Adatbázisok elmélete 26. előadás

Csima Judit
Budapesti Műszaki és Gazdaságtudományi Egyetem
Számítástudományi Tsz.
I. B. 136/b
csima@cs.bme.hu

2003. Május 13.

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 tranazkciót 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 $T_2T_1T_3$ soros sorrenddel azonos hatású ütemezés.

Az időbélyeges tranazkciókezelés 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 $t(T) \geq w(A)$, akkor enegedjük 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_2T_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$\leftrightarrow$ 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_2T_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, 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, aki még meghalhat) írt az adabázisba. 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 (persze időbélyegekkel is van ilyen)

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 iss 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, aki nem COMMIT-ált, nem olvasunk. (Nem olvasunk olyan értéket, amit olyan tranzakció írt, akinek 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 tranazkció 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 tranazkciókat, 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 történésit írjuk ide, például:

- T_i kezd: (T_i, BEGIN)
- T_i írja A-t: (T_i, A régi érték, A új érték)
  (néha elég csak a régi vagy csak az új érték, a naplózái 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

REDO protokoll-naplózás

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, COMMIT), (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

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 (erről később)
4. A COMMIT-tált tranzakciók írásait elölről kezdve (a legelső COMMIT-tól) megismételjük (ha már egyszer be volt írva, az se baj, akkor csak felülírjuk ugyanazzal). Ezt meg tudjuk tenni, mert ismerjük az új értékeket.

Megjegyzések a REDO helyreállításhoz

- Ha a napló háttértáron van, akkor mindent újra tudunk csináni, 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_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

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

Előnyök, hátrányok

Miért jó?

- REDO-nál elég a CHECKPOINT bejegyzésig menni a naplóban
- CHECKPOINT előtti naplórészek akár el is dobhatók (ha másra nem kellenek)

Hátrány: hosszú lehet a CHECKPOINT képzése, ezalatt nem indulhat új tranzakció, lassú lesz a futás

Megoldás erre: dinamikusan képezett CHECKPOINT, amikor nem állítunk le mindent

A CHECKPOINT ütemzése:

- adott idő letelte után
- adott lefutott tranazakció után

Ha ritkák a rendszerhibák, elég ritka CHECKPOINT.