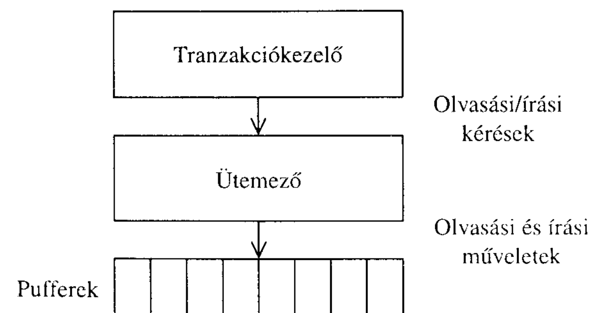
9. fejezet

Konkurenciavezérlés

A tranzakciók közötti egymásra hatás az adatbázis-állapot inkonzisztensé válását okozhatja, még akkor is, amikor a tranzakciók külön-külön megőrzik az állapot helyességét, és rendszerhiba sem történt. Ezért valamiképpen szabályoznunk kell, hogy a különböző tranzakciók egyes lépései milyen sorrendben következzenek egymás után. A lépések szabályozásának a feladatát az adatbázis-kezelő rendszer *ütemező* (scheduler) része végzi. Azt az általános folyamatot, amely biztosítja, hogy a tranzakciók egyidejű végrehajtása során megőrizzék a konzisztenciát, *konkurenciavezérlésnek* (concurrency control) nevezzük. Az ütemező szerepét a 9.1. ábrán láthatjuk.

Amint a tranzakciók az adatbáziselemek olvasását és írását kérik, ezek a kérések az ütemezőhöz kerülnek. Legtöbbször az ütemező közvetlenül végrehajtja az olvasásokat és írásokat, mégpedig először a pufferkezelőt hívja meg, amennyiben a szükséges adatbáziselem nincs a pufferben. Bizonyos esetekben azonban nem biztonságos azonnal végrehajtania a kéréseket. Az ütemezőnek késleltetnie kell a kérést, sőt valamilyen konkurenciavezérlési technikában az ütemező abortálhatja (leállíthatja a befejezés előtt, vagyis sikertelenül befejezheti) a kérést kiadó tranzakciót.

Először azt tanulmányozzuk hogyan biztosítható, hogy a konkurensen végrehajtott tranzakciók megőrizzék az adatbázis-állapot helyességét. Az elméleti követelményt *sorbarendezhetőségnek* (serializability) nevezzük, melynél van egy fontosabb, erősebb feltétel, amelyet *konfliktus-sorbarendezhetőségnek* (conflict-serializability) hívunk, és a legtöbb ütemező valójában ezt alkalmazza. Vizsgálni fogjuk az ütemezők legfontosabb megvalósítási technikáit: a zárolást, az időbélyegzést és az érvényesítést.



9.1. ábra. *Az ütemező fogadja a tranzakcióktól az olvasási/írási kéréseket, és vagy azonnal végrehajtja ezeket a pufferben, vagy késlelteti őket*

A zároláson alapuló ütemezésekről szóló rész tartalmazza a fontos „kétfázisú zárolás" fogalmát, amely a legelterjedtebb követelmény annak érdekében, hogy biztosítsuk az ütemezések sorbarendezhetőségét. A zármódok számos különböző halmazával ismerkedünk meg, amelyeket az ütemező a különféle alkalmazásokhoz alkalmazhat. A zárolási sémák közül azokat tanulmányozzuk, amelyekben a zárolható elemek beágyazott, illetve faszerkezetűek.

9.1. Soros és sorba rendezhető ütemezések

A konkurenciavezérlés tanulmányozását azzal kezdjük, hogy megvizsgáljuk, a konkurensen végrehajtott tranzakciók milyen feltételekkel tudják megőrizni az adatbázis-állapot konzisztenciáját. Az alapfeltevésünk, amelyet „helyességi elv"-nek (correctness principle) neveztünk, a 8.1.3. részben az volt, hogy ha minden egyes tranzakciót elkülönítve hajtunk végre (anélkül, hogy más tranzakció konkurensen futna), akkor az adatbázist konzisztens állapotból konzisztens állapotba alakítjuk. A gyakorlatban azonban a tranzakciók általában más tranzakciókkal egyidejűleg konkurensen futnak, emiatt a helyességi elvet közvetlenül nem alkalmazhatjuk. Így olyan „ütemezéseket" kell tekintenünk, amelyek biztosítják, hogy ugyanazt az eredményt állítják elő, mintha a tranzakciókat egyesével hajtottuk volna végre. Az egész fejezet fő témáját adják azok a módszerek, amelyek biztosítják, hogy a tranzakciók csak olyan módon legyenek konkurensen végrehajtva, mintha sorban egyesével futottak volna le.

9.1.1. Ütemezések

Az *ütemezés* (schedule) egy vagy több tranzakció által végrehajtott lényeges műveletek időrendben vett sorozata. Amikor a konkurenciavezérlést tanulmányozzuk, a lényeges olvasási és írási műveletek a központi memória puffereiben történnek, nem pedig lemezen. Vagyis egy A adatbáziselemet, amelyet valamelyik T tranzakció hozott be a pufferbe, ebben a pufferben nemcsak a T tudja olvasni vagy írni, hanem más tranzakciók is hozzáférhetnek az A-hoz. Idézzük fel a 8.1.4. részből, hogy a `READ` (`OLVASÁS`) és a `WRITE` (`ÍRÁS`) műveletek először meghívnak egy `INPUT` utasítást, hogy az adatbáziselemet a lemezről betöltsék, ha még nincs a pufferben, egyébként pedig a `READ` és `WRITE` műveletek közvetlenül a pufferben hozzáférnek az elemhez. Ezért csupán a `READ` és `WRITE` műveletek és a sorrendjük számít, amikor a konkurenciával foglalkozunk, és az `INPUT`, illetve `OUTPUT` műveleteket figyelmen kívül fogjuk hagyni.

9.1. példa: Tekintsünk két tranzakciót és az adatbázison való hatásukat, amikor egy meghatározott sorrendben hajtjuk végre a műveleteiket. A T_1 és T_2 tranzakciók fő

T_1	T_2
READ(A,t)	READ(A,s)
t := t+100	s := s*2
WRITE(A,t)	WRITE(A,s)
READ(B,t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)

9.2. ábra. *Két tranzakció*

műveletei a 9.2. ábrán találhatók. A *t* és *s* változók a T_1-nek, illetve T_2-nek megfelelő helyi változók, és *nem* adatbáziselemek.

Tételezzük fel, hogy az adatbázis-állapoton az egyetlen konzisztenciamegszorítás az $A = B$. Mivel a T_1 az A-hoz és a B-hez is hozzáad 100-at, és a T_2 az A-t és a B-t is megszorozza 2-vel, tudjuk, hogy az egyes tranzakciók egymástól elkülönítve futva megőrzik a konzisztenciát. □

9.1.2. Soros ütemezések

Azt mondjuk, hogy egy ütemezés *soros* (serial schedule), ha úgy épül fel a tranzakciós műveletekből, hogy először az egyik tranzakció összes műveletét tartalmazza, majd azután egy másik tranzakció összes műveletét stb., miközben nem cseréli fel a műveleteket. Pontosabban kifejezve, egy S ütemezés soros, ha bármely két T és T' tranzakcióra, ha T-nek van olyan művelete, amely megelőzi a T' valamelyik műveletét, akkor a T összes művelete megelőzi a T' valamennyi műveletét.

9.2. példa: A 9.2. ábrán szereplő tranzakcióknak két soros ütemezése van, az egyikben T_1 megelőzi T_2-t, a másikban T_2 előzi meg T_1-et. A 9.3. ábra azt az eseménysoro-

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250

9.3. ábra. *Soros ütemezés, amelyben T_1 megelőzi T_2-t*

T_1	T_2	A	B
		25	25
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	50	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		50
READ(A,t)			
t := t+100			
WRITE(A,t)		150	
READ(B,t)			
t := t+100			
WRITE(B,t)			150

9.4. ábra. *Soros ütemezés, amelyben T_2 megelőzi T_1-t*

zatot mutatja, amikor T_1 megelőzi T_2-t, és a kezdeti állapot $A = B = 25$. Azt a megállapodást követjük, hogy időrendi sorrendben függőlegesen lefelé írunk. Továbbá a megjelenített A és B értékek a központi memória pufferbeli értékeire utalnak, nem szükségképpen a lemezen tárolt értékeire.

A 9.4. ábrán látjuk a másik soros ütemezést, amelyben T_2 megelőzi T_1-et. A kezdeti állapot legyen megint $A = B = 25$. Megjegyezzük, hogy A és B végső értéke különböző a két ütemezésben, mégpedig mindkettő értéke 250, ha a T_1 fut először, és 150, ha a T_2 fut előbb. De nem is a végeredmény a központi kérdés addig, amíg a konzisztenciát megőrizzük. Általában nem várjuk el, hogy az adatbázis végső állapota független legyen a tranzakciók sorrendjétől. □

A soros ütemezést úgy ábrázolhatjuk, mint ahogyan a 9.3. ábrán vagy a 9.4. ábrán látható, a műveleteket az előfordulásuk sorrendjében soroljuk fel. Másrészt, mivel a soros ütemezésben a műveletek sorrendje csak magától a tranzakciók sorrendjétől függ, ezért a soros ütemezést néha a tranzakciók felsorolásával fogjuk megadni. Így a 9.3. ábra ütemezését (T_1, T_2) reprezentálja, a 9.4. ábráét pedig (T_2, T_1).

9.1.3. Sorba rendezhető ütemezések

A tranzakciókra vonatkozó helyességi elv szerint minden soros ütemezés megőrzi az adatbázis-állapot konzisztenciáját. Vajon van-e más ütemezés is, amely szintén biztosítja a konzisztencia megmaradását? Igen, ilyen létezik, ahogyan ezt a következő példa mutatja. Általában azt mondjuk, hogy egy ütemezés *sorba rendezhető* (serializable schedule), ha ugyanolyan hatással van az adatbázis állapotára, mint valamelyik soros ütemezés, függetlenül attól, hogy mi volt az adatbázis kezdeti állapota.

9.3. példa: A 9.5. ábrán látjuk a 9.1. példában szereplő két tranzakciónak egy sorba rendezhető, ám nem soros ütemezését. Ebben az ütemezésben T_2 azután van hatással

az A-ra, miután a T_1 volt, de mielőtt a T_1 hatással lenne a B-re. Mégis azt látjuk, hogy ebben az ütemezésben a két tranzakció hatása megegyezik a 9.3. ábrán látható (T_1, T_2) soros ütemezés hatásával. Ahhoz, hogy meggyőződjünk az állítás igazságáról, nemcsak azt az esetet kell megnéznünk, amely a 9.5. ábrán látható, amikor az adatbázis-állapot $A = B = 25$-ről indul, hanem bármely konzisztens adatbázis kiindulási állapotból kiindulva. Mivel minden konzisztens adatbázis-állapotban az $A = B = c$ valamely c konstanssal, nem nehéz levezetnünk, hogy a 9.5. ábra ütemezésében az A-nak is és a B-nek is $2(c + 100)$ lesz az értéke, és így bármelyik konzisztens állapotból indulunk ki, a konzisztenciát megőrizzük.

Másrészt tekintsük a 9.6. ábrán található ütemezést. Világos, hogy ez nem soros, de ami lényegesebb, nem is sorba rendezhető. Meggyőződhetünk arról, hogy nem sorba rendezhető, ugyanis legyen a kiindulási konzisztens állapotban $A = B = 25$, és az adatbázis inkonzisztens állapotba kerül, amikor $A = 250$ és $B = 150$ lesz. Megjegyezzük,

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250

9.5. ábra. *Sorba rendezhető, de nem soros ütemezés*

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		50
READ(B,t)			
t := t+100			
WRITE(B,t)			150

9.6. ábra. *Nem sorba rendezhető ütemezés*

hogy ebben a műveleti sorrendben, a T_1 dolgozik előbb az A-val, viszont T_2 dolgozik előbb a B-vel, ennek hatásaként másképpen kell kiszámolnunk A-t és B-t, vagyis $A := 2(A + 100)$, szemben $B := 2B + 100$-zal. A 9.6. ábrán található ütemezés olyan viselkedést mutat, amelyet a konkurenciavezérlési működésekkel el kell kerülnünk. □

9.1.4. A tranzakció szemantikájának hatása

A sorbarendezhetőségi vizsgálatainkban eddig a tranzakciók által végrehajtott műveleteket néztük meg részletesen annak érdekében, hogy meghatározzuk sorba rendezhető-e az ütemezés. Azonban a tranzakciók részletei is számítanak, ahogyan ezt a következő példából láthatjuk.

9.4. példa: Tekintsük a 9.7. ábrán látható ütemezést, amely csak a T_2 által végrehajtott számításokban különbözik a 9.6. ábrától, mégpedig abban, hogy a T_2 nem 2-vel szorozza meg A-t és B-t, hanem 1-gyel.[1] Ekkor A és B értéke az ütemezés végén megegyezik, és könnyen ellenőrizhetjük, hogy a konzisztens kezdeti állapottól függetlenül a végállapot is konzisztens lesz. Valójában az egyetlen végállapot az, amelyet vagy a (T_1, T_2) vagy a (T_2, T_1) soros ütemezés eredményez. □

Sajnos, az ütemező számára nem reális a tranzakciós számítások részleteinek figyelembevétele. Mivel a tranzakciók gyakran tartalmaznak általános célú programozási

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*1		
	WRITE(A,s)	125	
	READ(B,s)		
	s := s*1		
	WRITE(B,s)		25
READ(B,t)			
t := t+100			
WRITE(B,t)			125

9.7. ábra. *Egy olyan ütemezés, amely csak a tranzakciók részletezett viselkedése miatt sorba rendezhető*

[1] Valaki jogosan kérdezheti, hogy miért is viselkedik így egy tranzakció? Mégis a példa kedvéért ezt most hagyjuk figyelmen kívül. Valójában több elfogadható tranzakciót is helyettesíthetnénk a T_2 helyére, amely az A-t és B-t változatlanul hagyná. Például amikor a T_2 csak egyszerűen beolvassa az A-t és B-t, és kiíratja az értéküket. Vagy T_2 a felhasználótól kérhet be adatokat, hogy kiszámoljon egy F tényezőt, amivel megszorozza az A-t és a B-t, és előfordulhat olyan felhasználói input, amelyre az $F = 1$.

nyelven írt kódokat éppúgy, mint SQL vagy más magas szintű nyelv utasításait, néha nagyon nehéz megválaszolni azokat a kérdéseket, mint pl. „ez a tranzakció az A-t egy 1-től különböző konstanssal szorozta-e meg?". Az ütemezőnek azonban látnia kell a tranzakciók olvasási és írási kéréseit, így tudhatja, hogy az egyes tranzakciók mely adatbáziselemeket olvasták be, és mely elemek *változhattak* meg. Az ütemező feladatának az egyszerűsítésére megszokott az a feltételezés, hogy:

- Bármely A adatbáziselemnek egy T tranzakció olyan értéket ír be, amely az adatbázis-állapottól függ oly módon, hogy ne forduljon elő aritmetikai egybeesés.

Más szóval kifejezve, ha a T tudna az A-ra olyan hatással lenni, hogy az adatbázisállapot inkonzisztenssé váljék, akkor a T ezt meg is teszi. Ezt a feltevést a 9.2. részben pontosítjuk, amikor a sorbarendezhetőség biztosítására adunk meg elégséges feltételeket.

9.1.5. A tranzakciók és ütemezések jelölése

Ha elfogadjuk, hogy egy tranzakció által végrehajtott pontos számítások tetszőlegesek lehetnek, akkor nem szükséges a helyi számítási lépések részleteit néznünk, mint amilyen a `t := t+100`. Csak a tranzakciók által végrehajtott olvasások és írások számítanak. Így a tranzakciókat és az ütemezéseket rövidebben jelölhetjük. Ekkor $r_T(X)$ és $w_T(X)$ tranzakcióműveletek, és azt jelentik, hogy a T tranzakció olvassa (r, az angol *read* = olvasás rövidítése), illetve írja (w, az angol *write* = írás rövidítése) az X adatbáziselemet. Továbbá, mivel a tranzakcióinkat $T_1, T_2, \ldots$-vel fogjuk általában jelölni, így megállapodunk abban, hogy $r_i(X)$ és $w_i(X)$ ugyanazt jelöli, mint $r_{T_i}(X)$, illetve $w_{T_i}(X)$.

9.5. példa: A 9.2. ábrán látható tranzakciók az alábbi módon írhatók fel:

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$;
T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$;

Megjegyezzük, hogy nem említettük sehol a t és az s helyi változókat, és nem jelöltük azt sem, hogy mi történt a beolvasás után az A-val és B-vel. Intuíció alapján ezt úgy értelmezzük, hogy az adatbáziselemek megváltozásában a „legrosszabbat fogjuk feltételezni".

Egy másik példaként nézzük meg a T_1 és T_2-nek a 9.5. ábrán látható sorba rendezhető ütemezését. Ezt az ütemezést átírva:

$r_1(A)$; $w_1(A)$; $r_2(A)$; $w_2(A)$; $r_1(B)$; $w_1(B)$; $r_2(B)$; $w_2(B)$;
□

Pontosítva a jelölést:

1. Egy *tranzakció műveletét* (action of transaction) $r_i(X)$ vagy $w_i(X)$ formában fejezünk ki, amely azt jelenti, hogy a T_i tranzakció olvassa *(read)*, illetve írja *(write)* az X adatbáziselemet.

2. Egy T_i *tranzakció* az i indexű műveletekből álló sorozat.
3. A $\mathcal{T}$ tranzakciók halmazának egy S *ütemezése* olyan műveletek sorozata, amelyben minden $\mathcal{T}$ halmazbeli T_i tranzakcióra teljesül, hogy T_i műveletei ugyanabban a sorrendben fordulnak elő az S-ben, mint ahogy magában a T_i definíciójában szerepeltek. Azt mondjuk, hogy az S az őt alkotó tranzakciók műveleteinek *átlapolása* (interleaving).

Például a 9.5. példában található ütemezésben az összes 1-es indexű művelet ugyanabban a sorrendben szerepel, mint ahogy a T_1 definíciójában volt, és az összes 2-es indexű művelet ugyanabban a sorrendben fordul elő, mint ahogy a T_2 definíciójában szerepelt.

9.1.6. Feladatok

* **9.1.1. feladat:** Egy repülő-helyfoglalási rendszer végzi a T_1 tranzakciót, és az alábbi lépéseket hajtja végre:

i) A vevőtől lekérdezzük a keresett járat idejét és városait. A keresett járatokról az információ az A és B adatbáziselemekben található (valószínűleg lemezblokkokban), amelyeket a rendszer a lemezről érhet el.
ii) A vevőnek elmondjuk a feltételeket, és kiválasztjuk a járatot, amelynek adatai, beleértve a járatra való foglalás számát is, a B-ben található. A járatra való helyfoglalást a vevő végzi el.
iii) A vevő kiválasztja a járatra az ülőhelyet, a járat ülőhelyadatait a C adatbáziselem tartalmazza.
iv) A rendszer megkapja a vevő hitelkártyaszámát, és hozzáfűzi a számlát a számlák jegyzékéhez, mely jegyzék a járat D adatbáziselemében található.
v) A vevő telefonszámát és a járat adatait hozzáadjuk egy másik jegyzékhez az E adatbáziselemben, hogy egy faxot tudjunk küldeni a járatra való érvényesítéshez.

Fejezzük ki a T_1 tranzakciót r és w műveletek sorozataként!

*! **9.1.2. feladat:** Ha van két tranzakciónk, az egyik 4, a másik pedig 6 műveletből áll, akkor ezeknek a tranzakcióknak mennyi átlapolása (ütemezése) lehetséges?

9.2. Konfliktus-sorbarendezhetőség

Most egy olyan elégséges feltételt adunk meg, mely biztosítja egy ütemezés sorbarendezhetőségét. A piaci rendszerek ütemezői a tranzakciók sorbarendezhetőségére általában ezt az erősebb feltételt biztosítják, amelyet „konfliktus-sorbarendezhetőségnek" nevezünk. Ez a *konfliktus* (conflict) fogalmon alapul: amely olyan egymást követő műveletpár az ütemezésben, amelynek ha a sorrendjét felcseréljük, akkor legalább az egyik tranzakció viselkedése megváltozhat.

9.2.1. Konfliktusok

Azzal kezdjük, hogy vegyük észre a legtöbb műveletpár *nincs* konfliktusban a fenti értelemben. Ugyanis, tételezzük fel, hogy T_i és T_j különböző tranzakciók, vagyis $i \neq j$.

1. $r_i(X)$; $r_j(Y)$ sohasem konfliktus, még akkor sem, ha $X = Y$. Ennek az az oka, hogy egyik lépés sem változtatja meg az értékeket.
2. $r_i(X)$; $w_j(Y)$ nincs konfliktusban, feltéve, ha $X \neq Y$. Ennek az az oka, hogy a T_j írhatja az Y-t, mielőtt a T_i beolvasta az X-et, az X értéke ugyanis ettől nem változik. Annak sincs hatása a T_j-re, hogy a T_i olvassa az X-et, ugyanis ez nincs hatással arra, hogy milyen értéket ír be a T_j az Y-ba.
3. $w_i(X)$; $r_j(Y)$ nincs konfliktusban, ha $X \neq Y$, ugyanazért, mint a 2.
4. Szintén hasonlóan $w_i(X)$; $w_j(Y)$ sincs konfliktusban mindaddig, amíg $X \neq Y$.

Másrészt három esetben nem cserélhetjük fel a műveletek sorrendjét:

a) Ugyanannak a tranzakciónak két művelete, pl. $r_i(X)$; $w_i(Y)$ konfliktus. Ennek az az oka, hogy egyetlen tranzakción belül a műveletek sorrendje rögzített, és az adatbázis-kezelő rendszer ezt a sorrendet nem rendezheti át újra.
b) Különböző tranzakciók ugyanarra az adatbáziselemre való írása konfliktus. Vagyis $w_i(X)$; $w_j(X)$ konfliktus. Ennek az az oka, mint már írtuk, hogy az X értéke az marad, amelyet a T_j számolt ki. Ha felcseréljük a sorrendjüket, hogy $w_j(X)$; $w_i(X)$, akkor az X-nek a T_i által kiszámított értéke marad meg. Az a feltevésünk, hogy „nincs egybeesés", azt adja, hogy a T_i és a T_j által írt értékek lehetnek különbözőek, és ezért az adatbázis valamelyik kezdeti állapotára különbözni fognak.
c) Különböző tranzakcióknak ugyanabból az adatbáziselemből való olvasása és írása is konfliktus. Vagyis $r_i(X)$; $w_j(X)$ konfliktus, és $w_i(X)$; $r_j(X)$ is konfliktus. Ha átvisszük $w_j(X)$-et $r_i(X)$ elé, akkor a T_i által olvasott X-beli érték az lesz, amelyet a T_j írt, amiről pedig feltételeztük, hogy nem szükségképpen egyezik meg az X korábbi értékével. Tehát $r_i(X)$ és $w_j(X)$ sorrendjének cseréje befolyásolja, hogy T_i milyen értéket olvas X-ből, ez pedig befolyásolja a T_i működését.

Levonhatjuk a következtetést, hogy különböző tranzakciók bármely két műveletének sorrendje felcserélhető, hacsak nem

1. Ugyanarra az adatbáziselemre vonatkoznak, és
2. Legalább az egyik művelet írás.

Ezt az elvet kiterjesztve tetszőleges ütemezést véve annyi nem konfliktusos cserét készíthetünk, amennyit csak kívánunk, abból a célból, hogy az ütemezést soros ütemezéssé alakítsuk át. Ha ezt meg tudjuk tenni, akkor az eredeti ütemezés sorba rendezhető volt, ugyanis az adatbázis állapotára való hatása változatlan marad minden nem konfliktusos cserével.

Azt mondjuk, hogy két ütemezés *konfliktusekvivalens* (conflict-equivalent), ha szomszédos műveletek nem konfliktusos cseréinek sorozatával az egyiket átalakíthatjuk a másikká. Azt mondjuk, hogy egy ütemezés *konfliktus-sorbarendezhető* (conflict-serializable schedule), ha konfliktusekvivalens valamely soros ütemezéssel. Megjegyezzük, hogy a konfliktus-sorbarendezhetőség elégséges feltétele a sorbarendezhetőségnek, vagyis egy konfliktus-sorbarendezhető ütemezés sorba rendezhető ütemezés is egyben. Azonban a konfliktus-sorbarendezhetőség nem szükséges ahhoz, hogy egy ütemezés sorba rendezhető legyen, mégis általában ezt a feltételt ellenőrzik a piaci rendszerek ütemezői, amikor a sorbarendezhetőséget kell biztosítaniuk.

9.6. példa: Legyen az ütemezés

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

a 9.5. példából. Azt állítjuk, hogy ez az ütemezés konfliktus-sorbarendezhető. A 9.8. ábrán látható a cserék sorozata, amellyel ez az ütemezés átalakítható a (T_1, T_2) soros ütemezéssé, ahol az összes T_1-beli művelet megelőzi az összes T_2-beli műveletet. Aláhúztuk azokat a szomszédos műveletpárokat, amelyeket felcserélünk az egyes lépésekben. □

$r_1(A)$; $w_1(A)$; $r_2(A)$; $\underline{w_2(A); r_1(B)}$; $w_1(B)$; $r_2(B)$; $w_2(B)$;
$r_1(A)$; $w_1(A)$; $\underline{r_2(A); r_1(B)}$; $w_2(A)$; $w_1(B)$; $r_2(B)$; $w_2(B)$;
$r_1(A)$; $w_1(A)$; $r_1(B)$; $r_2(A)$; $\underline{w_2(A); w_1(B)}$; $r_2(B)$; $w_2(B)$;
$r_1(A)$; $w_1(A)$; $r_1(B)$; $\underline{r_2(A); w_1(B)}$; $w_2(A)$; $r_2(B)$; $w_2(B)$;
$r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$; $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$;

9.8. ábra. *Egy konfliktus-sorbarendezhető ütemezés szomszédos műveletek felcserélésével való átalakítása soros ütemezéssé*

9.2.2. Megelőzési gráfok és teszt a konfliktus-sorbarendezhetőségre

Viszonylag könnyű megvizsgálnunk egy S ütemezést, és eldöntenünk, hogy konfliktus-sorbarendezhető-e vagy nem. Az az alapötlet, hogy ha valahol konfliktusban álló műveletek szerepelnek az S-ben, ezeket a műveleteket végrehajtó tranzakcióknak ugyanabban a sorrendben kell előfordulniuk a konfliktusekvivalens soros ütemezésekben, mint ahogyan az S-ben voltak. Tehát a konfliktusban álló műveletpárok megszorítást adnak a feltételezett konfliktusekvivalens soros ütemezésben a tranzakciók sorrendjére. Ha ezek a megszorítások nem mondanak egymásnak ellent, akkor találhatunk konfliktusekvivalens soros ütemezést. Ha pedig ellentmondanak egymásnak, akkor tudjuk, hogy nincs ilyen soros ütemezés.

Adott a T_1 és T_2 tranzakcióknak, esetleg további tranzakcióknak, egy S ütemezése. Azt mondjuk, hogy T_1 *megelőzi* T_2-t, és $T_1 <_S T_2$-vel jelöljük, ha van a T_1-ben olyan A_1 művelet, és a T_2-ben olyan A_2, hogy

1. A_1 megelőzi A_2-t az S-ben,
2. A_1 és A_2 ugyanarra az adatbáziselemre vonatkoznak, és
3. A_1 és A_2 közül legalább az egyik írás művelet.

Megjegyezzük, hogy ezek pontosan azok a feltételek, amikor nem lehet felcserélni az A_1 és A_2 sorrendjét. Tehát, A_1 az A_2 előtt szerepel bármely az S-sel konfliktusekvivalens ütemezésben. Ennek eredményeként, ha ezek közül az ütemezések közül az egyik soros ütemezés, akkor ebben T_1-nek meg kell előznie T_2-t.

Ezeket a megelőzéseket a *megelőzési gráfban* (precedence graph) összegezhetjük. A megelőzési gráf csomópontjai az S ütemezés tranzakciói. Ha a tranzakciókat T_i-vel jelöljük az i függvényében, akkor a T_i-nek megfelelő csomópontot az i egésszel jelöljük. Az i csomópontból a j csomópontba vezet irányított él, ha $T_i <_S T_j$.

9.7. példa: A következő S ütemezés a T_1, T_2, T_3 három tranzakciót tartalmazza:

S: $r_2(A)$; $r_1(B)$; $w_2(A)$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $r_2(B)$; $w_2(B)$;

Ha az A-val kapcsolatos műveleteket nézzük meg, akkor több okot találunk, hogy miért igaz a $T_2 <_S T_3$. Például $r_2(A)$ az S-ben $w_3(A)$ előtt áll, és $w_2(A)$ az $r_3(A)$ és a $w_3(A)$ előtt is áll. A három észrevételünk közül bármelyik elegendő, hogy igazoljuk, valóban vezet él a 2-ből 3-ba a 9.9. ábrán szereplő megelőzési gráfban.

9.9. ábra. *A 9.7. példa S ütemezéséhez tartozó megelőzési gráf*

Hasonló módon ha megnézzük a B-vel kapcsolatos műveleteket, akkor szintén több okot találunk, hogy miért igaz a $T_1 <_S T_2$. Például $r_1(B)$ művelet a $w_2(B)$ művelet előtt áll. Tehát az S megelőzési gráfjában az 1-ből 2-be szintén vezet él. Tulajdonképpen ezek és csak ezek az élek azok, amelyeket az S ütemezésben szereplő műveletek sorrendjéből tudunk ellenőrizni. □

Van egy egyszerű szabály, amivel megmondhatjuk, hogy egy S ütemezés konfliktus-sorbarendezhető-e:

- Rajzoljuk fel az S megelőzési gráfját, és nézzük meg tartalmaz-e kört!

Ha igen, akkor S nem konfliktus-sorbarendezhető. Ha pedig a gráf körmentes, akkor S konfliktus-sorbarendezhető, továbbá a csomópontok bármelyik topologikus sorrendje[2] megadja a konfliktusekvivalens soros sorrendet.

9.8. példa: A 9.9. ábra megelőzési gráfja körmentes, így a 9.7. példa S ütemezése konfliktus-sorbarendezhető. A csomópontoknak, azaz a tranzakcióknak csak egyetlen sorrendje van, amely konzisztens a gráf éleivel, és ez: (T_1, T_2, T_3). Megjegyezzük,

[2] Egy körmentes gráf *topologikus sorrendje* a csomópontok bármely olyan rendezése, amelyben minden $a \to b$ élre, az a csomópont megelőzi a b csomópontot a topologikus rendezésben. Bármely körmentes gráfnak találhatunk topologikus rendezettséget úgy, hogy többszörösen ismételve töröljük azokat a csomópontokat, amelyeknek nincs megelőzője a maradék csomópontok között.

Miért nem szükséges konfliktus-sorbarendezhetőség a sorbarendezhetőséghez?

Egy példát már láttunk a 9.7. ábrán. Akkor megnéztük, hogy a T_2 által végrehajtott speciális számítások miatt hogyan vált az ütemezés sorba rendezhetővé. Pedig a 9.7. ábra ütemezése nem konfliktus-sorbarendezhető, ugyanis az A-t T_1 írja előbb, a B-t pedig a T_2. Mivel sem az A írását, sem a B írását nem lehet átrendezni, semmilyen módon nem kerülhet T_1 összes művelete a T_2 összes művelete elé, sem fordítva.

Azonban vannak olyan sorba rendezhető, de nem konfliktus-sorbarendezhető ütemezések is, amelyek nem függnek a tranzakciók által végrehajtott számításoktól. Például tekintsük a T_1, T_2, és T_3 három tranzakciót, amelyek mindegyike X értékét írja. A T_1 és T_2 az Y-nak is ír értéket, mielőtt az X-nek írnának értéket. Az egyik lehetséges ütemezés, amely itt éppen soros is, a következő:

S_1: $w_1(Y)$; $w_1(X)$; $w_2(Y)$; $w_2(X)$; $w_3(X)$.

Az S_1 ütemezés X értékének a T_3 által írt értéket, Y értékének pedig a T_2 által írt értéket adja. Ugyanezt végzi a következő ütemezés is:

S_2: $w_1(Y)$; $w_2(Y)$; $w_2(X)$; $w_1(X)$; $w_3(X)$.

Intuíció alapján átgondolva annak, hogy a T_1 és a T_2 milyen értéket ír az X-be, nincs hatása, ugyanis a T_3 felülírja az értékeket. Emiatt S_1 és S_2 az X-nek is és az Y-nak is ugyanazt az értéket adja. Mivel az S_1 soros, és az S_2-nek bármely adatbázis-állapotra ugyanaz a hatása, mint az S_1-nek, tehát S_2 sorba rendezhető. Ugyanakkor mivel nem tudjuk felcserélni $w_1(Y)$-t $w_2(Y)$-nal, és nem tudjuk felcserélni $w_1(X)$-et $w_2(X)$-szel, így cseréken keresztül nem lehet az S_2-t valamelyik soros ütemezéssé átalakítani. Tehát S_2 sorba rendezhető, de nem konfliktus-sorbarendezhető.

hogy az S-et valóban át lehet alakítani olyan ütemezéssé, amelyben a három tranzakció mindegyikének az összes művelete ugyanebben a sorrendben van, és ez a soros ütemezés:

S': $r_1(B)$; $w_1(B)$; $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$; $r_3(A)$; $w_3(A)$;

Ahhoz, hogy belássuk, megkaphatjuk az S-ből az S'-t szomszédos elemek cseréjével, az első észrevételünk, hogy az $r_1(B)$-t konfliktus nélkül az $r_2(A)$ elé hozhatjuk. Ezután három cserével a $w_1(B)$-t közvetlenül az $r_1(B)$ utánra tudjuk cserélni, ugyanis mindegyik közbeeső művelet az A-ra vonatkozik, és a B-re nem. Ezután az $r_2(B)$-t és a $w_2(B)$-t csak az A-ra vonatkozó műveleteken keresztül át tudjuk vinni pontosan a $w_2(A)$ utáni helyzetbe, amivel megkapjuk az S'-t. □

Megjegyezzük, hogy ezek pontosan azok a feltételek, amikor nem lehet felcserélni az A_1 és A_2 sorrendjét. Tehát, A_1 az A_2 előtt szerepel bármely az S-sel konfliktusekvivalens ütemezésben. Ennek eredményeként, ha ezek közül az ütemezések közül az egyik soros ütemezés, akkor ebben T_1-nek meg kell előznie T_2-t.

Ezeket a megelőzéseket a *megelőzési gráfban* (precedence graph) összegezhetjük. A megelőzési gráf csomópontjai az S ütemezés tranzakciói. Ha a tranzakciókat T_i-vel jelöljük az i függvényében, akkor a T_i-nek megfelelő csomópontot az i egésszel jelöljük. Az i csomópontból a j csomópontba vezet irányított él, ha $T_i <_S T_j$.

9.7. példa: A következő S ütemezés a T_1, T_2, T_3 három tranzakciót tartalmazza:

S: $r_2(A)$; $r_1(B)$; $w_2(A)$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $r_2(B)$; $w_2(B)$;

Ha az A-val kapcsolatos műveleteket nézzük meg, akkor több okot találunk, hogy miért igaz a $T_2 <_S T_3$. Például $r_2(A)$ az S-ben $w_3(A)$ előtt áll, és $w_2(A)$ az $r_3(A)$ és a $w_3(A)$ előtt is áll. A három észrevételünk közül bármelyik elegendő, hogy igazoljuk, valóban vezet él a 2-ből 3-ba a 9.9. ábrán szereplő megelőzési gráfban.

9.9. ábra. *A 9.7. példa S ütemezéséhez tartozó megelőzési gráf*

Hasonló módon ha megnézzük a B-vel kapcsolatos műveleteket, akkor szintén több okot találunk, hogy miért igaz a $T_1 <_S T_2$. Például $r_1(B)$ művelet a $w_2(B)$ művelet előtt áll. Tehát az S megelőzési gráfjában az 1-ből 2-be szintén vezet él. Tulajdonképpen ezek és csak ezek az élek azok, amelyeket az S ütemezésben szereplő műveletek sorrendjéből tudunk ellenőrizni. □

Van egy egyszerű szabály, amivel megmondhatjuk, hogy egy S ütemezés konfliktus-sorbarendezhető-e:

- Rajzoljuk fel az S megelőzési gráfját, és nézzük meg tartalmaz-e kört!

Ha igen, akkor S nem konfliktus-sorbarendezhető. Ha pedig a gráf körmentes, akkor S konfliktus-sorbarendezhető, továbbá a csomópontok bármelyik topologikus sorrendje[2] megadja a konfliktusekvivalens soros sorrendet.

9.8. példa: A 9.9. ábra megelőzési gráfja körmentes, így a 9.7. példa S ütemezése konfliktus-sorbarendezhető. A csomópontoknak, azaz a tranzakcióknak csak egyetlen sorrendje van, amely konzisztens a gráf éleivel, és ez: (T_1, T_2, T_3). Megjegyezzük, hogy az S-et valóban át lehet alakítani olyan ütemezéssé, amelyben a három tranzakció mindegyikének az összes művelete ugyanebben a sorrendben van, és ez a soros ütemezés:

S'': $r_1(B)$; $w_1(B)$; $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$; $r_3(A)$; $w_3(A)$;

Ahhoz, hogy belássuk, megkaphatjuk az S-ből az S'-t szomszédos elemek cseréjével, az első észrevételünk, hogy az $r_1(B)$-t konfliktus nélkül az $r_2(A)$ elé hozhatjuk. Ezután három cserével a $w_1(B)$-t közvetlenül az $r_1(B)$ utánra tudjuk cserélni, ugyanis mindegyik közbeeső művelet az A-ra vonatkozik, és a B-re nem. Ezután az $r_2(B)$-t és a $w_2(B)$-t csak az A-ra vonatkozó műveleteken keresztül át tudjuk vinni pontosan a $w_2(A)$ utáni helyzetbe, amivel megkapjuk az S'-t. □

[2] Egy körmentes gráf *topologikus sorrendje* a csomópontok bármely olyan rendezése, amelyben minden $a \to b$ élre, az a csomópont megelőzi a b csomópontot a topologikus rendezésben. Bármely körmentes gráfnak találhatunk topologikus rendezettséget úgy, hogy többszörösen ismételve töröljük azokat a csomópontokat, amelyeknek nincs megelőzője a maradék csomópontok között.

Miért nem szükséges konfliktus-sorbarendezhetőség a sorbarendezhetőséghez?

Egy példát már láttunk a 9.7. ábrán. Akkor megnéztük, hogy a T_2 által végrehajtott speciális számítások miatt hogyan vált az ütemezés sorba rendezhetővé. Pedig a 9.7. ábra ütemezése nem konfliktus-sorbarendezhető, ugyanis az A-t T_1 írja előbb, a B-t pedig a T_2. Mivel sem az A írását, sem a B írását nem lehet átrendezni, semmilyen módon nem kerülhet T_1 összes művelete a T_2 összes művelete elé, sem fordítva.

Azonban vannak olyan sorba rendezhető, de nem konfliktus-sorbarendezhető ütemezések is, amelyek nem függnek a tranzakciók által végrehajtott számításoktól. Például tekintsük a T_1, T_2, és T_3 három tranzakciót, amelyek mindegyike X értékét írja. A T_1 és T_2 az Y-nak is ír értéket, mielőtt az X-nek írnának értéket. Az egyik lehetséges ütemezés, amely itt éppen soros is, a következő:

S_1: $w_1(Y)$; $w_1(X)$; $w_2(Y)$; $w_2(X)$; $w_3(X)$.

Az S_1 ütemezés X értékének a T_3 által írt értéket, Y értékének pedig a T_2 által írt értéket adja. Ugyanezt végzi a következő ütemezés is:

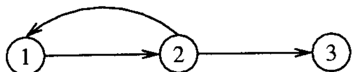
S_2: $w_1(Y)$; $w_2(Y)$; $w_2(X)$; $w_1(X)$; $w_3(X)$.

Intuíció alapján átgondolva annak, hogy a T_1 és a T_2 milyen értéket ír az X-be, nincs hatása, ugyanis a T_3 felülírja az értékeket. Emiatt S_1 és S_2 az X-nek is és az Y-nak is ugyanazt az értéket adja. Mivel az S_1 soros, és az S_2-nek bármely adatbázis-állapotra ugyanaz a hatása, mint az S_1-nek, tehát S_2 sorba rendezhető. Ugyanakkor mivel nem tudjuk felcserélni $w_1(Y)$-t $w_2(Y)$-nal, és nem tudjuk felcserélni $w_1(X)$-et $w_2(X)$-szel, így cseréken keresztül nem lehet az S_2-t valamelyik soros ütemezéssé átalakítani. Tehát S_2 sorba rendezhető, de nem konfliktus-sorbarendezhető.

9.9. példa: Tekintsük az alábbi ütemezést:

S_1: $r_2(A)$; $r_1(B)$; $w_2(A)$; $r_2(B)$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $w_2(B)$;

amely csak abban különbözik az S-től, hogy az $r_2(B)$ művelet három hellyel előbb szerepel. Az A-ra vonatkozó műveleteket megvizsgálva most is csak a $T_2 <_{S_1} T_3$ megelőzési kapcsolathoz jutunk. De, ha a B-t vizsgáljuk, akkor nemcsak $T_1 <_{S_1} T_2$ teljesül (ugyanis $r_1(B)$ és $w_1(B)$ a $w_2(B)$ előtt szerepel), hanem $T_2 <_{S_1} T_1$ is (ugyanis $r_2(B)$ a $w_1(B)$ előtt fordul elő). Emiatt az S_1 ütemezéshez tartozó megelőzési gráf az, amely a 9.10. ábrán látható.



9.10. ábra. *A 9.9. példa S_1 ütemezéséhez tartozó ciklikus megelőzési gráf, ez az ütemezés nem konfliktus-sorbarendezhető*

Ez a gráf nyilvánvalóan tartalmaz kört. Arra következtethetünk, hogy S_1 nem konfliktus-sorbarendezhető, ugyanis intuíció alapján láthatjuk, hogy bármely konfliktusekvivalens soros ütemezésben a T_1-nek T_2 előtt is és után is kellene állnia, így emiatt nem létezik ilyen ütemezés. □

9.2.3. Miért működik a megelőzési gráfon alapuló tesztelés?

Láttuk, hogy a megelőzési gráfban a kör túl sok megszorítást jelent a feltételezett konfliktusekvivalens soros ütemezésre nézve. Azaz, ha létezik $T_1 \to T_2 \to \ldots \to T_n \to T_1$ n darab tranzakcióból álló kör, akkor a feltételezett soros sorrendben T_1 műveleteinek meg kell előznie a T_2-ben szereplő műveleteket, amelyek megelőzik a T_3-belieket és így tovább egészen T_n-ig. De a T_n műveletei emiatt a T_1-beliek mögött vannak, ugyanakkor meg is kellene előzniük a T_1-belieket a $T_n \to T_1$ él miatt. Ebből következik, hogyha a megelőzési gráf tartalmaz kört, akkor az ütemezés nem konfliktus-sorbarendezhető.

A másik irányt egy kicsit nehezebb belátnunk. Azt kell megmutatnunk, hogy amikor a megelőzési gráf körmentes, akkor az ütemezés műveletei átrendezhetők szomszédos műveletek szabályos cseréivel úgy, hogy az ütemezés egy soros ütemezéssé váljon. Ha ezt meg tudjuk tenni, akkor bebizonyítottuk, hogy minden körmentes megelőzési gráffal rendelkező ütemezés egyben konfliktus-sorbarendezhető. A bizonyítás az ütemezésben részt vevő tranzakciók száma szerinti indukcióval történik.

Alapeset: Ha $n = 1$, vagyis csak egyetlen tranzakcióból áll az ütemezés, akkor az ütemezés már önmagában soros, emiatt biztosan konfliktus-sorbarendezhető.

Indukció: Legyen S

$T_1, T_2, \ldots, T_n$

n darab tranzakció műveleteiből álló ütemezés. Tételezzük fel, hogy S-nek körmentes megelőzési gráfja van. Ha a véges gráf körmentes, akkor van legalább egy olyan csomópontja, amelybe nem vezet él. Legyen a T_i tranzakciónak megfelelő i csomópont egy ilyen csomópont. Mivel nem vezet az i csomópontba él, nincs az S-ben olyan A művelet, hogy:

1. Valamelyik T_j, a T_i-től különböző tranzakcióra vonatkozzon,
2. T_i valamely műveletét megelőzi, és
3. Ezzel a művelettel konfliktusban van.

Ugyanis, ha lenne ilyen, akkor a megelőzési gráfban be kellene rajzolnunk egy élt a j csomópontból az i csomópontba.

Így lehetséges, hogy a T_i minden műveletét az S legelejére mozgatjuk át, miközben megtartjuk a sorrendjüket. Az ütemezés most a következő alakú:

(T_i műveletei) (a többi $n - 1$ tranzakció műveletei)

Most tekintsük az S második részét, vagyis a T_i-től különböző összes tranzakciónak a műveleteit. Mivel ezek a műveletek egymáshoz viszonyítva ugyanabban a sorrendben vannak, mint ahogyan az S-ben voltak, ennek a második résznek a megelőzési gráfja megegyezik az S olyan megelőzési gráfjával, amelyből elhagyjuk a T_i csomópontot és az ebből a csomópontból kimenő éleket.

Mivel az eredeti megelőzési gráf körmentes volt, és csomópontok, illetve élek törlésével nem válhatott ciklikussá, következik, hogy a második rész megelőzési gráfja is körmentes. Továbbá, mivel a második része $n - 1$ tranzakciót tartalmaz, alkalmazzuk rá az indukciós feltevést. Így tudjuk, hogy a második rész műveletei szomszédos műveletek szabályos cseréivel átrendezhetők soros ütemezéssé. Ily módon magát az S-et alakítottuk át olyan soros ütemezéssé, amelyben a T_i műveletei állnak legelöl, és a többi tranzakció műveletei ezután következnek valamilyen soros sorrendben. Az indukciót beláttuk, és így következik, hogy minden olyan ütemezés, amelynek körmentes a megelőzési gráfja, egyben konfliktus-sorbarendezhető.

9.2.4. Feladatok

9.2.1. feladat: Az alábbiakban úgy adunk meg két tranzakciót, hogy leírjuk az A és B két adatbáziselemre vonatkozó hatásukat, amelyekről feltehetjük, hogy egészek.

T_1: `READ(A,t); t := t+2; WRITE(A,t); READ(B,t); t := t*3; WRITE(B,t);`

T_2: `READ(B,s); s := s*2; WRITE(B,s); READ(A,s); s := s+3; WRITE(A,s);`

Tételezzük fel, hogy bármilyen, az adatbázis konzisztenciájára vonatkozó megszorításokat megőrzik ezek a tranzakciók, ha egymástól elkülönítve hajtjuk végre őket. Megjegyezzük, hogy $A = B$ *nem* konzisztenciára vonatkozó megszorítás.

a) Igazoljuk, hogy mindkét soros ütemezésnek az adatbázison való hatása megegyezik; azaz (T_1, T_2) és (T_2, T_1) ekvivalensek! Mutassuk be ezt a tényt úgy, hogy a két tranzakció hatását tetszőleges kezdeti adatbázis-állapotból vizsgáljuk meg!
b) Adjunk példákat a fenti 12 művelet sorba rendezhető és nem sorsorba rendezhető ütemezésére!
c) Mennyi soros ütemezése van a 12 műveletnek?
*!! d) Mennyi sorba rendezhető ütemezése van a 12 műveletnek?

9.2.2. feladat: A 9.2.1. feladat két tranzakcióját átírva olyanná, amely csak az olvasási és írási műveleteket tartalmazza, a következőt kapjuk:

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$;
T_2: $r_2(B)$; $w_2(B)$; $r_2(A)$; $w_2(A)$;

Válaszoljunk a következő kérdésekre:

*! a) A fenti nyolc művelet lehetséges ütemezései közül hány darab konfliktusekvivalens a (T_1, T_2) soros sorrenddel?
b) A nyolc művelet hány darab ütemezése ekvivalens a (T_2, T_1) soros sorrenddel?
!! c) A nyolc művelet hány ütemezése ekvivalens (nem feltétlenül konfliktusekvivalens) a (T_1, T_2) soros sorrenddel, feltéve, hogy a tranzakciók hatása az adatbázis-állapotra ugyanaz, mint amelyet a 9.2.1. feladatban leírtunk?
! d) Miért különbözik a fenti c) és a 9.2.1.d) feladat válasza?

! **9.2.3. feladat:** Tegyük fel, hogy a 9.2.2. feladat tranzakcióit megváltoztatjuk:

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$;
T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$;

Azaz a tranzakciók szemantikája ugyanaz marad, mint a 9.2.1. feladatban volt, de a T_2 annyiban változik, hogy az A-t előbb dolgozza fel, mint a B-t. Adjuk meg:

a) A konfliktus-sorbarendezhető ütemezések számát.
b) A sorba rendezhető ütemezések számát, feltéve, hogy a tranzakcióknak ugyanolyan a hatásuk az adatbázis-állapotra, mint a 9.2.1. feladatban.

9.2.4. feladat: Tekintsük az alábbi ütemezéseket:

* a) $r_1(A)$; $r_2(A)$; $r_3(B)$; $w_1(A)$; $r_2(C)$; $r_2(B)$; $w_2(B)$; $w_1(C)$;
b) $r_1(A)$; $w_1(B)$; $r_2(B)$; $w_2(C)$; $r_3(C)$; $w_3(A)$;
c) $w_3(A)$; $r_1(A)$; $w_1(B)$; $r_2(B)$; $w_2(C)$; $r_3(C)$;
d) $r_1(A)$; $r_2(A)$; $w_1(B)$; $w_2(B)$; $r_1(B)$; $r_2(B)$; $w_2(C)$; $w_1(D)$;
e) $r_1(A)$; $r_2(A)$; $r_1(B)$; $r_2(B)$; $r_3(A)$; $r_4(B)$; $w_1(A)$; $w_2(B)$;

Válaszoljunk mindegyik esetében a következő kérdésekre:

i) Mi az ütemezés megelőzési gráfja?
ii) Konfliktus-sorbarendezhető-e az ütemezés? Ha igen, akkor melyek az összes ekvivalens soros ütemezések?
! iii) Vannak-e olyan soros ütemezések, amelyek ekvivalensek (függetlenül attól, hogy hogyan hatnak a tranzakciók az adatokon), de nem konfliktusekvivalensek?

!! **9.2.5. feladat:** Azt mondjuk, hogy a T tranzakció *megelőzi* az U tranzakciót egy S ütemezésben, ha a T összes művelete megelőzi az U összes műveletét az S-ben. Megjegyezzük, hogyha az S-ben csak a T és U tranzakciók vannak, akkor az, hogy T megelőzi az U-t, ugyanazt jelenti, hogy az S a (T, U) soros ütemezés. De, ha az S más tranzakciókat is tartalmaz a T-n és U-n kívül, akkor nem biztos, hogy az S sorba rendezhető, és valójában a többi tranzakció hatása miatt még az is előfordulhat, hogy nem konfliktus-sorbarendezhető. Adjunk példát az S olyan ütemezésére, amely:

i) S-ben T_1 megelőzi T_2-t, és
ii) S konfliktus-sorbarendezhető, de
iii) Minden S-sel konfliktusekvivalens soros ütemezésben T_2 megelőzi T_1-et!

! **9.2.6. feladat:** Magyarázzuk meg, hogyan lehet bármely $n > 1$ esetén találni olyan ütemezést, amelynek a megelőzési gráfja tartalmaz n hosszúságú kört, de kisebb kört nem!

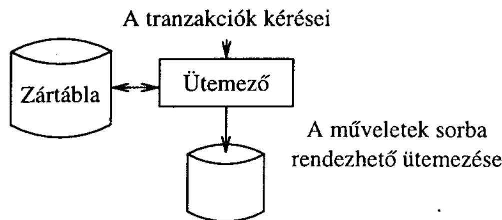
9.3. A sorbarendezhetőség biztosítása zárakkal

Képzeljünk el olyan tranzakcióhalmazt, amely megszorítások nélkül hajtja végre a műveleteit. Ezek a műveletek is egy ütemezést alkotnak, de nem valószínű, hogy ez az ütemezés sorba rendezhető lenne. Az ütemező feladata az, hogy megakadályozza az olyan műveleti sorrendeket, amelyek nem sorba rendezhető ütemezésekhez vezetnek. Ebben a részben az ütemező legáltalánosabb szerkezetét tekintjük, egy olyat, amelyben az adatbáziselemekre kiadott „zárak" akadályozzák meg a nem sorba rendezhető viselkedést. Intuíció alapján arról van szó, hogy a tranzakciók zárolják azokat az adatbáziselemeket, amelyekhez hozzáférnek, hogy megakadályozzák azt, hogy ugyanakkor más tranzakciók is hozzáférjenek ezekhez az elemekhez, mivel ekkor felmerülne a nem sorbarendezhetőség kockázata.

Ebben a fejezetben egy (nagyon is) leegyszerűsített zárolási sémával vezetjük be a zárolás fogalmát. Ebben a sémában csak egyféle zár van, amelyet a tranzakcióknak meg kell kapniuk az adatbáziselemre, ha bármilyen műveletet végre akarnak hajtani ezen az elemen. A 9.4. részben sokkal valósabb zárolási sémákat tanulmányozunk, különböző zármódokkal, beleértve az általános osztott/kizárólagos zárakat, amelyek az olvasási és írási jogoknak felelnek meg.

9.3.1. Zárak

A 9.11. ábrán egy ütemezőt láthatunk, amely zártábla segítségével végzi a feladatát. Emlékeztetünk arra, hogy az ütemező felelős azért, hogy fogadja a tranzakcióktól a kéréseket, és vagy megengedi a műveleteket az adatbázison, vagy addig késlelteti, amikor már biztonságosan végre tudja hajtani őket. A továbbiakban kifejtjük, hogyan irányítja ezt a döntést a zártábla felhasználásával.



9.11. ábra. *A döntéseket zártábla felhasználásával irányító ütemező*

Az lenne az ideális, ha az ütemező akkor, és csak akkor továbbítana egy kérést, ha annak végrehajtása nem vezethetne inkonzisztens adatbázis-állapotba, miután az összes aktív tranzakciót vagy véglegesen végrehajtottuk, vagy abortáltuk (vagyis leállítottuk a befejezése előtt, más szóval sikertelenül fejeztük be). Ezt a kérdést viszont túl nehéz lenne valós időben eldönteni. Így minden ütemező csak egy egyszerű tesztet hajt végre a sorbarendezhetőség biztosítására, azonban letilthat olyan műveleteket is, amelyek önmagunkban nem vezetnének inkonzisztenciához. A zárolási ütemező, mint a legtöbb ütemező, a konfliktus-sorbarendezhetőséget követeli meg, pedig mint ezt már tanultuk, ez erősebb követelmény, mint a sorbarendezhetőség.

Ha az ütemező zárakat használ, akkor a tranzakcióknak zárakat kell kérniük, és feloldaniuk az adatbáziselemek olvasásán és írásán felül. A zárak használatának két értelemben is helyesnek kell lennie, mind a tranzakciók szerkezetére, mind pedig az ütemezők szerkezetére alkalmazva.

- *Tranzakciók konzisztenciája* (consistency of transactions): A műveletek és a zárak az alábbi elvárások szerint kapcsolódnak egymáshoz:

  1. A tranzakció csak akkor olvashat vagy írhat egy elemet, ha már korábban zárolta az elemet, és még nem oldotta fel a zárat.
  2. Ha egy tranzakció zárol egy elemet, akkor később azt fel kell szabadítania.

- *Az ütemezések jogszerűsége* (legality of schedules): A zárak értelme feleljen meg a szándék szerinti elvárásnak: nem zárolhatja két tranzakció ugyanazt az elemet, csak úgy, ha az egyik előbb már feloldotta a zárat.

Kibővítjük a jelöléseinket a zárolás és a feloldás műveletekkel:

$l_i(X)$: T_i tranzakció az X adatbáziselemre *zárolást kér* (request a lock, l = „lock").
$u_i(X)$: T_i tranzakció az X adatbáziselem *zárolását feloldja* (release its lock, u = „unlock").

Így a tranzakciók konzisztenciafeltétele úgy is kimondható, hogy: „Amikor egy T_i tranzakcióban van egy $r_i(X)$ vagy egy $w_i(X)$ művelet, akkor van korábban egy $l_i(X)$ művelet, de közben nincs $u_i(X)$, és van később egy $u_i(X)$ művelet." Az ütemezések jogszerűsége azt mondja ki, hogy: „Ha egy ütemezésben van olyan $l_i(X)$ művelet, amelyet $l_j(X)$ követ, akkor e két művelet között lennie kell egy $u_i(X)$ műveletnek."

9.10. példa: Tekintsük a 9.1. példában szereplő T_1, T_2 tranzakciókat. Emlékeztetőül, a T_1 hozzáad az A és B adatbáziselemekhez 100-at, a T_2 pedig megduplázza az értéküket. Itt most úgy adjuk meg a tranzakciókat, hogy a zárolási és az aritmetikai számolási műveleteket is leírjuk segítségképpen, hogy emlékezzünk, mit tesznek a tranzakciók.[3]

T_1: $l_1(A)$; $r_1(A)$; `A := A+100;` $w_1(A)$; $u_1(A)$; $l_1(B)$; $r_1(B)$; `B := B+100;` $w_1(B)$; $u_1(B)$;
T_2: $l_2(A)$; $r_2(A)$; `A := A*2;` $w_2(A)$; $u_2(A)$; $l_2(B)$; $r_2(B)$; `B := B*2;` $w_2(B)$; $u_2(B)$;

Mindkét tranzakció konzisztens. Mindkettő felszabadítja az A-ra és B-re kiadott zárakat. Továbbá mindkettő csak olyan lépésekben dolgozik az A-n és a B-n, amikor előzőleg már zárolták az elemet, és még nem oldották fel a zár alól.

A 9.12. ábrán ennek a két tranzakciónak egy jogszerű ütemezése található. Helymegtakarítás miatt több műveletet írtunk egy sorban. Ez az ütemezés jogszerű, ugyanis a két tranzakció sohasem zárolja egyidejűleg az A-t vagy a B-t. Pontosabban, a T_2 nem végzi el az $l_2(A)$-t, csak miután a T_1 végrehajtotta az $u_1(A)$-t, és a T_1 nem végzi el az

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$;			
`A := A+100;`			
$w_1(A)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$;		
	`A := A*2;`		
	$w_2(A)$; $u_2(A)$;	250	
	$l_2(B)$; $r_2(B)$;		
	`B := B*2;`		
	$w_2(B)$; $u_2(B)$;		50
$l_1(B)$; $r_1(B)$;			
`B := B+100;`			
$w_1(B)$; $u_1(B)$;			150

9.12. ábra. *Konzisztens tranzakciók jogszerű ütemezése. Sajnos nem sorba rendezhető*

[3] Megjegyezzük, hogy a tranzakciók aktuális számításait rendszerint nem ábrázoljuk a jelenlegi jelölésünkben, ugyanis az ütemező sem tudja ezt figyelembe venni, amikor arról dönt, hogy engedélyezze vagy elutasítsa a tranzakciókéréseket.

$l_1(B)$-t, csak miután a T_2 végrehajtotta az $u_2(B)$-t. Láthatjuk a kiszámított értékek nyomon követésével, hogy bár ez az ütemezés jogszerű, mégsem sorba rendezhető. A 9.3.3. részben látni fogunk egy további feltételt, a „kétfázisú zárolás"-t, amivel biztosíthatjuk majd, hogy a jogszerű ütemezések konfliktus-sorbarendezhetők legyenek. □

9.3.2. A zárolási ütemező

A zároláson alapuló ütemező feladata, hogy akkor és csak akkor engedélyezze a kéréseket, ha a kérés jogszerű ütemezést eredményez. Ezt a döntést segíti a zártábla, amely minden adatbáziselemhez megadja azt a tranzakciót, ha van ilyen, amelyik pillanatnyilag éppen zárolja az adott elemet. Részletesen később a 9.5.2. részben beszélünk a zártábla szerkezetéről. Ha viszont csak egyféle zárolás van, mint ahogyan eddig feltételeztük, akkor úgy tekinthetjük a táblát, mint (X, T) párokból álló `Zárolások(elem, tranzakció)` relációt, ahol a T tranzakció zárolja az X adatbáziselemet. Az ütemezőnek csak le kell kérdeznie ezt a relációt, illetve egyszerű `INSERT` és `DELETE` utasításokkal kell módosítania.

9.11. példa: A 9.12. ábrán található ütemezés jogszerű, ahogyan ezt már láttuk, így a zárolási ütemező engedélyezhetné az összes kérést abban a sorrendben, ahogyan beérkeznek. Néha azonban előfordulhat, hogy nem lehet engedélyezni a kéréseket. Tekintsük a 9.10. példából a T_1 és T_2 tranzakciókat egy apró (de a 9.3.3. részben majd látni fogjuk, hogy lényeges) változtatással, mégpedig a T_1 is és a T_2 is előbb zárolja B-t, és csak azután oldja fel A zárolását.

T_1: $l_1(A)$; $r_1(A)$; `A := A+100;` $w_1(A)$; $l_1(B)$; $u_1(A)$; $r_1(B)$; `B := B+100;` $w_1(B)$; $u_1(B)$;
T_2: $l_2(A)$; $r_2(A)$; `A := A*2;` $w_2(A)$; $l_2(B)$; $u_2(A)$; $r_2(B)$; `B := B*2;` $w_2(B)$; $u_2(B)$;

A 9.13. ábrán látható, hogy amikor T_2 kéri B zárolását, az ütemezőnek el kell utasítania ezt a kérést, ugyanis T_1 még zárolja a B-t. Így T_2 áll, és a következő műveleteket a T_1 végzi. Végül a T_1 végrehajtja $u_1(B)$-t, amely felszabadítja a B-t. Most T_2 már zárolhatja a B-t, amelyet a következő lépésben végre is hajt. Megjegyezzük, hogy mivel T_2-nek várakoznia kellett, emiatt a B-t akkor szorozza meg 2-vel, miután a T_1 már hozzáadott 100-at B-hez, és ez konzisztens adatbázis-állapotot eredményez. □

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$;			
`A := A+100;`			
$w_1(A)$; $l_1(B)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$;		
	`A := A*2;`		
	$w_2(A)$;	250	
	$l_2(B)$; **Elutasítva**		
$r_1(B)$; `B := B+100;`			
$w_1(B)$; $u_1(B)$;			125
	$l_2(B)$; $u_2(A)$; $r_2(B)$;		
	`B := B*2;`		
	$w_2(B)$; $u_2(B)$;		250

9.13. ábra. *A zárolási ütemező késlelteti azt a kérést, amely jogtalan ütemezéshez vezetne*

9.3.3. A kétfázisú zárolás

Van egy meglepő feltétel, amellyel biztosítani tudjuk, hogy konzisztens tranzakciók jogszerű ütemezése konfliktus-sorbarendezhető legyen. Ezt a feltételt, amelyet a gyakorlatban elterjedt zárolási rendszerek leginkább követnek, *kétfázisú zárolásnak* (two-phase locking) vagy *2FZ-nek* (2PL) nevezzük. A 2FZ feltétel:

- Minden tranzakcióban minden zárolási művelet megelőzi az összes zárfeloldási műveletet.

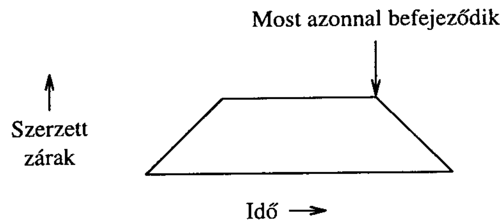
A „kétfázis", amelyre a 2FZ-vel hivatkozunk, abból adódik, hogy az első fázisban csak zárolásokat adunk ki, a második fázisban pedig csak megszüntetünk zárolásokat. A kétfázisú zárolás a konzisztenciához hasonlóan, a tranzakcióban a műveletek sorrendjére egy feltétel. Azokat a tranzakciókat, amelyek eleget tesznek a 2FZ feltételnek *kétfázisú zárolású tranzakcióknak* (two-phase-locked transaction) vagy 2FZ *tranzakcióknak* nevezzük.

9.12. példa: A 9.10. példában a tranzakciók nem engedelmeskednek a kétfázisú zárolási szabálynak. Például a T_1 előbb oldja fel az A zárolását, mint zárolná a B-t. A 9.11. példában található tranzakciók változata azonban már *eleget tesz* a 2FZ feltételnek. Megjegyezzük, hogy T_1 az A-t és B-t is az első öt műveleten belül zárolja, és a következő öt műveleten belül feloldja a zárakat; és a T_2 is hasonlóan viselkedik. Ha összehasonlítjuk a 9.12. és 9.13. ábrákat látjuk, hogy a kétfázisú zárolású tranzakciók hogyan működnek együtt az ütemezővel a konzisztencia biztosítására, ellenben a nem 2FZ tranzakciók esetén előfordulhat inkonzisztencia (és emiatt nem konfliktus-sorbarendezhető) viselkedés.

9.3.4. Miért működik a kétfázisú zárolás?

Igaz, bár közel sem nyilvánvaló, hogy a 2FZ példánkban észlelt előnyei általában is érvényesek. Intuíció alapján, mindegyik kétfázisú zárolású tranzakcióról azt gondolhatjuk, hogy rögtön végrehajtásra kerülnek, amint az első zárfeloldási kérés kiadásra kerül, ahogy ezt a 9.14. ábra javasolja. A 2FZ tranzakciók S ütemezésével konfliktus-ekvivalens soros ütemezés pont olyan, mint amelyikben a tranzakciók ugyanabban a sorrendben vannak, amilyenek az első zárfeloldásaik.[4]

[4] Bizonyos ütemezések esetén más konfliktusekvivalens soros ütemezések is lehetnek.



9.14. ábra. *Minden kétfázisú zárolású tranzakciónak van olyan pontja, amikor azt mondhatjuk, hogy azonnal befejeződik*

Megmutatjuk, hogyan lehet konzisztens, kétfázisú zárolású tranzakciók bármely S jogszerű ütemezését átalakítani konfliktusekvivalens soros ütemezéssé. A konverziót legjobban az S-ben részt vevő tranzakciók n száma szerinti indukcióval tudjuk leírni. A következőkben lényeges megjegyeznünk, hogy a konfliktusekvivalencia csak az olvasási és írási műveletekre vonatkozik. Amikor felcseréljük az olvasások és írások sorrendjét, akkor figyelmen kívül hagyjuk a zárolási és zárfeloldási műveleteket. Amikor megkaptuk az olvasási és írási műveletek sorrendjét, akkor úgy helyezzük el köréjük a zárolási és zárfeloldási műveleteket, ahogyan azt a különböző tranzakciók megkövetelik. Mivel minden tranzakció felszabadítja az összes zárolást a tranzakció befejezése előtt, tudjuk, hogy a soros ütemezés jogszerű lesz.

Alapeset: Ha $n = 1$, akkor semmit sem kell tennünk, az S már soros ütemezés.

Indukció: Tételezzük fel, hogy S n darab tranzakciót tartalmaz: $T_1, T_2, \ldots, T_n$, és legyen T_i az a tranzakció, amelyik a teljes S ütemezésben a legelső zárfeloldási műveletet végzi, mondjuk az $u_i(X)$-et. Azt állítjuk, hogy a T_i összes olvasási és írási műveletét az ütemezés legelejére tudjuk vinni anélkül, hogy konfliktusműveleteken kellene átnaladnunk.

Tekintsük a T_i valamelyik műveletét, mondjuk $w_i(Y)$-t. Megelőzheti-e ezt az S-ben valamely konfliktus művelet, például $w_j(Y)$? Ha így lenne, akkor az S ütemezésben $u_j(Y)$ és $l_i(Y)$ műveletek az alábbi módon helyezkednének el a művelet sorozatban

$$\ldots;\ w_j(Y);\ \ldots;\ u_j(Y);\ \ldots;\ l_i(Y);\ \ldots;\ w_i(Y);\ \ldots$$

Mivel T_i az első, amelyik zárat old fel, így az S-ben az $u_i(X)$ megelőzi az $u_j(Y)$-t, vagyis az S a következőképpen néz ki:

$$\ldots;\ w_j(Y);\ \ldots;\ u_i(X);\ \ldots;\ u_j(Y);\ \ldots;\ l_i(Y);\ \ldots;\ w_i(Y);\ \ldots$$

illetve az $u_i(X)$ előfordulhat még a $w_j(Y)$ előtt is. Mindegyik esetben az $u_i(X)$ az $l_i(Y)$ előtt van, ami azt jelenti, hogy a T_i *nem* lenne kétfázisú zárolású, amint azt feltételeztük. Ahogyan beláttuk, hogy nem létezhetnek konfliktuspárok az írásra, ugyanúgy be lehet látni bármely két lehetséges műveletre, az egyiket T_i-ből, a másikat pedig egy másik T_j-ből választva, hogy nem lehetnek konfliktuspárok.

Bebizonyítottuk, hogy valóban az S legelejére lehet vinni a T_i összes műveletét

konfliktusmentes olvasási és írási műveletpárok cseréjével. Ezután elhelyezhetjük a T_i zárolási és feloldási műveleteit. Vagyis az S-et a következő alakba írhatjuk át

(T_i műveletei) (a többi $n-1$ tranzakció műveletei)

Az $n-1$ tranzakcióból álló második része szintén konzisztens, 2FZ tranzakciókból álló jogszerű ütemezés, így az indukciós feltevést alkalmazhatjuk rá. Átalakítjuk a második részt konfliktusekvivalens soros ütemezéssé, ily módon a teljes S konfliktus-sorbarendezhetővé vált.

9.3.5. Feladatok

9.3.1. feladat: Az alábbiakban megadunk két tranzakciót a zárolási kéréseikkel és a tranzakciók szemantikájával. Megjegyezzük, hogy a 9.2.1. feladatban láttuk ezeknek a tranzakcióknak azt a szokatlan tulajdonságát, hogy úgy ütemezhetők, hogy ne legyenek konfliktus-sorbarendezhetők, de a szemantikájuk miatt mégis sorba rendezhetők.

A holtpont kockázata

Az egyik probléma, amelyet nem lehet a kétfázisú zárolással megoldani, a holtpontok bekövetkezésének a lehetősége, vagyis amikor az ütemező arra kényszeríti a tranzakciókat, hogy örökké várakozzanak egy olyan zárra, amelyet egy másik tranzakció tart zárolva. Például tekintsük a 9.11. példa 2FZ tranzakcióit, de most a T_2 dolgozza fel előbb a B-t:

T_1: $l_1(A)$; $r_1(A)$; `A := A+100;` $w_1(A)$; $l_1(B)$; $u_1(A)$; $r_1(B)$; `B := B+100;` $w_1(B)$; $u_1(B)$;
T_2: $l_2(B)$; $r_2(B)$; `B := B*2;` $w_2(B)$; $l_2(A)$; $u_2(B)$; $r_2(A)$; `A := A*2;` $w_2(A)$; $u_2(A)$;

A tranzakciós műveletek egy lehetséges végrehajtása:

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$;			
	$l_2(B)$; $r_2(B)$;		
`A := A+100;`			
	`B := B*2;`		
$w_1(A)$;		125	
	$w_2(B)$;		50
$l_1(B)$; **Elutasítva**	$l_2(A)$; **Elutasítva**		

Most egyik tranzakció sem folytatódhat, hanem örökké várakozniuk kell. A 10.3. részben megvizsgáljuk azokat a módszereket, amelyek megszüntetik ezt a helyzetet. Viszont vegyük észre, hogy nem tudjuk mind a két tranzakciót folytatni, ugyanis ha így lenne, akkor az adatbázis végső állapotában nem lehetne $A = B$.

T_1: $l_1(A)$; $r_1(A)$; `A := A+2`; $w_1(A)$; $u_1(A)$; $l_1(B)$; $r_1(B)$; `B := B*3`; $w_1(B)$; $u_1(B)$;
T_2: $l_2(B)$; $r_2(B)$; `B := B*2`; $w_2(B)$; $u_2(B)$; $l_2(A)$; $r_2(A)$; `A := A+3`; $w_2(A)$; $u_2(A)$;

Az alábbi kérdésekben az ütemezéseknek csak az írás és olvasás műveleteit tekintsük, a zárolási, feloldási és értékadási lépésektől tekintsünk el.

* a) Adjunk példát a zárolások miatt tiltott ütemezésre!

! b) Nyolc olvasás és írás művelet $\binom{8}{4} = 70$ sorrendjéből mennyi a jogszerű ütemezés (vagyis amelyeket zárakkal megengedünk)?

! c) A jogszerű ütemezések közül mennyi sorba rendezhető (a tranzakciók fent megadott szemantikája szerint)?

! d) A jogszerű és sorba rendezhető ütemezések közül mennyi konfliktus-sorbarendezhető?

!! e) Mivel a T_1 és a T_2 nem kétfázisú zárolású, azt várnánk, hogy bizonyos nem sorba rendezhető viselkedés fordulna elő. Vannak-e olyan jogszerű ütemezések, amelyek nem sorba rendezhetők? Ha igen, adjunk rá példát, ha nem, akkor magyarázzuk meg, hogy miért nem!

*! **9.3.2. feladat:** A 9.3.1. feladat tranzakcióit kétfázisú zárolásúvá írtuk át úgy, hogy az összes zárfeloldási műveletet a végére vittük:

T_1: $l_1(A)$; $r_1(A)$; `A := A+2`; $w_1(A)$; $l_1(B)$; $r_1(B)$; `B := B*3`; $w_1(B)$; $u_1(A)$; $u_1(B)$;
T_2: $l_2(B)$; $r_2(B)$; `B := B*2`; $w_2(B)$; $l_2(A)$; $r_2(A)$; `A := A+3`; $w_2(A)$; $u_2(B)$; $u_2(A)$;

Ezeknek a tranzakcióknak az összes írási és olvasási műveleteiből hány jogszerű ütemezést tudunk képezni?

9.3.3. feladat: A 9.2.4. feladat mindegyik ütemezése esetén tételezzük fel, hogy minden tranzakció közvetlenül azelőtt zárolja az adatbáziselemeket, mielőtt olvasná vagy írná az elemet, és minden tranzakció azonnal feloldja az elem zárolását, miután utoljára használta az elemet. Mondjuk meg, hogy mit tenne a zárolási ütemező ezeknek az ütemezéseknek mindegyikével, vagyis melyik kérést késleltetnénk, és mikor adnánk lehetőséget a tranzakció folytatására?

! **9.3.4. feladat:** Az alábbiakban megadott tranzakciók mindegyikénél tételezzük fel, hogy beszúrjuk a zárolás és feloldás műveletet minden egyes adatbáziselemhez, amihez hozzáférünk!

* a) $r_1(A)$; $w_1(B)$;
b) $r_2(A)$; $w_2(A)$; $w_2(B)$;

Adjuk meg, hogy a zárolási, feloldási, olvasási és írási műveleteknek hány sorrendje lehet az alábbi tranzakcióknál:

i) Konzisztens és kétfázisú zárolású.
ii) Konzisztens, de nem kétfázisú zárolású.
iii) Nem konzisztens, de kétfázisú zárolású.
iv) Sem nem konzisztens, sem nem kétfázisú zárolású.

9.4. Különböző zármódú zárolási rendszerek

A 9.3. rész zárolási sémája bemutatja a zárolás mögött álló legfőbb elveket, de túl egyszerű ahhoz, hogy a gyakorlatban is használható séma legyen. Az a legfőbb probléma, hogy a T tranzakciónak akkor is zárolnia kell egy X adatbáziselemet, ha csak olvasni akarja X-et, és nem akarja írni. Nem kerülhetjük el a zárolást ekkor sem, mert ha nem zárolnánk, akkor esetleg egy másik tranzakció azalatt írna az X-be új értéket, mialatt a T aktív, ami nem sorba rendezhető viselkedést okoz. Másrészről pedig miért is ne olvashatná több tranzakció egyidejűleg az X értékét mindaddig, amíg egyiknek sincs engedélyezve, hogy írjon az X-be.

Ez indokolja, hogy bevezessük a legelterjedtebb zárolási sémát, amikor két különböző zárat használunk, az egyiket az olvasáshoz (ezt „osztott zárnak" vagy „olvasási zárnak" nevezzük), és egyet az íráshoz (amelyet „kizárólagos zárnak" vagy „írási zárnak" hívunk). Ezután pedig megvizsgálunk egy fejlettebb sémát, amikor a tranzakcióknak később lehetőségük lesz osztott zárat „felminősíteni" kizárólagos zárrá. Megnézzük a „növelési zárakat" is olyan speciális írási műveletek kezelésére, amelyek növelik az adatbáziselemet. A lényeges különbség az, hogy a növelési műveletek felcserélhetők egymással, ellenben az általános írási műveletek nem. Ezek a példák elvezetnek a „kompatibilitási mátrix" segítségével megadott zárolási séma általános fogalmához, amely azt jelzi, hogy milyen zárakat engedélyezhetünk az olyan adatbáziselemekre, amelyek már zárolva vannak.

9.4.1. Osztott és kizárólagos zárak

Mivel ugyanannak az adatbáziselemnek két olvasási művelete nem eredményez konfliktust, így ahhoz nincs szükség zárolásra vagy más konkurenciavezérlési működésre, hogy az olvasási műveleteket egy bizonyos sorrendbe soroljuk. Mint a bevezetőben javasoltuk, továbbra is szükséges azt az elemet is zárolni, amelyet olvasunk, ugyanis ennek az elemnek az írását nem szabad közben megengednünk. Az íráshoz szükséges zár viszont „erősebb", mint az olvasáshoz szükséges zár, ugyanis ennek mind az olvasásokat, mind az írásokat meg kell akadályoznia.

Emiatt olyan zárolási ütemezőt tekintsünk, amely két különböző zárat alkalmaz: az *osztott zárakat* (shared locks) és a *kizárólagos zárakat* (exclusive locks). Intuíció alapján tetszőleges X adatbáziselemet vagy egyszer lehet zárolni kizárólagosan, vagy akárhányszor lehet zárolni osztottan, ha még nincs kizárólagosan zárolva. Amikor írni akarjuk az X-et, akkor az X-en kizárólagos zárral kell rendelkeznünk, de ha csak ol-

vasni szándékozzuk az X-et, akkor az X-en akár osztott, akár kizárólagos zár megfelel. Feltételezzük, hogy ha olvasni akarjuk az X-et, de írni nem, akkor előnyben részesítjük az osztott zárolást.

Az $sl_i(X)$ jelölést használjuk arra, hogy „a T_i tranzakció osztott zárat kér az X adatbáziselemre", és $xl_i(X)$ jelölést pedig arra, hogy „a T_i kizárólagos zárat kér az X-re". Továbbra is $u_i(X)$-szel jelöljük, hogy a T_i feloldja az X zárását, vagyis felszabadítja minden zár alól az X-et.

A három követelmény, a tranzakciók konzisztenciája, a tranzakciók 2FZ-je, és az ütemezések jogszerűsége, mindegyikének van megfelelője az osztott/kizárólagos zárolási rendszerben. Az alábbiakban összegezzük ezeket a követelményeket:

1. *Tranzakciók konzisztenciája:* Nem írhatunk kizárólagos zár fenntartása nélkül és nem olvashatunk valamilyen zár fenntartása nélkül. Pontosabban fogalmazva, bármely T_i tranzakcióban.

   a) Az $r_i(X)$ olvasási műveletet meg kell hogy előzze egy $sl_i(X)$ vagy $xl_i(X)$ úgy, hogy közben nincs $u_i(X)$.
   b) A $w_i(X)$ írási műveletet meg kell hogy előzze egy $xl_i(X)$ úgy, hogy közben nincs $u_i(X)$.

   Minden zárolást követnie kell ugyanannak az elemnek a zárolását feloldó műveletnek.
2. *Tranzakciók kétfázisú zárolása:* A zárolásoknak meg kell előzniük a zárak feloldását. Pontosabban fogalmazva, bármely T_i kétfázisú zárolású tranzakcióban egyetlen $sl_i(X)$ vagy $xl_i(X)$ műveletet sem előzhet meg egyetlen $u_i(Y)$ művelet sem semmilyen Y-ra.
3. *Az ütemezések jogszerűsége:* Egy elemet vagy egyetlen tranzakció zárol kizárólagosan, vagy több is zárolhatja osztottan, de a kettő egyszerre nem lehet. Pontosabban fogalmazva:

   a) Ha $xl_i(X)$ szerepel egy ütemezésben, akkor ezután nem következhet $xl_j(X)$ vagy $sl_j(X)$ valamely i-től különböző j-re anélkül, hogy közben ne szerepelne $u_i(X)$.
   b) Ha $sl_i(X)$ szerepel egy ütemezésben, akkor ezután nem következhet $xl_j(X)$ $j \neq i$-re anélkül, hogy közben ne szerepelne $u_i(X)$.

Megjegyezzük, hogy engedélyezett, hogy egy tranzakció ugyanazon elemre kérjen és tartson mind osztott, mind kizárólagos zárat, feltéve, ha ezzel nem kerül konfliktusba más tranzakciók zárolásaival. Ha a tranzakciók előre tudnák, milyen zárakra lesz szükségük, akkor biztosan csak a kizárólagos zárolást kérnék, de ha nem láthatók előre a zárolási igények, lehetséges, hogy egy tranzakció osztott és kizárólagos zárakat is kér különböző időpontokban.

9.13. példa: Nézzük az alábbi osztott és kizárólagos zárakat használó két tranzakciónak egy lehetséges ütemezését:

T_1: $sl_1(A)$; $r_1(A)$; $xl_1(B)$; $r_1(B)$; $w_1(B)$; $u_1(A)$; $u_1(B)$;
T_2: $sl_2(A)$; $r_2(A)$; $sl_2(B)$; $r_2(B)$; $u_2(A)$; $u_2(B)$;

A T_1 is és a T_2 is olvassa A-t és B-t, de csak a T_1 írja B-t. Egyik sem írja A-t.

A 9.15. ábrán T_1 és T_2 műveleteinek olyan ütemezése látható, amelyet a T_1 kezd az A osztott zárolásával. Ezután a T_2 következik, az A és B mindegyikét osztottan zárolja. Most a T_1-nek lenne szüksége a B kizárólagos zárolására, ugyanis olvassa is és írja is a B-t. Viszont nem kaphatja meg a kizárólagos zárat, hiszen a T_2-nek már osztott zárja van a B-n. Így az ütemező várakozni kényszeríti a T_1-et. Végül a T_2 feloldja a B zárját, és ekkor befejeződhet a T_1. □

T_1	T_2
$sl_1(A)$; $r_1(A)$;	
	$sl_2(A)$; $r_2(A)$;
	$sl_2(B)$; $r_2(B)$;
$xl_1(B)$ **Elutasítva**	
	$u_2(A)$; $u_2(B)$;
$xl_1(B)$; $r_1(B)$; $w_1(B)$;	
$u_1(A)$; $u_1(B)$;	

9.15. ábra. *Osztott és kizárólagos zárolást használó ütemezés*

Megjegyezzük, hogy a 9.15. ábrán látható ütemezés eredménye konfliktus-sorbarendezhető. A konfliktusekvivalens soros sorrend (T_2, T_1), hiába kezdődött el a T_1 előbb. Bár itt most nem bizonyítjuk, hogy konzisztens 2FZ tranzakciók jogszerű ütemezése konfliktus-sorbarendezhető, ugyanazok a meggondolások alkalmazhatók az osztott és kizárólagos zárakra is, mint amelyeket a 9.3.4. részben adtunk. A 9.15. ábrán a T_2 előbb old fel zárat, mint a T_1, így azt várjuk, hogy a T_2 megelőzi a T_1-et a soros sorrendben. Ezzel ekvivalensen megvizsgálva a 9.15. ábra olvasási és írási műveleteit, észrevehető, hogy az $r_1(A)$-t a T_2 összes műveletén át hátra tudjuk cserélgetni, amíg a $w_1(B)$-t nem tudjuk az $r_2(B)$ elé vinni, ami pedig szükséges lenne ahhoz, hogy ha a T_1 megelőzze T_2-t egy konfliktusekvivalens soros ütemezésben.

9.4.2. Kompatibilitási mátrixok

Ha több zármódot használunk, akkor az ütemezőnek valamilyen elvre van szüksége ahhoz, hogy mikor engedélyezzen egy zárolási kérést, ha már adva vannak más zárak is azon az adatbáziselemen. Míg az osztott/kizárólagos rendszerek egyszerűek, fogjuk látni, hogy a zárolási módoknak viszonylag összetettebb rendszerei is léteznek a gyakorlatban. A zárolást engedélyező elvek következő fogalmait előbb az egyszerű osztott/kizárólagos rendszerek környezetében vezetjük be.

A *kompatibilitási mátrix* minden egyes zármódhoz rendelkezik egy-egy sorral és egy-egy oszloppal. A sorok egy másik tranzakció által az X elemre már érvényes záraknak felelnek meg, az oszlopok pedig az X-re kért zármódoknak felelnek meg. A kompatibilitási mátrix használatának szabálya a zárolást engedélyező döntésekre az alábbi:

- *C* módú zárat akkor és csak akkor engedélyezhetünk, ha a táblázat minden olyan *R* sorára, amelyre más tranzakció már zárolta az *X*-et *R* módban, a *C* oszlopban „Igen" szerepel.

9.14. példa: A 9.16. ábrán osztott (*S*) és kizárólagos (*X*) zárak kompatibilitási mátrixa látható. Az *S* oszlop azt mondja meg, hogy akkor engedélyezhetünk osztott zárat egy elemre, ha arra az elemre jelenleg is csak osztott zárak vannak. Az *X* oszlop azt mondja meg, hogy csak akkor engedélyezhetünk kizárólagos zárat, ha jelenleg nincs más zár rajta. Megjegyezzük, hogy ezek a szabályok az ütemezések jogszerűségének a definícióját tükrözik erre a zárolási rendszerre. □

		Kért	zár
		S	*X*
Érvényes zár	*S*	Igen	Nem
ebben a módban	*X*	Nem	Nem

9.16. ábra. *Osztott (S) és kizárólagos (X) zárak kompatibilitási mátrixa*

9.4.3. Zárak felminősítése

Az a *T* tranzakció, amelyik osztott zárat helyez az *X*-re „barátságos" a többi tranzakcióhoz, ugyanis a többinek is lehetősége van az *X*-et a *T*-vel egy időben olvasni. Így kíváncsiak vagyunk arra, vajon még barátságosabb-e az a *T* tranzakció, amelyik beolvasni és új értékkel visszaírni akarja az *X*-et, előbb csak osztott zárat tesz az *X*-re, és később, amikor a *T* már készen áll az új érték beírásával, akkor *felminősíti a zárat kizárólagossá* (upgrade the lock to exclusive) (vagyis később kéri az *X* kizárólagos zárolását azon túl, hogy már osztott zárat tart fenn az *X*-en). Nincs akadálya, hogy a tranzakció ugyanarra az adatbáziselemre újabb különböző zármódú kéréseket adjon ki. Továbbra is fenntartjuk azt a megszokott jelölést, hogy $u_i(X)$ a T_i tranzakció által fennálló összes zárat feloldja az *X*-en, bár be lehetne vezetni zárolási módoktól függő feloldási műveleteket, ha lenne hasznuk.

9.15. példa: A következő példában a T_1 tranzakció a T_2-vel konkurensen tudja végrehajtani a számításait, amely nem lenne lehetséges, ha a T_1 a kezdetben kizárólagosan zárolta volna a *B*-t. A két tranzakció:

T_1: $sl_1(A)$; $r_1(A)$; $sl_1(B)$; $r_1(B)$; $xl_1(B)$; $w_1(B)$; $u_1(A)$; $u_1(B)$;
T_2: $sl_2(A)$; $r_2(A)$; $sl_2(B)$; $r_2(B)$; $u_2(A)$; $u_2(B)$;

Itt a T_1 beolvassa *A*-t és *B*-t, és végrehajtja a (valószínűleg hosszadalmas) számításokat velük, és a legvégén az eredményt beírja a *B* új értékének. Megjegyezzük, hogy a T_1 előbb osztottan zárolja a *B*-t, és később, miután az *A*-val és *B*-vel kapcsolatos számításait befejezte, kér egy kizárólagos zárat a *B*-re. A T_2 tranzakció csak beolvassa az *A*-t és *B*-t, és nem ír rájuk.

A 9.17. ábra a műveletek egy lehetséges ütemezését mutatja. T_2 egy osztott zárat kap a *B*-n, a T_1 előtt, de a negyedik sorban T_1 is képes osztottan zárolni a *B*-t. Így a T_1 rendelkezésére áll az *A* is és a *B* is, és az értékeik felhasználásával végre tudja hajtani a számításokat. Attól kezdve, hogy a T_1 megpróbálja a *B*-n a zárat felminősíteni kizárólagossá, az ütemező a kérést elutasítja, és arra kényszeríti a T_1-et, hogy várjon addig, amíg a T_2 felszabadítja a *B*-n a zárat. Ezután megkapja a T_1 a *B*-n a kizárólagos zárat, beírja *B*-t, és befejeződik a tranzakció.

T_1	T_2
$sl_1(A)$; $r_1(A)$;	
	$sl_2(A)$; $r_2(A)$;
	$sl_2(B)$; $r_2(B)$;
$sl_1(B)$; $r_1(B)$;	
$xl_1(B)$ **Elutasítva**	
	$u_2(A)$; $u_2(B)$;
$xl_1(B)$; $w_1(B)$;	
$u_1(A)$; $u_2(B)$;	

9.17. ábra. *A zárak felminősítésével több konkurens művelet lehet*

Megjegyezzük, hogy ha a T_1 a kezdéskor kért volna kizárólagos zárat a *B*-re, mielőtt beolvasta volna a *B*-t, akkor ezt a kérést az ütemező elutasította volna, ugyanis a T_2-nek már volt egy osztott zárja a *B*-n. A T_1 nem tudta volna elvégezni a számításait a *B* beolvasása nélkül, és így T_1-nek sokkal több dolga lett volna, miután a T_2 felszabadította a zárakat. Végeredményben a T_1 később fejezte volna be, mint most, amikor a felminősítő stratégiát alkalmazta, ha csak kizárólagos zárat használt volna a *B*-n. □

9.16. példa: Sajnos a felminősítés válogatás nélküli alkalmazása a holtpontok új és potenciálisan komoly forrását jelenti. Tételezzük fel, hogy a T_1 is és a T_2 is beolvassa az *A* adatbáziselemet, és egy új értéket ír vissza az *A*-ba. Ha mindkét tranzakció a felminősítéssel dolgozik, akkor előbb osztott zárat kapnak az *A*-ra, és azután minősítik ezt át kizárólagossá, így a 9.18. ábrán javasolt eseménysorozat következhet be, amikor a T_1 és a T_2 közel egyidejűleg kezdődik.

A T_1 és T_2 is kaphat osztott zárat az *A*-ra. Ezután mindkettő megpróbálja ezt felminősíteni kizárólagossá, de az ütemező mindkettőt várakozásra kényszeríti, hiszen a másik már osztottan zárolja az *A*-t. Emiatt egyik végrehajtása sem folytatódhat, vagy

T_1	T_2
$sl_1(A)$;	
	$sl_2(A)$;
$xl_1(A)$ **Elutasítva**	
	$xl_2(A)$ **Elutasítva**

9.18. ábra. *Két tranzakció általi felminősítés holtpontot okozhat*

mindkettőnek örökösen kell várakoznia, vagy addig, amíg a rendszer felfedezi, hogy holtpont alakult ki, abortálja a két tranzakció valamelyikét, és a másiknak engedélyezi az A-ra a kizárólagos zárat. □

9.4.4. Módosítási zárak

El tudjuk kerülni a 9.16. példában vázolt holtpont problémát egy harmadik zárolási mód, az úgynevezett *módosítási zárak* (update locks) használatával. Az $ul_i(X)$ módosítási zár a T_i tranzakciónak csak az X olvasására ad jogot, az X írására nem. Azonban csak a módosítási zárat lehet később felminősíteni. Az olvasási zárat nem lehet felminősíteni. Módosítási zárat akkor is engedélyezhetünk az X-en, amikor az X már osztott módon zárolva van, ha azonban az X-en már van egy módosítási zár, akkor ez megakadályozza, hogy bármilyen más újabb zárat, akár osztott, akár módosítási, akár kizárólagos zárat kapjon az X. Ennek az az oka, hogy ha nem utasítanánk el ezeket az újabb zárolásokat, akkor előfordulhat, hogy a módosításinak soha sem lenne lehetősége kizárólagossá való felminősítésre, ugyanis mindig valamilyen más zár lenne az X-en.

Ez a szabály nem szimmetrikus kompatibilitási mátrixot eredményez, ugyanis az U módosítási zár úgy néz ki, mintha osztott zár lenne, amikor kérjük, és úgy néz ki, mintha kizárólagos zár lenne, amikor már megvan. Emiatt az U és S zárak oszlopai megegyeznek, és az U és X sorai úgyszintén megegyeznek. A mátrixot a 9.19. ábrán láthatjuk.[5]

	S	X	U
S	Igen	Nem	Igen
X	Nem	Nem	Nem
U	Nem	Nem	Nem

9.19. ábra. *Osztott (S), kizárólagos (X) és módosítási (U) zárak kompatibilitási mátrixa*

9.17. példa: A módosítási zárak használata nem befolyásolja a 9.15. példát. A harmadik művelet az lenne, hogy a T_1 módosítási zárat tenne az B-re, nem pedig osztott zárat. A módosítási zárat viszont megkapná, ugyanis csak osztott zárak vannak a B-n, és a 9.17. ábrával megegyező műveletsorozat fordulna elő.

Módosítási zárakkal megszüntethető a 9.16. példában bemutatott probléma. Most mind a T_1, mind a T_2 előbb módosítási zárat kér az A-n, és csak később kizárólagos zárat. A T_1 és T_2 egy lehetséges leírása az alábbi:

T_1: $ul_1(A)$; $r_1(A)$; $xl_1(A)$; $w_1(A)$; $u_1(A)$;
T_2: $ul_2(A)$; $r_2(A)$; $xl_2(B)$; $w_2(A)$; $u_2(A)$;

A 9.18. ábrának megfelelő eseménysorozatot a 9.20. ábrán láthatjuk. Itt a T_2-t elutasítjuk, amelyik másodikként kérte az A módosítási zárolását. A T_1-nek megengedjük, hogy befejeződjön, és ezután folytatódhat a T_2. A zárolási rendszer hatékonyan megakadályozta a T_1 és T_2 konkurens végrehajtását, ebben a példában viszont lényeges mennyiségű konkurens végrehajtás vagy holtpontot vagy inkonzisztens adatbázisállapotot eredményez. □

[5] Megjegyezzük, hogy az ütemezések jogszerűségével kapcsolatban van egy további feltétel, amely nem tükröződik a mátrixban: egy olyan tranzakciónak, amely az X-en osztott zárat tart fenn, de módosítási zárat nem, nem adható az X-re kizárólagos zár, még akkor sem, ha általában nem tiltjuk a tranzakcióknak, hogy egy elemen több zárat tartsanak fenn.

T_1	T_2
$ul_1(A)$; $r_1(A)$;	
	$ul_2(A)$ **Elutasítva**
$xl_1(A)$; $w_1(A)$; $u_1(A)$;	
	$ul_2(A)$; $r_2(A)$;
	$xl_2(A)$; $w_2(A)$; $u_2(A)$;

9.20. ábra. *Helyes végrehajtás a módosítási zárak használatával*

9.4.5. Növelési zárak

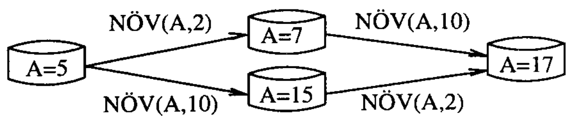
Egy másik érdekes zárolási mód, amely bizonyos helyzetekben hasznos lehet, a „növelési zár". Számos tranzakciónak csak az a hatása az adatbázison, hogy növeli vagy csökkenti a tárolt értékeket. Például:

1. Olyan tranzakció, amely pénzt utal át az egyik bankszámláról egy másikra.
2. Olyan tranzakció, amely repülőjegyeket árusít, és csökkenti az adott gépen a szabad ülőhelyek számát.

A növelési műveletek érdekes tulajdonsága, hogy tetszőleges sorrendben kiszámíthatók, ugyanis ha két tranzakció egy-egy konstanst ad hozzá ugyanahhoz az adatbáziselemhez, akkor nem számít, hogy melyiket hajtjuk végre előbb, ahogyan ezt a 9.21. ábrán látható adatbázis-állapot diagram javasolja. Másrészt a növelés nem cserélhető fel sem az olvasással, sem az írással. Ha azelőtt vagy azután olvassuk be az A-t, hogy valaki növelte, különböző értékeket kapunk, és ha azelőtt vagy azután növeljük az A-t, hogy más tranzakció új értéket írt be az A-ba, akkor is különböző értékei lesznek az A-nak az adatbázisban.

Vezessünk be a tranzakciókon mint egy lehetséges műveletet, a *növelési műveletet* (increment action), amelyet `NÖV(A,c)`-vel rövidítünk. Ez a művelet megnöveli c konstanssal az A adatbáziselemet, amelyről feltételezzük, hogy egyszerű szám. Megjegyezzük, hogy c negatív is lehet, ebben az esetben valójában csökkentjük az A értékét. Gyakorlatban alkalmazhatjuk a `NÖV`-öt a sor egy komponensére, annak ellenére, hogy maga a sor, és nem a komponense a zárolható elem.

Formálisan, a `NÖV(A,c)` a következő lépések atomi végrehajtására szolgál: `READ(A, t); t := t+c; WRITE(A,t);`. Nem ismertetjük azt a hardver és/vagy szoftverműködést, amely ezt a műveletet atomivá teszi, csak azt kell megjegyeznünk, hogy az atomiságnak ez az alakja alsóbb szintű, mint a tranzakcióknak a zárolások által támogatott atomisága.



9.21. ábra. *Két növelési művelet kiszámítása, mivel a végső adatbázis-állapot nem függ attól, hogy melyiket hajtottuk előbb végre*

Szükségünk van a növelési műveletnek megfelelő *növelési zárra* (increment lock), amelyet $il_i(X)$-szel fogunk jelölni, mely a T_i tranzakciónak az X-re vonatkozó növelési zárolásra kérése. A $növ_i(X)$ rövidítést arra a műveletre használjuk, amelyben a T_i tranzakció megnöveli az X adatbáziselemet valamely konstanssal. Annak nincs jelentősége, hogy pontosan mi is ez a konstans.

A növelési műveletek és zárak létezése szükségessé teszi, hogy több helyen módosítsuk a konzisztens tranzakciók, konfliktusok és jogszerű ütemezések definícióit. A változtatások az alábbiak:

a) Egy konzisztens tranzakció csak akkor végezheti el az X-en a növelési műveletet, ha egyidejűleg növelési zárat tart fenn az X-en. A növelési zár viszont nem teszi lehetővé sem az olvasási, sem az írási műveleteket.
b) Egy jogszerű ütemezésben bármennyi tranzakció bármikor fenntarthat az X-re növelési zárat. De ha egy tranzakció növelési zárat tart fenn az X-en, akkor egyidejűleg semelyik más tranzakció sem tarthat fenn sem osztott, sem kizárólagos zárat az X-en. Ezeket a követelményeket a kompatibilitási mátrix segítségével fejeztük ki, mely a 9.22. ábrán látható, ahol az I jelenti a növelési módú zárat (I az angol „increment" rövidítése).
c) A $növ_i(X)$ művelet konfliktusban áll az $r_j(X)$-szel, és a $w_j(X)$-szel is, $j \neq i$-re, de nem áll konfliktusban $növ_j(X)$-szel.

	S	X	I
S	Igen	Nem	Nem
X	Nem	Nem	Nem
I	Nem	Nem	Igen

9.22. ábra. *Osztott (S), kizárólagos (X) és növelési (I) zárak kompatibilitási mátrixa*

9.18. példa: Tekintsünk két tranzakciót, mindkettő beolvassa az A adatbáziselemet, és azután növeli a B-t. Lehet, hogy az A-t adják hozzá a B-hez, vagy egy olyan konstanssal növelik a B-t, amelynek a kiszámítása valamilyen más módon függ az A-tól.

T_1: $sl_1(A)$; $r_1(A)$; $il_1(B)$; $növ_1(B)$; $u_1(A)$; $u_1(B)$;
T_2: $sl_2(A)$; $r_2(A)$; $il_2(B)$; $növ_2(B)$; $u_2(A)$; $u_2(B)$;

Megjegyezzük, hogy a tranzakciók konzisztensek, ugyanis csak akkor végeznek növelést, amikor növelési zárral rendelkeznek, és csak akkor olvasnak, amikor osztott zárat tartanak fenn. A 9.23. ábra a T_1 és T_2-nek egy lehetséges ütemezését mutatja. A

T_1	T_2
$sl_1(A)$; $r_1(A)$;	
	$sl_2(A)$; $r_2(A)$;
	$il_2(B)$; $növ_2(B)$;
$il_1(B)$; $növ_1(B)$;	
	$u_2(A)$; $u_2(B)$;
$u_1(A)$; $u_1(B)$;	

9.23. ábra. *Növelési műveletekkel és zárakkal rendelkező tranzakciók ütemezése*

T_1 olvassa először az A-t, azután a T_2 beolvassa az A-t és növeli a B-t. Ezután viszont a T_1-nek is megengedjük, hogy növelési zárat kapjon a B-re, és folytatódjon.

Megjegyezzük, hogy az ütemezőnek a 9.23. ábrán egyik kérést sem kell késleltetnie. Például tételezzük fel, hogy T_1 növeli a B-t A-val, és T_2 növeli a B-t $2A$-val. Bármelyik sorrendben végrehajthatjuk, ugyanis az A értéke nem változik, és a növelést is bármely sorrendben elvégezhetjük.

Másképpen kifejezve, megnézhetjük a nem zárolási műveletek sorozatát a 9.23. ábra ütemezésében:

S: $r_1(A)$; $r_2(A)$; $növ_2(B)$; $növ_1(B)$;

Az utolsó műveletet, a $növ_1(B)$-t előrébb tudjuk hozni a második helyre, ugyanis ez nincs konfliktusban ugyanannak az elemnek egy másik növelésével, és biztosan nincs konfliktusban egy másik elem olvasásával. A cseréknek ez a sorozata mutatja, hogy az S konfliktusekvivalens a következő soros ütemezéssel:

$r_1(A)$; $növ_1(B)$; $r_2(A)$; $növ_2(B)$;.

Hasonlóan tudjuk az első műveletet, az $r_1(A)$-t cserékkel a harmadik helyre hátrább vinni, amely azt a soros ütemezést adja, amelyben a T_2 megelőzi T_1-et. □

9.4.6. Feladatok

9.4.1. feladat: A T_1, T_2 és T_3 tranzakciók minden alábbi ütemezésére:

a) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(D)$;
b) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(A)$;
c) $r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(D)$; $w_1(C)$; $w_2(D)$; $w_3(E)$;
* d) $r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(D)$; $w_1(A)$; $w_2(B)$; $w_3(C)$;
e) $r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(A)$; $w_1(A)$; $w_2(B)$; $w_3(C)$;

Végezzük el a következőket:

i) Illesszük be az osztott és a kizárólagos zárakat, és illesszük be a zárak feloldási műveleteit! Helyezzünk osztott zárat közvetlenül minden olyan olvasási művelet elé,

amelyik után nem következik ugyanannak a tranzakciónak ugyanarra az elemre való írási művelete! Helyezzünk kizárólagos zárat minden más olvasási és írási művelet elé! Helyezzük el minden tranzakció végére a szükséges zárfeloldásokat!

ii) Adjuk meg mi történik, amikor minden ütemezést osztott és kizárólagos zárakat támogató ütemező futtat!

iii) Illesszük be az osztott és kizárólagos zárakat oly módon, amely lehetővé teszi a felminősítést. Helyezzünk osztott zárat minden olvasás elé, és kizárólagos zárat minden írás elé, és helyezzük el a szükséges zárfeloldásokat a tranzakciók végére.

iv) Adjuk meg mi történik, amikor az iii)-ból minden ütemezést osztott, kizárólagos zárakat és felminősítést támogató ütemező futtat.

v) Illesszük be az osztott, kizárólagos és módosítási zárakat, a feloldási műveletekkel együtt. Helyezzünk osztott zárat minden olyan olvasási művelet elé, amelyiket nem fogunk felminősíteni, helyezzünk módosítási zárat minden olyan olvasási művelet elé, amelyeket felminősítünk, és helyezzünk kizárólagos zárat minden írási művelet elé! Helyezzük el a zárfeloldásokat a tranzakciók végére, mint rendszerint!

vi) Adjuk meg mi történik, amikor az v)-ből minden ütemezést osztott, kizárólagos és módosítási zárakat támogató ütemező futtat!

! **9.4.2. feladat:** Tekintsük az alábbi két tranzakciót:

T_1: $r_1(A)$; $r_1(B)$; $növ_1(A)$; $növ_1(B)$;
T_2: $r_2(A)$; $r_2(B)$; $növ_2(A)$; $növ_2(B)$;

Válaszoljunk a következőkre:

* a) Ezeknek a tranzakcióknak mennyi átlapolása (ütemezése) sorba rendezhető?
b) Ha T_2-ben megfordítanánk a növelések sorrendjét [vagyis $növ_2(B)$ után következne $növ_2(A)$], akkor mennyi sorba rendezhető ütemezés lenne?

9.4.3. feladat: Az alábbi ütemezések mindegyikére, illesszük be a megfelelő zárolásokat (olvasási, írási vagy növelési) minden művelet elé, és a zárak feloldási műveleteit a tranzakciók végére. Ezután magyarázzuk el mi történik, amikor az ütemezést egy olyan ütemező futtatja, amelyik ezt a három zárolási típust támogatja!

a) $r_1(A)$; $r_2(B)$; $növ_1(B)$; $növ_2(C)$; $w_1(C)$; $w_2(D)$;
b) $r_1(A)$; $r_2(B)$; $növ_1(B)$; $növ_2(A)$; $w_1(C)$; $w_2(D)$;
c) $növ_1(A)$; $növ_2(B)$; $növ_1(B)$; $növ_2(C)$; $w_1(C)$; $w_2(D)$;

9.4.4. feladat: A 9.1.1. feladatban láttunk egy repülőjárat-helyfoglalással kapcsolatos feltételezett tranzakciót. Ha a tranzakciókezelőnek lehetősége lenne osztott, kizárólagos, módosítási és növelési zárakat alkalmaznia, akkor milyen zárat javasolnánk a tranzakció minden egyes lépéséhez?

9.4.5. feladat: Egy konstans tényezővel való szorzási műveletet modellezhetünk egy saját művelettel. Tételezzük fel, hogy `MC(X,c)` a `READ(X,t); t := c*t; WRITE(X,t);` lépéseknek atomi végrehajtását jelenti. Bevezethetünk egy olyan zárolási módot is, amely lehetővé teszi a konstans tényezővel való szorzást.

a) Adjuk meg az olvasási, írási és konstanssal való szorzási zárolások kompatibilitási mátrixát.
! b) Adjuk meg az olvasási, írási, növelési és konstanssal való szorzási zárolások kompatibilitási mátrixát.

! **9.4.6. feladat:** A feladat kedvéért tegyük fel, hogy az adatbáziselemek kétdimenziós vektorok. Négy műveletet tudunk ezekkel a vektorokkal végrehajtani, és mindegyikhez saját típusú zárolás tartozik.

i) Megváltoztatjuk az értékeket az x tengely mentén (X zár).
ii) Megváltoztatjuk az értékeket az y tengely mentén (Y zár).
iii) Megváltoztatjuk a vektor hajlásszögét (A zár, ahol A az angol „angle" rövidítése).
iv) Megváltoztatjuk a vektor nagyságát (M zár, ahol M az angol „magnitude"-ból származik).

Válaszoljunk az alábbi kérdésekre:

* a) Mely műveletpárok felcserélhetők? Például, ha elforgatjuk a vektort úgy, hogy a hajlásszöge 120° legyen, és azután megváltoztatjuk az x koordinátáját 10-re, ez ugyanaz-e, mintha először változtatnánk meg az x koordinátáját 10-re, és azután változtatnánk a hajlásszöget 120°-ra?
b) Az a) válasz alapján mi a négy zártípus kompatibilitási mátrixa?
!! c) Tegyük fel, hogy megváltoztatjuk a négy műveletet úgy, hogy ahelyett, hogy új értéket adnánk egy mértéknek, inkább növeljük a mértéket (pl. „adjunk hozzá 10-et az x koordinátához", vagy „forgassuk el a vektort 30°-kal az óramutató irányába"). Mi lenne ekkor a kompatibilitási mátrix?

! **9.4.7. feladat:** Az alábbi ütemezésből egy művelet hiányzik:

$r_1(A)$; $r_2(B)$; ???; $w_1(C)$; $w_2(A)$;

Az a feladatunk, hogy találjunk bizonyos művelettípusú műveleteket a ??? helyettesítésére, amivel az ütemezés nem lenne sorba rendezhető! Adjuk meg az összes nem sorba rendezhető helyettesítést az alábbi művelettípusok mindegyikére:

* a) olvasási műveletek;
b) írási műveletek;
c) módosítási műveletek;
d) növelési műveletek.

9.5. A zárolási ütemező felépítése

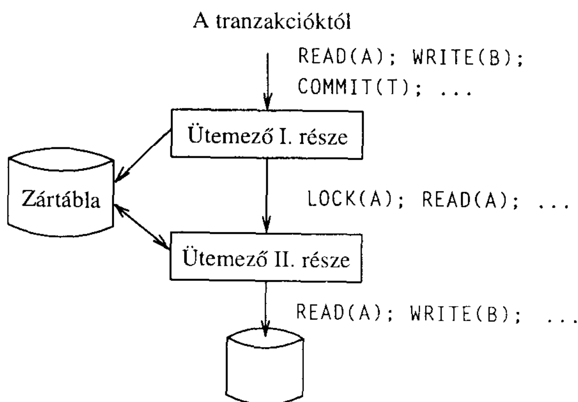
Eddig már számos zárolási sémát láttunk, most a leginkább arra van szükségünk, hogy megnézzük, hogyan működik egy olyan ütemező, amely ezek közül a sémák közül használja valamelyiket. Itt csak néhány megadott elven alapuló egyszerű ütemező felépítését tekintjük:

1. Maguk a tranzakciók nem kérnek zárakat, vagy figyelmen kívül hagyjuk, hogy ezt teszik. Az ütemező feladata, hogy zárolási műveleteket szúrjon be az adatokhoz hozzáférő olvasási, írási illetve egyéb műveletek sorába.
2. Nem a tranzakciók, hanem az ütemező oldja fel a zárakat, mégpedig akkor, amikor a tranzakciókezelő a tranzakció véglegesítésére vagy abortálására készül.

9.5.1. Zárolási műveleteket beszúró ütemező

A 9.24. ábra egy olyan két részből álló ütemezőt mutat be, amely olvasás, írás, végrehajtás, abortálás kéréseket fogad a tranzakcióktól. Az ütemező karbantartja a zártáblát, amelyet bár másodlagosan tárolt adatként ábrázoltunk, lehet, hogy részben vagy egészben a központi memóriában tárolunk. A zártábla által használt központi memória általában nem a lekérdezés-végrehajtás és a naplózás által használt pufferterület része. A zártábla az adatbázis-kezelő rendszernek csak egy komponense, és az operációs rendszer foglal le neki helyet ugyanúgy, mint az adatbázis-kezelő rendszer többi kódjának és belső adatainak.

A tranzakciók által kért műveletek általában az ütemezőn jutnak keresztül, és az adatbázison kerülnek végrehajtásra. Bizonyos körülmények esetén viszont *késleltett* a tranzakció, zárolásra vár, és a kérései (még) nem jutottak el az adatbázishoz. Az ütemező két része a következő műveleteket hajtja végre:



9.24. ábra. *Egy ütemező, amely beszúrja a zárolási kéréseket a tranzakciók kéréseinek sorába*

1. Az I. rész fogadja a tranzakciók által generált kérések sorát, és minden adatbázis-hozzáférési művelet elé, mint az olvasás, írás, növelés vagy a módosítás, beszúrja a megfelelő zárolási műveletet. Az adatbázis-hozzáférési műveleteket ezután átküldi a II. részhez. Az ütemező I. részének kell kiválasztania a megfelelő zárolási módot az ütemező által használt zármódok halmazából.
2. A II. rész fogadja az I. részen keresztül érkező zárolási és adatbázis-hozzáférési műveletek sorozatát, és mindegyiket pontosan végrehajtja. Ha egy zárolási vagy adatbázis-hozzáférési kérés érkezik a II. részhez, eldönti, hogy az igénylő a T tranzakciót késlelteti-e, mivel a zárat nem tudja engedélyezni. Ha így van, akkor magát a műveletet késlelteti, és hozzáadja azoknak a műveleteknek a listájához, amelyeket még végre kell hajtania a T tranzakciónak. Ha a T nem késleltetett (vagyis az összes előzőleg kért zár már korábban engedélyezve van), akkor

   a) Ha a művelet adatbázis-hozzáférés, akkor továbbítja az adatbázishoz, és végrehajtja.
   b) Ha zárolási művelet érkezik a II. részhez, megvizsgálja a zártáblát, hogy vajon a zár engedélyezhető-e.

      i) Ha igen, akkor úgy módosítja a zártáblát, hogy az éppen engedélyezett zárat is tartalmazza.
      ii) Ha nem, akkor egy bejegyzést kell elkészítenie a zártáblában, mely jelzi a zárolási kérést. Az ütemező II. része ezután késlelteti a T tranzakció további műveleteit mindaddig, amíg nem tudja engedélyezni a zárat.

3. Amikor a T tranzakciót véglegesítjük vagy abortáljuk, akkor a tranzakciókezelő értesíti az I. részt, hogy oldja fel az összes T által fenntartott zárat. Ha bármelyik tranzakció várakozik ezen zárak valamelyikére, akkor az I. rész értesíti a II. részt.
4. Amikor a II. rész értesül, hogy valamelyik X adatbáziselemen elérhetővé vált egy zár, akkor eldönti, hogy melyik a következő tranzakció vagy tranzakciók, amelyek megkapják most a zárat az X-re. A tranzakció(k), amely(ek) megkapták a zárat, a késleltetett műveleteik közül annyit végrehajthatnak, amennyit csak végre tudnak hajtani mindaddig, amíg vagy befejeződnek, vagy egy másik zárolási kéréshez érkeznek el, amelyet nem lehet engedélyezni.

9.19. példa: Ha csak egymódú zárak vannak, mint a 9.3. részben, akkor az ütemező I. részének a feladata egyszerű. Ha bármilyen műveletet lát az X adatbáziselemen, és még nem szúrt be zárolási kérést az X-re az adott tranzakcióhoz, akkor beszúrja a kérést. Amikor a tranzakciót véglegesítjük vagy abortáljuk, az I. rész törölheti ezt a tranzakciót, miután feloldotta a zárakat, így az I. részhez igényelt memória nem nő korlátlanul.

Amikor többmódú zárak vannak, az ütemezőnek szüksége lehet arra, hogy azonnal értesüljön, milyen későbbi műveletek fognak előfordulni ugyanazon az adatbáziselemen. Nézzük meg újból az osztott-kizárólagos-módosítási zárak esetét, a 9.15. példa tranzakcióit használva, amelyeket most a zárolások nélkül írunk fel:

T_1: $r_1(A)$; $r_1(B)$; $w_1(B)$;
T_2: $r_2(A)$; $r_2(B)$;

Az ütemező I. részéhez küldött üzenetnek nemcsak az olvasási és írási kéréseket kell tartalmaznia, hanem ugyanazon az elemen bekövetkező későbbi műveletekre vonatkozó jelzést is. Amikor az $r_1(B)$ érkezik be például, az ütemezőnek tudnia kell, hogy lesz-e később $w_1(B)$ művelet (vagy lehet-e ilyen művelet, ha a T_1 tranzakció kódjában elágazás szerepel). Több módon válhat az információ elérhetővé. Például, ha a tranzakció egy lekérdezés, akkor tudjuk, hogy semmit sem fog írni. Ha a tranzakció egy SQL-adatbázist módosítási utasítás, akkor a lekérdező processzor azonnal megadhatja azokat az adatbáziselemeket, melyeket olvashatunk és írhatunk is egyben. Ha a tranzakció beágyazott SQL-program, akkor a fordító hozzá tud férni az összes SQL-utasításhoz (és csakis ezekkel lehet írni az adatbázisba), és meghatározhatja, mely adatbáziselemek esélyesek az írásra.

A példánkban tételezzük fel, hogy a 9.17. ábrán javasolt sorrendben következnek be az események. Ekkor a T_1 először $r_1(A)$-t adja ki. Mivel nincs később kizárólagos zárrá való felminősítés erre a zárra, az ütemező beszúrja az $sl_1(A)$-t az $r_1(A)$ elé. Ezután a T_2 kérései – $r_2(A)$ és $r_2(B)$ – érkeznek az ütemezőhöz. Megint nincs később felminősítés, így az ütemező I. része a következő műveletsorozatot adja ki: $sl_2(A)$; $r_2(A)$; $sl_2(B)$; $r_2(B)$;.

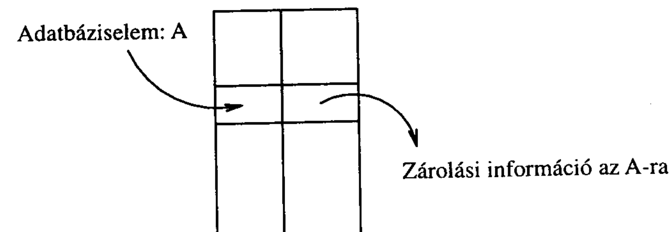
Ezután az $r_1(B)$ művelet érkezik be az ütemezőhöz azzal a figyelmeztetéssel, hogy ezt a zárat fel lehet minősíteni. Az ütemező I. része ekkor kibocsátja $ul_1(B)$; $r_1(B)$-t a II. résznek. Az utóbbi megnézi a zártáblát, és azt találja, hogy a T_1 engedélyezheti a módosítási zárat B-re, ugyanis csak osztott zárak vannak a B-n.

Amikor a $w_1(B)$ művelet beérkezik az ütemezőhöz, az I. rész kibocsátja az $xl_1(B)$; $w_1(B)$-t. A II. rész viszont nem teljesítheti az $xl_1(B)$ kérést, ugyanis a T_2-nek már van osztott zárja a B-n. A T_1-nek ezt és ez utáni minden műveletét késlelteti, egyben a II. rész tárolja a későbbi végrehajtáshoz. Végül a T_2 végrehajtja a véglegesítést, és az I. rész feloldja a zárakat az A-n és a B-n, amelyet a T_2 tartott fenn. Ugyanekkor felfedezi, hogy a T_1 várakozik a B zárolására. Értesíti a II. részt, és ez az $xl_1(B)$ zárolást most már végrehajthatónak találja. Beviszi ezt a zárat a zártáblába, és folytatja a T_1-től tárolt műveletek végrehajtását mindaddig, ameddig tudja. Az esetünkben a T_1 befejeződik. □

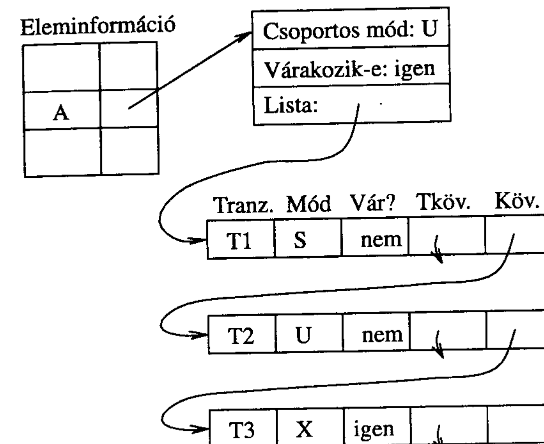
9.5.2. A zártábla

Absztrakt szinten a zártábla egy olyan reláció, amely összekapcsolja az adatbáziselemeket az elemre vonatkozó zárolási információval, mint ahogyan ezt a 9.25. ábra mutatja. A táblát például egy olyan tördelőtáblával lehet megvalósítani, amely az adatbáziselemek (címeit) használja tördelőkulcsként. Azok az elemek, amelyek nincsenek zárolva, nem fordulnak elő a táblában, így a méret csak a zárolt elemek számával arányos, nem pedig a teljes adatbázis méretével.

A 9.26. ábrán egy példát láthatunk, hogy milyen információk találhatók a zártábla-bejegyzésnél. Ez a példaszerkezet feltételezi, hogy az ütemező a 9.4.4. rész osztott-



9.25. ábra. *A zártábla az adatbáziselemekről a zárolási információkra történő leképezés*



9.26. ábra. *Zártábla-bejegyzések szerkezete*

kizárólagos-módosítási zársémáit alkalmazza. Az A-hoz, egy tipikus adatbáziselemhez, a bejegyzés a következő komponensekből álló sor:

1. A *csoportos mód* (group mode) a legszigorúbb feltételek összefoglalása, amivel egy tranzakció szembesül, amikor egy új zárolást kér az A-n. Ahelyett, hogy összehasonlítanánk a zárolási kérést a többi tranzakciónak ugyanazon az elemen fenntartott minden zárolásával, egyszerűsíthetjük az engedélyezési/elutasítási döntést azzal, hogy a kérést csak a csoportos móddal[6] hasonlítjuk össze. Az osztott-kizárólagos-módosítási (rövidítve: SXU) zárolási sémákhoz egyszerű a szabály: egy csoportos módban:

[6] A zároláskezelőnek viszont foglalkoznia kell azzal a lehetőséggel, hogy a kérést kiadó tranzakciónak már van egy másik módban zárja ugyanazon az elemen. Például az SXU zárolási rendszerre vonatkoztatva, a zároláskezelő elfogadhat egy X zár kérést, ha az igénylő tranzakció pont az, amely U zárat tart fenn ugyanazon az elemen. Azoknál a rendszereknél, amelyek nem támogatják, hogy egy tranzakció egy elemen több zárat is tartson, a csoportos mód mindig megadja mindazt, amit a zároláskezelőnek tudnia kell.

a) *S* azt jelenti, hogy csak osztott zárak vannak (*S* az angol „shared" rövidítése)
b) *U* azt jelenti, hogy egy módosítási zár van (*U* az angol „update" rövidítése), és lehet még egy vagy több osztott zár is.
c) *X* azt jelenti, hogy csak egy kizárólagos zár van (*X* az angol „eXclusive" szóból származik), és semmilyen más zár nincs.

A többi zárolási sémához is mindig találunk a csoportos mód összegzésének megfelelő rendszert. Ezeket a példákat feladatként javasoljuk elvégezni.

2. A *várakozási bit* (waiting bit) azt adja meg, hogy van-e legalább egy tranzakció, amely az *A* zárolására várakozik.

3. Az összes olyan tranzakciót leíró lista, amelyek vagy jelenleg zárolják az *A*-t, vagy az *A* zárolására várakoznak. Hasznos információk, amelyeket minden listabejegyzés tartalmazhat:

a) A zárolást fenntartó vagy a zárolásra váró tranzakció neve.
b) Ennek a zárnak a módja.
c) A tranzakció fenntartja-e a zárat vagy várakozik-e a zárra.

A 9.26. ábrán két kapcsolást mutatunk minden bejegyzéshez. Az egyik (`Köv`) magukhoz az adatbáziselemre vonatkozó bejegyzésekhez tartozó kapcsolás, a másik pedig (az ábrán `Tköv`) egy bizonyos tranzakció összes bejegyzéséhez kapcsolás. Az utóbbi kapcsolás akkor használható, amikor a tranzakciót véglegesítjük vagy abortáljuk, így könnyen megtalálhatjuk az összes zárat, amelyet fel kell oldanunk.

Zárolási kérések kezelése

Tételezzük fel, hogy a *T* tranzakció zárat kér az *A*-ra. Ha nincs az *A*-ra bejegyzés a zártáblában, akkor biztos, hogy zárak sincsenek az *A*-n, így létrehozhatjuk a bejegyzést, és engedélyezhetjük a kérést. Ha a zártáblában létezik bejegyzés az *A*-ra, akkor ezt felhasználjuk a zárolási kéréssel kapcsolatos döntésünkben. Megkeressük a csoportos módot, amely a 9.26. ábrán az *U*, vagyis „módosítási". Amikor már van módosítási zár egy elemen, akkor semmilyen más zárat nem engedélyezhetünk (kivéve azt az esetet, amikor maga a *T* tartja fenn az *U* zárat, és a többi zárak kompatibilisek *T* kérésével). Tehát a *T*-nek ezt a kérését elutasítjuk, és egy bejegyzést helyezünk el a listában, amely szerint *T* zárat kért (bármilyen módban kérte), és `Vár? = 'igen'`.

Ha a csoportos mód *X*, vagyis kizáró lenne, akkor ugyanez történne. Ha azonban a csoportos mód *S*, vagyis osztott lenne, akkor lehetne adni egy másik osztott vagy módosítási zárat. Ebben az esetben, a *T* bejegyzése a listában `Vár? = 'nem'`, és a csoportos módot az *U*-ra kellene cserélni, ha az új zár módosítási zár, egyébként pedig a csoportos mód az *S* maradna. Akár adtunk engedélyt a zárolásra, akár nem, az új lista bejegyzéshez megfelelő kapcsolat létesül, a `Tköv` és a `Köv` mezőkön keresztül. Megjegyezzük, hogy akár engedélyezzük a zárat, akár nem, a zártáblában a bejegyzés megadja az ütemezőnek azt, amit tudnia kell anélkül, hogy megvizsgálná a zárolások listáját.

Zárfeloldások kezelése

Most tételezzük fel, hogy a *T* tranzakció feloldja *A*-t. Ekkor *T* bejegyzését *A*-ra a listában töröljük. Ha a *T* által fenntartott zár nem egyezik meg a csoportos móddal (pl. *T* egy *S* zárat tart fenn, amíg a csoportos mód *U*), akkor nincs okunk, hogy megváltoztassuk a csoportos módot. Másrészt, ha a *T* által fenntartott zár van a csoportos módban, akkor meg kell vizsgálnunk a teljes listát, hogy megtaláljuk az új csoportos módot. A 9.26. ábrán található példában láttuk, hogy csak egyetlen *U* zár lehet egy elemen, így ha azt a zárat feloldjuk, az új csoportos mód csak az *S* lehetne (ha maradt még osztott zár), vagy semmi (ha nincs más zár jelenleg fenntartva).[7] Ha a csoportos mód *X*, akkor tudjuk, hogy nincsenek más zárolások, és ha a csoportos mód *S*, akkor el kell döntenünk, hogy van-e további osztott zár.

Ha a `Várakozik` értéke `'igen'`, akkor engedélyeznünk kell egy vagy több zárat a kért zárak listájáról. Több különböző megközelítés lehetséges, és mindegyiknek megvan a saját előnye:

1. *Első-beérkezett-első-kiszolgása* (first-come-first-served): Azt a zárolási kérést engedélyezzük, amelyik a legrégebb óta várakozik. Ez a stratégia azt biztosítja, hogy ne legyen kiéheztetés, vagyis a tranzakció ne várjon örökké egy zárra.
2. *Osztott záraknak elsőségadás* (priority to shared locks): Először az összes várakozó osztott zárat engedélyezzük. Ezután egy módosítási zárolást engedélyezünk, ha várakozik ilyen. A kizárólagos zárolást csak akkor engedélyezzük, ha semmilyen más igény nem várakozik. Ez a stratégia csak akkor engedi a kiéheztetést, ha a tranzakció *U* vagy *X* zárolásra vár.
3. *Felminősítésnek elsőségadás* (priority to upgrading): Ha van olyan *U* zárral rendelkező tranzakció, amely *X* zárrá való felminősítésre vár, akkor ezt engedélyezzük előbb. Másküldönben a fent említett stratégiák valamelyikét követjük.

9.5.3. Feladatok

9.5.1. feladat: Melyek a zártáblához a megfelelő csoportos módok, ha az alkalmazott zárolási módok az alábbiak:

a) osztott és kizárólagos zárak;
*! b) osztott, kizárólagos és növelési zárak;
!! c) a 9.4.6. feladatban szereplő zárolási módok.

9.5.2. feladat: A 9.2.4. feladat minden ütemezéséhez adjuk meg azokat a lépéseket, amelyeket ebben a fejezetben leírt zárolási ütemező végezne el.

[7] Valójában sohasem találunk „semmi" csoportos módot, ugyanis ha nincs sem zár, sem zárolási kérés egy elemen, akkor nincs bejegyzés sem a tárolási táblában erre az elemre.

9.6. Adatbáziselemekből álló hierarchiák kezelése

Most térjünk vissza a különféle zárolási sémák feltárásához, amelyet a 9.4. részben elkezdtünk. Különösen két olyan problémára összpontosítunk, amelyek akkor merülnek fel, amikor fastruktúra tartozik az adatainkhoz.

1. Az első fajta fastruktúra, amelyet figyelembe veszünk, a zárolható elemek hierarchiája. Ebben a részben megvizsgáljuk, hogyan engedélyezünk zárolást mind a nagy elemekre, pl. relációkra, mind a kisebb elemekre, mint pl. a reláció néhány sorát tartalmazó blokkokra vagy egyedi sorokra.
2. A másik lényeges hierarchiafajtát képezik a konkurenciavezérlési rendszerekben azok az adatok, amelyek önmagukban faszervezésűek. Jelentősebb példa a B-fa-indexek. A B-fák csomópontjait adatbáziselemeknek tekinthetjük, viszont ha így tekintjük, mint ahogyan azt a 9.7. részben látni fogjuk, az eddig tanult zárolási sémákat szegényesen használhatjuk, emiatt egy új megközelítésre van szükségünk.

9.6.1. Többszörös szemcsézettségű zárak

Emlékezzünk vissza, hogy az „adatbáziselem" kifejezést szándékosan definiálatlanul hagytuk, ugyanis a különböző rendszerek különböző méretű adatbáziselemeket zárolnak, mint pl. sorokat, lapokat vagy blokkokat, relációkat. Bizonyos alkalmazásoknál a kis adatbáziselemek előnyösek, mint amilyen a sorok, amíg másoknál a nagy elemek nyújtják a legtöbbet.

9.20. példa: Tekintsünk egy banki adatbázist. Ha a relációkat kezeljük adatbáziselemekként, akkor így csak egy zárat tudunk kiadni arra a teljes relációra, amely a számlák egyenlegét adja meg, ezért a rendszer nagyon kis konkurenciát engedélyezne. Mivel a legtöbb tranzakció a számla egyenlegét változtatja, vagy pozitívan vagy negatívan, a legtöbb tranzakciónak kizárólagosan kellene zárolnia a számlaegyenlegek relációt. Így csak egyetlen befizetést vagy kivételt tudnánk egyidejűleg elvégezni, nem számítana, hogy hány olyan processzor lenne, amely alkalmas lenne ezeknek a tranzakcióknak az elvégzésére. Jobb megközelítés, hogy egyedi oldalakat vagy adatblokkokat zárolunk. Így két olyan számla, amelynek a sorai különböző blokkban vannak, egyidejűleg módosítható. Ez biztosítja szinte a teljes konkurenciát, amely elérhető a rendszerben. A másik véglet az lenne, ha minden egyes sorra biztosítanánk zárolást, így bármilyen számlahalmazt egyszerre tudnánk módosítani, de a záraknak ennyire finom szemcséssége valószínűleg nem érné meg a sok fáradságot.

Ellentétes esetben, tekintsünk egy dokumentumokból álló adatbázist. Ezeket a dokumentumokat időnként szerkeszteni szokták, és a legtöbb tranzakció teljes dokumentumokhoz fér hozzá. Az adatbáziselem ésszerű megválasztása ekkor a teljes dokumentum. Mivel a legtöbb tranzakció *csak olvasási tranzakció* (vagyis nem végez írási műveletet), a zárolás csak azért szükséges, hogy elkerüljük a szerkesztés közben a dokumentumok olvasását. Ha kisebb szemcsézettségű elemeket zárolnánk, mint például paragrafusokat, mondatokat vagy szavakat, akkor ennek semmilyen előnyét sem látnánk, viszont sokkal költségesebb lenne. Az egyetlen tevékenység, amelyet a kisebb szemcsézettségű zárak támogatnának az, hogy a dokumentum egy részét tudnánk olvasni a dokumentum szerkesztése közben is. □

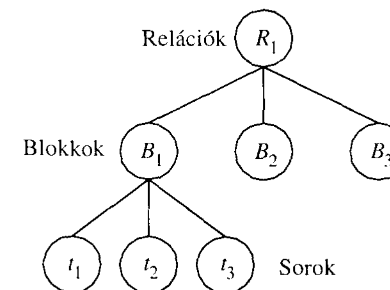
Bizonyos alkalmazások mind a nagy, mind a kis szemcsézettségű zárakat is tudják alkalmazni. Például a 9.20. példában tárgyalt banki adatbázisnál világos, hogy blokk- vagy sorszintű zárolás is szükséges, de néhány esetben a teljes számlareláció zárolása is szükséges lehet, annak érdekében, hogy ellenőrizzük a számlákat (pl. ellenőrizzük, hogy helyesek-e a számlaösszegek). De ha osztott zárat teszünk a számlarelációra annak érdekében, hogy kiszámoljunk a reláción valamilyen csoportfüggvényt, és egyidejűleg az egyéni számlák soraihoz kizárólagos zárat adunk, ez könnyen nem sorba rendezhető viselkedéshez vezethet, ugyanis a reláció valójában megváltozik, amíg egy feltehetően befagyasztott másolatát olvassuk a csoportfüggvényes lekérdezéshez.

9.6.2. A figyelmeztető zárak

A probléma megoldását, hogy hogyan kezeljük az újfajta zárolással kapcsolatos különféle szemcsézettségeken levő zárakat, „figyelmeztetés" nevű új zárat vezetünk be. Ezek a zárak akkor hasznosak, amikor adatbáziselemek beágyazott vagy hierarchikus struktúrákat mutatnak, mint azt a 9.27. ábrán láthatjuk. Itt az adatbáziselemek három szintjét figyelhetjük meg:

1. a relációk a legnagyobb zárolható elemek;
2. minden reláció egy vagy több blokkból vagy lapból épül fel, amelyekben a soraik vannak;
3. minden blokk egy vagy több sort tartalmaz.

Az adatbáziselemek hierarchiáján a zárak kezelésére szolgáló szabályok alkotják a *figyelmeztető protokollt* (warning protocol), amely tartalmazza mind a „közönséges" zárakat, mind a „figyelmeztető" zárakat. A zárolási sémát úgy adjuk meg, hogy a közönséges zárak S és X (osztott és kizárólagos). A figyelmeztető zárakat a közönséges



9.27. ábra. *Hierarchikusan szervezett adatbáziselemek*

zárak elé helyezett *I* előtaggal jelöljük (az angol „intention to" = szándékszik rövidítése). Például *IS* azt jelenti, hogy szándékunkban áll osztott zárat kapni egy részelemen. A figyelmeztető protokoll szabályai:

1. Ahhoz, hogy elhelyezzünk egy közönséges *S* vagy *X* zárat valamely elemen, a hierarchia gyökerénél kell kezdenünk.
2. Ha már annál az elemnél tartunk, amelyet zárolni akarunk, akkor nem kell tovább folytatnunk, hanem kérjük az *S* vagy *X* zárolást arra az elemre.
3. Ha az elem, amelyet zárolni szeretnénk, lejjebb van a hierarchiában, akkor elhelyezzünk egy figyelmeztetést ezen a csomóponton. Vagyis ha osztott zárat szeretnénk kérni egy részelemen, akkor ebben a csomópontban egy *IS* zárat kérünk. Ha kizárólagos zárat akarunk egy részelemen kérni, akkor ebben a csomópontban egy *IX* zárat kérünk. Amikor a jelenlegi csomópontban levő zárat megkaptuk, akkor az ehhez a csomóponthoz tartozó utódcsomóponttal folytatjuk (azzal, amelyikhez tartozó részfa tartalmazza azt a csomópontot, amelyet zárolni kívánunk). Ezután megfelelően a 2. vagy 3. lépéssel folytatjuk mindaddig, amíg elérjük a keresett csomópontot.

Ahhoz, hogy eldöntsük, engedélyezhetjük-e ezek közül a zárak közül valamelyiket vagy sem, a 9.28. ábrán található kompatibilitási mátrixot használjuk. Ennek a mátrixnak az értelmezéséhez először nézzük meg az *IS* oszlopot. Ha *IS* zárat kérünk egy *N* csomópontban, az *N* egy leszármazottját szándékozzuk olvasni. Ez a szándék csak abban az esetben okozhat problémát, ha már egy másik tranzakció korábban jogosulttá vált arra, hogy az *N* által reprezentált teljes adatbáziselemről egy új példányt készítsen, emiatt „Nem" található az *X*-hez tartozó sorban. Megjegyezzük, hogy ha más tranzakció azt tervezi, hogy csak egy részelemét írja, ezt az *N*-en *IX* zárral jelölve meg, akkor lehetőségünk van arra, hogy engedélyezzük az *IS* zárat az *N*-en, és a konfliktust alsóbb szinten oldhatjuk meg, ha valóban az írási szándék és az olvasási szándék egy közös elemhez kapcsolódik.

	IS	*IX*	*S*	*X*
IS	Igen	Igen	Igen	Nem
IX	Igen	Igen	Nem	Nem
S	Igen	Nem	Igen	Nem
X	Nem	Nem	Nem	Nem

9.28. ábra. *Osztott (S), kizárólagos (X), és szándékjelölő (I előtaggal jelölt) zárak kompatibilitási mátrixa*

Most tekintsük az *IX*-hez tartozó oszlopot. Ha az *N* csomópont egy részelemét szándékozzuk írni, akkor meg kell akadályoznunk az *N* által képviselt teljes elem olvasását vagy írását. Ekkor „Nem"-et látunk az *S* és *X* zármódok bejegyzéseiben. Azonban az *IS* oszloppal kapcsolatban leírtaknak megfelelően, más tranzakció, amely egy részelemet olvas vagy ír, a potenciális konfliktusokat az adott szinten kezeli le, így az *IX* nincs konfliktusban egy másik *IX*-szel vagy egy *IS*-sel az *N*-en.

Ezután nézzük az *S*-hez tartozó oszlopot. Az *N* csomópontnak megfeleltetett elem olvasása nincs konfliktusban sem egy másik olvasási zárral az *N*-en, sem egy olvasási zárral az *N* egy részelemén, amelyet az *N*-en *IS* reprezentál. Emiatt „Igen"-t találunk az *S* és az *IS* soraiban is. Azonban egy *X* vagy egy *IX* azt jelenti, hogy más tranzakció írni fogja legalábbis egy részét az *N* által reprezentált elemnek. Ezért nem tudjuk engedélyezni a teljes olvasását az *N*-nek, amelyet a „Nem" bejegyzés fejez ki az *S* oszlopban.

Végül a *X* oszlopban csak „Nem" bejegyzések vannak. Nem tudjuk megengedni az *N* csomópont egyik részének írását sem, ha más tranzakciónak már joga van olvasni vagy írni az *N*-et, vagy arra, hogy megszerezze ezt a jogot az *N* egy részelemére.

9.21. példa: Tekintsük a következő relációt

```
Film(filmCím, év, hossz, stúdióNév)
```

Tételezzük fel, hogy a teljes relációra és az egyedi sorokra követelünk zárolást. Ekkor a T_1 tranzakció, amely az alábbi kérdést tartalmazza:

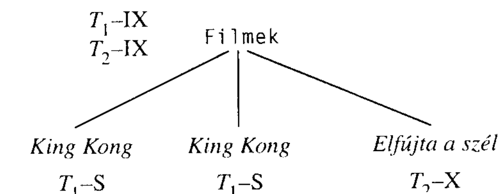
```
SELECT *
FROM Film
WHERE filmCím = 'King Kong';
```

azzal kezdődik, hogy *IS* módon zárolja a teljes relációt. Ezután veszi az egyedi sorokat (két film szerepel a *King Kong* filmcímmel), és *S* módú zárolást ad ki ezekre.

Most tételezzük fel, hogy mialatt az első lekérdezést végezzük, elkezdődik a T_2 tranzakció, amely a sorok év komponensét változtatja meg:

```
UPDATE Film
SET év = 1939
WHERE filmCím = 'Elfújta a szél';
```

Ekkor a T_2-nek szüksége van a reláció *IX* módú zárolására, ugyanis azt tervezi, hogy új értéket ír be az egyik sorba. Ez kompatibilis a T_1-nek a relációra vonatkozó *IS* zárolásával, így a zárat engedélyezzük. Amikor a T_2 elérkezik az *Elfújta a szél*hez tartozó sorhoz, ezen a soron nem talál zárat, így megkapja az *X* módú zárat, és átírja a sort. Ha a T_2 a *King Kong* filmek valamelyikéhez próbált volna új értéket beírni, akkor várnia kellett volna, amíg a T_1 felszabadítja az *S* zárakat, ugyanis az *S* és *X* nem kompatibilisek. A 9.29. ábrán láthatjuk a zárak kollekcióját. □



9.29. ábra. *Engedélyezett zárak a* `Film` *soraihoz hozzáférő két tranzakcióhoz*

Csoportos mód a szándékzárolásokhoz

A 9.28. ábrán szereplő kompatibilitási mátrix olyan helyzetet mutat be, amelyet eddig még nem láttunk a zármódok erejét illetően. A korábbi zárolási sémákban, valahányszor lehetőségünk volt arra, hogy egy adatbáziselemet egyszerre kétféleképpen, *M* és *N* módban is zároljunk, ezek közül az egyik *dominánsabb* volt a másiknál, mégpedig abban az értelemben, hogy amikor az egyik mód sorában és oszlopában minden olyan pozícióban „Nem" áll, amelyben a másik mód sorában vagy oszlopában a „Nem" áll. Például a 9.19. ábrán látjuk, hogy az *U* dominánsabb az *S*-nél, és az *X* dominánsabb az *S*-nél is és az *U*-nál is. Egy előnye annak, hogy tudjuk, mindig van egy domináns zár egy elemen az, hogy több zárolás hatását össze tudjuk foglalni egy „csoportos mód"-dal, amint azt a 9.5.2. részben tárgyaltuk.

Amint a 9.28. ábrán látjuk, az *S* és *IX* módok közül egyik sem dominánsabb a másiknál. Továbbá egy elemet *S* és *IX* módok mindkettőjében zárolhatunk egyidejűleg, feltéve, hogy ugyanaz a tranzakció kérte a zárolást (vigyázzunk, hogy a „nem" bejegyzések a kompatibilitási mátrixban csak azokra a zárakra alkalmazhatók, amelyeket *más* tranzakciók tartanak fenn). Egy tranzakció mindkét zárolást kérheti, ha egy teljes elemet akar beolvasni, és azután a részelemek csak kis részhalmazát akarja írni. Ha egy tranzakciónak *S* és *IX* zárolásai is vannak egy elemen, akkor ez korlátozza a többi tranzakciót olyan mértékben, ahogy bármelyik zár teszi. Vagyis elképzelhetünk egy másik *SIX* zárolási módot, amelynek a sorai és oszlopai a „Nem"-et tartalmazzák az *IS* bejegyzés kivételével mindenhol. Az *SIX* zárolási mód csoportmódként szolgál, ha van olyan tranzakció, amelynek van *S*, illetve *IX* módú, de nincs *X* módú zárolása.

Elképzelhetjük ugyanezt a helyzetet a 9.22. ábrán levő mátrixnál a növelési zárolásokra. Vagyis egy tranzakció az *S* és az *I* módokban is fenntarthatna zárakat. Ez a helyzet viszont ekvivalens az *X* módú zárolással, így ekkor az *X*-et csoportos módként használhatnánk.

9.6.3. Fantomok és a beszúrások helyes kezelése

Amikor a tranzakciók egy zárolható elem új részelemeit hozzák létre, néhány kedvező lehetőség rosszra fordulhat. Az a probléma, hogy csak létező egyedeket tudunk zárolni. Nem könnyű olyan adatbáziselemeket zárolni, amelyek nem léteznek, de később beszúrhatók. A következő példával világítjuk meg ezt az esetet.

9.22. példa: Tegyük fel, hogy ugyanaz a `Film` relációnk van, mint a 9.21. példában, és az első tranzakció, amelyet végrehajtunk a T_3, amely az alábbi lekérdezés:

```
SELECT SUM(hossz)
FROM Film
WHERE stúdióNév = 'Disney';
```


T_3-nak be kell olvasnia az összes Disney-filmről a sorokat, így azzal kezdődhet, hogy a relációt *IS* zárolja, és a Disney-filmekhez tartozó minden sort *S* zárolja[8].

Most egy T_4 tranzakció is megjelenik, és beszúr egy új Disney-filmet. Úgy tűnik, hogy a T_4-nek nincs szüksége zárolásokra, de a T_3 eredményét helytelenné változtatja. Ez a tény önmagában nem konkurenciaprobléma, ugyanis a (T_3, T_4) soros sorrend azzal ekvivalens, ami valójában történt. Lehetne még más *X* elem is, amelyet a T_3 és a T_4 is úgy ír, hogy a T_4 írja előbb, és így az összetettebb tranzakcióknak *lehetne* nem sorba rendezhető viselkedése.

Pontosabban kifejezve, tegyük fel, hogy D_1 és D_2 korábban létező Disney-filmek, és D_3 a T_4 által beszúrt új Disney-film. Legyen *L* a T_3 által kiszámolt Disney-filmek hosszainak az összege, és legyen az a konzisztenciamegszorítás az adatbázison, hogy az *L*-nek egyenlőnek kell lennie azon a Disney-filmek hosszának az összegével, amelyek léteztek, amikor az *L*-t utoljára kiszámoltuk. Ekkor a figyelmeztetési protokoll alatt jogszerű az alábbi eseménysorozat:

$r_3(D_1)$; $r_3(D_2)$; $w_4(D_3)$; $w_4(X)$; $w_3(L)$; $w_3(X)$;

Itt $w_4(D_3)$-et használtuk arra, hogy a T_4 tranzakció létrehozza a D_3-at. A fenti ütemezés nem sorba rendezhető. Ténylegesen az *L* értéke nem a D_1, D_2 és D_3 hosszának az összege, amelyek a jelenleg létező Disney-filmek. Továbbá az a tény, hogy *X* értékét a T_3 írta, és nem a T_4, kizárja azt a lehetőséget, hogy a T_3 a T_4 előtt következzen a feltételezett ekvivalens soros elrendezésben. □

A 9.22. példában az a probléma, hogy az új Disney-filmnek van egy *fantom* (phantom) sora, amelyet zárolni kellett volna, de nem tettük meg, ugyanis még nem létezett akkor, amikor a zárolásokat elvégeztük. Mégis van egy egyszerű út, hogy elkerüljük a fantomokat. A sorok beszúrását és törlését az egész relációra vonatkozó írásként kell tekintenünk. Így a T_4 tranzakciónak a 9.22. példában meg kell kapnia az *X* zárat a `Film` reláción. Minthogy a T_3 már *IS* módban zárolta ezt a relációt, és az a mód nem kompatibilis az *X* móddal, a T_4-nek várnia kell, amíg a T_3 befejeződik.

9.6.4. Feladatok

9.6.1. feladat: Tekintsünk a változatosság kedvéért egy objektumorientált adatbázist. A *C* osztály objektumait két blokkban tároljuk, a B_1-ben és a B_2-ben. A B_1 tartalmazza az O_1 és O_2 objektumokat, míg a B_2 tartalmazza az O_3, O_4 és O_5 objektumokat. Az osztálykiterjedések, a blokkok és az objektumok zárolható adatbáziselemekből álló hierarchiát alkotnak. Adjuk meg a zárolási kérések sorozatát és a figyelmeztető protokoll alapú ütemező feladatát az alábbi kérési sorozatokhoz. Feltehetjük, hogy minden kérés éppen azelőtt fordul elő, mint amikor szükségünk van rá, és minden zárfeloldás a tranzakció befejeztével történik.

[8] Ha viszont sok Disney-film lenne, akkor hatékonyabb lehetne csak egy *S* zárat kérni a teljes relációra.

* a) $r_1(O_1)$; $w_2(O_2)$; $r_2(O_3)$; $w_1(O_4)$;
b) $r_1(O_5)$; $w_2(O_5)$; $r_2(O_3)$; $w_1(O_4)$;
c) $r_1(O_1)$; $r_1(O_3)$; $r_2(O_1)$; $w_2(O_4)$; $w_2(O_5)$;
d) $r_1(O_1)$; $r_2(O_2)$; $r_3(O_1)$; $w_1(O_3)$; $w_2(O_4)$;$w_3(O_5)$; $w_1(O_2)$;

9.6.2. feladat: Változtassuk meg a 9.22. példában található eseménysorozatot úgy, hogy a $w_4(D_3)$ művelet a teljes `Film` relációnak a T_4 általi írása legyen. Ezután mutassunk be egy figyelmeztető protokoll alapú ütemező működést ezen a kérési sorozaton!

!! **9.6.3. feladat:** Mutassuk be, hogyan adjuk hozzá a növelési zárakat a figyelmeztető protokoll alapú ütemezőhöz!

9.7. Faprotokoll

Ebben a fejezetben az elemekből álló fákkal kapcsolatosan egy másik problémát tekintünk. A 9.6. részben a beágyazott szerkezetű adatbáziselemekkel létrehozott fákkal foglalkoztunk, amelyben a gyerekek a szülők részei voltak. Most maguknak az elemeknek a kapcsolati sémájából álló fastruktúrákkal foglalkozunk. Az adatbáziselemek diszjunkt adatdarabok, azonban csak egyféleképpen, a szülőkön keresztül lehet elérni egy csomópontot. A B-fák az ilyen típusú adatoknak fontos példái. Tudjuk, hogy csak egy bizonyos útvonalon jutunk el egy elemhez, és ez lényeges szabadságot ad nekünk abban, hogy a már látott kétfázisú zárolási megközelítéstől eltérő módon kezeljük a zárakat.

9.7.1. Fa alapú zárolások idítékai

Tekintsünk egy B-fa-indexet olyan rendszerben, amely az egyedi csomópontokat (vagyis blokkokat) zárolható adatbáziselemekként kezeli. A csomópont a zárolás szemcsézettségének a megfelelő szintje, ugyanis nem előnyös, ha kisebb darabokat kezelünk elemekként. Ha pedig a teljes B-fát kezeljük adatbáziselemként, akkor ez megakadályozza az index olyan konkurens használatát, mint amilyen elérhető a 9.7. rész tárgyát alkotó működési mechanizmus által.

Ha a zármódoknak egy szabványos halmazát használjuk, mint az osztott, kizárólagos és módosítási zárak, valamint használjuk a kétfázisú zárolást, akkor a B-fa konkurens használata szinte lehetetlen. Ennek az az oka, hogy az indexet használó minden tranzakciónak a B-fa gyökércsomópontját kell először zárolnia. Ha a tranzakció 2FZ, akkor nem lehet addig feloldani a gyökéren a zárolást, amíg meg nem szerezte az összes zárat, amelyre szüksége van, mind a B-fa-csomópontokon mind pedig más adatbáziselemeken.[9] Továbbá, mivel elvben bármely tranzakció, amely beszúrásokat vagy törléseket végez, a B-fa gyökerének az átírásával fejeződhet be, ily módon a tranzakciónak legalább egy módosítási zárolásra szüksége van a gyökércsomóponton, vagy kizárólagos zárra van szüksége, ha a módosítási mód nem elérhető. Így csak egyetlen nem csak olvasási tranzakció férhet hozzá bármikor a B-fához.

[9] Ezenkívül jó oka van annak, amiért a tranzakciók minden zárat addig tartanak, amíg készen nem állnak a véglegesítésre. Lásd a 10.1. részt.

Mégis az esetek többségében majdnem közvetlenül levezethetjük, hogy egy B-fa csomópontját nem kell átírni, még akkor sem, ha a tranzakció beszúr vagy töröl egy sort. Például, ha a tranzakció beszúr egy sort, de a gyökérnek az a gyereke, amelyhez hozzáférünk, nincs teljesen tele, akkor tudjuk, hogy a beszúrás nem kerül fel a gyökérig. Hasonlóan, ha a tranzakció egyetlen sort töröl, és a gyökérnek abban a gyerekében, amelyhez hozzáfértünk, a minimum számnál több kulcs és mutató van, akkor biztosak lehetünk, hogy a gyökér nem változik meg.

Így, amikor a tranzakció a gyökérnek egyik gyereke felé irányul, és észleli azt a (teljesen szokványos) helyzetet, ami kizárja a gyökér átírását, azonnal szeretnénk feloldani a gyökéren a zárat. Ugyanezt a megfigyelést alkalmazhatjuk a B-fa bármely belső csomópontjának a zárolására is, bár a konkurens B-fánál a legtöbb lehetőség abból származik, hogy a gyökéren a zárat korán oldjuk fel. Sajnos, a gyökéren levő zárolás korai feloldása ellentmond a 2FZ-nek, így nem lehetünk biztosak abban, hogy a B-fához hozzáférő több tranzakciónak az ütemezése sorba rendezhető lesz. A megoldás egy speciális protokoll a B-fákhoz hasonló fastruktúrájú adatokhoz hozzáférő tranzakciók részére. A protokoll ellentmond a 2FZ-nek, de azt a tényt használja, hogy az elemekhez való hozzáférés lefelé halad a fán, a sorbarendezhetőség biztosítása érdekében.

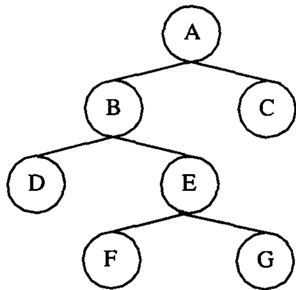
9.7.2. Faszerkezetű adatok hozzáférési szabályai

Az alábbi megszorítások a zárakon a *faprotokollt* (tree protocol) adják. Tételezzük fel, hogy csak egyféle zár van, amelyet az $l_i(X)$ alakú zárolási kérésekkel ábrázolunk, de ezt az ötletet bármely zárolási módokból álló halmazra általánosíthatjuk. Tételezzük fel, hogy a tranzakciók konzisztensek, az ütemezéseknek jogszerűnek kell lenniük (vagyis az ütemező csak akkor adja meg az elvárt megszorításokat a zárak engedélyezésével, amikor nincs konfliktusban azokkal a zárakkal, amelyek már a csomóponton vannak), és ugyanakkor nincs kétfázisú zárolási követelmény a tranzakciókon.

1. Egy tranzakciónak az első zárja a fa bármely csomópontján lehet.[10]
2. Rákövetkező zárakat csak akkor lehet szerezni, ha a tranzakciónak jelenleg van zárja a szülő csomóponton.
3. A csomópontok zárját bármikor feloldhatjuk.
4. Egy tranzakció nem zárolhatja újból azt a csomópontot, amelyen feloldotta a zárat, még akkor sem, ha még tartja a csomópont szülőjén a zárat.

9.23. példa: A 9.30. ábra a csomópontok hierarchiáját, míg a 9.31. ábra ezeken az adatokon három tranzakció műveleteit mutatja. T_1 az A gyökéren kezdődik, és lefelé folytatódik B, C és D felé. T_2 a B-n kezdődik, és az E felé próbál haladni, de először

[10] A 9.7.1. rész B-fa példájában az első zárnak mindig a gyökéren kell lennie.



9.30. ábra. *Zárolható elemekből álló fa*

T_1	T_2	T_3
$l_1(A)$; $r_1(A)$;		
$l_1(B)$; $r_1(B)$;		
$l_1(C)$; $r_1(C)$;		
$w_1(A)$; $u_1(A)$;		
$l_1(D)$; $r_1(D)$;		
$w_1(B)$; $u_1(B)$;		
	$l_2(B)$; $r_2(B)$;	
		$l_3(E)$; $r_3(E)$;
$w_1(D)$; $u_1(D)$;		
$w_1(C)$; $u_1(C)$;		
	$l_2(E)$; **Elutasítva**	
		$l_3(F)$; $r_3(F)$;
		$w_3(F)$; $u_3(F)$;
		$l_3(G)$; $r_3(G)$;
		$w_3(E)$; $u_3(E)$;
	$l_2(E)$; $r_2(E)$;	
		$w_3(G)$; $u_3(G)$;
	$w_2(B)$; $u_2(B)$;	
	$w_2(E)$; $u_2(E)$;	

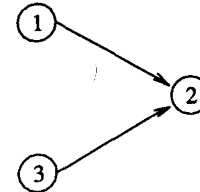
9.31. ábra. *A faprotokollt követő három tranzakció*

elutasítjuk, ugyanis már a T_3-nak van zárja az E-n. A T_3 tranzakció az E-n kezdődik, és folytatja az F-fel és G-vel. Megjegyezzük, hogy a T_1 nem 2FZ tranzakció, ugyanis az A-n előbb töröljük a zárat, mielőtt megszerezzük a zárat a D-n. Hasonlóan a T_3 sem 2FZ tranzakció, de a T_2 véletlenül éppen 2FZ. □

9.7.3. Miért működik a faprotokoll?

A faprotokoll az ütemezésben részt vevő tranzakciókon egy soros sorrendet kényszerít ki. A következőképpen definiálhatjuk a megelőzési sorrendet. Azt mondjuk, hogy $T_i <_S T_j$ az S ütemezésben, ha a T_i és T_j tranzakciók egyrészt közösen zárolnak egy csomópontot, másrészt a T_i zárolja a csomópontot először.

9.24. példa: A 9.31. ábra S ütemezésében a T_1 és T_2 közösen zárolják a B-t, és a T_1 zárolja először. Így $T_1 <_S T_2$. Azt találjuk még, hogy T_2 és T_3 közösen zárolják az E-t, és a T_3 zárolja először, így $T_3 <_S T_2$. A T_1 és T_3 között viszont nincs megelőzés, hiszen nincs olyan csomópont, amelyet közösen zárolnak. Így ezekből a megelőzési relációkból levezetett megelőzési gráf a 9.32. ábrán látható. □



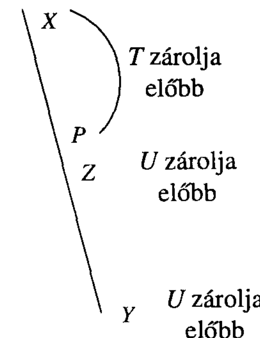
9.32. ábra. *A 9.31. ábra ütemezéséből származó megelőzési gráf*

Ha a fent definiált megelőzési relációkból rajzolt megelőzési gráf nem tartalmaz kört, akkor azt állítjuk, hogy a tranzakciók bármely topologikus sorrendje egy ekvivalens soros ütemezés. Például a 9.31. ábrához vagy a (T_1, T_3, T_2) vagy a (T_3, T_1, T_2) az ekvivalens soros ütemezés. Ennek az az oka, hogy az ilyen soros ütemezésben minden csomóponthoz ugyanabban a sorrendben nyúlnak a tranzakciók, mint az eredeti ütemezésben.

Ahhoz, hogy megértsük a fent leírt megelőzési gráfnak miért kell körmentesnek lennie, először vegyük észre a következőt:

- Ha két tranzakció közösen zárol néhány elemet, akkor ugyanabban a sorrendben zárolják mindegyiket.

Tekintsünk valamilyen T és U tranzakciókat, amelyek két vagy több elemet közösen zárolnak. Először, megjegyezzük, hogy mindegyik tranzakció faformájú halmazát zárolja az elemeknek, és a két fa metszete maga is fa. Emiatt van egy legmagasabb X



9.33. ábra. *Két tranzakció által közösen zárolt elemek útja*

elem, amelyet a T is és az U is zárol. Tételezzük fel, hogy T zárolja az X-et először, de van egy másik Y elem, amelyet az U előbb zárol, mint a T. Ekkor az elemekből álló fában van út az X-től az Y-ba, és a T-nek is és az U-nak is zárolnia kell minden elemet az út mentén, ugyanis egyik sem zárolhat úgy egy csomópontot, hogy ne lenne már ennek a szülőjén zárja.

Tekintsük az első elemet az út mentén, mondjuk legyen Z, amelyet az U zárol először, mint azt a 9.33. ábrán látjuk. Ekkor T előbb zárolja Z-nek a P szülőjét, mint az U. Ekkor viszont a T még mindig tartja a zárolást P-n, amikor zárolja Z-t, így U még nem zárolta P-t, amikor a Z-t zárolja. Az nem lehet, hogy Z lenne az első elem, amelyet az U a T-vel közösen zárol, mivel mindkettő zárolta az ősét, X-et (amely lehet a P is, csak a Z nem). Így az U addig nem zárolhatja a Z-t, amíg meg nem szerezte a P-n a zárat, amely azután van, hogy a T zárolta a Z-t. Arra következtetünk, hogy a T megelőzi az U-t minden csomópontban, amelyet közösen zárolnak.

Most tekintsük a $T_1, T_2, \ldots, T_n$ tranzakciók tetszőleges halmazát, amely eleget tesz a faprotokollnak, és az S ütemezésnek megfelelően zárolja a fa valamely csomópontjait. Először azok a tranzakciók, amelyek zárolják a gyökeret, ezt valamilyen sorrendben végzik, és olyan szabály alapján, amelyet éppen megfigyeltünk:

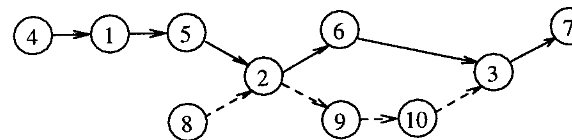
- Ha a T_i előbb zárolja a gyökeret, mint a T_j, akkor a T_i minden T_j-vel közös $T_i <_S T_j$ csomópontot előbb zárol, mint a T_j. Vagyis $T_i <_S T_j$, de nem $T_j <_S T_i$.

A fa csomópontjainak a száma szerinti teljes indukcióval megmutathatjuk, hogy a teljes tranzakcióhalmazhoz van az S-sel ekvivalens soros sorrend.

Alapeset: Ha csak egyetlen csomópont van, a gyökér, akkor ahogyan már megfigyeltük, a megfelelő sorrend az, ahogy a tranzakciók a gyökeret zárolják.

Indukció: Ha egynél több csomópont van a fában, tekintsük a gyökér mindegyik részfájához az olyan tranzakciókból álló halmazt, amelyek egy vagy több csomópontot zárolnak abban a részfában. Megjegyezzük, hogy a gyökeret zároló tranzakciók egy vagy több részfához tartozhatnak, de egy olyan tranzakció, amely nem zárolja a gyökeret, az csak egyetlen részfához tartozik. Például a 9.31. ábrán található tranzakciók közül csak a T_1 zárolja a gyökeret, és az mindkét részfához tartozik, a B gyökerű fához is és a C gyökerű fához is. A T_2 és a T_3 viszont csak a B gyökerű fához tartoznak.

Az indukciós feltevés szerint, van soros sorrend az összes olyan tranzakcióhoz, amelyek ugyanabban a tetszőleges részfában zárolnak csomópontokat. Csupán egybe kell olvasztanunk a különböző részfákhoz tartozó soros sorrendeket. Mivel a tranzakcióknak ezekben a listáiban csak azok a tranzakciók közösek, amelyek a gyökeret zároló tranzakciók, és megállapítottuk, hogy ezek a tranzakciók minden közös csomópontot ugyanabban a sorrendben zárolnak, ahogy a gyökeret zárolják, nem fordulhat elő két, a gyökeret zároló tranzakció különböző sorrendben két részlistán. Pontosabban, ha T_i és T_j előfordul a gyökér valamely C gyermekéhez tartozó listán, akkor ezek a C-t ugyanabban a sorrendben zárolják, ahogyan a gyökeret zárolják, és emiatt a listán is ebben a sorrend-



9.34. ábra. *A részfákhoz tartozó soros sorrendek egyesítése az összes tranzakcióhoz tartozó soros sorrendé*

ben fordulnak elő. Így felépíthetjük a soros sorrendet a teljes tranzakcióhalmazhoz azokból a tranzakciókból kiindulva, amelyek a gyökeret zárolják, a megfelelő sorrendjükben, és beleolvasztjuk azokat a tranzakciókat, amelyek nem zárolják a gyökeret, a részfáik soros sorrendjével konzisztens tetszőleges sorrendben.

9.25. példa: Tegyük fel, hogy van 10 darab tranzakció $T_1, T_2, \ldots, T_{10}$, és ezekből T_1, T_2 és T_3 ugyanabban a sorrendben zárolja a gyökeret. Tegyük fel azt is, hogy a gyökérnek van két gyereke, az elsőt a T_1-től a T_7-ig zárolják a tranzakciók, és a másodikat T_2, T_3, T_8, T_9 és T_{10} zárolja. Tegyük fel, hogy az első részfához a soros sorrend $(T_4, T_1, T_5, T_2, T_6, T_3, T_7)$. Megjegyezzük, hogy ennek a sorrendnek tartalmaznia kell T_1, T_2 és T_3-at ebben a sorrendben. Legyen továbbá a második részfához a soros sorrend $(T_8, T_2, T_9, T_{10}, T_3)$. Mint az előző esetben, a T_2 és T_3 tranzakciók, amelyek a gyökeret zárolják, abban a sorrendben fordulnak elő, ahogyan a gyökeret zárolták.

Ezeknek a tranzakcióknak a soros sorrendjére felállított megszorításokat a 9.34. ábrán mutatjuk be. A folyamatos vonalak a gyökér első gyerekének a rendezése szerinti megszorításokat jelölik, és a szaggatott vonalak pedig a második gyereknél levő rendezést jelölik. Ennek a gráfnak több topologikus sorrendjéből a $(T_4, T_8, T_1, T_5, T_2, T_9, T_6, T_{10}, T_3, T_7)$ az egyik. □

9.7.4. Feladatok

9.7.1. feladat: Tegyük fel, hogy végrehajtjuk az alábbi műveleteket a 4.23. ábrán szereplő B-fán. Ha a faprotokollt alkalmazzuk, mikor tudjuk feloldani az írási zárakat minden egyes megvizsgált csomóponton?

* a) Beszúrjuk a 10-et.
b) Beszúrjuk a 20-at.
c) Töröljük az 5-öt.
d) Töröljük a 23-at.

! **9.7.2. feladat:** Tekintsük az alábbi tranzakciókat, amelyek a 9.30. ábrán található fán működnek.

T_1: $r_1(A)$; $r_1(B)$; $r_1(E)$;
T_2: $r_2(A)$; $r_2(C)$; $r_2(B)$;
T_3: $r_3(B)$; $r_3(E)$; $r_3(F)$;

Válaszoljunk a következőkre:

* a) Hányféleképpen lehet T_1-et és T_2-t átlapolni, ha a faprotokollt követjük?
b) Hányféleképpen lehet T_1-et és T_3-t átlapolni, ha a faprotokollt követjük?
!! c) Hányféleképpen lehet mind a hármat átlapolni, ha a faprotokollt követjük?

! **9.7.3. feladat:** Tegyük fel, hogy van nyolc tranzakció $T_1, T_2, \ldots, T_8$, amelyekből a páratlan számú tranzakciók, T_1, T_3, T_5 és T_7, a fa gyökerét zárolják ebben a sorrendben. Három gyereke van a gyökérnek, az elsőt T_1, T_2, T_3 és T_4 zárolja ebben a sorrendben. A második gyereket T_3, T_6 és T_5 zárolja ebben a sorrendben, és a harmadik gyereket T_8 és T_7 zárolja ebben a sorrendben. A tranzakcióknak hány olyan soros sorrendje van, amely konzisztens ezekkel az állításokkal?

!! **9.7.4. feladat:** Tegyük fel, hogy az olvasáshoz, illetve íráshoz megfelelő osztott, illetve kizárólagos zárakkal rendelkező faprotokollt használjuk. A 2. szabályt, amely megköveteli, hogy zárolva legyen a szülő ahhoz, hogy a csomópontot zároljuk, meg kell változtatnunk, hogy megakadályozzuk a nem sorba rendezhető viselkedést. Mi az osztott és kizárólagos záraknak megfelelő helyes 2. szabály? *Útmutató:* ugyanolyan típusú zárolása szükséges-e a szülőknek, mint amilyen a gyereken van?

9.8. Konkurenciavezérlés időbélyegzőkkel

A következőkben a zárolástól különböző két másik módszert nézünk meg, amelyeket néhány rendszerben használnak a tranzakciók sorbarendezhetőségének biztosítására:

1. *Időbélyegzés* (timestamping). Minden tranzakcióhoz hozzárendelünk egy „időbélyegzőt", minden adatbáziselem utolsó olvasását és írását végző tranzakció időbélyegzőjét rögzítjük, és összehasonlítjuk ezeket az értékeket, hogy biztosítsuk, hogy a tranzakciók időbélyegzőinek megfelelő soros ütemezés ekvivalens legyen a tranzakciók aktuális ütemezésével. Ez a megközelítés lesz a jelenlegi rész témája.
2. *Érvényesítés* (validation). Megvizsgáljuk a tranzakciók időbélyegzőit és az adatbáziselemeket, amikor a tranzakció véglegesítésre kerül. Ezt az eljárást a tranzakciók „érvényesítésének" nevezzük. Az a soros ütemezés, amely az érvényesítési idejük alapján rendezi a tranzakciókat, ekvivalens kell hogy legyen az aktuális ütemezéssel. Az érvényesítési megközelítést a 9.9. részben tárgyaljuk.

Mindkét megközelítés *optimista* abban az értelemben, hogy feltételezik: nem fordul elő nem sorba rendezhető viselkedés, és csak akkor tisztázza a helyzetet, amikor a megszegés nyilvánvaló. Ezzel ellentétben, minden zárolási módszer azt feltételezi, hogy a dolgok rosszra fordulnak, hacsak a tranzakciókat azonnal meg nem akadályozzák a nem sorba rendezhető viselkedésbe kerülésben. Az optimista megközelítések abban különböznek a zárolásoktól, hogy az egyetlen ellenszerük, amikor valami rosszra fordul, hogy azt a tranzakciót, amely nem sorba rendezhető viselkedésbe próbált kerülni, abortáljuk (leállítjuk), és aztán újraindítjuk. A zárolási ütemezők ezzel ellentétben késleltetik a tranzakciókat, de nem abortálják őket.[11] Általában az optimista ütemezők akkor jobbak a zárolásinál, amikor sok tranzakció csak olvasási, ugyanis az ilyen tranzakciók önmagukban soha nem okozhatnak nem sorba rendezhető viselkedést.

9.8.1. Időbélyegzők

Annak érdekében, hogy az időbélyegzést konkurenciavezérlési módszerként használjuk, az ütemezőnek minden egyes T tranzakcióhoz hozzá kell rendelnie egy egyedi számot, a TS(T) *időbélyegzőt* (ahol TS az angol *timestamp* rövidítése). Az időbélyegzőket növekvő sorrendben kell kiadni abban az időpontban, amikor a tranzakció az elindításáról először értesíti az ütemezőt. Két megközelítés az időbélyegzők generálásához:

a) Az egyik lehetőség, hogy az időbélyegzőket a rendszeróra felhasználásával hozzuk létre, feltéve, hogy az ütemező nem működik annyira gyorsan, hogy két tranzakcióhoz ugyanazt az órapercegést rendelné időbélyegzőként.
b) A másik megközelítés szerint az ütemező karbantart egy számlálót. Minden alkalommal, amikor egy tranzakció elindul, a számláló növekszik 1-gyel, és ez az új érték lesz a tranzakció időbélyegzője. Ebben a megközelítésben az időbélyegzőknek semmi közük sincs az „idő"-höz, azonban azzal a bármely időbélyegző-generáló rendszer esetén szükséges fontos tulajdonsággal rendelkeznek, miszerint egy később elindított tranzakció nagyobb időbélyegzőt kap, mint egy korábban elindított tranzakció.

Bármelyik módszert is használjuk az időbélyegzők generálására, az ütemezőnek karban kell tartania a jelenleg aktív tranzakciók és időbélyegzőik tábláját.

Ahhoz, hogy időbélyegzőket használjunk konkurenciavezérlési módszerként, minden egyes X adatbáziselemhez hozzá kell kapcsolnunk két időbélyegzőt és egy további bitet:

1. RT(X), az X *olvasási ideje* (ahol RT az angol *read time* rövidítése), amely a legmagasabb időbélyegző, ami egy olyan tranzakcióhoz tartozik, amely már olvasta az X-et.
2. WT(X), az X *írási ideje* (ahol WT az angol *write time* rövidítése), amely a legmagasabb időbélyegző, ami egy olyan tranzakcióhoz tartozik, amely már írta X-et.
3. C(X), az X *véglegesítési bitje* (C az angol *commit bit* szóból származik), amely akkor és csak akkor igaz, ha a legújabb tranzakció, amely az X-et írta, már véglegesítve van. Ennek a bitnek az a célja, hogy elkerüljük azt a helyzetet, amelyben egy T tranzakció egy másik U tranzakció által írt adatokat olvas be, és utána az U-t

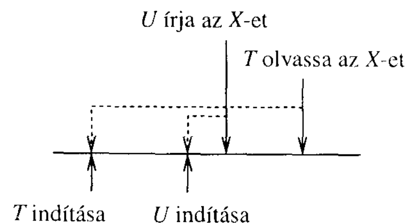
[11] Ez nem azt jelenti, hogy az a rendszer, amely zárolási ütemezőt használ, soha nem abortáltatja a tranzakciót. Például a 10.3. részben tárgyaljuk a holtpontok feloldását szolgáló tranzakcióabortáltatást. Egy zárolási ütemező viszont soha nem használ tranzakcióabortálást egyszerűen mint egy választ a zárolási kéréshez, amelyet nem lehet engedélyezni.

abortáljuk. Ez a probléma, amikor a *T* nem véglegesített adatok „piszkos olvasását" hajtja végre, bizonyosan az adatbázis-állapot inkonzisztensé válását is okozhatja. Így bármely ütemezőhöz szükség van olyan mechanizmusra, amely megakadályozza a piszkos olvasást.[12]

9.8.2. Fizikailag nem megvalósítható viselkedések

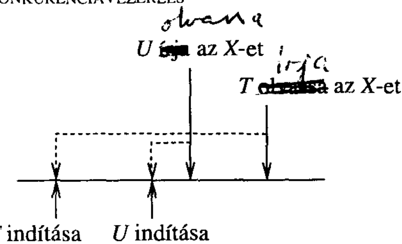
Azért, hogy megértsük az időbélyegzőn alapuló ütemező felépítését és szabályait, emlékeztetnünk kell arra, hogy az ütemező feltételezi, hogy a tranzakciók időbélyegző szerinti sorrendje egyúttal olyan soros sorrend, amely a végrehajtás sorrendjét is jelenti. Így az ütemező feladata azon túl, hogy hozzárendeli az időbélyegzőket a tranzakciókhoz, és módosítja az RT-t, WT-t és C-t az adatbáziselemek számára, még az is, hogy ellenőrzi, amikor egy olvasás vagy írás fordul elő, hogy az úgy történt *volna-e* a valós időben is, ha minden tranzakciót azonnal, az időbélyegző általi időpillanatban hajtottuk volna végre. Ha nem, akkor azt mondjuk, hogy a viselkedés *fizikailag nem megvalósítható*. Kétféle probléma merülhet fel:

1. *Túl késői olvasás:* A *T* tranzakció megpróbálja olvasni az *X* adatbáziselemet, de az *X* írási ideje azt jelzi, hogy az *X* jelenlegi értékét azután írtuk, miután a *T*-t már elméletileg végrehajtottuk. Vagyis TS(*T*) < WT(*X*). A 9.35. ábra mutatja ezt a problémát. A vízszintes tengely jelenti azt a valós időt, amikor az események előfordulnak. A pontozott vonalak kapcsolják össze az aktuális eseményt azzal az időponttal, amikor a tranzakciók időbélyegzője szerint elméletileg végre kellett volna hajtani az eseményt. Így látjuk, hogy az *U* tranzakciót a *T* tranzakció után indítottuk el, mégis az *X* értékét előbb írta, mielőtt a *T* beolvasta volna az *X*-et. *T*-nek nem az *U* által írt értéket kellene olvasnia, ugyanis elméletileg az *U*-t a *T* után hajtjuk végre. A *T*-nek viszont nincs más választása, ugyanis az *X*-nek az *U* által írt értéke az egyetlen, amelyet a *T* most be tud olvasni. A megoldás, hogy a *T*-t abortáljuk, amikor ez a probléma felmerül.
2. *Túl késői írás:* A *T* tranzakció megpróbálja írni az *X* adatbáziselemet, de az *X* olvasási ideje azt jelzi, hogy van egy másik tranzakció is, amelynek a *T* által beírt értéket kellene olvasnia, ám ehelyett más értéket olvas. Vagyis WT(*X*) < TS(*T*) < RT(*X*). A 9.36. ábra mutatja ezt a problémát, amely egy *U* tranzakciót mutat, amelyet a *T* után indítottunk el, mégis előbb olvassa az *X*-et, mint a *T*-nek lehetősége lett volna írnia az *X*-et. Amikor a *T* megpróbálja írni az *X*-et, úgy találjuk, hogy RT(*X*) > TS(*T*), ami azt jelenti, hogy az *U* tranzakció már beolvasta az *X*-et, amelyet elméletileg a *T*-nél később kellett volna elvégeznie. Valamint úgy találjuk, hogy WT(*X*) < TS(*T*), ami azt jelenti, hogy semelyik más tranzakció sem írta az *X*-et, amellyel felülírta volna a *T* általi értéket, így érvénytelenítette volna a *T* hatását, és olyan érték került volna az *X*-be, amelyet az *U* beolvashat.



9.35. ábra. *A T tranzakció túl késői olvasást próbál végezni*

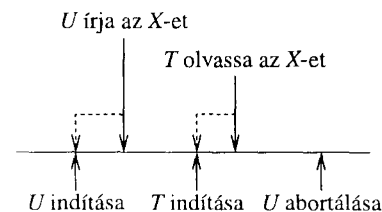
[12] Bár a piaci rendszerek általában a felhasználóra bízzák, hogy megengedhetők-e a piszkos olvasások.



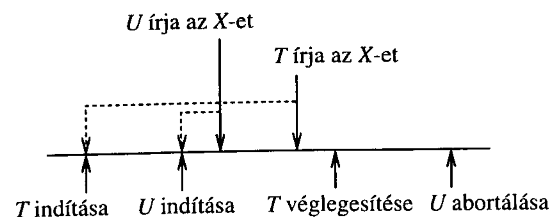
9.36. ábra. *A T tranzakció túl késői írást próbál végezni*

9.8.3. Piszkos adatok problémái

Van egy problémákból álló osztály, amelynek kezelésére bevezették a véglegesítési bitet. A problémák egyike a „piszkos olvasás", amelyet a 9.37. ábra szemléltet. Itt a *T* tranzakció olvassa az *X*-et, és ezen *X*-et utoljára az *U* írta. Az *U* időbélyegzője kisebb, mint a *T*-é, és a valóságban a *T* általi olvasás az *U* általi írás után történik, így az esemény úgy tűnik, hogy fizikailag megvalósítható. Mégis lehetséges, hogy miután a *T* beolvasta az *U* által írt *X*-beli értéket, az *U* tranzakciót abortáljuk. Esetleg az *U* talált hibás feltételt a saját adataiban, mint pl. 0-val való osztást, vagy mint ahogyan később látni fogjuk a 9.8.4. részben, az ütemező kényszeríti ki az *U* abortálását, ugyanis az valamilyen fizikailag nem megvalósítható viselkedést eredményező dolgot próbált végezni. Így, bár nincs fizikailag nem megvalósítható abban, hogy a *T* olvassa az *X*-et, mégis jobb a *T* olvasását akkorra elhalasztani, amikor az *U* véglegesítését vagy abortálását már elvégeztük. Meg tudjuk mondani, hogy az *U* még nincs véglegesítve, ugyanis a C(*X*) véglegesítési bit hamis lesz.



9.37. ábra. *A T tranzakció piszkos adat olvasást tud végezni, ha akkor olvassa az X-et, amikor az ábrán látható*



9.38. ábra. *Egy írást elhagyunk, ugyanis van egy későbbi időbélyegzővel ellátott írás, azonban az író tranzakció később abortál*

Egy másik lehetséges problémát a 9.38. ábra szemléltet. Itt az U, a T-nél későbbi időbélyegzőjű tranzakció írja először az X-et. Amikor a T írni próbál, a megfelelő műveletet semmit sem végez. Nyilvánvalóan nincs más V tranzakció, amelynek az X-ből a T általi értékét kellene beolvasnia, és ehelyett az U általi értéket olvasná, ugyanis ha a V megpróbálná olvasni az X-et, abortálnia kellene a túl késői olvasás miatt. A későbbi X olvasásoknál az U általi értéket kell olvasni, vagy az X még későbbi, de nem a T általi értékét. Ezt az ötletet, miszerint azokat az írásokat kihagyhatjuk, amelyeknél már elvégeztünk egy későbbi írási idejű írást, *Thomas-féle írási szabálynak* nevezzük.

A Thomas-féle írási szabállyal azonban van egy lényegi probléma. Ha az U-t később abortáljuk, amint az a 9.38. ábrán látható, akkor az U által írt X értéket ki kell törölnünk, továbbá az előző értéket és írási időt vissza kell állítanunk. Minthogy a T-t véglegesítettük, úgy látszik, hogy a T által írt X értéket kell a későbbi olvasáshoz használnunk. Mi viszont már kihagytuk a T általi írást, és már túl késő, hogy helyrehozhassuk ezt a hibát.

Sokféle módon lehet kezelni a most vázolt problémát, azonban egy viszonylag egyszerű elvet mutatunk be, amely az időbélyegzőn alapuló ütemezőre épül.

- Amikor a T tranzakció írja az X adatbáziselemet, az írás „kísérleti", és vissza lehet állítani, ha a T-t abortáljuk. A $\texttt{C}(X)$ véglegesítési bitet hamisra állítjuk, egyúttal az ütemező másolatot készít az X régi értékéről, és az előző $\texttt{WT}(X)$-ről.

9.8.4. Az időbélyegzőn alapuló ütemezések szabályai

Összegezhetjük azokat a szabályokat, amelyeket az időbélyegzőket használó ütemezőnek követnie kell ahhoz, hogy biztosan ne fordulhasson elő semmiféle fizikailag nem megvalósítható viselkedés. Az ütemezőnek egy T tranzakciótól érkező olvasási vagy írási kérésre adott válaszában az alábbi választásai lehetnek:

a) Engedélyezi a kérést.
b) Abortálja a T-t (ha a T megsérti a fizikai valóságot), és egy új időbélyegzővel újraindítja a T-t (azt az abortálást, amelyet újraindítás követ gyakran *visszagörgetésnek* vagy *rollbacknek* nevezzük).
c) Késlelteti a T-t, és később dönti el, hogy abortálja a T-t, vagy engedélyezi a kérést (ha a kérés olvasás és az olvasás piszkos is lehet, mint a 9.8.3. részben).

A szabályok a következők:

1. Tegyük fel, hogy az ütemezőhöz érkező kérés $r_T(X)$.

   a) Ha $\texttt{TS}(T) \geq \texttt{WT}(X)$, az olvasás fizikailag megvalósítható.

      i) Ha $\texttt{C}(X)$ igaz, engedélyezzük a kérést. Ha $\texttt{TS}(T) > \texttt{RT}(X)$, akkor $\texttt{RT}(X) := \texttt{TS}(T)$, egyébként nem változtatjuk meg $\texttt{RT}(X)$-t.
      ii) Ha $\texttt{C}(X)$ hamis, késleltessük a T-t addig, amíg $\texttt{C}(X)$ igazzá válik, vagy addig, amíg az X-et író tranzakció abortál.

   b) Ha $\texttt{TS}(T) < \texttt{WT}(X)$, az olvasás fizikailag nem megvalósítható. Visszagörgetjük a T-t, vagyis abortáljuk T-t, és újraindítjuk egy új, nagyobb időbélyegzővel.

2. Tegyük fel, hogy az ütemezőhöz érkező kérés $w_T(X)$.

   a) Ha $\texttt{TS}(T) \geq \texttt{RT}(X)$ és $\texttt{TS}(T) \geq \texttt{WT}(X)$, az írás fizikailag megvalósítható, és az alábbiakat kell végrehajtani:

      i) Írjuk be az új X értéket.
      ii) Állítsuk be $\texttt{WT}(X) := \texttt{TS}(T)$.
      iii) Állítsuk be $\texttt{C}(X) := \texttt{hamis}$.

   b) Ha $\texttt{TS}(T) \geq \texttt{RT}(X)$, de $\texttt{TS}(T) < \texttt{WT}(X)$, akkor az írás fizikailag megvalósítható, de az X-nek már egy későbbi értéke van. Ha $\texttt{C}(X)$ igaz, az X előző írását végző tranzakció véglegesítve van, és egyszerűen figyelmen kívül hagyjuk a T írását, megengedjük, hogy a T folytatódjon, és ne változtassa meg az adatbázist. Ha viszont a $\texttt{C}(X)$ hamis, akkor késleltetnünk kell a T-t, mégpedig az 1a)ii) pontban leírtak szerint.
   c) Ha $\texttt{TS}(T) < \texttt{RT}(X)$, az írás fizikailag nem megvalósítható, és a T-t vissza kell görgetnünk.

3. Tegyük fel, hogy az ütemezőhöz érkező kérés a T véglegesítése. Meg kell találnunk (az ütemező karbantartási listája alapján) az összes olyan X adatbáziselemet, amelybe a T írt, és állítsuk be $\texttt{C}(X)$-et $\texttt{igaz}$-ra. Ha vannak az X véglegesítésére várakozó tranzakciók (az ütemező egy másik karbantartási listáján találjuk meg), ezeknek a tranzakcióknak megengedjük, hogy folytatódjanak.
4. Tegyük fel, hogy az ütemezőhöz érkező kérés a T abortálása, vagy a T visszagörgetésére való döntés, mint az 1.b) vagy 2.c) esetekben. Ekkor bármely olyan tranzakcióra, amely egy X elem T általi írására várakozott, meg kell ismételnünk ezt az olvasási vagy írási kísérletet, és meglátjuk, hogy a művelet most jogszerű-e, miután az abortált tranzakció írásait visszavontuk.

9.26. példa: A 9.39. ábrán három tranzakció T_1, T_2 és T_3 ütemezése látható, amelyek három A, B és C adatbáziselemhez férnek hozzá. Az események előfordulásának valós

T_1	T_2	T_3	A	B	C
200	150	175	RT = 0 WT = 0	RT = 0 WT = 0	RT = 0 WT = 0
$r_1(B)$;				RT = 200	
	$r_2(A)$;		RT = 150		
		$r_3(C)$;			RT = 175
$w_1(B)$;				WT = 200	
$w_1(A)$;			WT = 200		
	$w_2(C)$; **Abortál**;				
		$w_3(A)$;			

9.39. ábra. *Három tranzakció időbélyegzőn alapuló ütemező alatti végrehajtása*

ideje a szokás szerint a lapon lefelé nő. Most azonban a tranzakciók időbélyegzői és az elemek olvasási és írási ideje is jelölve vannak. Tegyük fel, hogy kezdetben minden adatbáziselemhez az olvasási és az írási idő is 0. A tranzakciók abban a pillanatban kapnak időbélyegzőt, amikor értesítik az ütemezőt az elindításukról. Megjegyezzük, hogy bár a T_1 hajtja végre az első adathozzáférést, mégsem neki van a legkisebb időbélyegzője. Tegyük fel, hogy T_2 az első, amelyik az indításáról értesíti az ütemezőt, és a T_3 volt a következő, és T_1-et indítottuk el utoljára.

Az első műveletben a T_1 beolvassa a B-t. Mivel a B írási ideje kisebb, mint a T_1 időbélyegzője, ez az olvasás fizikailag megvalósítható, és engedélyezzük a végrehajtást. B olvasási idejét 200-ra állítjuk, a T_1 időbélyegzőjére. A második és a harmadik olvasási művelet hasonlóan jogszerű, és mindegyik adatbáziselem olvasási idejének értékét az őt olvasó tranzakció időbélyegzőjére állítjuk.

A negyedik lépésben T_1 írja a B-t. Mivel a B olvasási ideje nem nagyobb, mint a T_1 időbélyegzője, az írás fizikailag megvalósítható. Mivel a B írási ideje nem nagyobb, mint a T_1 időbélyegzője, ténylegesen végre kell hajtanunk az írást. Amikor ezt elvégezzük, a B írási idejét 200-ra növeljük, amely az őt felülíró T_1 tranzakció időbélyegzője.

Ezután T_2 megpróbálja írni a C-t. C-t viszont már beolvasta a T_3 tranzakció, amelyet elméletileg a 175-ös időpontban hajtottunk végre, míg a T_2-nek az értéket a 150-es időpontban kellett volna beírnia. Így a T_2 olyan dologgal próbálkozik, amely fizikailag nem megvalósítható viselkedést eredményezne, és a T_2-t vissza kell görgetnünk.

Az utolsó lépés, hogy a T_3 írja az A-t. Mivel az A írási ideje 150, kevesebb, mint a T_3 időbélyegzője, ami 175, az írás jogszerű. Viszont az A-nak már egy későbbi értéke van tárolva ebben az adatbáziselemben, mégpedig a T_1 által beírt érték, elméletileg a 200-as időpontban. Így a T_3-at nem görgetjük vissza, de be sem írjuk az értékét. □

9.8.5. Többváltozatú időbélyegzők

Az időbélyegzés egyik fontos változata karbantartja az adatbáziselemek régi változatait is, az adatbázisban magában tárolt jelenlegi változaton kívül. A cél az, hogy megengedjünk olyan $r_T(X)$ olvasásokat, amelyek egyébként a T tranzakció abortálását

okoznák (ugyanis az X jelenlegi változatát egy T-nél későbbi írta felül) úgy, hogy az X-nek a T időbélyegzőjű tranzakcióhoz megfelelő régebbi változatának a beolvasásával folytatjuk T-t. A módszer különösen hasznos, ha az adatbáziselemek lemezblokkok vagy lapok, ugyanis ekkor csak annyit kell a pufferkezelőnek tennie, hogy bizonyos blokkok a memóriában legyenek, amelyek néhány jelenleg aktív tranzakció számára hasznosak lehetnek.

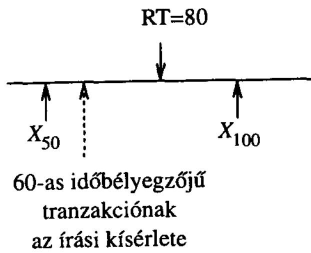
9.27. példa: Tekintsük a 9.40. ábrán szereplő, az A adatbáziselemhez hozzáférő tranzakciók halmazát. Ezek a tranzakciók egy közönséges időbélyegzőn alapuló ütemező alatt működnek, és amikor a T_3 megpróbálja az A-t olvasni, azt találja, hogy a WT(A) nagyobb, mint a saját időbélyegzője, és abortálni kell. Viszont megvan az A-nak a T_1 által írt, és a T_2 által felülírt régi értéke, amely alkalmas lenne a T_3-nak, hogy olvassa. Ebben a változatban az A-nak 150-es volt az írási ideje, ami kevesebb, mint a T_3 175-ös időbélyegzője. Ha az A-nak ez a régi értéke hozzáférhető lenne, a T_3 engedélyt kaphatna az olvasásra, még ha ez az A-nak nem is a „jelenlegi" értéke. □

T_1	T_2	T_3	T_4	A
150	200	175	225	RT = 0 WT = 0
$r_1(A)$;				RT = 150
$w_1(A)$;				WT = 150
	$r_2(A)$;			RT = 200
	$w_2(A)$;			WT = 200
		$r_3(A)$; **Abortál;**		
			$r_4(A)$;	RT = 225

9.40. ábra. *A T_3-at abortálnunk kell, ugyanis nem tud hozzáférni az A régi értékéhez*

A többváltozatú időbélyegzés ütemező az alábbiakban különbözik az 9.8.4. részben leírt ütemezőtől:

1. Amikor egy új $w_T(X)$ írás fordul elő, ha ez jogszerű, akkor az X adatbáziselemnek egy új változatát hozzuk létre. Az írási ideje TS(T), és X_t-vel fogunk rá hivatkozni, ahol t = TS(T).
2. Amikor egy $r_T(X)$ olvasás fordul elő, az ütemező megkeresi az X-nek azt az X_t változatát, amelyre $t \leq$ TS(T), de nincs más $X_{t'}$ változata, amelyre $t < t' \leq$ TS(T) lenne. Vagyis az X-nek azt a változatát, amelyet a T elméleti végrehajtása előtt közvetlenül írtak, olvassa be a T.
3. Az írási időket egy elem *változataihoz* kapcsoljuk, és soha nem változtatjuk meg.
4. Az olvasási időket is a változatokhoz kapcsoljuk. Arra használjuk őket, hogy visszautasítsunk bizonyos írásokat, mégpedig azokat, amelyek ideje kisebb, mint az előző verzió olvasási ideje. A 9.41. ábrán mutatjuk be ezt a problémát, ahol az X változatai az X_{50} és az X_{100}, a korábbi a 80-as időpontban olvasásra került, és megjelent a 60-as időbélyegzőjű T tranzakció általi új írás. Ez az írás a T abortálá-



9.41. ábra. *Egy tranzakció az X egyik változatát próbálja írni, amely az eseményt fizikailag megvalósíthatatlanná tenné*

sát kell hogy okozza, ugyanis az X-beli értékét a 80-as időbélyegzőjű tranzakciónak kellett volna olvasnia, ha T végrehajtását engedtük volna.

5. Amikor egy X_t változat t írási ideje olyan, hogy nincs a t-nél kisebb időbélyegzőjű aktív tranzakció, akkor törölhetjük az X-nek az X_t-t *megelőző* változatait.

9.28. példa: Tekintsük újból a 9.40. ábrán szereplő műveleteket, amikor többváltozatú időbélyegzést használunk. Először, az A-nak három változata létezik: A_0, amelyik a tranzakciók elindítása előtt létezik, A_{150}, amelyet a T_1 írt, és A_{200}, amelyet a T_2 írt. A 9.42. ábra mutatja azt az eseménysorozatot, amikor a változatokat létrehozzuk, és amikor ezeket beolvassuk. Megjegyezzük, hogy T_3-at nem kell abortálni, ugyanis be tudja olvasni az A-nak egy korábbi változatát. □

T_1	T_2	T_3	T_4	A_0	A_{150}	A_{200}
150	200	175	225			
$r_1(A)$;				Olvasás		
$w_1(A)$;					Létrehozás	
	$r_2(A)$;				Olvasás	
	$w_2(A)$;					Létrehozás
		$r_3(A)$;			Olvasás	
			$r_4(A)$;			Olvasás

9.42. ábra. *Többváltozatú konkurenciavezérlést alkalmazó tranzakciók végrehajtása*

9.8.6. Az időbélyegzők és zárolások

Általában az időbélyegzés azokban a helyzetekben kiváló, amikor a tranzakciók többsége csak olvasási, vagy ritka az az eset, hogy konkurens tranzakciók ugyanazt az elemet próbálják meg olvasni és írni. Az erősen konfliktusos helyzetekben jobb a zárolásokat használni. Ehhez az ökölszabályhoz az érvek az alábbiak:

- A zárolások gyakran késleltetik a tranzakciókat azzal, hogy a zárakra várnak, és még holtpontok is kialakulhatnak, amikor néhány tranzakció hosszú ideje várakozik, és ezután az egyiket vissza kell görgetni.

- Ha viszont a konkurens tranzakciók gyakran olvasnak és írnak közös elemeket, akkor a visszagörgetés lesz gyakori, ami még több késedelmet okoz, mint egy zárolási rendszer.

Több piaci rendszer érdekes kompromisszumot alkalmaz. Az ütemező felosztja a tranzakciókat csak olvasási tranzakciókra és olvasási/írási tranzakciókra. Az olvasási/írási tranzakciókat kétfázisú zárolást használva hajtjuk végre úgy, hogy a zárolt elemek hozzáférését megakadályozzuk a többi és a csak olvasási tranzakciók esetén is.

A csak olvasási tranzakciókat a többváltozatú időbélyegzéssel hajtjuk végre. Amikor az olvasási/írási tranzakciók létrehozzák egy adatbáziselem új változatait, ezeket a változatokat úgy kezeljük, ahogyan a 9.8.5. részben leírtuk. Csak olvasási tranzakciónak megengedjük, hogy egy adatbáziselem bármelyik változatát olvassa, amelyik megfelel az időbélyegzőjének. Csak olvasási tranzakciót emiatt soha nem kell abortálnunk, és csak nagyon ritkán kell késleltetnünk.

9.8.7. Feladatok

9.8.1. feladat: Az alábbiakban több eseménysorozatot találunk, beleértve az *indítási* eseményeket is, ahol az st_i (az angol *start* rövidítéssel) azt jelenti, hogy a T_i tranzakciót elindítottuk. Ezek a sorozatok valós időt jelentenek, és az időbélyegzőn alapuló ütemező a tranzakciókhoz az időbélyegzőket az indítási sorrendjük szerint adja. Mondjuk meg, hogy mi történik, amikor ezeket végrehajtjuk.

* a) st_1; st_2; $r_1(A)$; $r_2(B)$; $w_2(A)$; $w_1(B)$;
b) st_1; $r_1(A)$; st_2; $w_2(B)$; $r_2(A)$; $w_1(B)$;
c) st_1; st_2; st_3; $r_1(A)$; $r_2(B)$; $w_1(C)$; $r_3(B)$; $r_3(C)$; $w_2(B)$; $w_3(A)$;
d) st_1; st_3; st_2; $r_1(A)$; $r_2(B)$; $w_1(C)$; $r_3(B)$; $r_3(C)$; $w_2(B)$; $w_3(A)$;

9.8.2. feladat: Mondjuk meg, hogy mi történik az alábbi eseménysorozatok folyamán, ha többváltozatú, időbélyegzőn alapuló ütemezőt használunk. Mi történik helyette, ha az ütemező nem támogat többszörös többváltozatokat?

* a) st_1; st_2; st_3; st_4; $w_1(A)$; $w_2(A)$; $w_3(A)$; $r_2(A)$; $r_4(A)$;
b) st_1; st_2; st_3; st_4; $w_1(A)$; $w_3(A)$; $r_4(A)$; $r_2(A)$;
c) st_1; st_2; st_3; st_4; $w_1(A)$; $w_4(A)$; $r_3(A)$; $w_2(A)$;

!! **9.8.3. feladat:** Észrevettük a zároláson alapuló ütemezők tanulmányozásában, hogy számos okból alakulhatnak ki holtpontok a zárakat elnyert tranzakciók esetén. A $C(X)$ véglegesítési bitet használó időbélyegzőn alapuló ütemező esetén kialakulhat-e holtpont?

9.9. Konkurenciavezérlés érvényesítéssel

Az *érvényesítés* (validation) az optimista konkurenciavezérlés másik típusa, amelyben a tranzakcióknak megengedjük, hogy zárolások nélkül hozzáférjenek az adatokhoz, és a megfelelő időben ellenőrizzük a tranzakció sorba rendezhető viselkedését. Az érvényesítés alapvetően abban különbözik az időbélyegzőtől, hogy itt az ütemező egy nyilvántartást vezet arról, mit tesznek az aktív tranzakciók ahelyett, hogy az összes adatbáziselemhez feljegyezné az olvasási és írási időt. Mielőtt a tranzakció írni kezdene értékeket az adatbáziselemekbe, egy „érvényesítési fázison" megy keresztül, ahol a beolvasott és írandó elemek halmazait összehasonlítjuk más aktív tranzakciók írásai halmazaival. Ha a fizikailag nem megvalósítható viselkedés kockázata lépne fel, akkor a tranzakciót visszagörgetjük.

9.9.1. Érvényesítésen alapuló ütemező felépítése

Amikor az érvényesítést használjuk konkurenciavezérlési működésként, az ütemezőnek meg kell adnunk minden *T* tranzakcióhoz a *T* által olvasott adatbáziselemek halmazát, és a *T* által írt elemek halmazát. Ezek a halmazok az RS(*T*) *olvasási halmaz* (ahol *RS* az angol *read set* rövidítése), és a WS(*T*) *írási halmaz* (ahol *WS* az angol *write set* rövidítése). A tranzakciókat három fázisban hajtjuk végre:

1. *Olvasás.* Az első fázisban a tranzakciók beolvassák az adatbázisból az összes elemet az olvasási halmazba. A tranzakció ki is számítja a lokális címhelyen az összes eredményt, amelyet be fog írni.
2. *Érvényesítés.* A második fázisban, az ütemező érvényesíti a tranzakciót oly módon, hogy összehasonlítja az olvasási és írási halmazait a többi tranzakcióéval. Az érvényesítési eljárást a 9.9.2. részben fogjuk leírni. Ha az érvényesítés hibát jelez, akkor a tranzakciót visszagörgetjük, egyébként pedig folytatódik a harmadik fázissal.
3. *Írás.* A harmadik fázisban a tranzakció az írási halmazában levő elemek értékeit beírja az adatbázisba.

Intuitív alapon minden sikeresen érvényesített tranzakcióról azt gondolhatjuk, hogy az érvényesítés pillanatában került végrehajtásra. Így az érvényesítésen alapuló ütemező a tranzakciók feltételezett soros sorrendjével dolgozik. Az alapja annak a döntésnek, hogy érvényesítsen-e vagy sem egy tranzakciót az, hogy a tranzakciók viselkedése konzisztens legyen ezzel a soros sorrenddel.

Ahhoz a döntéshez, hogy az ütemező érvényesítheti-e a tranzakciót, fenntart három halmazt:

1. *KEZD*, a már elindított, de még nem teljesen érvényesített tranzakciók halmaza. Ebben a halmazban minden *T* tranzakcióhoz az ütemező karbantartja a KEZD(*T*)-t a *T* indításának időpontjában.
2. *ÉRV*, a már érvényesített, de a 3. fázisban az írásokat még nem befejezett tranzakciók halmaza. Ebben a halmazban minden *T* tranzakcióhoz az ütemező karbantartja a KEZD(*T*)-t és az ÉRV(*T*)-t a *T* érvényesítésekor. Megjegyezzük, hogy ÉRV(*T*) az az idő, amikor a *T* végrehajtását gondoljuk a végrehajtás feltételezett soros sorrendjében.
3. *BEF*, a 3. fázist befejezett tranzakciók halmaza. Ezekhez a *T* tranzakciókhoz az ütemező rögzíti a KEZD(*T*)-t, az ÉRV(*T*)-t és a BEF(*T*)-t a *T* befejezésekor. Elméletben ez a halmaz nő, de amint látni fogjuk, nem kell megjegyeznünk a *T* tranzakciót, ha BEF(*T*) < KEZD(*U*) bármely *U* aktív tranzakcióra (vagyis bármely *U*-ra a *KEZD*-ben vagy az *ÉRV*-ben). Az ütemező így időnként tisztogathatja a *BEF* halmazt, hogy megakadályozza a méretének a korlátlan növekedését.

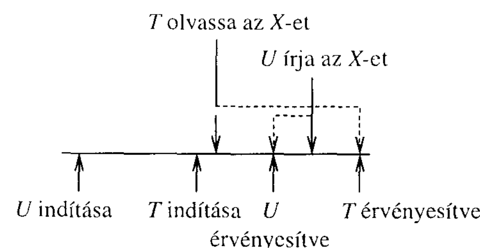
9.9.2. Az érvényesítési szabályok

Ha az ütemező elvégzi a karbantartását, akkor a 9.9.1. részben leírt információ elég ahhoz, hogy észlelje a tranzakciók feltételezett soros sorrendjének (a tranzakciók érvényesítési sorrendjének) bármely lehetséges megsértését. A szabályok megértése szempontjából először vizsgáljuk meg, hogy mi lehet hibás, amikor megpróbáljuk a *T* tranzakciót érvényesíteni.

1. Tegyük fel, hogy van olyan *U* tranzakció, hogy:

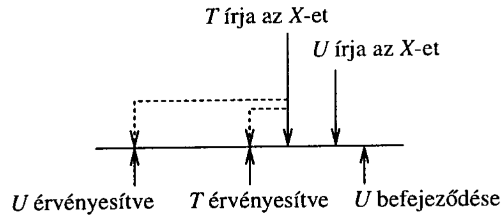
   a) *U* az *ÉRV*-ben vagy a *BEF*-ben van, vagyis az *U*-t már érvényesítettük.
   b) BEF(*U*) > KEZD(*T*), vagyis az *U* nem fejeződött be a *T* indítása előtt.[13]
   c) RS(*T*) ∩ WS(*U*) nem üres, legyen *X* a metszetben levő adatbáziselem.

   Ekkor lehetséges, hogy az *U* azután írja az *X*-et, miután a *T* olvassa az *X*-et. Tulajdonképpen lehet, hogy az *U* még nem írta az *X*-et. Az az eset, amikor az *U* nem időben írta az *X*-et, a 9.43. ábrán látható. Az ábrához magyarázatként megjegyez-



9.43. ábra. *A T-t nem érvényesíthetjük, ha egy korábbi tranzakció most ír valamit, amelyet a T-nek olvasnia kellett volna*

[13] Megjegyezzük, hogy ha *U* az *ÉRV*-ben van, akkor az *U* még nem fejeződött be a *T* érvényesítésekor. Ebben az esetben a BEF(*U*) technikailag nem definiált. Viszont tudjuk, hogy a KEZD(*T*)-nél nagyobbnak kell lennie ebben az esetben.



9.44. ábra. *A T tranzakció nem érvényesíthető akkor, ha egy korábbi tranzakció előtt tudna írni*

zük, hogy a pontozott vonalak kapcsolják össze a valós idejű eseményeket azzal az idővel, amikor elő kellett volna fordulniuk, ha a tranzakciókat az érvényesítés pillanatában hajtottuk volna végre. Mivel nem tudjuk, hogy a *T*-nek be kell-e olvasnia az *U*-tól származó értékét vagy sem, vissza kell görgetnünk *T*-t, hogy elkerüljük annak a kockázatát, hogy a *T* és az *U* műveletei nem lesznek konzisztensek a feltételezett soros sorrenddel.

2. Tegyük fel, hogy van olyan *U* tranzakció, amelyre:

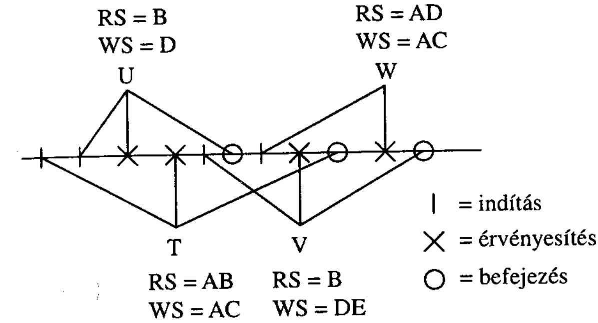
   a) *U* az *ÉRV*-ben van; vagyis az *U*-t már sikeresen érvényesítettük.
   b) BEF(*U*) > ÉRV(*T*); vagyis az *U*-t nem fejeztük be mielőtt a *T* az érvényesítési fázisába lépett volna.
   c) WS(*T*) ∩ WS(*U*) ≠ ∅ legyen *X* mind a két írási halmazban.

Ekkor a lehetséges problémát a 9.44. ábra mutatja. Mind a *T*-nek, mind az *U*-nak írnia kell az *X* értékét, és ha megengedjük a *T* érvényesítését, lehetséges, hogy az *U* előtt fogja írni az *X*-et. Mivel nem lehetünk biztosak a dolgunkban, visszagörgetjük a *T*-t, hogy biztosan ne szegjük meg azt a feltételezett soros sorrendet, amelyben *T*-t követi az *U*.

A fent leírt két problémával kerülhetünk csak olyan helyzetbe, amikor a *T* által végzett írás fizikailag nem megvalósítható. A 9.43. ábrán, ha az *U* a *T* elindítása előtt fejeződött volna be, akkor a *T* biztosan olyan *X* értéket olvasna, amelyet vagy *U* vagy valamely későbbi tranzakció írt. A 9.44. ábrán, ha az *U* a *T* érvényesítése előtt fejeződik be, akkor biztos, hogy az *U* a *T* előtt írta az *X*-et. Így a *T* tranzakció érvényesítésére vonatkozó észrevételeinket az alábbi szabállyal foglalhatjuk össze:

- Összehasonlítjuk az RS(*T*)-t a WS(*U*)-val, és ellenőrizzük, hogy RS(*T*) ∩ WS(*U*) = = ∅, minden olyan *U*-ra, amely még nem fejeződött be a *T* elindítása előtt, vagyis, ha BEF(*U*) > KEZD(*T*), és az *U* egy korábban érvényesített tranzakció.
- Összehasonlítjuk a WS(*T*)-t a WS(*U*)-val, és ellenőrizzük, hogy WS(*T*) ∩ WS(*U*) = ∅, minden olyan *U*-ra, amely még nem fejeződött be a *T*-t érvényesítése előtt, vagyis, ha BEF(*U*) > ÉRV(*T*), és az *U* egy korábban érvényesített tranzakció.

9.29. példa: A 9.45. ábra egy idővonalat ábrázol, amely mentén *T*, *U*, *V* és *W* négy tranzakció végrehajtási és érvényesítési kísérletei láthatók. Az ábrán jelöltük mindegyik tranzakció olvasási és írási halmazait. A *T*-t indítjuk el elsőnek, de az *U*-t érvényesítjük elsőnek.



9.45. ábra. *Négy tranzakció és az érvényesítéseik*

1. Az *U* érvényesítése: amikor az *U*-t érvényesítjük, nincs más érvényesített tranzakció, és így nem kell semmit sem ellenőriznünk. Az *U*-t sikeresen érvényesítjük, és értéket írunk a *D* adatbáziselembe.
2. A *T* érvényesítése: amikor a *T*-t érvényesítjük, az *U* már érvényesítve van, de még nincs befejezve. Így ellenőriznünk kell, hogy a *T*-nek sem az olvasási, sem az írási halmazában nincs semmi közös a WS(*U*) = {*D*}-vel. Mivel RS(*T*) = {*A*, *B*}, és WS(*T*) = = {*A*, *C*}, mindkét ellenőrzés sikeres, ily módon a *T*-t érvényesítjük.
3. A *V* érvényesítése: amikor a *V*-t érvényesítjük, az *U* már érvényesítve van, és befejeződött, a *T* már érvényesítve van, de még nem fejeződött be. Továbbá a *V*-t az *U* befejeződése előtt indítjuk el. Így össze kell hasonlítanunk mind az RS(*V*)-t, mind a WS(*V*)-t a WS(*T*)-vel, azonban csak az RS(*V*)-t kell összehasonlítanunk a WS(*U*)-val. Azt találjuk, hogy:

   - RS(*V*) ∩ WS(*T*) = {*B*} ∩ {*A*, *C*} = ∅.
   - WS(*V*) ∩ WS(*T*) = {*D*, *E*} ∩ {*A*, *C*} = ∅.
   - RS(*V*) ∩ WS(*U*) = {*B*} ∩ {*D*} = ∅.

   Így a *V*-t sikeresen érvényesítjük.

4. A *W* érvényesítése: Amikor a *W*-t érvényesítjük, azt tapasztaljuk, hogy az *U* a *W* elindítása előtt befejeződött, és így nem kell elvégeznünk a *W* és *U* összehasonlítását. A *T* a *W* érvényesítése előtt fejeződött be, de nem fejeződött be a *W* elindítása előtt, ezért csak az RS(*W*)-t kell összehasonlítanunk a WS(*T*)-vel. A *V* már érvényesítve van, de még nem fejeződött be, így össze kell hasonlítanunk mind az RS(*W*)-t, mind a WS(*W*)-t a WS(*T*)-vel. Ezek az ellenőrzések:

   - RS(*W*) ∩ WS(*T*) = {*A*, *D*} ∩ {*A*, *C*} = {*A*}.
   - RS(*W*) ∩ WS(*V*) = {*A*, *D*} ∩ {*D*, *E*} = {*D*}.
   - WS(*W*) ∩ WS(*V*) = {*A*, *C*} ∩ {*D*, *E*} = ∅.

Mivel a metszetek nem mind üresek, a *W*-t nem érvényesítjük, hanem a *W*-t visszagörgetjük, és így nem ír értéket sem az *A*-ba, sem a *C*-be.

□

Csak egy pillanat

Úgy gondoljuk, hogy az érvényesítés egy pillanat alatt vagy észrevétlenül rövid idő alatt játszódik le. Például úgy képzeljük, hogy el tudjuk dönteni, hogy egy *U* tranzakció már érvényesített-e akkor, amikor a *T* tranzakció érvényesítése elindult. Előfordulhat-e, hogy az *U* érvényesítése a *T*-t érvényesítése alatt fejeződik be?

Ha egyprocesszoros rendszeren futtatunk, és csak egy ütemező végzi a feldolgozást, akkor valóban azt gondolhatjuk az érvényesítésről és az ütemező többi tevékenységéről, hogy egy pillanat alatt bekövetkeznek. Azért, mert ha az ütemező a *T*-t érvényesíti, akkor nem érvényesítheti ezalatt az *U*-t is, így a *T* érvényesítése alatt az *U* érvényesítési állapota sem változhat meg.

Ha többprocesszoros rendszer alatt futtatunk, és több ütemező végzi a feldolgozást, akkor lehet, hogy az egyik a *T*-t érvényesíti, mialatt a másik az *U*-t érvényesíti. Ha így van, akkor a többprocesszoros rendszer olyan szinkronizációs működésére kell támaszkodnunk, amely biztosítja, hogy az érvényesítést atomi tevékenységként végezzük el.

9.9.3. Három konkurenciavezérlés működésének összehasonlítása

A sorbarendezhetőséghez három megközelítést néztünk meg – a zárolásokat, az időbélyegzőket és az érvényesítést – mindegyiknek megvannak az előnyei. Először hasonlítsuk őket össze a tár felhasználása szempontjából:

- *Zárak:* A zártábla által lefoglalt tár a zárolt adatbáziselemek számával arányos.
- *Időbélyegzők:* Egy naiv megvalósításban minden adatbáziselemhez akár hozzáférünk jelenleg, akár nem, az olvasási és írási időkhöz szükségünk van tárra. Egy körültekintőbb megvalósítás azonban úgy kezeli az összes olyan időbélyegzőt, amely a legkorábbi aktív tranzakciók előtti, hogy „mínusz végtelen" értékűnek tekinti, és nem rögzíti ezeket. Ebben az esetben, a zártáblával analóg táblában tudjuk tárolni az olvasási és írási időket, amelyben csak a legújabban elért adatbáziselemek szerepelnek.
- *Érvényesítés:* Tárat használunk az időbélyegzőkhöz és minden jelenleg aktív tranzakció olvasási/írási halmazaihoz, hozzávéve még egy pár olyan tranzakciót, amelyek azután fejeződnek be, miután valamelyik jelenleg aktív tranzakció elkezdődött.

Így mindegyik megközelítésben az összes aktív tranzakcióra felhasznált tár a tranzakciók által hozzáfért adatbáziselemek számának az összegével megközelítőleg arányos. Az időbélyegzés és az érvényesítés kicsit több helyet használhat fel, ugyanis nyomon kell követnünk a korábban véglegesített tranzakciók bizonyos hozzáféréseit, amelyeket a zártábla nem rögzített volna. Az érvényesítéssel kapcsolatban egy lényeges probléma, hogy a tranzakcióhoz tartozó írási halmazt az írás elvégzése előtt kell már ismernünk (de a tranzakció helyi számításai befejeződése után).

Összehasonlíthatjuk a módszereket abból a szempontból is, hogy késleltetés nélkül befejeződnek-e a tranzakciók. A három módszer hatékonysága attól függ, hogy vajon a tranzakciók közötti *egymásra hatás* erős vagy gyenge (milyen valószínűséggel akar egy tranzakció hozzáférni egy olyan elemhez, amelyhez egy konkurens tranzakció már hozzáfért).

- A zárolás késlelteti a tranzakciókat, azonban elkerüli a visszagörgetéseket, még ha erős is az egymásra hatás. Az időbélyegzők és az érvényesítés nem késlelteti a tranzakciókat, azonban visszagörgetést okozhat, amely a késleltetésnek egy problémásabb formája, azonfelül erőforrásokat is pazarol.
- Ha gyenge az egymásra hatás, akkor sem az időbélyegzés, sem az érvényesítés nem okoz sok visszagörgetést, és előnyösebb lehet a zárolásnál, ugyanis ezeknek általában alacsonyabbak a költségei, mint a zárolási ütemezőnek.
- Amikor szükséges a visszagörgetés, az időbélyegzők hamarabb feltárják a problémákat, mint az érvényesítés, amelyek mindig hagyják, hogy a tranzakció elvégezze az összes belső munkáját, mielőtt megnézné, hogy vissza kell-e görgetni a tranzakciót.

9.9.4. Feladatok

9.9.1. feladat: A következő eseménysorozatokban, az $R_i(X)$ azt jelöli, hogy „T_i tranzakciót elindítjuk, melynek az olvasási halmaza az X adatbáziselemek listája". V_i azt jelenti, hogy „T_i megpróbálja az érvényesítést", és $W_i(X)$ azt jelenti, hogy „T_i befejeződik, és az írási halmaza az X". Mondjuk meg, mi történik, amikor minden sorozatot érvényesítésen alapuló ütemező hajt végre.

* a) $R_1(A,B)$; $R_2(B,C)$; V_1; $R_3(C,D)$; V_3; $W_1(A)$; V_2; $W_2(A)$; $W_3(B)$;
b) $R_1(A,B)$; $R_2(B,C)$; V_1; $R_3(C,D)$; V_3; $W_1(A)$; V_2; $W_2(A)$; $W_3(D)$;
c) $R_1(A,B)$; $R_2(B,C)$; V_1; $R_3(C,D)$; V_3; $W_1(C)$; V_2; $W_2(A)$; $W_3(D)$;
d) $R_1(A,B)$; $R_2(B,C)$; $R_3(C)$; V_1; V_2; V_3; $W_1(A)$; $W_2(B)$; $W_3(C)$;
e) $R_1(A,B)$; $R_2(B,C)$; $R_3(C)$; V_1; V_2; V_3; $W_1(C)$; $W_2(B)$; $W_3(A)$;
f) $R_1(A,B)$; $R_2(B,C)$; $R_3(C)$; V_1; V_2; V_3; $W_1(A)$; $W_2(C)$; $W_3(B)$;

9.10. Összefoglalás

- *Konzisztens adatbázis-állapotok:* A tervezők által megadott megszorításokat kielégítő adatbázis-állapotokat, legyenek azok implikáltak vagy deklaráltak, konzisztensnek nevezzük. Fontos, hogy a műveletek megőrizzék az adatbázis konzisztenciáját, vagyis az adatbázist konzisztens állapotból konzisztens állapotba vigyék.
- *Konkurens tranzakciók konzisztenciája:* Normálisan több tranzakció egyidejűleg fér hozzá az adatbázishoz. Az egymástól elszigetelten futó tranzakciókról feltételezzük, hogy megőrzik az adatbázis konzisztenciáját. Az ütemező feladata annak

biztosítása, hogy a konkurensen működő tranzakciók is megőrizzék az adatbázis konzisztenciáját.

- *Ütemezések:* A tranzakciók műveletekből állnak, többségében az adatbázis olvasásából és írásából. Egy vagy több tranzakció ilyen műveleteinek sorozatát ütemezésnek nevezzük.
- *Soros ütemezések:* Ha a tranzakciókat sorban egymás után, egyenként hajtjuk végre, akkor az ütemezést sorosnak mondjuk.
- *Sorba rendezhető ütemezések:* Az olyan ütemezést, amelynek az adatbázisra való hatása ekvivalens valamely soros ütemezéssel, sorba rendezhetőnek hívunk. Több tranzakciótól származó műveletek átlapolása előfordulhat egy sorba rendezhető ütemezésben, míg maga az ütemezés nem soros, ugyanakkor nagyon elővigyázatosaknak kell lennünk, milyen műveletsorozatokat engedélyezünk, különben az átlapolás az adatbázist nem konzisztens állapotba fogja átalakítani.
- *Konfliktus-sorbarendezhetőség:* Egyszerű teszt, amely a sorbarendezhetőségre elégséges feltétel, mely szerint az ütemezést konfliktus nélküli szomszédos műveletek cseréinek sorozatával sorossá alakíthatjuk át. Az ilyen ütemezést konfliktus-sorbarendezhetőnak nevezzük. Konfliktus fordulhat elő, ha megpróbáljuk ugyanannak a tranzakciónak két műveletét felcserélni, vagy két, ugyanahhoz az adatbáziselemhez hozzáférő műveletet cserélünk meg, amelyek közül legalább az egyik művelet írás.
- *Megelőzési gráf:* Könnyű teszt a konfliktus-sorbarendezhetőséghez, hogy elkészítjük az ütemezés megelőzési gráfját. A csomópontok a tranzakcióknak felelnek meg, a *T*-ből vezet él az *U*-ba, $T \to U$, ha az ütemezésben a *T* valamelyik művelete konfliktusban áll az *U*-nak egy későbbi műveletével. Egy ütemezés akkor és csak akkor konfliktus-sorbarendezhető, ha a megelőzési gráf körmentes.
- *Zárolás:* A legáltalánosabb megközelítés a sorba rendezhető ütemezések biztosításához, hogy zároljuk az adatbáziselemeket mielőtt hozzáférnénk, és feloldjuk a zárakat, miután befejeztük az elemhez való hozzáférést. A zárak megakadályozzák a többi tranzakció hozzáférését az adott elemhez.
- *Kétfázisú zárolás:* Önmagában a zárolás nem biztosítja a sorbarendezhetőséget. A kétfázisú zárolásban minden tranzakció előbb olyan fázisba lép, amikor csak zárakat igényel, és ezután lép olyan fázisba, amikor csak feloldja a zárakat. A kétfázisú zárolás biztosítja a sorbarendezhetőséget.
- *Zárolási módok:* Ahhoz, hogy elkerüljük a tranzakciók felesleges zárolásait, a rendszerek általában több zárolási módot is alkalmaznak, minden módhoz külön szabályok tartoznak, amelyekkel megadjuk, hogy mikor engedélyezhetjük a zárat. A legáltalánosabb rendszer csak olvasáshoz osztott zárakat, az írást is tartalmazó hozzáférésekhez pedig kizárólagos zárakat alkalmaz.
- *Kompatibilitási mátrixok:* A kompatibilitási mátrix hasznos összefoglalása annak, hogy mikor jogos egy bizonyos zármódú zárat engedélyeznünk, amikor ugyanabban vagy más módban, ugyanazon az elemen más zárak is adottak lehetnek.
- *Módosítási zárak:* Az ütemező lehetőséget nyújt arra, hogy egy tranzakció, amely előbb olvassa, és azután írja az elemet, előbb módosítási zárat helyezzen el, és később minősítse át ezt a zárat kizárólagossá. Módosítási zárat akkor is engedélyezhetünk, amikor már vannak osztott zárak az adott elemen, de ha egyszer kiadtunk egy módosítási zárat erre az elemre, ez megakadályozza, hogy más zárakat engedélyezzünk.

- *Növelési zárak:* Abban a speciális esetben, amikor egy tranzakció csak hozzáad vagy levon egy konstanst egy elemből, a növelési zár megfelelő. Ugyanannak az elemnek a növelési zárjai nem mondanak ellent egymásnak, viszont az osztott és kizárólagos zárakkal konfliktust alkotnak.
- *Szemcsézettség hierarchiájú elemek zárolása:* Amikor nagy és kis elemeket – relációkat, lemezblokkokat és esetleg sorokat – kell zárolnunk, a zárak figyelmeztető rendszere biztosítja a sorbarendezhetőséget. A tranzakciók szándékot kifejező zárakat helyeznek el a nagy elemekre, hogy figyelmeztessék a többi tranzakciót, hogy ennek egy vagy több részeleméhez való hozzáférést szándékoznak elvégezni.
- *Faelrendezésű elemek zárolása:* Ha az adatbáziselemekhez csak úgy tudunk hozzáférni, hogy egy fán haladunk lefelé, mint egy B-fa-indexen, akkor egy nem kétfázisú zárolási stratégiával tudjuk biztosítani a sorbarendezhetőséget. A szabályok szerint zárolnunk kell a szülőt, amikor a gyerekre igényelünk zárat, de később a szülőn való zár feloldható, és további zárakat is kiadhatunk.
- *Optimista konkurenciavezérlés:* A zárolás helyett az ütemező felteheti, hogy a tranzakciók sorba rendezhetők lesznek, és csak akkor abortálja a tranzakciót, ha valamilyen lehetséges nem sorba rendezhető viselkedést tapasztal. Ez a megközelítés, amelyet optimistának nevezünk, két részre oszlik, az időbélyegzőn alapuló és az érvényesítésen alapuló ütemezésre.
- *Időbélyegzőn alapuló ütemezők:* Az ütemezőnek ez a típusa időbélyegzőt rendel a tranzakciókhoz, amint elkezdődnek. Az adatbáziselemekhez hozzárendelt olvasási és írási idők az adott műveleteket legújabban végrehajtó tranzakcióknak az időbélyegzői. Ha egy lehetetlen helyzetet észlelne, mint például amikor egy tranzakció egy későbbi tranzakció által felülírt értéket olvasna be, a megsértő tranzakciót visszagörgeti, vagyis abortálja, és utána újraindítja.
- *Érvényesítésen alapuló ütemezők:* Ezek az ütemezők azután érvényesítik a tranzakciókat, miután beolvastak mindent, amire szükségük van, de még mielőtt írtak volna. Azok a tranzakciók, amelyek valamely más tranzakció írásának feldolgozása alatt álló elemet olvastak be, vagy fognak írni, kétes eredményhez vezethetnek, így ezeket a tranzakciókat nem érvényesítjük. Azokat a tranzakciókat, amelyeket nem érvényesítünk, visszagörgetjük.
- *Többváltozatú időbélyegzők:* A gyakorlatban elterjedt technika, hogy a csak olvasási tranzakciókat időbélyegzőkkel ütemezzük, de többszörös változatokkal. Vagyis egy elem írásakor nem írjuk felül az elem korábbi értékeit mindaddig, ameddig azok a tranzakciók be nem fejeződnek, amelyeknek esetleg szüksége lenne ezekre a korábbi értékre. Az írási tranzakciókat a hagyományos zárakkal ütemezzük.

9.11. Irodalomjegyzék

A [6] könyv az ütemezésről, valamint a zárolásról nyújt lényeges forrásanyagot. A [3] ugyanennek egy másik fontos forrása. A konkurenciavezérlés legújabb eredményei a [12]-ben és a [11]-ben találhatók.

Valószínűleg a legjelentősebb cikk a tranzakciófeldolgozásban a [4] a kétfázisú zárolásról. A figyelmeztető protokoll a szemcsézettség hierarchiákra az [5]-ből származik. A fák nem kétfázisú zárolása a [10]-ből ered. A kompatibilitási mátrixot a zárolási módok tanulmányozására a [7]-ben vezették be.

Az időbélyegzők mint konkurenciavezérlési módszerek a [2]-ben és az [1]-ben fordulnak elő. Az érvényesítésen alapuló ütemezés a [8]-ból származik. A többszörös változatok használatát a [9]-ben tanulmányozták.

1. P. A. Bernstein, Goodman, N., „Timestamp-based algorithms for concurrency controll in distributed database systems", *Proc. Intl. Conf. on Very Large Databases* (1980), pp. 285–300.
2. P. A. Bernstein, Goodman, N., Rothnie, J. B. Jr., Papadimitriou, C. H., „Analysis of serializability in SDD-1: a system of distributed databases (the fully redundant case)", *IEEE Trans. on Software Engineering* **SE-4**:3 (1978), pp. 154–168.
3. P. A. Bernstein, Hadzilacos, V., Goodman, N., *Concurrency Control and Recovery in Database Systems,* Addison-Wesley, Reading MA, 1987.
4. Eswaran, K. P., Gray, J. N., Lorie, R. A., Traiger, I. L., „The notions of consistency and predicate locks in a database system", *Comm. ACM* **19**:11 (1976), pp. 624–633.
5. Gray, J. N, Putzolo, F., Traiger, I. L, „Granularity of locks and degrees of consistency in a shared data base", in Nijssen, G. M. (ed.), *Modeling in Data Base Management Systems,* North Holland, Amsterdam, 1976.
6. Gray, J. N, Reuter, A., *Transaction Processing: Concepts and Techniques,* Morgan-Kaufmann, San Francisco, 1993.
7. Korth, H. F., „Locking primitives in a database system", *J. ACM* **30**:1, (1983), pp. 55–79.
8. Kung, H.-T., Robinson, J. T., „Optimistic concurrency control", *ACM Trans. on Database Systems* **6**:2 (1981), pp. 312–326.
9. Papadimitriou, C. H., Kanellakis, P. C., „On concurrency control by multiple versions", *ACM Trans. on Database Systems* **9**:1 (1984), pp. 89–99.
10. Silberschatz, A., Kedem, Z., „Consistency in hierarchical database systems", *J. ACM* **27**:1 (1980), pp. 72–80.
11. Thompsian, A., „Concurrency control: methods, performance, and analysis", *Computing Surveys* **30**:1 (1998), pp. 170–231.
12. Thuraisingham, B., Ko, H.-P., „Concurrency control in trusted database management systems: a survey", *SIGMOD Record* **22**:4 (1993), pp. 52–60.

10. fejezet

Bővebben a tranzakciókezelésről

Ebben a fejezetben a tranzakciókezelés olyan kérdéseiről lesz szó, amelyekkel a 8. és a 9. fejezetben nem foglalkoztunk. Megnézzük, hogyan egyeztethető össze az előző két fejezet nézőpontja: milyen kölcsönhatásban áll egymással a helyreállítás, a tranzakciók abortáltathatóságának és a sorbarendezhetőség fenntartásának a szükségessége? Ezután megtárgyaljuk a tranzakciók közötti holtpontkezelés lehetőségeit. Holtpont tipikusan abban az esetben alakul ki, amikor több tranzakciónak kell várnia egy olyan erőforrásra (például egy zárra), amely az adott pillanatban egy másik tranzakció birtokában van.

A fejezet során bevezetést nyerünk az osztott adatbázisok világába is. Közelebbről megvizsgáljuk az esetleg többszörözött példányok segítségével megosztott adatok zárolási problémáját. Azt a kérdést is áttekintjük, hogy mi alapján lehet dönteni egy olyan tranzakció abortálásáról, illetve véglegesítéséről, amely egyszerre több helyszínen is végez műveletet.

Végül tárgyalásra kerülnek a „hosszú tranzakciókból" eredő problémák. Léteznek olyan alkalmazások, például CAD[1] vagy „munkafolyamat" rendszerek, amelyekben az emberi és a számítógépes eljárások akár több napon keresztül is kölcsönhatásban vannak egymással. Hasonlóan a rövid tranzakciós rendszerekhez (banki műveletek, repülőjegy-foglalás) itt is szükség van az adatbázis-állapot konzisztenciájának a megőrzésére. A 9. fejezetben bevezetett konkurenciavezérlő módszerek azonban nem működnek ésszerűen, amikor a zárak napokra vannak kiosztva, vagy az érvényesítési döntéseket több nappal ezelőtt végbement események kapcsán kell meghoznunk.

10.1. Tranzakciók, melyek nem véglegesített adatokat olvasnak

A 8. fejezetben néhány naplózási eljárással és ezeknek a rendszerhiba utáni helyreállításban betöltött szerepével ismerkedtünk meg. Az adatbázison végrehajtott számításokat olyan folyamatoknak tekintettük, amely során az értékek a nem felejtő lemez, a

[1] *Computer Aided Design,* magyarul Számítógéppel Támogatott Tervezés. *A fordító megjegyzése.*