <_{H_2} q_j$. Pořadí nekonfliktních operací není zajímavé.

Ne každá historie je serializovatelná:

	Neserializovatelná historie	
Krok	T_1	T_2
1	BOT	
2	READ(A)	
3	WRITE(A)	
4		BOT
5		READ(A)
6		WRITE(A)
7		READ(B)
8		WRITE(B)
9		COMMIT
10	READ(B)	
11	WRITE(B)	
12	COMMIT	

Důvod:

Transakce T_1 předchází transakci T_2 při zpracování objektu **A**, ale transakce T_2 předchází transakci T_1 při zpracování objektu **B**.

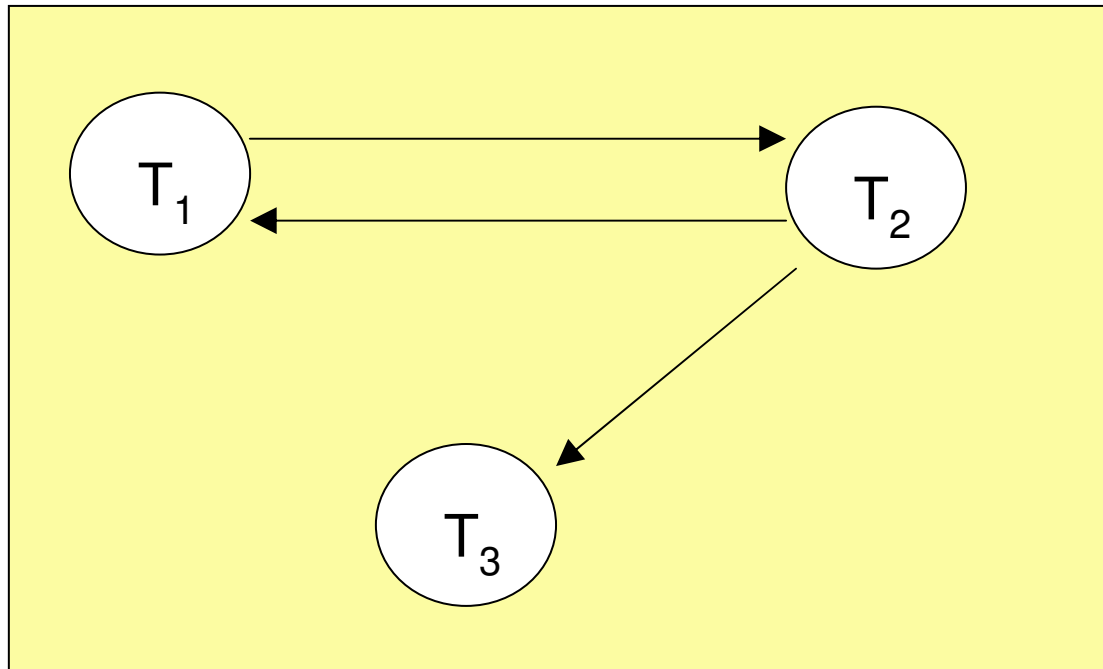
Tato historie není proto ekvivalentní ani sériovému provedení transakcí v pořadí T_1T_2 ani sériovému provedení transakcí v pořadí T_2T_1 .

Teorie serializovatelnosti (II)

Příklad: Mějme historii H například se třemi transakcemi T_1 , T_2 , T_3 :

$w_2(B) < r_1(B)$; $w_1(A) < r_2(A)$; $w_2(C) < r_3(C)$; $w_2(A) < r_3(A)$

Závislostní graf: $T_2 \rightarrow T_1$ $T_1 \rightarrow T_2$ $T_2 \rightarrow T_3$ $T_2 \rightarrow T_3$



Historie H je serializovatelná právě tehdy, když její závislostní graf nemá cykly.

Zamykání:

2 typy zámků:

- SLOCK: tzv. sdílený (Shared) zámek.
- XLOCK: tzv. výlučný (eXclusive) zámek.

Dobře formulovaná transakce

- Před každou operací READ se na daném DB objektu uplatní zámek SLOCK,
- před každou operací WRITE se na daném DB objektu uplatní zámek XLOCK
- operace UNLOCK se na daném DB objektu může provést pouze tehdy, když je na daném DB objektu uplatněn zámek SLOCK/XLOCK
- každá operace SLOCK/XLOCK je v někdy v následujícím běhu transakce následována příslušnou akcí UNLOCK.

Kompatibilita zámků

		Existující zámek		
		Není	SLOCK	XLOCK
Požadovaný zámek	SLOCK	OK	OK	Konflikt
	XLOCK	OK	Konflikt	Konflikt

Legální historie:

Každá historie dodržující kompatibilitu zámků je historií **legální**.

Akce a transakce

Akce na objektech: READ, WRITE, XLOCK, SLOCK, UNLOCK

Akce globální: BEGIN, COMMIT, ROLLBACK

T'	BEGIN		T''	BEGIN	
	SLOCK	A		SLOCK	A
	XLOCK	B		READ	A
	READ	A		XLOCK	B
	WRITE	B		WRITE	B
	COMMIT			ROLLBACK	

Na další stránce se zbavme se operací COMMIT a ROLLBACK převedením na (z hlediska konzistence) ekvivaletní transakční model:

Jednoduchá transakce (simple transaction):

- 1) Obsahuje akce READ, WRITE, XLOCK, SLOCK a UNLOCK.
- 2) COMMIT se nahradí sekvencí příkazů UNLOCK A, pro každý objekt A, na nějž bylo v průběhu transakce aplikováno SLOCK A nebo XLOCK
- 3) ROLLBACK se nahradí sekvencí akcí:
 - WRITE A pro každý objekt A, na nějž T aplikovala akci WRITE A
 - UNLOCK A pro každý objekt A, na nějž T aplikovala akci SLOCK A nebo XLOCK A

T'	SLOCK	A	T''	SLOCK	A
	XLOCK	B		READ	A
	READ	A		XLOCK	B
	WRITE	B		WRITE	B
	UNLOCK	A		WRITE (undo)	B
	UNLOCK	B		UNLOCK	A
				UNLOCK	B

Dvoufázové transakce

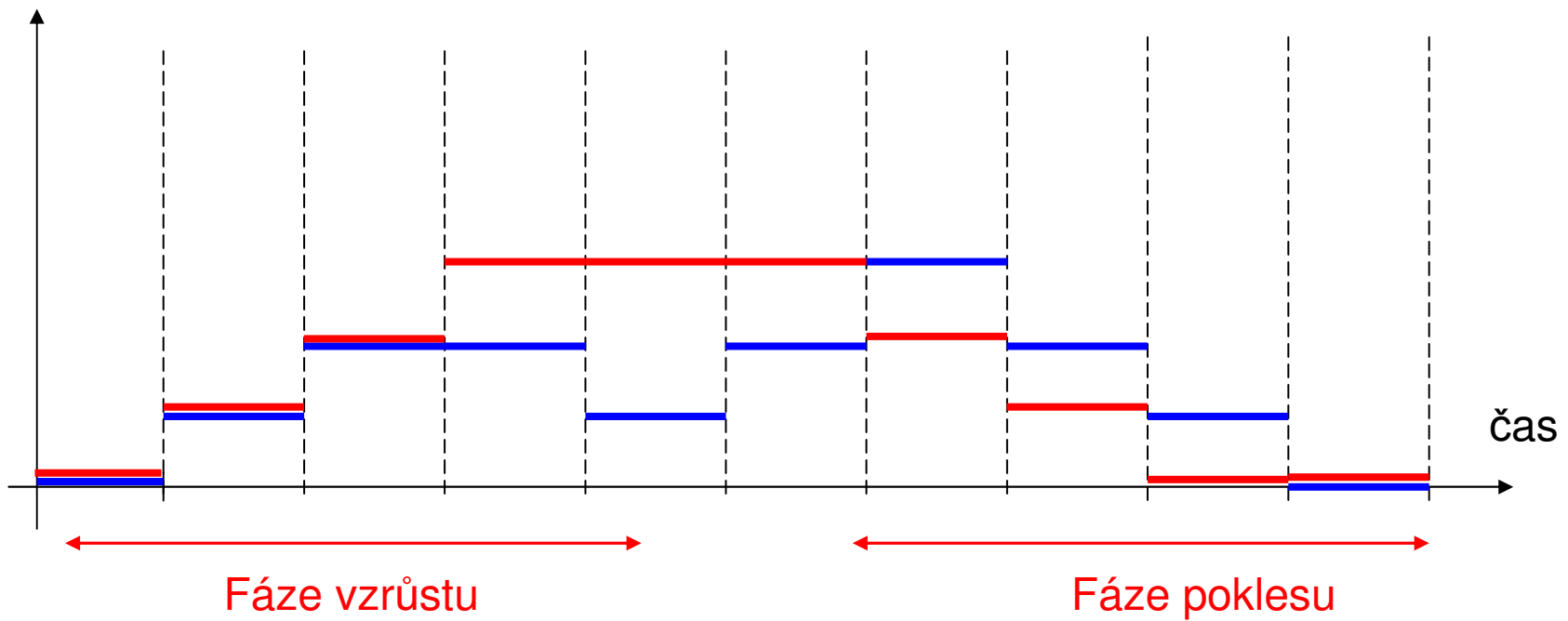
Všechny akce LOCK jsou provedeny před všemi akcemi UNLOCK.

Fáze vzrůstu (growing phase) - během ní se provedou všechny akce LOCK

Fáze poklesu (shrinking phase) - během ní se provedou všechny akce UNLOCK

U dvoufázové transakce se **fáze vzrůstu** a **fáze poklesu nepřekrývají**

počet uplatněných zámek



Transakce je serializovatelná (s výjimkou fantom problémů), právě když:

- je **dobře formulovaná** (všechny akce prokyty zámky)
- zamykat **výhradně** všechny data. jejichž obsah **modifikuje (legální)**
- je **dvoufázová** - neměla by uvolňovat zámky dříve než budou všechny zámky aplikovány
- **výhradní** zámky drží až do **COMMIT/ROLLBACK**

Stupně izolace

(stále neuvažujeme fantomy – náš zjednodušený model)

	Transakce	Názvy	Protokol zamykání
0°	0° T nepřepisuje dirty data jiné transakce, je-li tato stupně 1° a více	anarchie	dobře formulován pro WRITE
1°	1° T nemá lost updates	browse	dvoufázový pro XLOCK a dobře formulovaný pro WRITE
2°	2° nemá lost updates a dirty reads	*stabilita kursoru*	dvoufázový pro XLOCK a dobře formulovaný
3°	3° nemá lost updates, dirty reads a má repeatable reads	isolovaná transakce serializovatelná opakovatelné čtení	dvoufázový a dobře formulovaný

Cursor

```
char title[51], year[11], result[102], star_name[51];
```

```
EXEC SQL DECLARE CURSOR movie_cursor FOR  
SELECT title, CAST (year_released AS CHARACTER(10))  
FROM movie_titles;
```

```
while (/* cyklus pres jednotlivé filmy */)
{
    EXEC SQL FETCH NEXT FROM movie_cursor INTO :title, :year ;
    ...
}
```

Stabilita kursoru

SQL databázové systémy obvykle implementují poněkud vylepšený protokol 2° nazývaný **stabilita kursoru**.

Většina SQL databázových systémů aplikuje sdílený zámek na věty adresované nějakým kurorem => **stabilita kursoru**. Jedna z konkrétních implementací je popsána na stránce

<http://jazz.external.hp.com/training/sqltables/c5s38.html>

Stabilita kurzoru

Při operaci FETCH se provede:

1. Ukazatele ve zdrojových tabulkách kurzoru se posunou tak, že ukazují na dalšího kandidáta na záznam kurzoru
2. Záznamy, na něž ukazatele ve zdrojových tabulkách ukazují, se zamknou SLOCK zámkem
3. Ověří se, zda tento kandidát na záznam kurzoru (sestavený ze záznamů zdrojových tabulek, na něž momentálně ukazují ukazatele) opravdu patří do kurzoru.
4. Pokud ne, uvolní se zámkové záznamů zdrojových tabulek kurzoru a jde se na bod 1
5. Pokud ano, zůstanou příslušné záznamy zdrojových tabulek kurzoru, které se podílejí na aktuálním záznamu kurzoru, uzamčeny až do uzavření kurzoru. V případě, že v průběhu této transakce je daný záznam kurzoru modifikován, změní se u příslušných záznamů zdrojových tabulek kurzoru zámeček z SLOCK na XLOCK.

Podstatné je, že je-li nastaven stupeň izolovanosti tak, že zahrnuje stabilitu kurzoru, operace FETCH před přesunem na další záznam kurzoru **neuvolňuje zámkové** aktuálních záznamů zdrojových tabulek kurzoru.

SET TRANSACTION ISOLATION LEVEL [**READ UNCOMMITTED**]
[**READ COMMITTED**]
[**REPEATABLE READ**]
[**SERIALIZABLE**]

- READ UNCOMMITTED** - 1° browse - pro read-only transakce
- READ COMMITTED** - stabilita kursoru (vylepšený 2°)
- REPEATABLE READ** - 3° bez ochrany fantomů
- SERIALIZABLE** - 3° včetně ochrany fantomů

Metoda časových značek

- Každá transakce obdrží při zahájení časovou značku (při 1 tiku za ms vystačí 32 bitů na 49 dní)
- Při přístupu transakce k objektu za účelem čtení dostane objekt nejvyšší značku ze všech časových značek transakcí, jež čtou daný objekt
- Při přístupu transakce k objektu za účelem zápisu dostane objekt časovou značku dané transakce

Omezení:

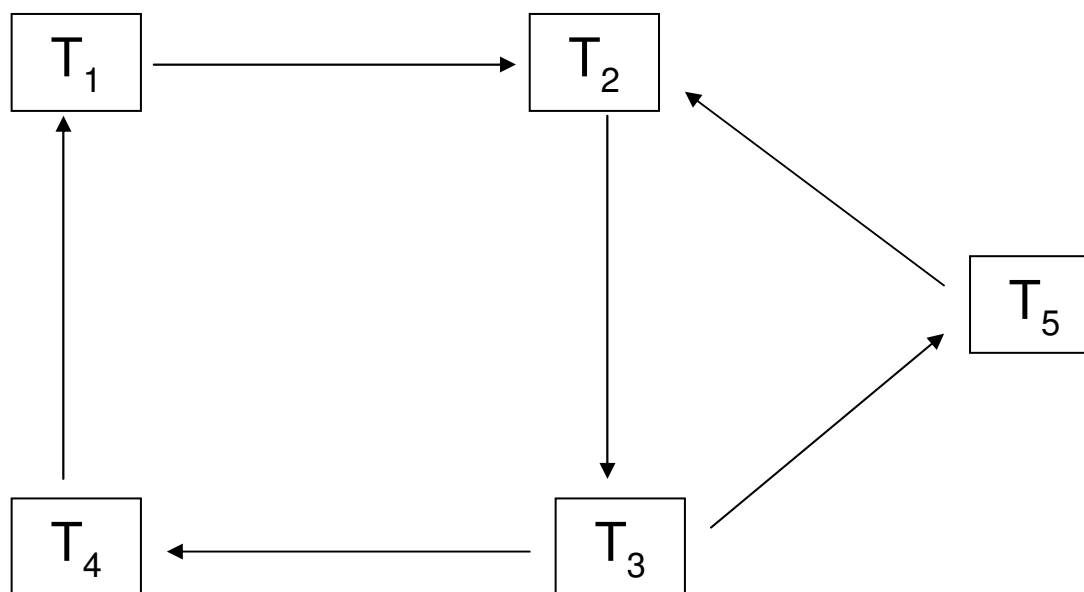
- Transakce se značkou **t** nesmí číst objekty se značkami **tw > t**.
Následuje ROLLBACK
- Transakce se značkou **t** nesmí přepsat objekty se značkami **tr > t**.
Následuje ROLLBACK
- Dvě transakce smějí číst tentýž objekt v libovolném okamžiku
- Když chce transakce se značkou **t** přepsat objekt se značkou **tw > t**, musí počkat, až bude značka **tw** odstraněna.

Vzájemné uváznutí transakcí (deadlock)

Krok	T ₁	T ₂	
1	BOT		
2	LockX(A)		
3		BOT	
4		LockS(B)	
5		Read(B)	
6	Read(A)		
7	Write(A)		
8	LockX(B)		T ₁ musí čekat na T ₂
9		LockS(A)	T ₂ musí čekat na T ₁
10	

Vzájemné uváznutí transakcí (deadlock)

Graf vzájemného čekání:



Odstranění cyklů – strategie:

- Přerušovat co nejmladší transakci (ovlivnit co nejméně dalších transakcí)
- Přerušovat transakci s max. počtem zámků
- Nepřerušovat transakci, která byla již vícekrát přerušena
- Přerušit transakci, která se účastní více cyklů

Ochrana před fantomy:

Jediná spolehlivá ochrana jsou **predikátové zámky**.

SELECT * FROM T Where P1()

Predikát P1() se uloží do seznamu aktivních predikátových zámků.

Chci paraleleně provést **INSERT INTO T** .

Musí se ověřit, zda náhodou vkládané věta nevyhovuje některému z aktivních predikátových zámků.

Musí se ověřit, zda náhodou vkládané věta nevyhovuje některému z aktivních predikátových zámků. Pokud ano, konflikt.

Predikátové zámky výpočetně i implementačně náročné. Výrobci DB systémů se jim vyhýbají.

Když ne predikátové zámky, co tedy?

- Časové značky
- MVCC – multiversion Concurrency Control

MVCC – multiversion Concurrency Control

- Používá časové značky
- Snapshot isolation
 - Na začátku transakce se udělá „snapshot“ databáze.
 - Změny prováděné během transakce jsou vidět v této transakci, ale nikoliv v transakcích paralelních.
 - Po skončení transakce se provede commit pouze tehdy, když updaty, které provedla, nejsou v konfliktu s updaty transakcí, které commitovaly poté, co jsme udělali snapshot.

ISO:

Postgre SQL

**READ UNCOMMITTED
READ COMMITTED
REPEATABLE READ
SERIALIZABLE**

**READ COMMITTED
READ COMMITTED
SERIALIZABLE
SERIALIZABLE**

READ COMMITTED in PostgreSQL:

Snapshot se dělá na začátku SELECTU

SERIALIZABLE in PostgreSQL:

Snapshot se dělá na začátku transakce.

PostgreSQL – manuál sekce 12.2.2.1:

Class	Value
1	10
1	20
2	100
2	200

Spustíme současně:

- Insert výsledku `SELECT SUM(value) FROM mytab WHERE class = 1;` do mytab
- Insert výsledku `SELECT SUM(value) FROM mytab WHERE class==2;` do mytab

Jaký bude výsledek?