Adatbázisok elmélete 23. előadás

Katona Gyula Y.
Budapesti Műszaki és Gazdaságtudományi Egyetem
Számítástudományi Tsz.
I. B. 137/b
kiskat@cs.bme.hu
http://www.cs.bme.hu/˜kiskat

2004

Összefüggések az adategységek között

Eddig nem néztük azt, hogy mik azok az adategységek, amikre a zárakat lehet kérni és kapni. Hallgatólagosan feltettük, hogy ezeket egymástól függetlenül lehet zárolni, nincs közöttük semmi szervezettség, semmi összefüggés.

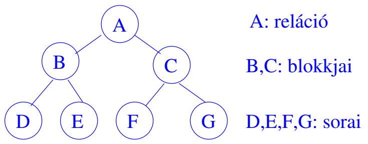
A valóságban két különböző esetben sem alkalmazható ez a megközelítés:

1. Ha az adategységek egymásba ágyazottak (pl. reláció, blokk, rekord), ekkor még további megkötéseket szeretnénk a zárolásra, az eddigi módszereket ki kell egészíteni.
2. Ha tudjuk, hogy egymáshoz képest hogyan helyezkednek el az adategységek a tárolási struktúrában, akkor jobb módszereket találhatunk, mint az eddigiek, illetve láthatjuk, hogy valamilyen eddig tanult módszer biztosan előnytelen lesz. Ez lesz például a B-fában tárolt adatok esete.

Adatbáziselemekből álló hierarchiák

A valóságban zárolhatunk teljes relációkat, de zárolhatjuk külön-külön ezek egyes blokkjait, sőt sorait is.
Minél nagyobb egységre rakunk zárat, annál könnyebb lesz a záradminisztráció, de annál több lesz a zárfeloldásra várás is és ezzel együtt a holtpont.
Alkalmazástól függ, hogy mi éri meg jobban, de egy valami mindig közös: Elvárjuk azt, hogy ha az A adategység egy reláció, B pedig ennek egy blokkja, akkor az A-ra rakott zár zárolja B-t is, azaz pl. az egyszerű tranzakciómodellben ne lehessen $l_j(B)$-t kapni, ha $l_i(A)$ után még nem volt $u_i(A)$. Ezt az eddigi technika még nem biztosítja, eddig ilyen összefüggéseket nem is vettünk figyelembe.

Egy ilyen lehetséges hierarchikus helyzet:



Figyelmeztető zármodell

Cél: olyan sorosítható ütemezés kikényszerítése, ami még az adategységek közötti hierarchiát is figyelembe veszi. Ez a hierarchia egy fával adott.

Egyszerű változat esetén háromféle zárművelet lesz: (lényegében az egyszerű tranzakciómodellnek felel meg):

- $LOCK_i(A)$: T_i zárolja A-t (explicit lock) és minden leszármazottját (implicit lock), kizárólagosan, azaz ezek után más tranzakció se A-ra, se ennek leszármazottjára nem kaphat zárat.
- $WARN_i(A)$: T_i figyelmeztetést rak A-ra (gyerekeire nem), ez annak jelzésére szolgál, hogy T_i majd zárat akar kapni A valamely leszármazottjára
- $UNLOCK_i(A)$: felszabadítja az A-ra rakott $LOCK$-ot és $WARN$-t, az implicit zár is lekerül A leszármazottjairól

A zárak használata

1. Az i-edik tranzakció, T_i, csak akkor olvashatja vagy írhatja az A adategységet, ha előtte zárat kért és kapott rá ($LOCK_i(A)$ vagy $LOCK_i$ A valamelyik ősén) és ezt a zárat még azóta nem engedte fel.
2. $LOCK_i(A)$ és $WARN_i(A)$ után mindig van $UNLOCK_i(A)$.
3. Ha $LOCK_i$ van A-n, akkor se $WARN_j$ se $LOCK_j$ nem kerülhet már rá (ha $j \neq i$), de két különböző tranzakciónak lehet $WARN$-ja ugyanott. Vagyis a kompatibilitási mátrix:

	LOCK	WARN
LOCK	N	N
WARN	N	I

A figyelmeztető protokoll

A T_i tranzakció követi a figyelmeztető protokollt, ha zárkérései a fentieken kívül még a következőket is tudják:

(a) T_i első zárkérése $WARN_i$ vagy $LOCK_i$ a gyökérre

(b) Ezután $LOCK_i$ vagy $WARN_i$ csak akkor kérhető egy adategységre, ha $WARN_i$ már van az apján.

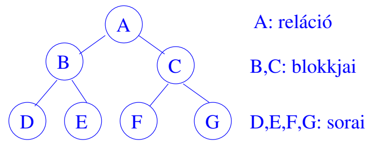
(c) $UNLOCK_i$ csak akkor kérhető egy adategységre, ha már nincs sem explicit $LOCK_i$, sem $WARN_i$ az adategység leszármazottjain

(d) Kétfázisú zárkérés van: $UNLOCK_i$ után nincs se $LOCK_i$, se $WARN_i$.

Az (a) és (b) pontok miatt a zárkérések felülről lefelé kúsznak a fában, a zárelengedések pedig a (c) miatt alulról felfele mennek az egyes tranzakciók esetén.

Példa

Az adategységek hierarchiája legyen (mint előbb):



Az alábbi zárkérésekből és zárelengedésekből álló sorozat legális és mindhárom tranzakció követi a figyelmeztető protokollt.

$WARN_1(A)$, $WARN_2(A)$, $WARN_3(A)$, $WARN_1(B)$, $LOCK_2(C)$, $LOCK_1(D)$, $UNLOCK_2(C)$, $UNLOCK_1(D)$, $UNLOCK_2(A)$, $UNLOCK_1(B)$, $LOCK_3(B)$, $WARN_3(C)$, $LOCK_3(F)$, $UNLOCK_1(A)$, $UNLOCK_3(B)$, $UNLOCK_3(F)$, $UNLOCK_3(C)$, $UNLOCK_3(A)$

Azért legális, mert minden LOCK és WARN fel van engedve később és egyszerre csak több WARN van ugyanott, más nem.

A tranzakciók zárkérései pedig egyrészt 2PL szerint mennek, másrészt a zárkérések a fában felülről lefele mennek, a zárelengedések pedig alulról felfele.

Figyelmeztető protokoll II.

A figyelmeztető protokollra igaz az alábbi tétel:
Tétel. *Ha a figyelmeztető zármodellben, egy legális ütemezésben minden tranzakció követi a figyelmeztető protokollt, akkor az ütemezés sorosítható és soha nem lesz egyszerre két különböző tranzakciónak zárja (se explicit, se implicit) ugyanazon az adategységen.*

Bizonyítás: A sorosíthatóságot a kétfázisúság biztosítja (pontosabban nem bizonyítjuk).

Zárkonfliktus pedig azért nem lesz, mert

- Két különböző tranzakciónak explicit LOCK-ja nem lehet soha ugyanott, ha az ütemezés legális.
- Egy explicit és egy implicit LOCK (két különböző tranzakciótól) nem lehet ugyanott: nem lehet, hogy egy A adatelemen $LOCK_i$ van és ezzel egyidejűleg egy leszármazottján (akin így implicit T_i lock van) van $LOCK_j$ is, mert ekkor A-n $WARN_j$-nek is kell lennie, de az nem lehet egy legális ütemezésben.
- Két különböző tranzakciónak implicit LOCK-ja sem lehet ugyanazon a C adategységen, mert ekkor C két különböző felmenőjén a két különböző tranzakció két LOCK-jának kellene lennie, ami az előző pont értelmében nem lehet.

Megjegyzés: Lehetne bonyolultabb zármód esetén is nézni figyelmeztető zárolást és figyelmeztető protokollt (az RLOCK/WLOCK modelnek megfelelően): lenne külön írási és olvasási figyelmeztetés is.

B-fában tárolt adategységek

Tekintsük most azt a helyzetet, amikor a zárolható adategységek egy fa csúcsaiban helyezkednek el, de a fa most nem az adategységek egymásba ágyazottságát mutatja (az adategységek most diszjunktak), hanem azt, hogy hogyan lehet elérni az adatokat. Például a B-fa esetén a levelekhez csak úgy juthatunk el, ha a gyökértől indulva végigjárunk egy lefele vezető utat. Ahhoz, hogy beolvashassuk azt a levelet, ami nekünk kell, előtte be kell olvasnunk az összes felmenőjét (és ha csúcsvágás vagy csúcsösszevonás úgy kívánja, írnunk is kell őket).

Ilyenkor a szokásos technikák mennek ugyan, de nagyon előnytelenek lehetnek. Például a 2PL esetén egész addig kell tartani a zárat a gyökéren, amíg le nem értünk a levélhez, ami indokolatlanul sok várakozáshoz vezet.

Kéne másik módszer, ami ebben a speciális esetben biztosítja a sorosíthatóságot, de nem olyan szigorú, mint a 2PL. Ez lesz a faprotokoll.

Faprotokoll

Egyszerű tranzakciómodellben vagyunk (de ezt is lehetne RLOCK/WLOCK modellre kibővíteni), azaz

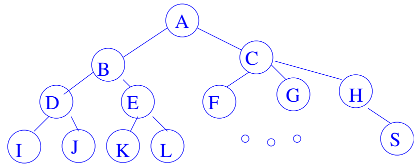
- egy zár van csak, ezt meg kell kapni íráshoz és olvasáshoz is
- zár után mindig van UNLOCK
- nincs két különböző tranzakciónak zárja ugyanott

A T_i tranzakció követi a faprotokollt, ha

1. Az első zárat bárhova elhelyezheti.
2. A későbbiekben azonban csak akkor kaphat zárat A-n, ha ekkor zárja van A apján.
3. Zárat bármikor fel lehet oldani (nem 2PL).
4. Nem lehet újrazárolni, azaz ha T_i elengedte egy A adategység zárját, akkor később nem kérhet rá újra (még akkor sem, ha A apján még megvan a zárja).

Tétel. *(Bizonyítás nélkül) Ha minden tranzakció követi a faprotokollt egy legális ütemezésben, akkor az ütemezés sorosítható lesz, noha nem feltétlenül lesz 2PL.*

Példa



Tekintsük ezt a B_3-fát. Az A-tól H-ig levő adategységek a fa belső csúcsai, itt mutatók és keresést segítő kulcsok vannak, a levelekben (I-től S-ig) pedig a keresési kulcs szerint rendezetten vannak a tárolt adatok, amik között keresünk. Most feltesszük, hogy egy levélben egy tárolt elem van.

Ha mondjuk az I-ben, J-ben és K-ban tárolt elemek keresési kulcsa $1, 3$ és 10, és be akarunk szúrni egy olyan elemet, ahol a kulcs értéke 4, akkor először olvasni kell A-t, B-t és D-t, majd írni is kell D-t.

Ekkor a megfelelő (faprotokoll szerinti, legális) ütemezés eleje $LOCK_i(A)$, $LOCK_i(B)$, $UNLOCK_i(A)$ mert B beolvasása után látjuk, hogy neki csak két gyereke van, ha kell is csúcsvágás, az A-t biztos nem érinti, A-t nem kell majd írni. Csak addig kellett fogni A-n a zárat, amíg B-re is megkaptuk.

Ezután $LOCK_i(D)$, $UNLOCK_i(B)$, mert látjuk, hogy D-nek csak két gyereke van, ezért B-t biztos nem kell írni.

Innen tovább: $UNLOCK_i(D)$, amikor már megtörtént az új levél beszúrása és D-ben is beállítottuk a mutatókat.

Példa II.

Tanulság:

- Faprotokoll szerint ment az ütemezés $\Longrightarrow$ jó lesz
- Nem 2PL és ezzel nyertünk is sokat, mert amint megvolt $UNLOCK_i(A)$, akkor rögtön indulhat a következő beszúrás, ha az a fa jobb oldali ágán fut le. Ha 2PL lett volna, akkor $LOCK_i(D)$-ig kellene várni ezzel.

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 $T_2T_1T_3$ 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_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↔ 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 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 (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 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, 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 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.