Adatbázisok elmélete 18. előadás

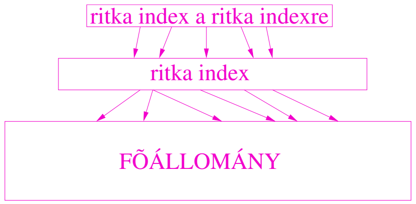
Katona Gyula Y.
Budapesti Műszaki és Gazdaságtudományi Egyetem
Számítástudományi Tsz.
I. B. 137/b
kiskat@cs.bme.hu
http://www.cs.bme.hu/˜kiskat

2004

Többszintes ritka index

Az indexen belül keresni arányos az indexállomány blokkjainak számával. Ez jóval kisebb, mint a főállomány lapszáma, de még mindig nagyon nagy lehet.

Ezért: többszintű index, vagyis index az indexre:



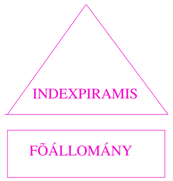
- a felső index még kisebb lesz, könnyebb lesz benne keresni
- a középső szint egyszerre indexe a főállománynak és "főállománya" a felső indexnek
- keresés: a legfelső szinten megkeressük a legnagyobb olyan bejegyzést, ami még kisebb a keresettnél és innen két lap beolvasásával (a megfelelő középső szintű indexlap és aztán a főállomány megfelelő lapja) megvan a keresett rekord
- a többi művelet hasonlóan megy (persze, ha módosul a főállomány, akkor esetleg mindegyik indexállományt is módosítani kell)

- eggyel több szint lesz, azaz eggyel több lapelérés kell a kezdeti keresés után, de a kezdeti keresés lerövidül
- nem csak kétszintű lehet a ritka index, hanem több is $\Longrightarrow$ dinamikusan is változhat $\Longrightarrow$ B-fa

B-fa

A ma ismert egyik legjobb és legelterjedtebb megoldás, m elágazásos B-fa vagy B_m-fa, lényegében ahogy algelből tanultuk:

- a fa levelei: a főállomány blokkjai
- a főállomány (a levelek) rendezett a keresési kulcs szerint
- minden levél ugyanolyan messze van a gyökértől
- a fa belső csúcsai: a különböző szintű indexek lapjai
- egy csúcs gyerekei: az indexlapon levő mutatóknak megfelelő eggyel lejjebb levő indexlapok (illetve alul levelek)
- m: egy lapra m indexrekord fér rá (kicsit más lesz egy belső csúcs szerkezete, mint algelből volt)
- minden lap legalább félig kitöltött, kivéve esetleg a gyökeret (minden csúcsnak legalább $\frac{m}{2}$ gyereke van, kivéve esetleg a gyökeret)



Műveletek

- **keresés**: ahogy algelből volt, a csúcsokban található kulcs-bejegyzések és mutatók mentén; arányos a fa magasságával, ami $O(log_m n)$, ha n blokkja van a főállománynak
- **beszúrás**: ahogy algelből volt, beszúrás után esetleg csúcsvágás(ok), de max. $O(log_m n)$
- **törlés**: ahogy algelből volt, törlés után esetleg csúcsösszevonás(ok), de max. $O(log_m n)$

Megjegyzések:

1. Ha m nagy $\Longrightarrow$ ritkán kell csúcsvágás/csúcsösszevonás.
2. általában m úgy van választva, hogy a fa magassága max. 4 legyen, ha az első lap a belső memóriában van, akkor elég 3 I/O művelet mindenhez

Sűrű index

Eddig feltettük, hogy a főállomány szabad és (lényegében) rendezett a keresési kulcs szerint. Hogyan érjük ezt el? Hogyan lehet több kulcs szerint is keresni?

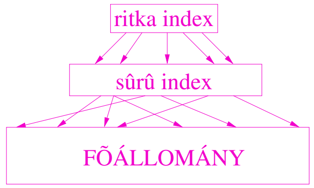
Erre megoldás a sűrű index:

- a főállomány minden rekordjához van egy indexbejegyzés $\Longrightarrow$ ugyanannyi bejegyzés lesz a sűrű indexben, mint ahány rekord van a főállományban, csak persze kisebbek a bejegyzések $\Longrightarrow$ sűrű index = főállomány kicsiben
- ez nem önálló állományszervezési technika (ellentétben a ritka index változataival), hanem csak kiegészítés, ami lehetővé teszi, hogy a főállományt szabadnak és rendezettnek tételezhessük fel

Haszna:

- szabaddá teszi a rekordokat (a főállomány ugyan kötött, de a sűrű index bejegyzései szabadon mozgathatók: építhető rá ritka index)
- rendezettnek mutatja a főállományt: a sűrű indexet úgy rendezzük, ahogy akarjuk
- sokkal kisebb, mint a főállomány, mégis egy az egyben megfelel neki

Tipikus használata:



A sűrű index ráépül a főállományra, erre építjük a valódi állományszervezést. A sűrű index miatt a főállomány szabadnak és rendezettnek tűnik.

Műveletek ebben a struktúrában

Úgy dolgozunk, mintha a sűrű index lenne a főállomány, innen már csak egy plusz lapelérés a valódi főállomány.

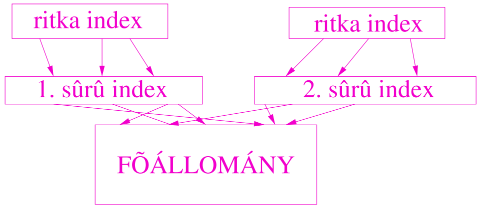
- **keresés:** keresés sűrűben, onnan egy lapelérés
- **beszúrás:** beszúrás sűrűbe, aztán valahova berakjuk a főállományba
- **törlés**: keresés, törlés a főállományból, törlés a sűrűből is

Hátrány

- plusz egy lapelérés kell a sűrű miatt
- karban kell tartani a sűrű indexet is mindig, amikor a főállomány változik

Nagy előnye a sűrű indexnek

Lehetővé teszi, hogy egy főállományra több különböző kulcs szerint is legyen index:



Itt minden sűrű index rendezett a megfelelő kulcs szerint és persze ha változik a főállomány, akkor mindegyik sűrűt is változtatni kell.

Példa:
A Személy(név, telefonszám, személyiszám, ...) sémában a személyiszám az elsődleges kulcs, ezért a rendszer eszerint rendezetten tárolja az adatokat és erre biztosan létre is hoz valami keresési struktúrát.

De ha mi szeretnénk a név-re is: kell egy sűrű index: invertált állomány.

Különböző technikák összevetése

- **hash**: konstans (gyakran 1) lapelérés átlagosan, de legrosszabb esetben lassú
- **ritka index**: korlátos viselkedés legrosszabb esetben is, dinamikus bővülés támogatása, rendezettség figyelembe vétele; B-fa esetén a gyakorlatban konstans lapelérés
- **sűrű index**: önmagában nem jó, csak kiegészítésül szolgál

Számolási példa

Egy állományt sűrű index, majd erre épített **1**-szintes ritka index segítségével szeretnénk tárolni. Adjon értelmes alsó becslést a szükséges lapok számára az alábbi feltételek mellett:

- az állomány $\mathbf{3 \cdot 10^6}$ rekordból áll,
- egy rekord hossza **300** Byte,
- egy lap mérete **1000** Byte,
- a kulcshossz **45** Byte,
- egy mutató hossza **5** Byte.

Megoldás:
A főállományban $\mathbf{3 \cdot 10^6}$ rekord van, mivel rekordok nem lóghatnak át laphatáron, ezért ehhez kell $\mathbf{10^6}$ lap.

A sűrű indexben annyi bejegyzés lesz, mint ahány rekord van a főállományban, azaz $\mathbf{3 \cdot 10^6}$.

Egy lapra pontosan **20** bejegyzés fér: ez $\mathbf{1,5 \cdot 10^5}$ lap.

Ez azt is jelenti, hogy a ritka indexben lesz legalább $\mathbf{1,5 \cdot 10^5}$ bejegyzés, ehhez kell még $\mathbf{7,5 \cdot 10^3}$ lap.

Ez összesen **1 157 500** lap.

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**.

Megoldandó problémák

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

(READ $A)_1$, (READ $A)_2$, $(A++)_1$,$(A++)_2$, (WRITE $A)_1$, (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.)

Alapfeltevések

- feltesszük, hogy a tranzakciók elemi műveletei: adat olvasása (READ A), számolás az adattal (pl. $A++$), adat írása (WRITE A)
- a fenti elemi utasításokat tartalmazó műveletsort a lekérdezésfeldolgozó állítja elő, elemzi a magas szintű lekérdezést/módosítást és azt ilyen elemi utasításokból álló sorozattá alakítja
- természetesen megengedett, hogy több helyről olvassunk, mielőtt számolunk, megengedett, hogy több adatból számoljunk ki valamit, illetve, hogy úgy írjunk, hogy nem is olvastuk ki azt az adatot
- ha egy A adatelemet ki kell olvasni, akkor ha már a belső memóriában (pufferban) van, akkor onnan olvasunk, különben a tárkezelővel még be kell előbb hozni a háttértárról
- ha írjuk az A adatelemet, akkor alapértelmezés szerint a pufferbe írjuk, kivéve speciális eseteket, amikor elő lesz írva, hogy valami azonnal kerüljön ki biztonságos háttértárra

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_1T_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
Read(A,t)		x	y
$t := t + 100$			
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_1T_2 soros ütemezés, se a T_2T_1 soros ütemezés hatása nem az, hogy (x, y)-ból $(2 \cdot (x + 100), 2 \cdot y + 100)$ lesz.

A T_1T_2 ütemezés $(2 \cdot (x + 100), 2 \cdot (y + 100))$ eredményt ad,
a T_2T_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 számolástól függetlenül ugyanaz lesz a hatás mint a T_1T_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):

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

illetve

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

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.