

Adatbázisok elmélete 26. 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

REDO helyreállítás (emlékeztető)

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)

REDO protokoll-naplózás (emlékeztető)

Fő szabály:

- Mielőtt az a lemezen módosítunk egy X adatelemet, a (T, X, v) és a $(T, COMMIT)$ bejegyzést is ki kell írunk 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. Zárok elengedése

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

UNDO/REDO protokoll-naplózás

- UNDO hátránya: COMMIT után mihamarabb írjunk \Rightarrow sok írás
- REDO hátránya: Nem írunk, amíg nincs COMMIT \Rightarrow nagy memóriagigény

UNDO/REDO

Fő elv:

- Mielőtt az adatbázis bármely X elemének értékét a lemezen módosítanánk, a (T, X, v, w) naplóbejegyzésnek a lemezre kell kerülnie.

Nagyobb szabadság, hogy mikor írjunk.

Nagyobb méretű napló. $\Rightarrow v, w$ nagyon nagy is lehet!

Előnyök, hátrányok

A CHECKPOINT ütemezése:

- adott idő letelte után
- adott lefutott tranzakció után

Ha ritkák a rendszerhibák, elég ritka CHECKPOINT.

Példa

T	t	A_M	B_M	A_D	B_D	Napló
LOCK(A)				8	8	(T, BEGIN)
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, 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, 8, 16)
(FLUSH LOG)						
OUTPUT(A)	16	16	16	16	8	
						(T, COMMIT)
OUTPUT(B)	16	16	16	16	16	
UNLOCK(A)						
UNLOCK(B)						

UNDO/REDO visszaállítás

- A legkorábbiól kezdve állítsuk vissza minden befejezett tranzakció hatását. (REDO)
- A legutolsótól kezdve állítsuk tegyük semmissé minden be nem fejezett tranzakció hatását. (UNDO)

CHECKPOINT képzés működés közben

1. Írjuk a naplóba a (START CHECKPOINT (T_1, \dots, T_k)) bejegyzést, ahol T_i az összes éppen aktív tranzakció
2. (FLUSH LOG)
3. Írjuk a lemezre az összes piszkos puffert
4. (END CHECKPOINT)
5. (FLUSH LOG)

Mindenképp elég visszamenni legfeljebb az előző CHECKPOINT-ig (mint a REDO-nál).

Példa

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

Példa

(T_1 , BEGIN)
 (T_1 , A, 4, 5)
 (T_2 , BEGIN)
 (T_1 , COMMIT)
 (T_2 , B, 9, 10)
 (START CHECKPOINT (T_2))
 (T_2 , C, 14, 15)
 (T_3 , BEGIN)
 (T_3 , D, 19, 20)
 (END CHECKPOINT)
 (T_2 , COMMIT)
 (T_3 , COMMIT)

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.

⇒ Időnként archiválunk

Osztott adatbázisok

- Adatok vízszintes felosztása
 - ★ Egy bank több fiókja, saját ügyfelek
 - ★ Üzlethálózat boltjai, saját eladások
 - ★ Könyvtár több fiókkal, saját katalógussal
- Adatok függőleges felbontása
 - ★ Bankban ⇒ ügyfél adatok helyben, hitelkártya adatok a központban
 - ★ Üzletláncban ⇒ eladások helyben, megrendelések a központban
- Adatok többszörözése
 - ★ Párhuzamosítás miatt
 - ★ Kommunikáció csökkentése miatt ⇒ gyakran szükséges adatok mindenhol (címjegyzék, telefonkönyv, chase-elés)

Archiválás működés közben

Ha leállítjuk a rendszert, nyugodtan lehet menteni.

Ha nem lehet leállítani ⇒

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)

Osztott tranzakciók

A tranzakciók műveletei most különböző helyeken történhetnek.

Mikor lesz kész az egész tranzakció? ⇒ ha minden része kész

Hogyan vesszük észre? Mi van ha közben ABORT vagy hiba van?

Példa

Áruházlánc központja lekérdezi minden boltban a mobiltelefon készletet. Ha valahol túl sok van, átküld belőle oda, ahol kevés van.

⇒ megszakadhat a kapcsolat menet közben, rossz az algoritmus, stb.

Kétfázisú véglegesítés (2PC)

- Alapelvek
 - ★ Minden állomás naplózza saját eseményeit
 - ★ Van egy koordinátor állomás, aki a döntést hozza majd
 - ★ Az állomások üzeneteket küldenek egymásnak, ezeket is naplózzák (ki- és bejövőt is)
- Első fázis
 - ★ A koordinátor saját naplójába (**T, Felkészül**)
 - ★ Ezt mindenhova elküldi (még magának is)
 - ★ Ha egy állomás megkapta az üzenetet, eldönti, hogy a nála található részre COMMIT vagy ABORT lesz majd
 - * Ha COMMIT várható (már csak ez lenne hátra) \Rightarrow (**T, Készenáll**) a saját naplóba
A koordinátornak elküldeni (**T, Készenáll**)-t
 - * Ha ABORT várható \Rightarrow (**T, ABORT-Legyen**) a saját naplóba
A koordinátornak elküldeni (**T, ABORT-Legyen**)-t
 - ★ Második fázis
 - * Ha a koordinátor a (**T, Készenáll**)-t megkapta mindenkitől \Rightarrow
 - (**T, COMMIT**) a saját naplójába
 - Mindenhova elküldi a (**T, COMMIT-Lesz**)-t
 - * Ha a koordinátor a (**T, ABORT-Legyen**)-t kapta legalább egy állomástól
 - (**T, ABORT**) a saját naplójába
 - Mindenhova elküldi a (**T, ABORT-Lesz**)-t
 - * Ha egy állomás a (**T, COMMIT-Lesz**)-t kapja \Rightarrow (**T, COMMIT**)
 - * Ha egy állomás a (**T, ABORT-Lesz**)-t kapja \Rightarrow (**T, ABORT**)

Osztott zárolás

Ha nincs adattöbbszörözés \Rightarrow ✓

Ha van \Rightarrow összhangban kell tartani a példányokat \Rightarrow globális (logikai) LOCK és lokális LOCK

Egyszerű modell

Minden LOCK globális és az egyik állomás nyilvántartja ezeket zárállomás

Költség: egy LOCK-hoz 3 üzenet \Rightarrow igénylés, engedélyezés, feloldás

\Rightarrow a zárállomás nagyon leterhelt lehet

Elsődleges példány

Van egy elsődleges példány, ha valaki zárolni akar valamit, akkor az elsődleges példányt tároló állomáshoz fordul.

Költség: mint előbb, de nem koncentrált forgalom

Helyreállítás

Egy adott állomáson:

- ★ Ha az utolsó bejegyzés COMMIT, ABORT, COMMIT-Lesz vagy ABORT-Lesz akkor a napló szerint helyreállítunk
- ★ Ha az utolsó bejegyzés Készenáll, akkor nem világos a helyzet, vagy várunk, vagy kommunikálunk a többivel, vagy ...
- ★ Ha nincs semmilyen bejegyzés, akkor ABORT (vagy várunk)

Osztott RLOCK/WLOCK

Alapelvek:

- Semelyik két tranzakciónak nem lehet globális WLOCK A-ja
- Ha egy tranzakciónak van globális WLOCK A-ja, akkor egy másiknak nem lehet globális RLOCK A-ja
- Lehet több tranzakciónak globális RLOCK A-ja
- Minden állomás az érvényes globális lock-ok figyelembevételével működik

Hogyan lehet megszerezni egy globális RLOCK-ot vagy WLOCK-ot? \Rightarrow többféle modell

WALL (write locks all)

- Globális RLOCK A megszerzéséhez elég egy lokális RLOCK A_i
- Globális WLOCK A megszerzéséhez kell minden lokális WLOCK A_i
- Globális RLOCK A megszerzése:
 - ★ Ha az i állomás akar egy RLOCK A_i -t, nem kell üzenni, megnézzük milyen zár van A_i -n
 - ★ Ha itt WLOCK A_i van, akkor elutasítja a kérést, ha semmi vagy RLOCK A_i , akkor engedélyezi
 - ★ Ha engedélyezi, akkor az i állomás felteszi az RLOCK A_i -t \Rightarrow globális RLOCK A
- Globális WLOCK A megszerzése
 - ★ Ha az i állomás akar egy WLOCK A_i -t, akkor üzen minden másik helyre ahol van A_j
 - ★ Ha itt RLOCK A_j vagy WLOCK A_j van, akkor elutasítja a kérést, ha semmi akkor engedélyezi
 - ★ Ha mindenholnan engedélyezés jött, az i állomás felteszi a WLOCK A_i -t, mindenhova üzen, hogy WLOCK A_j -t \Rightarrow globális WLOCK A

\Rightarrow Ha az egyik állomás kért és kapott WLOCK A -t, akkor másik nyilván nem kaphat később sem WLOCK A -t, sem RLOCK A -t

\Rightarrow Ha az egyik állomás kért és kapott RLOCK A -t, akkor másik nyilván nem kaphat később WLOCK A -t, de kaphat RLOCK A -t

k az n -ből protokoll

Közös általánosítás:

Legyen n , hogy hány példány van A -ból és legyen $n \geq k \geq \lceil (n+1)/2 \rceil$

- Globális RLOCK A megszerzéséhez kell, hogy lokális RLOCK A_i legyen legalább $n+1-k$ db A_i -n
- Globális WLOCK A megszerzéséhez kell, hogy lokális WLOCK A_i legyen legalább k db A_i -n

$k = n \Rightarrow$ WALL

$k = \lceil (n+1)/2 \rceil \Rightarrow$ többségi zárolás

k választásával hangolható a költség.

Miért jó a protokoll? \Rightarrow hasonlóan a többségi bizonyításhoz

Többségi zárolás

Csak az különbözik, hogy hogyan lehet globális zárat szerezni:

- Globális RLOCK A megszerzéséhez kell, hogy lokális RLOCK A_i legyen az A_i -k többségén
- Globális WLOCK A megszerzéséhez kell, hogy lokális WLOCK A_i legyen az A_i -k többségén

\Rightarrow Több üzenet szükséges az RLOCK megszerzéséhez, de kevesebb a WLOCK-hoz, mint az előbb.

Miért jó a többségi zárolás?

Két tranzakció nem tud egyszerre WLOCK A -t szerezni, mert mindkettőnek a példányok több, mint felére kellene WLOCK-ot kapnia \Rightarrow van olyan példány, amire mindkettő kapna

Hasonlóan nem lehet egy tranzakciónak WLOCK A -ja, egy másiknak RLOCK A -ja.

Viszont lehet két különböző tranzakciónak RLOCK A -ja, hiszen egy példányon is lehet ilyen.

Vége

Itt az anyag vége!