Adatbázisok elmélete

Tranzakciókezelés

Katona Gyula Y.

Számítástudományi és Információelméleti Tanszék
Budapesti Műszaki és Gazdaságtudományi Egyetem

Tranzakciókezelés

Eddig hallgatólagosan feltettük, hogy

- egy felhasználó van csak
- a lekérdezések/módosítások hiba nélkül lefutnak

A valóságban ez nincs így, két nagyobb gond is lehet, aminek kezelése a tranzakciókezelő dolga:

- **Többfelhasználós működés**: egyidejű hozzáférést kell biztosítani több felhasználónak, de úgy, hogy az adatbázis konzisztens maradjon (pl. banki rendszerek, helyfoglalás)
- **Rendszerhibák utáni helyreállítás**: ha a külső tár megmarad, de a belső sérül (vagy egyszerűen csak nem fut le valami) és emiatt az adatbázis inkonzisztens állapotba kerül, akkor újra konzisztens állapotba kell hozni (vagy visszacsinálni valamit, vagy befejezni valamit)

Ez két (néha egymással is ellentétes) kívánság, de az alapeszköz ugyanaz lesz: a **tranzakció**.

Tranzakciók MySQL-ben

```
START TRANSACTION [WITH CONSISTENT SNAPSHOT];
  SELECT ...;
  INSERT ...;
  UPDATE ...;
  ...;
COMMIT;
```

Többfelhasználós működés

A lekérdezésfeldolgozó a magas szintű utasításokból álló lekérdezéseket/módosításokat elemi utasításokra bontja, (pl: olvass ki valahonnan valamit, írj be valahova valamit, számolj valamit). Egy felhasználó egy lekérdezése/módosítása ilyen elemi utasítások sorozatává alakul.

1. felhasználó: $u_1, u_2, \ldots, u_{10}$
2. felhasználó: $v_1, v_2, \ldots, v_{103}$

De ez a két utasítássorozat nem elkülönülve jön, hanem összefésülődnek:

$$u_1, v_1, v_2, u_2, u_3, v_3, \ldots, v_{103}, u_{10}$$

A saját sorrend megmarad mindkettőn belül, de amúgy össze vannak keveredve, így lesz lehetséges a több felhasználó egyidejű kiszolgálása. Ebből viszont baj származhat, mert olyan állapot is kialakulhat, ami nem jött volna létre, ha egymás után futnak le a tranzakciók.

Példa

1. felhasználó: READ A, $A++$, WRITE A
2. felhasználó: READ A, $A++$, WRITE A

Ha ezek úgy fésülődnek össze, hogy

$$(\text{READ } A)_1, (\text{READ } A)_2, (A++)_1, (A++)_2, (\text{WRITE } A)_1, (\text{WRITE } A)_2$$

akkor a végén csak eggyel nő A értéke, holott kettővel kellett volna.

Rendszerhibák

Ha rendszerhiba van (a belső memória meghibásodik) vagy csak ABORT van (a tranzakciókezelő ütemező része kilő egy alkalmazást futás közben), akkor emiatt félbemaradhat valami, aminek nem lenne szabad.

Példa: átutalunk egyik helyről a másik helyre pénzt:

$$A := A - 50 \quad B := B + 50$$

Ha az a közepén meghal: hibás állapot jön létre.

Tranzakció

Alapfogalom mindkét problémakör megoldásában a **tranzakció**: egy felhasználóhoz tartozó elemi utasítások olyan sorozata, melynek fő jellemzője az **atomiság** (**A**tomicity): vagy az összes utasításnak végre kell hajtódnia vagy egynek sem szabad. Ez lesz az egyik dolog, amit mindenáron el akarunk majd érni.

További elvárások:

- **konzisztencia**, **C**onsistency: az adatbázis konzisztens állapotok között mozog, (hogy mit jelent a konzisztens, az a valóságtól függ, pl. banki összegek stimmelése), nem konzisztens állapot csak ideiglenesen állhat fenn (a rendszerhibák utáni helyreállításnál lesz ez fontos)
- **elkülönítés**, **I**solation: több tranzakció egyidejű futása után úgy kell kinéznie az adatbázisnak, mintha a tranzakciók nem lettek volna összefésülve (az ütemező dolga lesz ennek biztosítása)
- **tartósság**, **D**urability: a befejezett tranzakciók hatása nem veszhet el

Többfelhasználós működés, alapfogalmak

Cél: párhuzamos hozzáférés biztosítása, de úgy, hogy a konzisztencia megmaradjon
Feltételezzük, hogy ha a tranzakciók egymás után, elkülönítve futnak, akkor konzisztens állapotból konzisztens állapotba jut a rendszer. Csak azokat az összefésülődéseit akarjuk megengedni a tranzakcióknak, amelyeknek a hatása ekvivalens valamelyik izolálttal.

ütemezés: egy vagy több tranzakció műveleteinek valamilyen sorozata (fontos, hogy a tranzakciókon belüli sorrend megmarad)

soros ütemezés: olyan ütemezés, amikor a különböző tranzakciók utasításai nem keverednek, először lefut az egyik összes utasítása, aztán a másiké, aztán a harmadiké, . . .

sorosítható ütemezés: olyan ütemezés, amelynek hatása azonos a résztvevő tranzakciók **valamely** soros ütemezésének hatásával (azaz a végén minden érintett adatelem pont úgy néz ki, mint a soros ütemezés után)

Sorosíthatóság

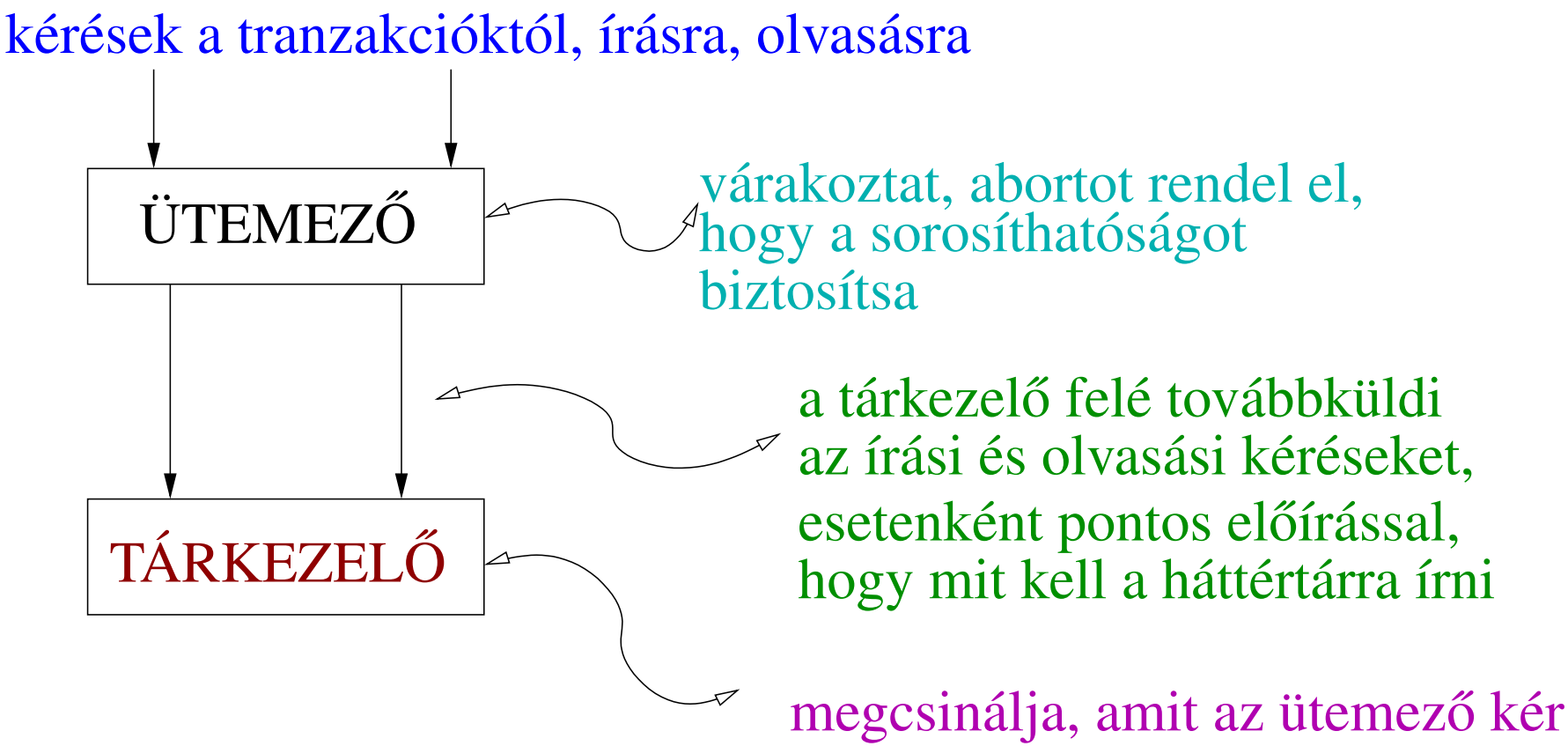
Megjegyzés: Az mindegy, hogy melyik soros ütemezéssel lesz ekvivalens a sorosítható ütemezés. Mivel a soros ütemezésekről feltettük, hogy jók, ezért ha valamelyikkel ekvivalens, az már elég.

Cél: olyan sorrend (összefésülődés) kikényszerítése, ami sorosítható ütemezés

Módszer: az ütemező (az adatbáziskezelő része) felelős azért, hogy csak ilyen sorrendek legyenek. Figyeli a tranzakciók műveleteit és késleltet/ABORT-ál tranzakciókat. (Nemsokára részletesebben is nézzük.)

A tárkezelővel való együttműködés

Az előbbiek miatt az ütemező és a tárkezelő szorosan együttműködnek:



Az ütemező eszközei a sorosíthatóság elérésére

Az ütemezőnek több lehetősége is van arra, hogy kikényszerítse a sorosítható ütemezéseket:

- zárak (ezen belül is még: protokoll elemek, pl. 2PL)
- időbélyegek (time stamp)
- érvényesítés

Fő elv lesz: inkább legyen szigorúbb és ne hagyjon lefutni egy olyat, ami sorosítható, mint hogy fusson egy olyan, aki nem az.
Mindegyik technikára igaz lesz, hogy biztosra megy, azaz olyanokat is ki fog lőni, amik sorosíthatók lennének.

Példa

T_1	T_2	A	B
		x	y
Read(A,t)			
$t := t + 100$			
Write(A,t)		$x + 100$	
	Read(A,s)		
	$s := 2 \cdot s$		
	Write(A,s)	$2 \cdot (x + 100)$	
Read(B,t)			
$t := t + 100$			
Write(B,t)			$y + 100$
	Read(B,s)		
	$s := 2 \cdot s$		
	Write(B,s)		$2 \cdot (y + 100)$

A táblázat baloldali részén azt jelezzük, hogy milyen műveleteket végeznek a tranzakciók, a jobboldalon pedig az látszik, hogy eközben mi történik az A és B adategységekkel. Ezek kezdeti értékei x és y.

Read(A,t)= olvassuk be A értékét a t változóba
Write(A,t)= írjuk ki a t változó értékét A-ba

Látható, hogy ez nem egy soros ütemezés, mert össze vannak fésülődve a két tranzakció utasításai.
Viszont sorosítható, mert a hatása A-n és B-n is azonos a $T_1\,T_2$ soros ütemezés hatásával, (x, y)-ból $(2 \cdot (x + 100), 2 \cdot (y + 100))$ lesz.

Példa nem sorosíthatóra

T_1	T_2	A	B
		x	y
Read(A,t)			
$t := t + 100$			
Write(A,t)		$x + 100$	
	Read(A,s)		
	$s := 2 \cdot s$		
	Write(A,s)	$2 \cdot (x + 100)$	
	Read(B,s)		
	$s := 2 \cdot s$		
	Write(B,s)		$2 \cdot y$
Read(B,t)			
$t := t + 100$			
Write(B,t)			$2 \cdot y + 100$

Ez nem egy sorosítható ütemezés, mert se a $T_1 T_2$ soros ütemezés, se a $T_2 T_1$ soros ütemezés hatása nem az, hogy (x, y)-ból $(2 \cdot (x + 100), 2 \cdot y + 100)$ lesz.
A $T_1 T_2$ ütemezés $(2 \cdot (x + 100), 2 \cdot (y + 100))$ eredményt ad,
a $T_2 T_1$ pedig $(2 \cdot x + 100, 2 \cdot y + 100)$-t.

Egyszerűsítések

Ha ismert, hogy mikor és mit akarnak írni és olvasni a tranzakciók és még az is ismert, hogy pontosan mit számolnak, akkor minden esetben el tudjuk dönteni, hogy egy ütemezés sorosítható-e.
A gyakorlatban azonban nem vizsgáljuk meg ennyire alaposan a történéseket, (mert pl. nem is tudnánk vagy mert az macerás), hanem az alábbi egyszerűsítésekkel dolgozunk:

- Nem vizsgáljuk meg, hogy mit számolnak a tranzakciók, hanem feltételezzük a legrosszabbat: valami olyat csinálnak a beolvasott adattal, ami teljesen egyedi. Azaz, feltesszük, hogy ha tud olyat csinálni, amitől inkonzisztens lesz a DB (az ütemezés hatása nem lesz azonos valamelyik soroséval), akkor azt teszi. $\Longrightarrow$
- Csak az írásokat és olvasásokat tartjuk nyilván, ezek alapján döntünk arról, hogy egy ütemezést sorosíthatónak tekintünk-e. Ha csak egyetlen olyan lehetséges számolás is van, amivel az írásokból és olvasásokból álló ütemezés nem sorosítható, akkor nem tekintjük sorosíthatónak.
- Ez néha kilő persze olyan ütemezéseket is, amik (ha megnéznénk a belső számolásokat is, akkor) sorosíthatók lennének, de ez nem baj.

Példák

A korábban látott két ütemezés átírva úgy, hogy a számolások ne látszódjanak:

T_1	T_2
r(A)	
w(A)	
	r(A)
	w(A)
r(B)	
w(B)	
	r(B)
	w(B)

T_1	T_2
r(A)	
w(A)	
	r(A)
	w(A)
	r(B)
	w(B)
r(B)	
w(B)	

r(A) jelentése beolvassuk A-t; w(A) jelentése kiírjuk A-t

Látszik, hogy az első esetben bármilyen számolást is csinálnak a tranzakciók a beolvasott adattal a kiírás előtt, a számolástől függetlenül ugyanaz lesz a hatás mint a $T_1\,T_2$ soros ütemezésnél.
A második esetben, ahogy már láttuk is, lehetséges olyan számolás, ami esetén nem lesz azonos a hatás semelyik sorossal, így ez a csak írásokat és olvasásokat tartalmazó ütemezés nem sorosítható. (Persze lehetséges olyan számolás, amivel kiegészítve sorosítható lenne, de most kegyetlenek vagyunk: ha van egy olyan, amivel rossz, akkor rossz.)

Jelölés: A táblázat helyett így fogjuk az ütemezéseket megadni (a két előbbi esetben például):

$r_1(A),\ w_1(A),\ r_2(A),\ w_2(A),\ r_1(B),\ w_1(B),\ r_2(B),\ w_2(B)$

illetve

$r_1(A),\ w_1(A),\ r_2(A),\ w_2(A),\ r_2(B),\ w_2(B),\ r_1(B),\ w_1(B)$

Feltevések még

Általános elv (ahogy az előbb már ki is derült), hogy inkább legyünk szigorúak és minősítsünk rossznak egy olyat, ami sorosítható lenne, ha jobban megnéznénk, mint hogy sorosíthatónak mondjunk egy olyat, ami esetleg nem az $\Longrightarrow$ mindig egy erősebb feltételt fogunk tesztelni, aki ezt is túléli az biztos sorosítható.
Általában nem egy már adott ütemezésről kell eldönteni, hogy az sorosítható-e, hanem olyan technikákat, protokollokat használunk, amikkel elérjük, hogy csak sorosítható ütemezések jöjjenek létre.

Sorosíthatóság biztosítása zárakkal

Elve: A tranzakciók zárolják azokat az adatelemeket, amivel dolgoznak, és amíg valami zár alatt van, addig a többi tranzakció nem, vagy csak korlátozottan fér hozzá.

Egyszerű tranzakciómodell

Csak egyféle zárkérés van (LOCK), mindegyik művelethez ezt a zárat kell megkapni. Ezen kívül van még zárelengedés (UNLOCK). Az ütemezésekben nem csak írás és olvasás lesz, hanem a zárkérések és zárelengedések is benne lesznek. Csak olyan zárkéréseket tartalmazó ütemezéseket akarunk majd megengedni, amik eleget tesznek néhány követelménynek.
A legális ütemezés jellemzői:

1. Az i-edik tranzakció, T_i, csak akkor olvashatja vagy írhatja az A adategységet, ha előtte zárat kért és kapott rá ($LOCK_i(A)$) és a zárat még azóta nem engedte fel (nem volt még azóta $UNLOCK_i(A)$).
2. Ha T_i zárolja az A adategységet, akkor később valamikor el is kell engednie a zárat ($LOCK_i(A)$ után mindig van $UNLOCK_i(A)$).
3. Egyszerre két különböző tranzakciónak nem lehet zárja ugyanazon az adategységen.

Példa

Példa legális zárkérésre, ütemezésre ebben a modellben:

$$l_1(A),\ r_1(A),\ w_1(A),\ u_1(A),\ l_2(A),\ r_2(A),\ w_2(A),\ u_2(A),$$

$$l_1(B),\ r_1(B),\ w_1(B),\ u_1(B),\ l_2(B),\ r_2(B),\ w_2(B),\ u_2(B),$$

Példa arra, hogy hogyan dolgozhat az ütemező azon az egyszerű tranzakciómodellben, hogy legális ütemezés alakuljon ki

Tegyük fel, hogy a következő sorrendben jönnek zárkérések és műveleti kérések az ütemezőhöz (két tranzakció van):

$l_1(A),\ r_1(A),\ w_1(A),\ l_1(B),\ u_1(A),\ l_2(A),\ r_2(A),\ w_2(A)$

Eddig minden rendben van, minden kérést teljesíteni lehet. Ha azonban a további kérések

$l_2(B),\ u_2(A),\ r_2(B),\ w_2(B),\ u_2(B),\ r_1(B),\ w_1(B),\ u_1(B)$

akkor ez már így nem mehet, mert T_2 nem kaphatja meg a kért zárat B-n, hiszen T_1 még tartja.

Emiatt az ütemező késlelteti T_2-t (T_2 vár T_1-re) és előbb engedi futni T_1-et, aztán jöhet T_2:

$r_1(B),\ w_1(B),\ u_1(B),\ l_2(B),\ u_2(A),\ r_2(B),\ w_2(B),\ u_2(B)$

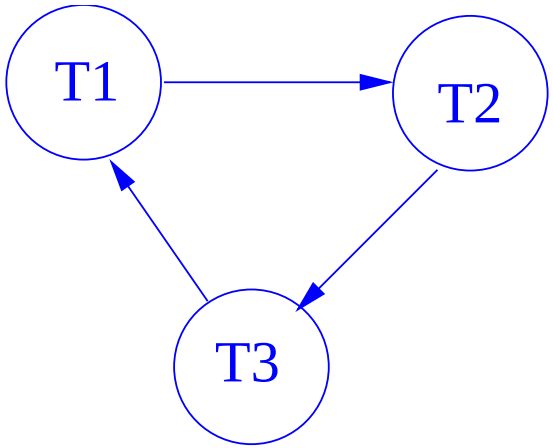
lesz az az ütemezés, ami le fog futni, ez már legális lesz.

Holtpont

Láttuk, hogy az ütemező úgy kényszeríti ki a legális ütemezést, hogy várakoztatja a tranzakciókat. Ebből problémák lehetnek, ha a tranzakciók egymásra várnak:
Holtpont (deadlock, patt): néhány zárkérés után akkor van holtpont, ha van egy olyan részhalmaza a tranzakcióknak, akik közül egyik se tud tovább futni, mert vár egy szintén ebben a részhalmazban levő másikra (vár egy olyan zár elengedésére, amit egy másik, ebbe a részhalmazba tartozó, tranzakció tart).
Például:
$l_1(A),\ l_2(B),\ l_3(C),\ l_1(B),\ l_2(C),\ l_3(A)$
sorrendben érkező zárkérések esetén egyik tranzakció se tud tovább futni.
Az ilyen helyzeteket el kell kerülni, illetve ha már kialakultak, akkor fel kell ismerni és meg kell szüntetni.

Várakozási gráf

A felismerésben segít a zárkérések sorozatához tratozó **várakozási gráf**: csúcsai a tranzakciók és akkor van él T_i-ből T_j-be, ha T_i vár egy olyan zár elengedésére, amit T_j tart éppen.
Például az előbbi, holtponthoz vezető zárkéréssorozat várakozási gráfja a hat zárkérés után:



Vegyük észre, hogy a várakozási gráf változik az ütemezés során, ahogy újabb zárkérések érkeznek vagy zárelengedések történnek.

Holtpont felismerése

A várakozási gráf segítségével fel lehet ismerni a holtpontot az alábbi tétel miatt:

Tétel

Az ütemezés során egy adott pillanatban pontosan akkor nincs holtpont, ha az adott pillanathoz tartozó várakozási gráf DAG (nincs benne irányított kör).

Bizonyítás.

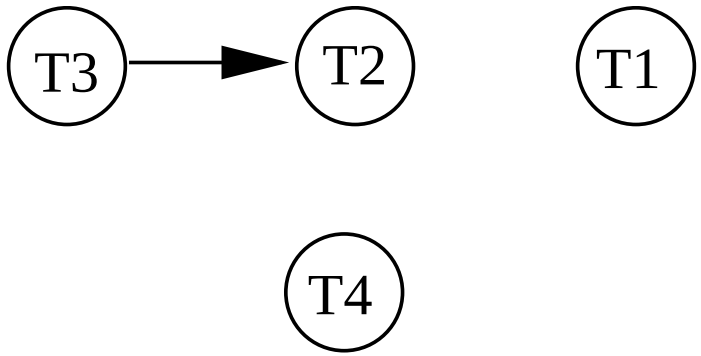
$\Rightarrow$: Ha van irányított kör a várakozási gráfban, akkor a körbeli tranzakciók egyike se tud lefutni, mert vár a mellette levőre. Ez holtpont.
$\Leftarrow$: Ha a gráf DAG, akkor van topológikus rendezése a tranzakcióknak és ebben a sorrendben le tudnak futni a tranzakciók. (Az első nem vár senkire, mert nem megy belőle ki él, így lefuthat; ezután már a másodikba se megy él, az is lefuthat . . .) □

Példa

Nézzük az alábbi írásokból és olvasásokból álló ütemezést:

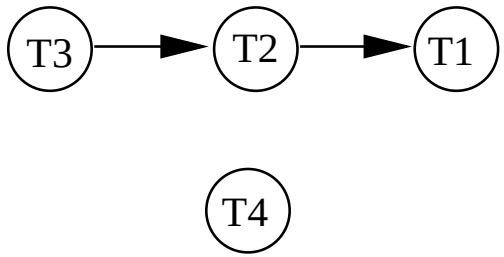
$$r_1(A),\ \ r_2(B),\ \ w_1(C),\ \ r_3(D),\ \ r_4(E),\ \ r_3(B),\ \ w_2(C),\ \ w_4(A),\ \ w_1(D)$$

Tegyük fel, hogy a zárkérések mindig közvetlenül megelőzik a műveletet, a zárelengedések pedig a tranzakciók végén, egyszerre történnek. Hogyan alakul a várakozási gráf ezen sorozat esetén? Lesz-e valamikor holtpont?
Az elején $l_1(A),\ \ r_1(A),\ \ l_2(B),\ \ r_2(B),\ \ l_1(C),\ \ w_1(C),\ \ l_3(D),\ \ r_3(D),\ \ l_4(E),\ \ r_4(E)$ zárkérések és műveletek vannak, eddig még senki nem vár senkire.
Ezután $l_3(B)$ jön $r_3(B)$ miatt, de T_3-nak várnia kell T_2-re:

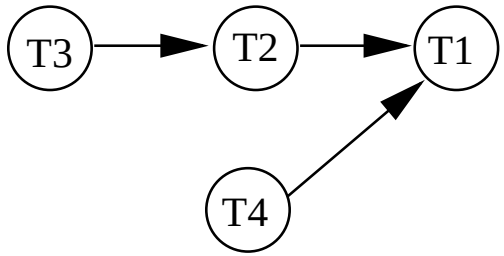


Példa

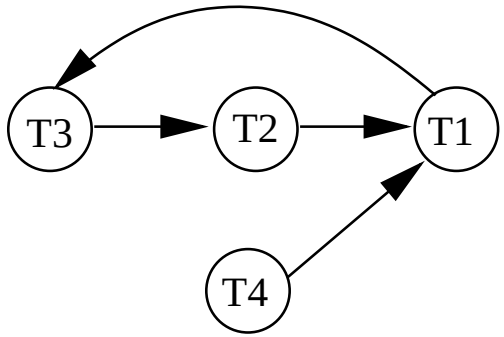
Ezután $l_2(C)$ jön $w_2(C)$ miatt, de T_2-nek is várnia kell T_1-re:



Ezután $l_4(A)$ jön $w_4(A)$ miatt, de T_4-nek is várnia kell T_1-re:



Végül $l_1(D)$ jön $w_1(D)$ miatt, de T_1-nek meg T_3-ra kell várnia:



És ez már holtpont: van irányított kör a gráfban.

Megoldások holtpont ellen

1. Rajzoljuk folyamatosan a várakozási gráfot és ha holtpont alakul ki, akkor ABORT-áljuk az egyik olyan tranzakciót, aki benne van a kialakult irányított körben. Ez egy megengedő megoldás (optimista), hagyja az ütemező, hogy mindenki úgy kérjen zárat, ahogy csak akar, de ha baj van, akkor erőszakosan beavatkozik. Az előző példa esetében mondjuk kilövi T_2-t, ettől lefuthat T_3, majd T_1 és T_4 is.
2. Pesszimista hozzáállás: ha hagyjuk, hogy mindenki össze-vissza kérjen zárat, abból baj lehet. Előzzük inkább meg a holtpont kialakulását valahogyan. Lehetőségek:

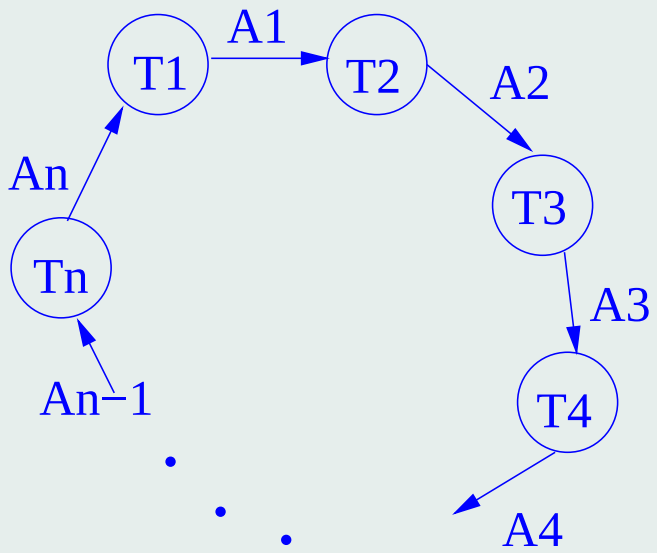
(a) Minden egyes tranzakció előre elkéri az összes zárat, ami neki kelleni fog. Ha nem kapja meg az összeset, akkor egyet se kér el, el se indul.
Ilyenkor biztos nem lesz holtpont, mert ha valaki megkap egy zárat, akkor le is tud futni, nem akad el. Az csak a baj ezzel, hogy előre kell mindent tudni.

(b) Feltesszük, hogy van egy sorrend az adategységeken és minden egyes tranzakció csak eszerint a sorrend szerint növőleg kérhet újabb zárakat, azaz ha egy adategységre kért már zárat, akkor kisebb sorszámúra már nem kérhet később. Itt lehet, hogy lesz várakozás, de holtpont biztos nem lesz.

Megoldások holtpont ellen

Bizonyítás.

Ha valamely pillanatban lenne irányított kör a várakozási gráfban:



ahol T_i vár T_{i+1}-re az A_i adategység miatt, akkor $A_1 < A_2 < A_3 < \ldots A_n < A_1$ áll fenn abban az esetben, ha mindegyik tranzakció betartotta, hogy növőleg kér zárat. Ez azonban ellentmondás. □

Tehát ez a protokoll is megelőzi a holtpontot, de itt is előre kell tudni, hogy milyen zárakat fog kérni egy tranzakció.

Még egy módszer, ami szintén optimista, mint az első:

Időkorlát alkalmazása: ha egy tranzakció kezdete óta túl sok idő telt el: ABORT.

Ehhez az kell, hogy ezt az időkorlátot jól tudjuk megválasztani.

Zárak és sorosíthatóság

Eddig azt láttuk csak, hogy mennyi baj lehet a zárak alkalmazásával (holtpont, éhezés). Most nézzük, hogy mire jók a zárak.
A zárak segítségével el lehet majd érni, hogy az ütemezések sorosíthatók legyenek, de pusztán az, hogy használjuk a zárakat, még nem ad sorosítható ütemezést.
Példa olyan legális, zárakat használó ütemezésre, ami nem sorosítható: a korábbi, nem sorosítható, írásokból és olvasásokból álló ütemezésbe zárakat rakunk:

$$l_1(A),\ r_1(A),\ w_1(A),\ u_1(A),\ l_2(A),\ r_2(A),\ w_2(A),\ u_2(A),$$
$$l_2(B),\ r_2(B),\ w_2(B),\ u_2(B),\ l_1(B),\ r_1(B),\ w_1(B),\ u_1(B)$$

Sorosíthatóság és zárak

Zárakat használunk, figyelünk arra, hogy legális legyen az ütemezés, és még figyelünk valamire, ami biztosítja a sorosíthatóságot.
Egyszerű tranzakció modellben vagyunk (egy fajta zár van csak és a korábbi három feltevés mindig fennáll, azaz a zárkérés legális) és még valamit felteszünk:
A sorosíthatóságról pusztán a zárkérések alapján fogunk dönteni, nem nézzük azt, hogy ezeken kívül milyen műveletek (írások/olvasások) vannak. Pontosabban:
Nem foglalkozunk azzal, hogy $LOCK_i(A)$ és $UNLOCK_i(A)$ között mi történik, feltesszük hogy valami teljesen egyedi írás és olvasás is. Ez hasonlít ahhoz a helyzethez, mint amikor a konkrét számolásokat elhanyagoltuk: feltesszük, hogy a lehető legrosszabb történik azalatt, amíg a tranzakciónál van a zár.
Így persze megint igaz lesz az, hogy olyan ütemezéseket is rossznak minősítünk, amik igazából sorosíthatók lennének, ha megnéznénk, hogy írások vagy olvasások történnek, de ez nem baj, mert szigorúbbak lehetünk, csak az a fontos, hogy olyan ne legyen sorosíthatónak minősítve, aki nem az.

Sorosítási gráf az egyszerű tranzakciómodellben

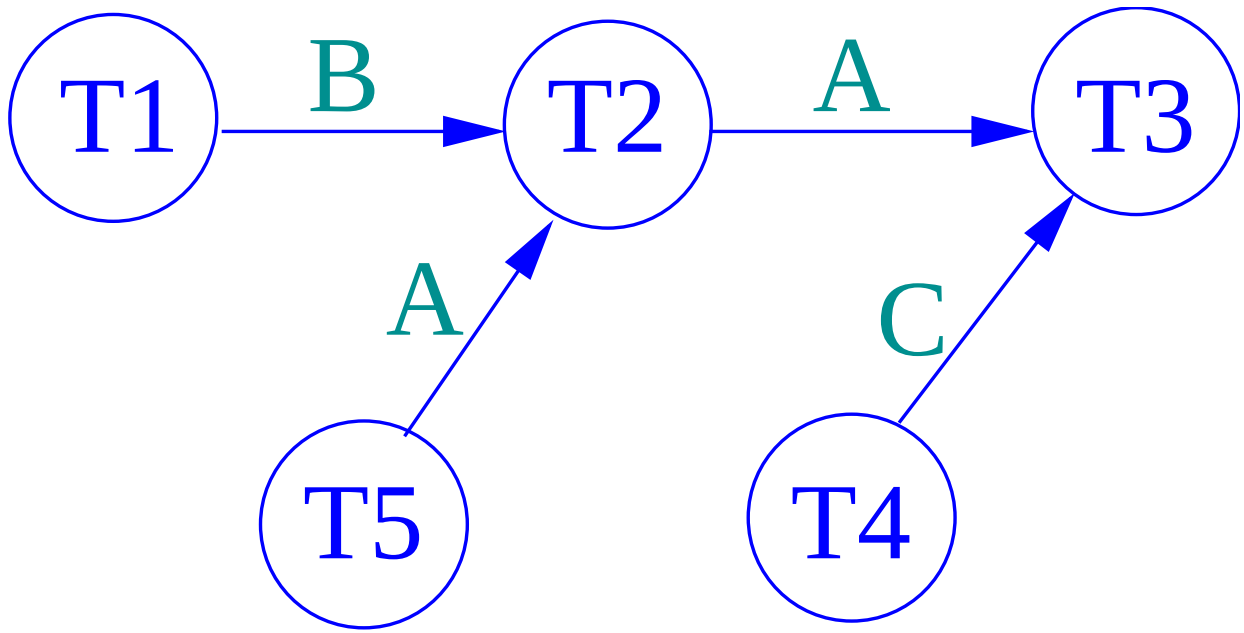
Az előbbiek értelmében tehát egy olyan legális ütemezésről akarjuk eldönteni, hogy sorosítható-e, amiben csak zárkérések és zárelengedések vannak.
Mikor lesz egy ilyen ütemezés sorosítható, függetlenül attól, hogy milyen írások és olvasások történnek valójában?
Ennek megválaszolásában segít a **sorosítási gráf**: csúcsai a tranzakciók és akkor van él T_i-ből T_j-be, ha az ütemezésben van olyan $u_i(A) \quad \ldots \quad l_j(A)$ rész, ahol $u_i(A)$ (T_i elengedi A zárját) és $l_j(A)$ (T_j megkapja A zárját) között A-ra senki se kap zárat.
Ekkor minden olyan soros ütemezésben, ami ekvivalens lehet a miénkkel, biztos, hogy T_j-nek T_i után kell jönnie.
Ez azért van így, mert feltettük, hogy T_i is és T_j is bármit csinálhat A-val, amíg nála van a zár és ha pl. T_i írja, T_j meg olvassa A-t, akkor már csak a $T_i, \ldots, T_j$ sorrend lesz a jó.

Példa sorosítási gráfra

Az alábbi, csak zárkéréseket és zárelengedéseket tartalmazó ütemezés legális (HF: leellenőrizni):

$$l_5(A),\ l_1(B),\ u_5(A),\ l_4(C),\ u_1(B),\ l_2(A),\ l_2(B),\ u_2(A),$$
$$l_3(A),\ u_3(A),\ u_4(C),\ u_2(B),\ l_3(C),\ u_3(C)$$

Az ehhez tartozó sorosítási gráf:



Tétel a sorosítási gráfról

Tétel

Egy csak zárkéréseket és zárelengedéseket tartalmazó egyszerű tranzakció modellbeli ütemezés pontosan akkor sorosítható, ha az előbbi módszerrel felrajzolt sorosítási gráf DAG.

Bizonyítás.

$\Rightarrow$: Ha nem DAG a gráf, akkor van benne irányított kör. Ebben a körben levő tranzakciók közül egyik sem előzheti meg a többit egy ekvivalens soros ütemezésben, amiből következik, hogy nincs ekvivalens soros ütemezés.
$\Leftarrow$: Teljes indukcióval: $n = 1$-re (1 tranzakció van csak) világos, egy ilyen ütemezés maga soros.
Legyen most az ütemezésben n tranzakció. Ha a gráf DAG, akkor létezik topologikus rendezése. Azt látjuk be, hogy a topologikus sorrend szerinti soros ütemezés ekvivalens lesz az eredeti ütemezéssel.
Ha T_i a topologikus rendezés szerinti első tranzakció, akkor nem megy bele él, vagyis ő csak olyan adategységeket használ, amiket az eredeti ütemezésben előtte senki. Így az ő összes utasítását előre mozgathatjuk, a hatás nem változik.
Ami ezután marad, az $n - 1$ tranzakció utasításaiból álló ütemezés, aminek a sorosítási gráfja szintén DAG, tehát ennek az indukció szerint létezik soros ekvivalense (a maradék tranzakciók topologikus sorrendjének megfelelően), ami T_i-vel kiegészítve soros ekvivalense lesz az eredetinek. □

Következmény

Következmény: A bizonyításból látszik, hogy a soros ekvivalensek és a lehetséges topologikus sorrendek megfelelnek egymásnak, vagyis annyi soros ekvivalens lesz, ahány különböző topologikus sorrend van.
Például a korábban látott sorosítási gráf esetén 8 darab topologikus sorrend van, így nyolc soros ekvivalens van:
$T_5\ T_4\ T_1\ T_2\ T_3$,
$T_4\ T_5\ T_1\ T_2\ T_3$,
$T_4\ T_1\ T_5\ T_2\ T_3$,
$T_5\ T_1\ T_4\ T_2\ T_3$,
$T_1\ T_5\ T_4\ T_2\ T_3$,
$T_1\ T_4\ T_5\ T_2\ T_3$,
$T_5\ T_1\ T_2\ T_4\ T_3$,
$T_1\ T_5\ T_2\ T_4\ T_3$,

Az ütemező lehetőségei a sorosíthatóság kikényszerítésére

1. Figyeli a sorosítási gráfot (amit a zárkérések alapján készít) és ha kör keletkezne, akkor az egyik körbeli tranzakciót ABORT-álja.
   (Előny: nem óvatoskodik, nem korlátoz feleslegesen;
   Hátrány: drasztikus megoldás az ABORT)
2. Protokollt ír elő a tranzakciók számára, amit minden egyes tranzakciónak be kell tartania:
   2PL (two-phase locking, kétfázisú protokoll): a T_i tranzakció követi a kétfázisú protokollt, ha $UNLOCK_i$ után nincs $LOCK_i$, azaz ha nem kér már zárat miután elengedett már egyet.

Tétel

Ha az egyszerű tranzakciómodellbeli legális ütemezésben minden tranzakció követi a 2PL-t, akkor az ütemezéshez tartozó sorosítási gráf DAG, azaz az ütemezés sorosítható.

Nem bizonyítjuk.

Bonyolultabb zármodellek

Többfajta zár van, aszerint, hogy a tranzakciók mit akarnak csinálni az adattal. (Persze akkor, ha van több különböző művelet, nem csak írás és olvasás.)
Cél, hogy minél jobban tükrözzék a zárkérési lehetőségek a lehetséges műveleteket, mert így kevesebb lesz a várakozás (több olyan eset lesz, amikor lehet két tranzakciónak zárja ugyanott, ha a megfelelő műveletek mehetnek együtt) és megalapozottabb lesz a döntés a sorosíthatóságról (nem leszünk annyira feleslegesen szigorúak).
Példa: Legyen három művelet: olvasás, írás és növelés (increment).
Ez utóbbi azt jelenti, hogy az adategység aktuális értékét megnöveljük eggyel.
Ekkor bevezethetünk három zárat: RLOCK, WLOCK és INCR, a kézenfekvő használattal (a megfelelő művelet csak akkor mehet, ha a tranzakció megkapta a hozzá tartozó zárat).

Kompatibilitási mátrix

Egy mátrix segítségével adjuk meg, hogy különböző tranzakcióknak milyen zárai lehetnek egyszerre egy adategységen.
Ez a **kompatibilitási mátrix**: a sorok és az oszlopok is a lehetséges záraknak felelnek meg és a Z_i sor Z_j oszlopában pontosan akkor van `I`, ha egy tranzakció megkaphatja egy adategységre a Z_j zárat akkor, ha egy másik tranzakció Z_i zárat tart fenn ezen az adategységen. Ha nem kaphatja meg, akkor `N` áll a Z_i sor Z_j oszlopában.
Akkor lehet két különböző tranzakciónak Z_i és Z_j zárja ugyanazon az adategységen, ha mindegy, hogy a két zárnak megfelelő műveletek milyen sorrendben hajtódnak végre.
Ez alapján az RLOCK/WLOCK modell és az RLOCK/WLOCK/INCR modell mátrixai:

	RLOCK	WLOCK
RLOCK	I	N
WLOCK	N	N

	RLOCK	WLOCK	INCR
RLOCK	I	N	N
WLOCK	N	N	N
INCR	N	N	I

A kompatibilitási mátrix használata

1. Ez alapján dönti el az ütemező, hogy egy ütemezés/zárkérés legális-e, illetve ez alapján várakoztatja a tranzakciókat. Minél több az `I` a mátrixban, annál kevesebb lesz a várakoztatás.
2. A mátrix alapján keletkező várakozásokhoz elkészített várakozási gráf segítségével az ütemező kezeli a holtpontot (ami tetszőleges zármodell esetén ugyanazt jelenti és a gráfot is ugyanúgy kell felépíteni).
3. A mátrix alapján készíti el az ütemező a sorosítási gráfot egy zárkérés-sorozathoz: a sorosítási gráf csúcsai a tranzakciók és akkor van él T_i-ből T_j-be, ha van olyan A adategység, amelyre az ütemezés során Z_k zárat kért és kapott T_i, ezt elengedte, majd ezután A-ra legközelebb T_j kért és kapott olyan Z_l zárat, hogy a mátrixban a Z_k sor Z_l oszlopában `N` áll.
   Vagyis olyankor lesz él, ha a két zár nem kompatibilis egymással, nem mindegy a két művelet sorrendje.

Sorosíthatóság

A sorosíthatóságról most is pusztán a zárkérések alapján fogunk dönteni, a sorosítási gráf segítségével:

Tétel

Egy csak zárkéréseket és zárelengedéseket tartalmazó legális ütemezés pontosan akkor sorosítható valamelyik zármodellben, ha a zármodellhez tartozó kompatibilitási mátrix alapján az előbbi módszerrel felrajzolt sorosítási gráf DAG.

Bizonyítás.

Pontosan ugyanúgy megy, ahogyan eddig. □

Az ütemező egyik lehetősége a sorosíthatóság elérésére, hogy folyamatosan figyeli a sorosítási gráfot és ha irányított kör keletezne, akkor ABORT-ot rendel el.

Sorosíthatóság II.

Másik lehetőség a protokollal való megelőzés. Tetszőleges zármodellben értelmes a 2PL és igaz az alábbi tétel:

Tétel

Ha valamilyen zármodellben egy legális ütemezésben minden tranzakció követi a 2PL-t, akkor az ütemezéshez tartozó sorosítási gráf DAG, azaz az ütemezés sorosítható.

Nem bizonyítjuk.
Megjegyzés: Minél gazdagabb a zármodell, minél több az `I` a kompatibilitási mátrixban, annál valószínűbb, hogy a sorosítási gráf DAG lesz minden külön protokoll nélkül. Ez azt jelenti, ilyenkor egyre jobb lesz az ABORT-os módszer (ritkán kellhet).

Mit látunk mi ebből?

Az adatbáziskezelő működése során az ütemező munkájába nem (nagyon) szólhatunk bele. Miért hasznos mégis tudni, hogy hogyan működik?

- Abba beleszólhatunk, hogy mennyire törekedjen sorosíthatóságra az adatbáziskezelő (akár azt is mondhatjuk, hogy semennyire). Ehhez nem árt, ha tudjuk, hogy mi is a sorosíthatóság, mit nyerünk vele és mibe kerül (bonyolult ütemező, lassabb futás).
- Ha ismerjük a különféle ütemezési technikákat: jobban fogjuk érteni az előforduló ABORT-okat, és majd az ABORT utáni visszaállítást is.

Sorosíthatóság időbélyegekkel

Eddig a zárakat vizsgáltuk, mint egy lehetséges technikát a sorosíthatóság kikényszerítésére. Másik lehetőség: **időbélyeges tranzakciókezelés**.
Ez optimistább, illetve agresszívabb, mint a zárak használata: hagyja a tranzakciókat szabadon futni (ellentétben a záraknál látott protokollokkal), de ha baj lenne, akkor agresszívan közbelép (ABORT).
Akkor jó, ha ritkán lesz ABORT, ha valószínűleg kevés lesz a sorosítási probléma.
Fő elv: minden tranzakciónak van egy időbélyege: $t(T_i)$ a T_i tranzakcióé. Az időbélyegek egyediek, növő sorrendben adja ki őket az ütemező, ahogy indulnak a tranzakciók.
Az ütemező célja: az időbélyegek növő sorrendjéhez tartozó soros ütemezéssel azonos hatású ütemezést enged csak lefutni, minden olyan kérést letilt (és a megfelelő tranzakciót ABORT-álja), ami ez ellen tesz.
Például, ha $t(T_1) = 120$, $t(T_2) = 90$ és $t(T_3) = 130$, akkor a cél a $T_2 T_1 T_3$ soros sorrenddel azonos hatású ütemezés.

Az időbélyeges tranzakciókezelés szabályai

Megkülönböztetünk írás és olvasás műveletet, továbbá minden A adatelemhez hozzárendelünk egy olvasási és egy írási időt $(r(A), w(A))$, melyek jelentése:

- r(A)= a legnagyobb olyan $t(T_i)$, amire igaz, ahogy T_i olvasta már A-t
- w(A)= a legnagyobb olyan $t(T_i)$, amire igaz, ahogy T_i írta már A-t

Az ütemező mit csinál, hogy kikényszerítse az időbélyegek szerinti növő soros ütemezés hatását?

- minden induló tranzakciónak legenerál egy időbélyeget, egyedit, növően, ez lesz a tranzakció egész futása alatt az ő időbélyege
- ha a T tranzakció bármit csinálni szeretne egy A adategységgel, akkor mielőtt ezt megengedné, megvizsgálja $t(T)$, és $r(A)$ illetve $w(A)$ kapcsolatát és a következőképpen cselekszik.

Szabályok

1. Ha T olvasná A-t, de $t(T) < w(A)$, akkor ABORT T (mert egy nagyobb időbélyegű, azaz T után következő tranzakciónak már megengedte, hogy írja)
2. Ha T írná A-t, de $t(T) < r(A)$, akkor ABORT T (mert egy nagyobb időbélyegű, azaz T után következő tranzakciónak már megengedte, hogy olvassa)
3. Ha T olvasná A-t, $t(T) \geq w(A)$, de $t(T) < r(A)$, akkor T olvashatja A-t és $r(A)$ marad, ami volt és persze $w(A)$ is (mert ugyan egy nagyobb időbélyegű tranzakciónak már megengedtük, hogy olvassa A-t, de ez nem baj, ettől még kijöhet a kívánt soros ütemezés hatása)
4. Ha T írná A-t, $t(T) \geq r(A)$, de $t(T) < w(A)$, akkor nem történik meg az írás, de nem is lesz ABORT T se és $r(A)$ és $w(A)$ marad, ami volt (mivel egy nagyobb időbélyegű tranzakciónak már megengedtük, hogy írja A-t, ezért a kívánt soros hatásban úgyse látszódik ez az írás)
5. Ha T olvasná vagy írná A-t, és $t(T) \geq w(A)$ és $t(T) \geq r(A)$, akkor engedjük és $r(A)$ illetve $w(A)$ változik, attól függően, hogy írás vagy olvasás történt

Példa

Legyenek a tranzakciók időbélyegei $t(T_1) = 20$, $t(T_2) = 10$ (vagyis cél a $T_2 T_1$ hatása) és tekintsük az alábbi kéréssorozatot:
$READ_2(A)$, $READ_1(A)$, $WRITE_1(C)$, $WRITE_2(C)$, $WRITE_2(A)$
Hogyan változnak az olvasási és írási idők (kezdetben nullák) és mit csinál az ütemező? Lesz-e ABORT?

Kérés	r(A)	w(A)	r(C)	w(C)	Magyarázat
	0	0	0	0	kezdetben minden nulla
$READ_2(A)$	10	0	0	0	5. eset $\Longrightarrow$ mehet
$READ_1(A)$	20	0	0	0	5. eset $\Longrightarrow$ mehet
$WRITE_1(C)$	20	0	0	20	5. eset $\Longrightarrow$ mehet
$WRITE_2(C)$	20	0	0	20	4. eset $\Longrightarrow$ nem mehet, de nincs ABORT se
$WRITE_2(A)$	20	0	0	20	2. eset $\Longrightarrow$ ABORT T_2

Megjegyzések

1. A szabályok 4. pontjánál (ahol nem volt se ABORT, se írás) egyes források ABORT-ot rendelnek el. Ennek oka, hogy az általunk definált szabályok alkalmazása esetén előfordulhat a következő kellemetlen jelenség:
   Ha az a T_i tranzakció, aki beállította $w(A)$ értékét (aminél az írni akaró T tranzakció időbélyege kisebb) esetleg ABORT-ál és emiatt vissza kell csinálni T_i összes hatását, akkor T hatásának látszania kellene, de nem fog, pedig T lefutott hiba nélkül. Ha a 4.pont esetén ABORT-ot rendelünk el, akkor ez a gond nincsen. Vannak azonban technikák, amikkel akkor is meggátolható ez a jelenség, ha úgy járunk el, ahogy megadtuk a 4. pontnál a tennivalókat (most nem nézzük, hogy mik ezek a technikák), ezért nem kell az ABORT ebben az esetben.
2. Az időbélyeges módszer a zárhasználat alternatívája. Az időbélyeges módszernél ha sok az ABORT, akkor sokat kell majd dolgoznunk a visszaállítással (ezért akkor javasolt, ha kevés a közös elemeken történő írás); a zárak hátránya pedig az, hogy karban kell tartani a zártáblát és a korlátozások miatt sok lehet a várakozás és a holtpont.
3. Vannak még más módszerek is a sorosíthatóság elérésére, pl. érvényesítés.

Időbélyegek↔ zárak

Egyik se jobb egyértelműen, mint a másik. Van, hogy mind a kettő ugyanazokat a kéréseket hagyja lefutni:

(a) Ha T_2 előbb indul, mint T_1, akkor a $READ_2(B)$, $READ_1(A)$, $WRITE_1(C)$, $WRITE_2(C)$ műveletsort egy időbélyegesen dolgozó ütemező nem hagyja lefutni, mert a $T_2 T_1$ soros sorrenddel ez nem lesz azonos hatású. Viszont RLOCK/WLOCK zárolás esetén van olyan legális zárkérés, amit az ütemező sorosíthatónak fog találni.

(b) A $READ_1(A)$, $WRITE_2(A)$, $WRITE_1(A)$, $WRITE_1(B)$, $WRITE_2(B)$, $WRITE_3(A)$ műveletsort (itt T_1 indul előbb), bárhogy is kérjük a zárakat legálisan, nem hagyja lefutni egy RLOCK/WLOCK zárakat használó ütemező (T_2 és T_1 között mindkét irányban lesz él a sorosítási gráfban), de időbélyeggel lefut ez a műveletsor.

Védekezés hibák ellen, helyreállítás

Alapprobléma: nem fut le valamelyik tranzakció (sérül az atomiság) és emiatt inkonzisztens lesz az adatbázis.
Ennek okai lehetnek:

1. tranzakcióhiba, programhiba
2. ütemező által elrendelt ABORT (holtpont vagy sorosíthatóság miatt)
3. rendszerhiba: belső tár sérül
4. médiahiba: háttértár is sérül

Cél mindegyik esetben az, hogy újra konzisztens állapotba hozzuk az adatbázist (visszacsinálás vagy befejezés) úgy, hogy a tartósság megmaradjon: ha egy tranzakció már befejezte a munkáját, akkor annak hatása ne vesszen el.
Az utolsó fajta hibával nem foglalkozunk, erre a szokásos eljárások mennek (archiválás, duplikálás).

Alapfogalmak

Feltevés, hogy a végig lefutott tranzakciók konzisztens állapotból konzisztens állapotba viszik az adatbázist, ezért baj csak akkor lehet, ha félbemaradnak.
Fontos eszköz a hiba utáni helyreállításban:
COMMIT pont: az a pont, amikor a tranzakció minden érdemi munkával megvan, programhiba vagy ütemező miatt ABORT már biztos nem lehet. Nem biztos, hogy ekkor minden hatása látszik is már a tranzakciónak, lehet, hogy nincs minden írása véglegesítve, de minden készen áll már erre.
Fontos fogalom még:
Piszkos adat: Olyan adat, amit még nem COMMIT-ált tranzakció (azaz olyan, ami még meghalhat) írt az adatbázisba. Ha ilyet olvas egy másik tranzakció, akkor baj lehet, ha az első ABORT-ál, de a második nem.

T_1	T_2
LOCK(A)	
READ(A)	
A:=A+100	
WRITE(A)	
LOCK(B)	
UNLOCK(A)	
	LOCK(A)
	READ(A)
	$A := A \cdot 25$
READ(B)	
	WRITE(A)
	COMMIT
	UNLOCK(A)
$B := \frac{B}{A}$	
$\Downarrow$	
ABORT	

Példa piszkos adatból eredő hibára zárolásos ütemezés esetén :
Ha az osztáskor A értéke éppen 0, akkor T_1 ABORT-ál, és emiatt sok baj lesz:

- B-n zár marad, ezt fel kell oldani
- T_1 félig futott csak le, amit eddig számolt, azt vissza kell csinálni
- T_2 rossz adatot olvasott (mert a T_1 által A-ba beleírt értéket visszavontuk), így T_2-t is vissza kell csinálni

Összességében ebben az esetben T_1 és T_2 minden hatását ki kell irtani a DB-ből.
Ha esetleg közben még mások is olvasták a T_1 vagy a T_2 által írt értékeket, akkor **lavina**: egymás után kell ABORT-okat elrendelni a tranzakcióknál piszkos adatból eredő hiba miatt.

Megoldások piszkos adat és lavina ellen

Különböző megoldások a tranzakcióhibákból (programhiba vagy rendszer általi ABORT) származó problémákra:

- Olyan tranzakciótól, ami nem COMMIT-ált, nem olvasunk. (Nem olvasunk olyan értéket, amit olyan tranzakció írt, aminek még nem volt COMMIT).
- Hagyjuk, hogy minden tranzakció azt csináljon, amit akar, ha lavina lesz, akkor majd megoldjuk (UNDO protokoll, nem lesz részletesen, de létezik)
- Zárolási protokollt kényszerítünk a tranzakciókra, ami biztosítja, hogy nem lesz piszkos adatból probléma, lavina:

  szigorú 2PL:
  - 2PL
  - DB-be írás csak COMMIT után
  - zárak elengedése csak írás után

Szigorú 2PL protokoll

Tétel

Ha mindegyik tranzakció a szigorú 2PL protokollt követi, akkor az ütemezés sorosítható lesz és lavinamentes.

Bizonyítás.

Mivel a tranzakciók követik a 2PL protokollt, ezért az ütemezés sorosítható lesz. Azért lesz lavinamentes is, mert egy T_i tranzakció csak akkor olvashatja egy másik T_j tranzakció írását, ha T_j már elengedte a zárakat, de az meg csak COMMIT után lehet, amikor T_j már biztos nem száll el. □

Megjegyzések:

1. Elég az írások, a COMMIT és a zárkérések sorrendjét figyelni, ahhoz hogy jó ütemezés legyen és ráadásul ezt minden tranzakció meg tudja maga tenni, nem kell a többire figyelnie.

2. Mivel írás csak COMMIT után van, nem kell azzal sem bajlódni, hogy visszagörgessük az elszállt tranzakciókat, mert ezeknek még úgysem látszik semmi hatásuk.

Rendszerhibák utáni helyreállítás

Az eddigiekben azzal foglalkoztunk, hogy a tranzakciók hibái esetén mit lehet tenni.
Erre a szigorú 2PL jó megoldást nyújt, de mi van a komolyabb hibák, a rendszerhibák esetén?
Azt mindig feltesszük, hogy a háttértár nem sérül, csak a belső memória, a puffer egy része száll el.
A belső tár sérülése elleni védekezés két részből áll:

1. Felkészülés a hibára: **naplózás**
2. Hiba után helyreállítás: **a napló segítségével egy konzisztens állapot helyreállítása**

Természetesen a naplózás és a hiba utáni helyreállítás összhangban vannak, de van több különböző naplózási protokoll (és ennek megfelelő helyreállítás).

Napló

A tranzakciók legfontosabb történéseit írjuk ide, például:

- T_i kezd: (T_i, BEGIN)
- T_i írja A-t: (T_i, A, régi érték, új érték)
  (néha elég csak a régi vagy csak az új érték, a naplózási protokolltól függően)
- T_i COMMIT-ál: (T_i, COMMIT)
- T_i ABORT-ál: (T_i, ABORT)

A napló időrendben tartalmazza a történéseket és tipikusan a háttértáron tartjuk, amiről feltesszük, hogy nem sérül.
Fontos, hogy a naplóbejegyzéseket mikor írjuk át a pufferből a lemezre.

UNDO protokoll-naplózás

Fő szabályok:

- Ha a T tranzakció módosítja az X adatbáziselemet, akkor a (T, X, régi érték) naplóbejegyzést azelőtt kell a lemezre írni, mielőtt az X új értékét a lemezre írná a rendszer.
- Ha a tranzakció hibamentesen befejeződött, akkor a COMMIT naplóbejegyzést csak azután szabad a lemezre írni, ha a tranzakció által módosított összes adatbáziselem már a lemezre íródott, de ezután rögtön.

UNDO protokoll

A (nem teljesen szigorú) 2PL kiegészítése, vagyis a zárkérések 2PL szerint történnek, ezen felül pedig a műveletek és ezek naplózása az alábbi sorrendben történik:

1. A tranzakciók történéseinek feljegyzése a naplóba, a belső táron: (T_i, BEGIN), (T_i, A régi érték) vagy (T_i, ABORT)
2. Tényleges írás az adatbázisba a háttértáron, nem a pufferben: OUTPUT(A)
3. COMMIT után a napló háttértárra írása.
4. Zárak elengedése

Megjegyzések

- nincs lavina, mert zárelengedés csak COMMIT után (azaz piszkos adatot nem olvashatunk)
- sorosítható, mert 2PL
- vissza lehet hozni konzisztens állapotba a DB-t, akkor is, ha a belső tár sérül, erre lesz majd mindjárt az UNDO helyreállítás

Példa

T	t	A_M	B_M	A_D	B_D	Napló
				8	8	(T, BEGIN)
LOCK(A)				8	8	
LOCK(B)				8	8	
READ(A, t)	8	8		8	8	
$t := t \cdot 2$	16	8		8	8	
WRITE(A, t)	16	16		8	8	(T, A, 8)
READ(B, t)	8	16	8	8	8	
$t := t \cdot 2$	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	(T, B, 8)
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						(T, COMMIT)
FLUSH LOG						
UNLOCK(A)						
UNLOCK(B)						

FLUSH LOG: napló kiírása a háttértárra t: lokális változó

A_M, B_M, A_D, B_D: Az A és B cellák tartalma memóriában illetve a lemezen.

UNDO helyreállítás

Ha hiba történt $\Longrightarrow$ konzisztens állapot visszaállítása
$\Longrightarrow$ nem befejezett tranzakciók hatásának törlése

- Első feladat: Eldönteni melyek a sikeresen befejezett és melyek nem befejezett tranzakciók.
  - Ha van (T, BEGIN) és van (T, COMMIT) $\Longrightarrow$ minden változás a lemezen van $\surd$
  - Ha van (T, BEGIN), de nincs (T, COMMIT) $\Longrightarrow$ lehet olyan változás, ami nem került még a lemezre, de lehet olyan is ami kikerült $\Longrightarrow$ ezeket vissza kell állítani
- Második feladat: visszaállítás
  A napló végétől visszafelé (pontosabban a hibától) haladva: megjegyezzük, hogy mely T_i-re találtunk (T_i, COMMIT) és (T_i, ABORT) bejegyzéseket.
  Ha van egy (T_i, X, v) bejegyzés:
  - Ha láttunk már (T_i, COMMIT) bejegyzést (visszafelé haladva), akkor T_i már befejeződött, értékét kiírtuk a tárra $\Longrightarrow$ nem csinálunk semmit
  - Minden más esetben (vagy volt (T_i, ABORT) vagy semmi) $\Longrightarrow$ X-be visszaírjuk v-t
- Harmadik feladat: Ha végeztünk, minden nem teljes T_i-re írjunk (T_i, ABORT) a napló végére.

Példa

T	t	A_M	B_M	A_D	B_D	*Napló*
				8	8	(T, BEGIN)
LOCK(A)				8	8	
LOCK(B)				8	8	
READ(A, t)	8	8		8	8	
$t := t \cdot 2$	16	8		8	8	
WRITE(A, t)	16	16		8	8	$(T, A, 8)$
READ(B, t)	8	16	8	8	8	
$t := t \cdot 2$	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	$(T, B, 8)$
(FLUSH LOG)						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						(T, COMMIT)
(FLUSH LOG)						
UNLOCK(A)						
UNLOCK(B)						

Megjegyzések

- Mi van ha a helyreállítás közben hiba történik? Már bizonyos értékeket visszaállítottunk, de utána elakadunk.
  $\Longrightarrow$ Kezdjük előről a visszaállítást! Ha már valami vissza volt állítva, legfeljebb még egyszer „visszaállítjuk” $\Longrightarrow$ nem történik semmi.

- **Ez így nagyon sokáig tarthat, mert el kell mennünk a napló elejéig.**

CHECKPOINT képzése

Meddig menjünk vissza a naplóban? Honnan tudjuk, hogy mikor vagyunk egy biztosan konzisztens állapotnál?
Erre való a **CHECKPOINT**. Ennek képzése:

1. Megtiltjuk új tranzakció indítását
2. Megvárjuk, amíg minden futó tranzakció COMMIT vagy ABORT módon véget ér
3. Minden puffert a háttértárra írunk, ekkor az adatbázis állapota biztosan konzisztens lesz
4. A naplóba beírjuk, hogy (CHECKPOINT)
5. A naplót is háttértárra írjuk: (FLUSH LOG)

Ezután nyilván elég az első CHECKPOINT-ig visszamenni, hiszen előtte minden T_i már valahogy befejeződött.
$\Longrightarrow$ Teljesen le kell állítani a rendszert.

CHECKPOINT képzése működés közben

1. A naplóba beírjuk, hogy (START CHECKPOINT ($T_1, \ldots, T_k$)), ahol T_i az összes éppen aktív tranzakció
2. A naplót háttértárra írjuk: FLUSH LOG
3. Megvárjuk, hogy minden fenti T_i végetérjen. (Közben más tranzakciók elindulhatnak.)
4. Ha mind befejeződött: (END CHECKPOINT) és (FLUSH LOG)

Visszaállítás

- Visszafelé olvasva, ha előbb (END CHECKPOINT) van $\Longrightarrow$ elég visszamenni a következő START CHECKPOINT-ig.
- Ha előbb (START CHECKPOINT ($T_1, \ldots, T_k$))-ot találunk $\Longrightarrow$ ezek nem mindegyike fejeződött be (meg esetleg mások sem, amik még később kezdődtek) $\Longrightarrow$ elég visszamenni a legkorábban kezdődött T_i elejére

Példa

(T_1, BEGIN)
$(T_1, A, 5)$
(T_2, BEGIN)
$(T_2, B, 10)$
$(\text{START CHECKPOINT } (T_1, T_2))$ $\leftarrow$
$(T_2, C, 15)$
(T_3, BEGIN)
$(T_1, D, 20)$
(T_1, COMMIT)
$(T_3, E, 25)$ $\leftarrow$
(T_2, COMMIT)
(END CHECKPOINT)
$(T_3, F, 30)$ $\leftarrow$

- T_3 nem fejeződött be
  $\Longrightarrow F \rightarrow 30$
- T_3 nem fejeződött be
  $\Longrightarrow E \rightarrow 25$
- (START CHECKPOINT)
  $\Longrightarrow \surd$

Példa

(T_1, BEGIN)
$(T_1, A, 5)$
(T_2, BEGIN)
$(T_2, B, 10)$ $\leftarrow$
(START CHECKPOINT (T_1, T_2)) $\leftarrow$
$(T_2, C, 15)$ $\leftarrow$
(T_3, BEGIN)
$(T_1, D, 20)$
(T_1, COMMIT) $\leftarrow$
$(T_3, E, 25)$ $\leftarrow$

- T_3 nem fejeződött be $\Longrightarrow E \rightarrow 25$
- T_1 befejeződött $\Longrightarrow$ T_1-et nem bántjuk
- T_2 nem fejeződött be $\Longrightarrow C \rightarrow 15$
- (START CHECKPOINT) $\Longrightarrow$ elég visszamenni T_2 elejéig
- T_2 nem fejeződött be $\Longrightarrow B \rightarrow 10$

REDO protokoll-naplózás

Fő szabály:

- Mielőtt a lemezen módosítunk egy X adatelemet, a (T, X, v) és a $(T$, COMMIT) bejegyzést is ki kell írnunk a naplóba.

REDO protokoll

Ez a szigorú 2PL kiegészítése, vagyis a zárkérések 2PL szerint történnek, ezen felül pedig a műveletek és ezek naplózása az alábbi sorrendben történik:

1. A tranzakciók történéseinek feljegyzése a naplóba, a belső táron: $(T_i$, BEGIN), $(T_i, A$, új érték), $(T_i$, ABORT)
2. COMMIT után a napló háttértárra írása
3. Tényleges írás az adatbázisba a háttértáron, nem a pufferben
4. Zárak elengedése

Megjegyzések:

- nincs lavina, mert zárelengedés csak COMMIT után
- sorosítható, mert 2PL
- vissza lehet hozni konzisztens állapotba a DB-t, akkor is, ha a belső tár sérül, erre lesz majd mindjárt a REDO helyreállítás
- Különbség a az UNDO protokollhoz képest:
  - Az adat változás utáni értékét jegyezzük fel a naplóba
  - Máshová rakjuk a COMMIT-ot, a kiírás elé $\Longrightarrow$ megtelhet a puffer
  - Az UNDO protokoll esetleg túl gyakran akar írni $\Longrightarrow$ itt el lehet halasztani az írást

Példa

T	t	A_M	B_M	A_D	B_D	*Napló*
				8	8	(T, BEGIN)
LOCK(A)				8	8	
LOCK(B)				8	8	
READ(A, t)	8	8		8	8	
$t := t \cdot 2$	16	8		8	8	
WRITE(A, t)	16	16		8	8	(T, A, 16)
READ(B, t)	8	16	8	8	8	
$t := t \cdot 2$	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	(T, B, 16)
						(T, COMMIT)
(FLUSH LOG)						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
UNLOCK(A)						
UNLOCK(B)						

REDO helyreállítás

Ha rendszerhiba történt és megsérült a belső tár, akkor az alábbiakat tesszük:

1. Minden zárat feloldunk
2. A napló mentett részét nézzük visszafele, megkeressük azokat a tranzakciókat, amikre volt már COMMIT (a többi nem érdekes, mert ha még nem volt a COMMIT-juk kimentve, akkor nem is írtak a DB-be)
3. Addig megyünk vissza a naplóban, amíg biztosan konzisztens állapotot nem találunk (eleje vagy CHECKPOINT)
4. A COMMIT-tált tranzakciók írásait előlről kezdve (a legelső COMMIT-ált elejétől) megismételjük (ha már egyszer be volt írva, az se baj, akkor csak felülírjuk ugyanazzal). Ezt meg tudjuk tenni, mert ismerjük az új értékeket.
5. Minden nem befejezett T_i tranzakcióra (T_i,ABORT)-ot írunk a napló végére, (FLUSH LOG)

Megjegyzések a REDO helyreállításhoz

- Ha a napló háttértáron van, akkor mindent újra tudunk csinálni, ami meg még nem került ki, azzal kapcsolatban változtatás se történt, nem kell visszacsinálni semmit.
- Ha a helyreállítás során lenne újra hiba, akkor a napló marad, mert az már kint van, ez alapján újra kezdhetjük a helyreállítást.
- Eredmény: a háttértárra kikerült COMMIT-oknak megfelelő tranazakciók eredménye látszik, a többiekéből pedig semmi.

Példa

T	t	A_M	B_M	A_D	B_D	*Napló*
				8	8	(T, BEGIN)
LOCK(A)				8	8	
LOCK(B)				8	8	
READ(A, t)	8	8		8	8	
$t := t \cdot 2$	16	8		8	8	
WRITE(A, t)	16	16		8	8	$(T, A, 16)$
READ(B, t)	8	16	8	8	8	
$t := t \cdot 2$	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	$(T, B, 16)$
						(T, COMMIT)
(FLUSH LOG)						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
UNLOCK(A)						
UNLOCK(B)						

Példa

T_1	napló
	(T_1, BEGIN)
LOCK(A)	
LOCK(B)	
	(T_1, A, x)
	(T_1, B, y)
	(T_1, COMMIT)
WRITE(A)	
WRITE(B)	
UNLOCK(A)	
UNLOCK(B)	

(T_1, A, x) jelentése: T_1 A-ba x-et írja
Ekkor a tényleges írás nem történik meg, csak a naplóba kerül ez bele, a tényleges írás csak a COMMIT után jön.

Ha a belső tár hibája a COMMIT háttértárra írása előtt történik, akkor még semmi valódi írás nem volt, azaz semmit se kell csinálni. Ha azonban a COMMIT után van, akkor a naplóban megvan minden utasítás, újra meg lehet csinálni T_1-et.

CHECKPOINT képzése

1. Megtiltjuk új tranzakció indítását
2. Megvárjuk, amíg minden futó tranzakció COMMIT vagy ABORT módon véget ér
3. Minden puffert a háttértárra írunk, ekkor az adatbázis állapota biztosan konzisztens lesz
4. A naplóba beírjuk, hogy CHECKPOINT
5. A naplót is háttértárra írjuk

CHECKPOINT képzése működés közben

1. A naplóba beírjuk, hogy (START CHECKPOINT ($T_1, \ldots, T_k$)), ahol T_i az összes éppen aktív tranzakció
2. A naplót háttértárra írjuk: FLUSH LOG
3. Az összes olyan adatelemet kiírjuk a lemezre, amit olyan tranzakciók indítottak, amik még a CHECKPOINT előtt befejeződtek, de még nem írtak ki mindent a lemezre.
4. (END CHECKPOINT) és (FLUSH LOG)

Visszaállítás

- Visszafelé olvasva, ha előbb (END CHECKPOINT) van $\Longrightarrow$ elég visszamenni a következő START CHECKPOINT-ig. $\Longrightarrow$ innen előre minden itt szereplő T_i-re és minden később kezdődő más tranzakcióra REDO
- Ha előbb (START CHECKPOINT ($T_1, \ldots, T_k$))-ot találunk $\Longrightarrow$ ezek nem mindegyike írta ki adatai (meg esetleg mások sem, amik még később kezdődtek) $\Longrightarrow$ elég visszamenni az előző (START CHECKPOINT)-hoz $\Longrightarrow$ onnan előre REDO

Előnyök, hátrányok

A CHECKPOINT ütemezése:

- adott idő letelte után
- adott lefutott tranzakció után

Ha ritkák a rendszerhibák, elég ritka CHECKPOINT.

Védekezés lemezhiba ellen

- **A naplót külön lemezen tartjuk**
- **Nem dobjuk el a napló CHECKPOINT előtti részét sem**
- **REDO vagy UNDO/REDO protokollt használunk**

Így elvileg a kezdeti adatbázis ismeretében vissza tudjuk állítani a legutolsó állapotot.
De a napló egy idő után nagyobb lesz, mint az adatbázis.
$\Longrightarrow$ Időnként archiválunk

Archiválás működés közben

Ha leállítjuk a rendszert, nyugodtan lehet menteni.
Ha nem lehet leállítani $\Longrightarrow$

1. (START DUMP) a naplóba
2. Megfelelő CHECKPOINT kialakítása
3. Adatok mentése valamilyen sorrendben
4. Napló mentése
5. (END DUMP)

Helyreállítás

1. Megkeressük a legutolsó teljes mentést (volt (END DUMP))
2. Módosítjuk az adatbázist a napló segítségével a CHECKPOINT-tól kezdve (ezért kell REDO vagy UNDO/REDO)