Databázové transakce (10)

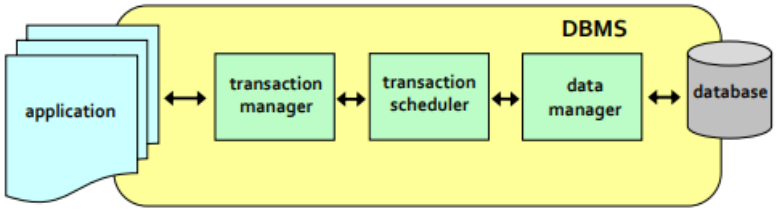
= sekvence akcí na databázových objektech (+ aritmetika apod.)

- Let us have a bank database with table ***Accounts*** and the following transaction to transfer the money (pseudocode):

```
transaction PaymentOrder(amount, fromAcc, toAcc)
    {
     1. SELECT Balance INTO X FROM Accounts WHERE accNr = fromAcc;
     2. if (X < amount) AbortTransaction("Not enough money!");
     3. UPDATE Accounts SET Balance = Balance - amount WHERE accNr = fromAcc;
     4. UPDATE Accounts SET Balance = Balance + amount WHERE accNr = toAcc;
     5. CommitTransaction;
    }
```

Správa transakcí v DBMS

- aplikace spouští transakce
- **transaction manager** vykoná transakce
- **plánovač** (**scheduler**) dynamicky naplánuje paralelní vykonávání tr., vytvoří **rozvrh** (**schedule**)
- **data manager** vykoná určité operace v transakci



- **ukončení transakce:**
  - **úspěšné** – ukončeno příkazem **COMMIT**, vykonané akce potvrzeny
  - **neúspěšné** – transakce je zrušena
    - **ukončení transakčním kódem** – příkaz **ABORT** (či **ROLLBACK**) – uživatel může být upozorněn
    - **přerušení systémem** – DBMS přeruší transakci
      - např. porušení nějakého integritního omezení – uživatel upozorněn
      - nebo plánovačem (např. deadlock) – uživatel neupozorněn
    - **selhání systému** – HW selhání, výpadek proudu - transakci nutno restartovat

ACID

- hlavní úkoly transakcí – vynucení ACID vlastností, max. výkon (propustnost) – paralelizace

- **Atomicity** – částečné vykonání transakce není povoleno (vše nebo nic)
  - zabraňuje nesprávnému přerušení transakce (nebo selhání)
  - = konzistence na DBMS úrovni

- **Consistency** – jakákoliv transakce převede DB z jednoho konzistentního stavu do druhého
  - = konzistence na aplikační úrovni

- **Isolation** – transakce prováděné paralelně nevidí efekty těch ostatních, dokud nejsou commitnuty

- **Durability** – jakmile je transakce commitnutá, již tak zůstane (i při výpadku proudu, apod.)
    - důležité logování

Transakce

- vykonaná transakce je sekvence základních DB akcí/operací:
    - **T = <A_T^1, A_T^2, ... , COMMIT or ABORT>**

- pro teď uvažujme statickou DB (žádné inserty/delete, jen update), nechť A je databázový objekt (tabulka, řádka, atribut)

Example:
Subtract 5 from A (some attribute), such that A>0.

```
T = <READ(A),                 // action 1
if (A ≤ 5) then ABORT
else WRITE(A – 5),            // action 2
COMMIT>                       // action 3
or
T = <READ(A),                 // action 1
if (A ≤ 5) then ABORT         // action 2
else ... >
```

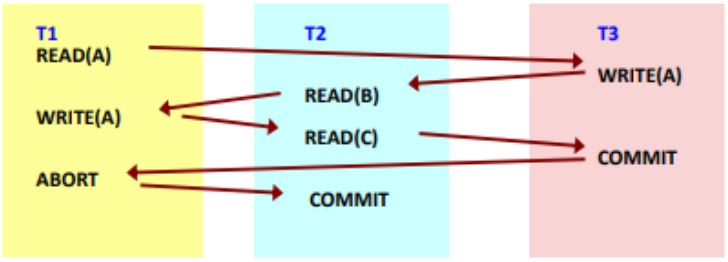
- **READ**(A) – přečte A z databáze
- **WRITE**(A) = zapíše A do databáze
- **COMMIT** – potvrdí vykonané akce jako platná, skončí transakci
- **ABORT** – zruší vykonané akce, ukončí transakci s chybou

- SQL příkazy SELECT, INSERT, UPDATE mohou být nahlíženy jako transakce implementované pomocí základních akcí; v sql je užit ROLLBACK místo abort

Schedules

- **databázový program** = navržený (neběžící) kus kódu, který bude vykonán jako transakce
    - nelineární (větvení, smyčky, skoky,...)

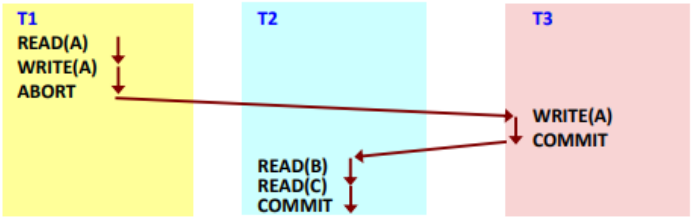
- **rozvrh (historie)** = seřazený seznam všech akcí přicházejících z různých transakcí (transakce jsou prokládány)
    - „runtime“ historie všech již současně vykonaných akcí několika transakcí
    - lineární (sekvence primitivních operací bez kontrolních konstruktů)



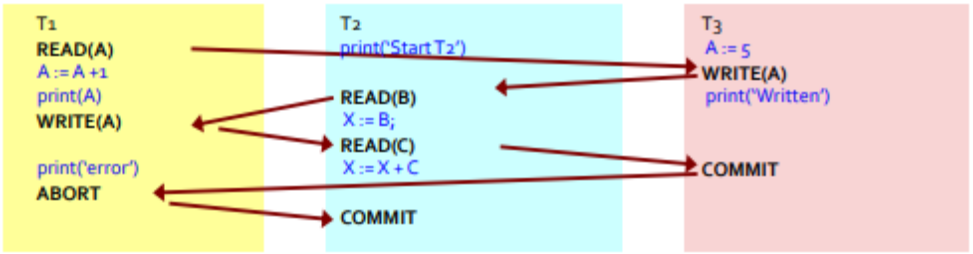
Serializovatelnost

- **serial schedules** = takové, kde jsou všechy akce transakce sdruženy dohromady
    - žádná akce není prokládána

- máme-li množinu transakcí S, můžeme získat |S|! seriál schedules
    - díky definici ACID jsou všechny schedules ekvivalentní, nezáleží na tom, je-li nějaká vykonána před jinou (a pokud ano, tak nejsou nezávislé a měly by být spojené do jedné transakce)



- **Proč prokládat transakce?**
  - každý schedule vede k prokládanému sekvenčnímu provedení transakcí (není zde žádné paralelní vykonávání DB operací)
  - Proč tedy prokládat, je-li počet kroku stejný jako v serial schedule?
    - paralelní vykonávání ne-DB operací s DB operacemi
    - odpověď úměrná složitosti transakce (OldestEmployee vs. ComputeTaxes)



- schedule je **serializovatelný**, vede-li jeho provedení ke konzistentnímu DB stavu, tj. je-li schedule ekvivalentní k jakémukoliv seriál schedule
  - prozatím jen commitnuté transakce a statickou DB
  - ne-DB operace pomíjíme, tam nelze zajistit konzistenci
  - silná vlastnost (zabezpečuje Isolation a Consistency v ACID)

- **view serializability** rozšiřuje s. zahrnutím přerušených transakcí a dynamickou DB
  - je ale NP-complete, takže v praxi namísto toho conflict serializability apod.

Konflikty

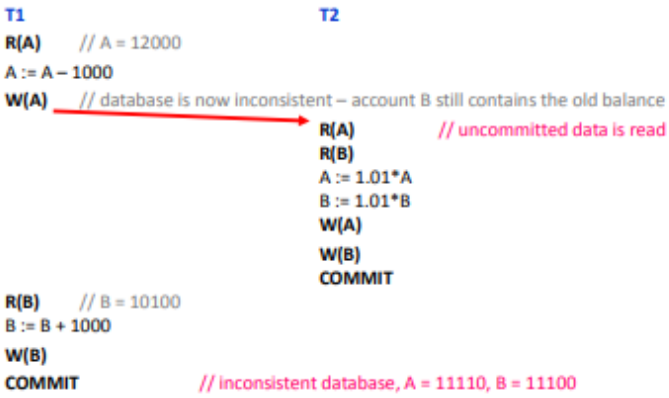
- kvůli zajištění serializovatelnosti (Consistency, Isolation) nesmí být akce prokládání náhodné
- existují tři typy lokálních závislostí ve schedule, tzv. **konfliktní páry**

- jsou čtyři možnosti čtení/zápisu stejného zdroje ve schedule:
  - **read-read** – ok, čtením transakcí se neovlivňují
  - **write-read (WR)** – T1 zapisuje, T2 čte → čtení necommitnutých dat
  - **read-write (RW)** – T1 čte, T2 zapisuje → neopakovatelné čtení
  - **write-write (WW)** – T1 zapisuje, pak T2 zapisuje → přepisování necommitnutých dat

WR (write-read)

- čtení necommitnutých dat (tzv. **dirty read**)
- transakce T2 čte A, které bylo dříve akzualizováno transakcí T1, ale T1 zatím necommitnula, takže T2 čte potencionálně nekonzistentní data

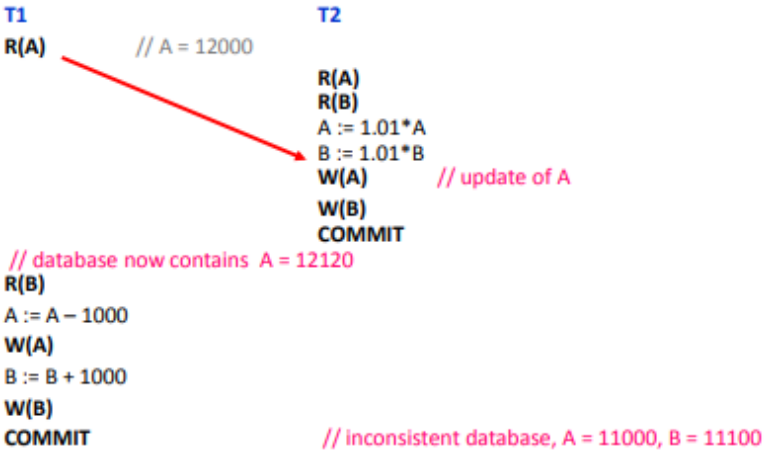
Example: T1 transfers 1000 USD from account A to account B (A = 12000, B = 10000)
T2 adds 1% per account



RW (read-write)

- neopakovatelné čtení (**unrepeatable read**)
- transakce T2 zapisuje A, které bylo čteno dříve v T1, která ještě neskončila
- T1 nemůže opakovat čtení A, které již nyní obsahuje jinou hodnotu

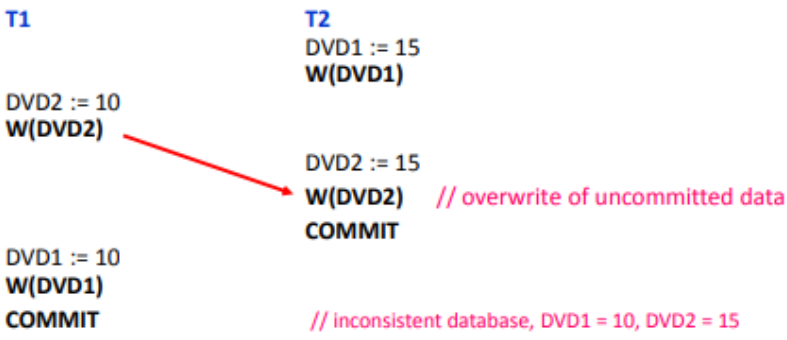
Example: T1 transfers 1000 USD from account A to account B (A = 12000, B = 10000)
T2 adds 1% per account



WW (write-write)

- přepisování necommitnutých dat (**blind write**)
- transakce T2 přepisuje A, které bylo dříve zapsáno v T1, která stále běží
- ztráta aktualizace (originální hodnota A ztracena)

Example: Set the same price to all DVDs.
(let's have two instances of this transaction, one setting price to 10 USD, second 15 USD)

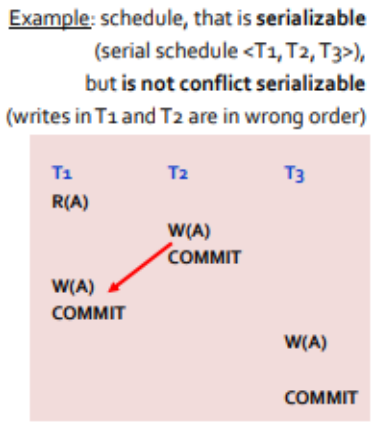


Serializovatelnost konfliktů

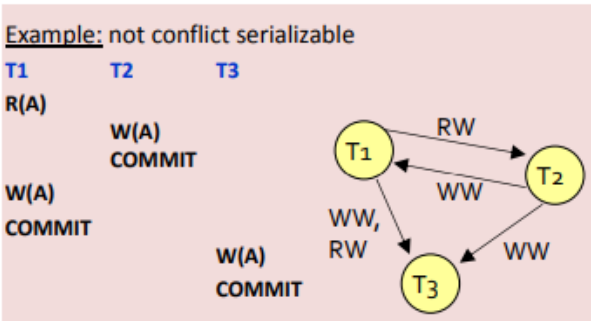
- dva schedule jsou **konfliktně ekvivalentní**, pokud sdílí množinu konfliktních párů
- schedule je **konfliktně serializovatelný**, pokud je konfliktně ekvivalentní s nějakým seriál schedule, tj. žádné „opravdové“ konflikty

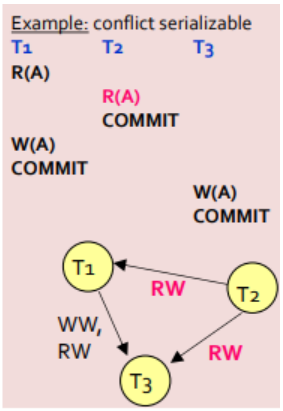
- nebere v úvahu:
  - zrušené transakce (ABORT/ROLLBACK), takže schedule může být neobnovitelný
  - dynamické DB (inserting, deleting), takže může vzniknout tzv. fantom

  … CS tedy není dostatečnou podmínkou zajištění ACID (view ser. je)



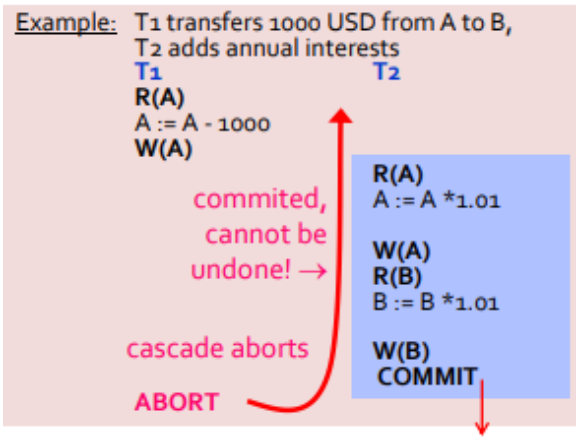
- detekce – **precedence graph** (prioritní graf?) na schedule
  - nody T jsou commitnuté transakce
  - hrany reprezentují RW, WR a WW konflikty na schedule
  - schedule je konfliktně serializovatelný, pokud je jeho prec. graph acyklický





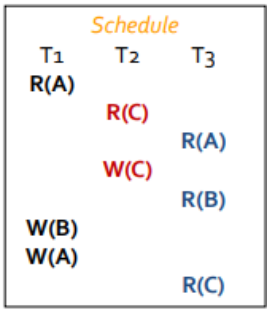
(Ne)obnovitelný schedule

- rozšiřujeme transakční model o ABORT, které přináší další nebezpečí – **unrecoverable sch**.
- jedna transakce se přeruší, takže musí být provedeny undo pro každý zápis, ale to nelze pro již commitnuté transakce, které četly změny způsobené přerušenou transakcí
- durability vlastnost ACID
- v rec. sch. je transakce T commitnuta po všech ostatních transakcích, které ovlivnily T commit (tj. změnily data, které pak četla T)
- je-li čtení změněných dat povoleno jen pro commitnuté transakce, vyhýbáme se též kaskádnímu přerušení transakcí

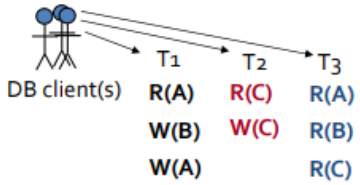


Locking protocols

- transakční scheduler pracuje pod nějakým protokolem, který umožňuje garanci ACID vlastností a maximální propustnost



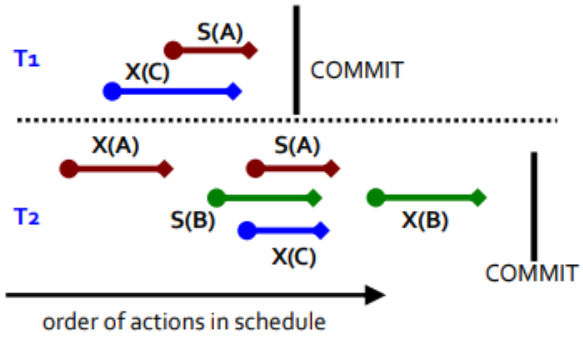
- **pesimistický dohled** (mnoho souběžných zatížení)
  - zamykací protokoly, time stamps
- **optimistický dohled** (nemnoho souběžných zatízení)
- Proč protokol?
  - scheduler nemůže vytvořit celý schedule dopředu
  - plánování je vytvářeno v lokálním časovém kontextu - dynamické provádení transakcí, větvení v kódu



- zamykání DB entit může být použito pro určování pořadí čtení a zápisů, a tedy k zabezpečení konfliktní serializovatelnosti
- **exkluzivní zámky**
  - **X(A)** zamkne A, takže R/W A jsou povoleny jen vlastníkovi zámku
  - může být uděleno jen jedné transakci
- **sdílené zámky**
  - **S(A)** – povolení jen čtení A
  - může být uděleno (sdíleno) několika transakcím
- odemykání pomocí **U(A)**
- je-li pro transakci potřeba zámek, který není dostupný, provedení transakce je pozastaveno a čeká na uvolnění zámku
  - ve schedule je zapsán požadavek zámku, následující prázdnými řádky čekání
- (ode/za)mykací kód je přidán pomocí transaction schedulera

Example: schedule with locking

T1	T2
	X(A)
X(C)	
	W(A)
R(C)	
W(C)	
	U(A)
S(A)	
	S(B)
R(A)	
	R(B)
U(C)	
	X(C)
U(A)	
	S(A)
COMMIT	
	W(C)
	R(A)
	U(B)
	U(C)
	U(A)
	X(B)
	W(B)
	U(B)
	COMMIT

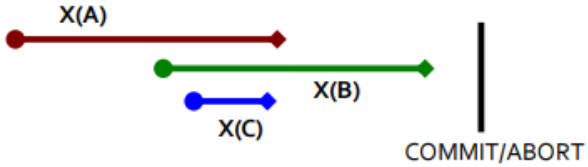


2PL (Two-phase locking protocol)

- řídí se dvěma pravidly pro vytváření schedule:
  - chce-li transakce číst (či psát) transakci A, musí nejdříve získat sdílený (či exkluzivní) zámek na A
  - transakce nemůže žádat o zámek, pokud již jeden uvolnila (nehledě na zamknutou entitu)

- dvě fáze – locking a unlocking

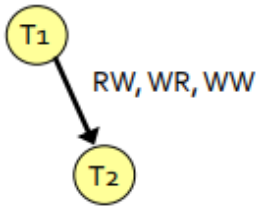
  Example: 2PL adjustment of the second transaction in the previous schedule



- 2PL restrikce schedule zajišťuje, že prioritní graf je acyklický, tj. schedulle je konfliktně serializovatelný
- 2PL NEgarantuje obnovitelnost schedule

  Example: 2PL-compliant schedule, but not recoverable, if T1 aborts

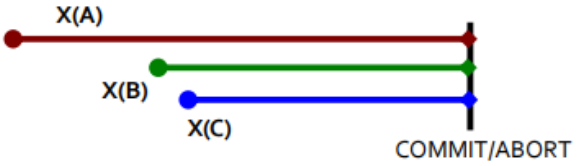
T1	T2
X(A)	
R(A)	
W(A)	
U(A)	
	X(A)
	R(A)
	A := A *1.01
	W(A)
	X(B)
	U(A)
	R(B)
	B := B *1.01
	W(B)
	U(B)
	COMMIT
ABORT / COMMIT	



Striktní 2PL

- striktní 2PL protokol dělá druhé pravidlo 2PL silnějším, takže pravidla jsou nyní:
  - chce-li transakce číst (či psát) transakci A, musí nejdříve získat sdílený (či exkluzivní) zámek na A
  - všechny zámky jsou uvolněny po skončení transakce

Example: strict 2PL adjustment of second transaction in the previous example



Insertions of U(A) are not needed (implicit at the time of COMMIT/ABORT).

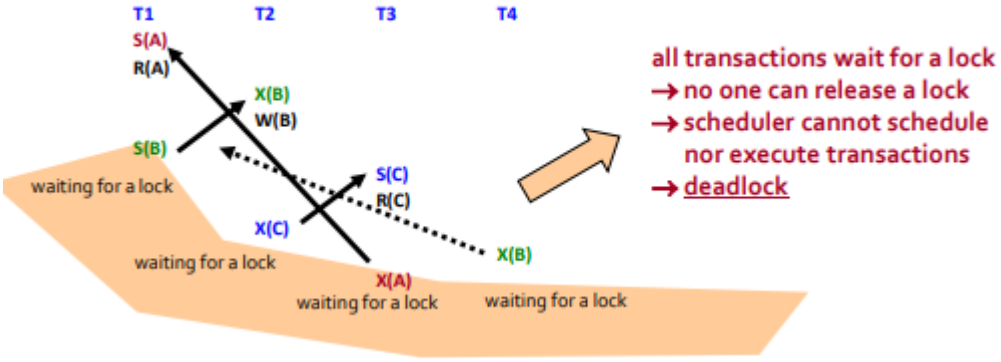
- zajišťuje i obnovitelnost schedule a vyhýbá se kaskádním abortům

Example: schedule built using strict 2PL

Deadlock

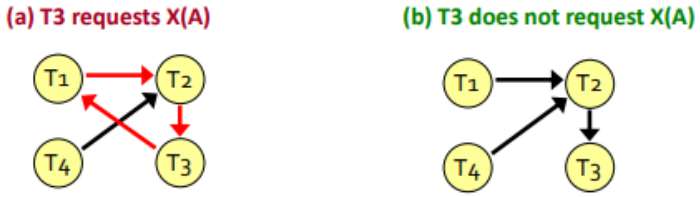
- během vykonávání transakce se může stát, že transakce T_1 požádá o zámek, který byl již slíben T_2, ale T_2 jej nemůže uvolnit, protože čeká na jiný zámek držený T_1
- může být zobecněno na více transakcí: T_1 čeká na T_2, T_2 čeká na T_3,…, T_n čeká na T_1
- striktní 2PL nemůže zabránit deadlockům (nemluvě o slabších protokolech)

Example:



- **detekce** – deadlock může být detekován opakovanou kontrolou waits-for grafu
  - waits-for graf je dynamický graf zachytývající čekání transakcí na zámky
  - nody jsou aktivní transakce
  - hrany označují čekání transakcí na zámky držené ostatními transakcemi
  - cyklus ve grafu = deadlock

Example: waits-for graph for the previous example



- deadlocky ale nejsou moc časté, takže **řešení** může být jednoduché
  - přerušit čekající transakci a restartovat ji (uživatel nezaznamená)
  - testovat waits-for graf – pokud nastane deadlock, přerušit a restartovat transakci v cyklu
    - přeruší se taková transakce, která:
      - drží nejméně zámků
      - vykonala nejméně práce
      - je daleko od dokončení
    - přerušená transakce již není přerušena, nastane-li další deadlock

- lze jim i **zabránit** (**prioritizací**)
  - každá transakce má prioritu (např. timestamp)
  - pokud T1 požádá o zámek držený T2, lock manager zvolí mezi dvěma strategiemi
    - **wait-die** – má-li T1 vyšší prioritu, může počkat; pokud ne, je přerušena a rest.
    - **wound-wait** – má-li T1 vyšší prioritu, T2 je přerušena; jinak T1 čeká

- **Coffmanovy podmínky**
  - deadlocky mohou nastat, pokud jsou SOUČASNĚ splněny VŠECHNY podmínky:
    - **mutual exclusion** – zdroje může používat v jednom okamžiku jeden proces
    - **resource holding (hold & wait)** – proces může žádat o další prostředky, i když už nějaké byly přiděleny
    - **no preemption** – zdroje mohou být uvolněny jen dobrovolně
    - **circular wait** – transakce mohou žádat a čekat na zdroje v cyklech

Phantom

- uvažujme nyní dynamickou databázi (s inserty a delete)
- pokud jedna transakce pracuje s nějakou množinou datových entit, zatímco jiná tuto množinu mění (insert/delete), mohlo by to vést k nekonzistenci v DB (inserializable schedule)

- **Proč?**
  - T1 zamkne všechny entity, které jsou v danou chvíli relevantní
    - např. splnění nějaké WHERE podmínky SELECT příkazu
  - během vykonávání T1 by mohla nová transakce T2 logicky rozšířit množinu entit
    - tj. v daný moment by množství zámků definovaných WHERE mohlo být větší
    - takže nějaké entity jsou zamklé a některé ne

T1: find the oldest male and female employees
(**SELECT * FROM** Employees ...) + **INSERT INTO** Statistics ...
T2: insert new employee Phill and delete employee Eve (employee replacement)
(**INSERT INTO** Employees ..., **DELETE FROM** Employees ...)

Initial state of the database: **{[Peter, 52, m], [John, 46, m], [Eve, 55, f], [Dana, 30, f]}**

T1	T2
lock men, i.e.,	
S(Peter)	
S(John)	
M = max(R(Peter), R(John))	
	Insert(Phill, 72, m)
	X(Eve)
	Delete(Eve)
	COMMIT
lock women, i.e.,	
S(Dana)	
F = max(R(Dana))	
Insert(M, F) // result is inserted into table Statistics	
COMMIT	

phantom
a new male employee can be inserted, although **all men** should be locked

Although the schedule is strict 2PL compliant, the result [Peter, Dana] is not correct as it does not follow the serial schedule T1, T2, resulting in [Peter, Eve], nor T2, T1, resulting [Phill, Dana].

- **prevence:**
  - neexistují-li indexy, vše relevantní musí být zamknuto
    - např. celá tabulka či dokonce několik tabulek
  - existují-li indexy (např. B^+-stromy) na entitách definovaných lock-podmínkou, je možné „číhat“ na fantoma v indexové úrovni – **index locking**
    - externí pokus o modifikaci množiny je identifikován aktualizovanými index zámky
    - jelikož index většinou udržuje jen jeden atribut, aplikace je limitovaná
  - zobecněním index locking je **predicate locking**, kdy zámky jsou požadovány pro logické množiny, ne pro jednotlivé instance dat
    - to je ale těžké implementovat

Optimistické (nezamykací) protokoly

- pokud současně vykonávané transakce nejsou často v konfliktu (nesoupeří o zdroje), locking overhead je zbytečně velký

- 3-fázový optimistický protokol
  - **čtení** – transakce čte data z DB, ale zapisuje do soukromého lokálního datového pr.
  - **validace** – chce-li transakce commitnout, přepošle to soukromý datový pr. do transaction managera (tj. požadavek na update DB)
    - TM rozhodne, jestli je update v konfliktu s dalšími transakcemi
      - pokud ano, transakce je přerušena a restartována
      - pokud ne, nastává třetí fáze
  - **zápis** – soukromý datový prostor je zkopírován do databáze