

Adatbázisok II.

2. ZH elméleti része

A jegyzetet CSONKA Szilvia írta. A jegyzet első számú forrása a gyakorlat.
Az elméleti kérdések kidolgozása HÉRA Zoltán jegyzete alapján készült.

Információk

- pótZH: vizsgaidőszak 1. hetében.
- A ZH $\frac{1}{4}$ 9-kor kezdődik.

1. Rész

1.1. Elméleti kérdések

1. Írja le az UNDO naplózás szabályait és a lemezre írások sorrendjét.

- U_1 : Ha a T tranzakció módosítja az X adatbáziselementet, akkor $\langle T, X, v \rangle$ típusú naplóbejegyzést azt megelőzően kell lemezre írni, mielőtt X új értékét lemezre írná a rendszer.
- U_2 : Ha a tranzakció hibamentesen teljesen befejeződött, akkor a COMMIT naplóbejegyzést csak azt követően szabad lemezre írni, hogy a tranzakció által módosított összes adatbáziselem már lemezre íródott, de ezután viszont a lehető leggyorsabban.
- Összefoglalva az U_1 és U_2 szabályokat, az egy tranzakcióhoz tartozó lemezre írásokat a következő sorrendben kell megejteni:
 - a) Az adatbáziselemek módosítására vonatkozó naplóbejegyzések kiírása.
 - b) Maguknak a módosított adatbáziselemeknek a kiírása.
 - c) A COMMIT naplóbejegyzés kiírása.

2. Adja meg a helyreállítás lépéseit UNDO napló esetén.

(1) Let $S = \text{set of transactions with}$
 $\langle T_i, \text{start} \rangle$ in log, but no
 $\langle T_i, \text{commit} \rangle$ (or $\langle T_i, \text{abort} \rangle$) record in log
(2) For each $\langle T_i, X, v \rangle$ in log,
in reverse order (latest \rightarrow earliest) do:
- if $T_i \in S$ then - write (X, v)
- output (X)
(3) For each $T_i \in S$ do
- write $\langle T_i, \text{abort} \rangle$ to log
(4) Flush log

3. Adja meg az egyszerű ellenőrzőpont képzés lépéseit.

- A CHECKPOINT megadja, hogy meddig kell visszaolvasnunk a naplózásban egy esetleges hiba bekövetkeztét követő helyreállítás során (úgynevezett ellenőrzőpont).
- **Képzése**
 - a) Megtiltjuk az új tranzakciók indítását.
 - b) Megvárjuk, amíg minden futó tranzakció COMMIT vagy ABORT módon véget ér.
 - c) A naplót a pufferből a háttértárra írjuk (FLUSH LOG).
 - d) A naplóba beírjuk, hogy CHECKPOINT.

- e) A naplót újra a háttértárra írjuk: FLUSH LOG.
- f) Újra fogadjuk a tranzakciókat.
- Ezután nyilván elég a CHECKPOINT-ig visszamenni, hiszen előtte minden T_i már valahogy befejeződött.
 - Probléma: Hosszú ideig tarthat, amíg az aktív tranzakciók befejeződnek. (Új tranzakciókat sokáig nem lehet kiszolgálni) → Megoldás: CHECKPOINT képzése működés közben
4. Adja meg a működés közbeni ellenőrzőpont képzés lépéseit UNDO napló esetén.
- $\langle \text{START CKPT } (T_1, \dots, T_k) \rangle$ naplóbejegyzés készítése, majd lemezre írása (FLUSHLOG) ahol T_1, \dots, T_k az éppen aktív tranzakciók nevei .
 - Meg kell várni a T_1, \dots, T_k tranzakciók mindegyikének normális vagy nem normális befejeződését, nem tiltva közben újabb tranzakciók indítását.
 - Ha a T_1, \dots, T_k tranzakciók mindegyike befejeződött, akkor $\langle \text{END CKPT} \rangle$ naplóbejegyzés elkészítése, majd lemezre írása (FLUSH LOG).
5. Írja le a REDO naplázás szabályait és a lemezre írások sorrendjét.
- A REDO naplázás szabályai:

R1: Mielőtt az adatbázis bármely X elemét a lemezen módosítanánk, az X módosítására vonatkozó összes naplóbejegyzésnek, azaz $\langle T, X, v \rangle$ -nek és $\langle T, \text{COMMIT} \rangle$ -nak a lemezre kell kerülnie.
 - A lemezre írás sorrendje:

Minthogy a COMMIT bejegyzést csak akkor írhatjuk a naplóba, miután a tranzakció teljesen és hibamentesen befejeződött, így a COMMIT bejegyzés csak a módosításokat leíró bejegyzések után állhat, ezért úgy is összegezhetjük R₁ szabálya hatását, hogy: ha helyrehozó naplázást használunk, akkor az egy tranzakcióra vonatkozó lemezre írásoknak a következő sorrendben kell megtörténniük:

 - a) Az adatbáziselemek módosítását leíró naplóbejegyzések lemezre írása.
 - b) A COMMIT naplóbejegyzés lemezre írása.
 - c) Az adatbáziselemek értékének tényleges cseréje a lemezen.
6. Adja meg a helyreállítás lépéseit REDO napló esetén.
- ```

(1) Let S = set of transactions with
<Ti, commit> (and no <Ti, end>) in log
(2) For each <Ti, X, v> in log, in forward
order (earliest → latest) do:
 - if Ti ∈ S then Write(X, v)
 Output(X)
(3) For each Ti ∈ S, write <Ti, end>
(4) FLUSH LOG

```
7. Adja meg a működés közbeni ellenőrzőpont képzés lépéseit REDO napló esetén.
- a)  $\langle \text{START CKPT}(T_1, \dots, T_k) \rangle$  naplóbejegyzés elkészítése és lemezre írása, ahol  $T_1, \dots, T_k$  az összes éppen aktív tranzakció.
  - b) Az összes olyan adatbáziselem kiírása lemezre, melyeket olyan tranzakciók írtak a pufferekbe, melyek a START CKPT naplóba írásakor már befejeződtek, de puffereik lemezre még nem kerültek.
  - c)  $\langle \text{END CKPT} \rangle$  bejegyzés naplóba írása, és a napló lemezre írása.
8. Adja meg a működés közbeni ellenőrzőpont képzés lépéseit UNDO/REDO napló esetén.
- a) Írunk a naplóba  $\langle \text{START CKPT}(T_1, \dots, T_k) \rangle$  naplóbejegyzést, ahol  $T_1, \dots, T_k$  az aktív tranzakciók, majd írjuk a naplót lemezre.

- b) Írjuk a lemezre az összes piszkos puffert, tehát azokat, melyek egy vagy több módosított adatbáziselementet tartalmaznak. A helyrehozó naplózástól eltérően itt az összes piszkos puffert lemezre írjuk, nem csak a már befejezett tranzakciók által módosítottakat.
- c) Írunk <END CKPT> bejegyzést a naplóba, majd írjuk a naplót lemezre.

## 1.2. Feldatok

### Források

- A feladatsor és további elméleti részek NIKOVITS Tibor honlapján: [http://people.inf.elte.hu/nikovits/AB2/ab2\\_feladat9.txt](http://people.inf.elte.hu/nikovits/AB2/ab2_feladat9.txt).
- További kapcsolódó források a gyakorlatvezető honlapjáról:
  - UW\_redo\_naplo.doc
  - UW\_undo\_naplo.doc
  - UW\_undo\_redo\_naplo.doc
- Az előadás honlapján található (<http://people.inf.elte.hu/kiss/15ab2/13ab2osz.htm>) *naplo.ppt* egyes diái (a 8., 9., 10. előadás anyagai között megtalálható) szintén segítségül szolgálnak az alábbi feladatok megoldásának megértéséhez.

#### 8.1.1 Feladat (könyv 463. old.)

Tegyük fel, hogy az adatbázisra vonatkozó konziszenciamegszorítás:  $0 \leq A \leq B$ .  
 Állapítsuk meg, hogy a következő tranzakció megőrizik-e az adatbázis konziszenciáját.

**T1:**  $A := A + B; B := A + B;$

- *Megjegyzés:* itt valójában  $B := A + 2B$  fog végrehajtódni, ugyanis az értékadásokat szekvenciálisan, egymás után fogjuk végrehajtani.
- *Megoldás:*

```

||| INPUT(A) //A-t beolvassuk a memóriapufferbe a lemezről
||| INPUT(B) //B-t is
||| READ(A, t) //a memóriából a t lokális változóba másoljuk A-t
||| READ(B, s) //B-t pedig s-be
||| t := t + s //végrehajtjuk a kért értékadások közül az elsőt
||| WRITE(A, t) //az eredményt a memóriapufferbe írjuk
||| s := s + t //végrehajtjuk a második értékadást is
||| WRITE(B, s) //ezt is a memóriába írjuk
||| OUTPUT(A) //a memóriából a lemezre mentjük A eredményét
||| OUTPUT(B) //B-ét is

```

- *Mejegyzés:* itt nem kérdezték a naplózást, így azt ebben a feladatban nem kell részleteznünk.

#### 8.2.4 Feladat (könyv 476. old.)

A következő naplóbejegyzés-sorozat a T és U két tranzakcióra vonatkozik:

```

||| <start T>
||| <T, A, 10>
||| <start U>
||| <U, B, 20>
||| <T, C, 30>
||| <U, D, 40>
||| <T, A, 11>
||| <U, B, 21>

```

```
|| <COMMIT U>
|| <T, E, 50>
|| <COMMIT T>
```

Adjuk meg a helyreállítás-kezelő tevékenységeit, ha az utolsó lemezre került naplóbejegyzés:

- a) <start U>
- b) <COMMIT U>
- c) <T, E, 50>
- d) <COMMIT T>

– *Megoldás UNDO helyreállítás (lásd. naplo.ppt 54. dia) esetén:*

- a) Az adatvesztéssel járó hiba bekövetkeztéig a következők futottak le:

```
|| <start T>
|| <T, A, 10>
|| <start U>
```

*Megoldás:*

```
|| WRITE(A, 10) //Visszaállítjuk A T tranzakció lefutása előtti állapotát.
|| OUTPUT(A) //Lemezre írjuk A helyreállított változatát.
|| <T, ABORT> //Naplózzuk, hogy T tranzakció eredményét visszaállítottuk.
|| //Azaz helyreállítottuk, a nem teljesen lefutott tranzakció
|| // okozta nem konzisztesn állapotot, hogy ismét konziszten
|| // legyen az adatbázis.
|| FLUSH LOG //Naplázást is lementjük.
```

- b) Az adatvesztéssel járó hiba bekövetkeztéig a következők futottak le:

```
|| <start T>
|| <T, A, 10>
|| <start U>
|| <U, B, 20>
|| <T, C, 30>
|| <U, D, 40>
|| <T, A, 11>
|| <U, B, 21>
|| <COMMIT U>
```

*Megoldás:*

```
|| WRITE(A, 11)
|| OUTPUT(A)
|| WRITE(C, 30)
|| OUTPUT(C)
|| WRITE(A, 10)
|| OUTPUT(A)
|| <T, ABORT> //Naplóbejegyzés: helyreállítottuk A állapotát a T előttire
|| FLUSH LOG //Lementjük a fenti naplóbejegyzést is.
```

*Megjegyzés:* Visszaállítjuk a dolgokat a napló végétől az eleje felé haladva. Az *U* tranzakció által végrehajtott módosításokat nem kell helyreállítanunk, ugyanis a hiba előtt ez már sikeresen lefutott és le is lett mentve (*COMMIT*) az eredménye, tehát nem zavar bele az adatbázis konziszenciájába.

- c) (Gyakorlaton nem volt.)

d) minden le lett mentve az adatvesztéssel járó hiba előtt, tehát nem szükséges semmit sem helyreállítani. :)

- **Megoldás REDO helyreállítás (lásd. naplo.ppt 78. dia) esetén:**

*Megjegyzés:* a dián a végéről lemaradt a "FLUSH LOG".

a) (Nem volt gyakorlaton.)

b) Az adatvesztéssel járó hiba bekövetkeztéig a következők futottak le:

```
|| <start T>
|| <T, A, 10>
|| <start U>
|| <U, B, 20>
|| <T, C, 30>
|| <U, D, 40>
|| <T, A, 11>
|| <U, B, 21>
|| <COMMIT U>
```

*Megoldás:*

```
|| WRITE(B, 20)
|| OUTPUT(B)
|| WRITE(D, 40)
|| OUTPUT(D)
|| WRITE(B, 21)
|| OUTPUT(B)
|| FLUSH LOG
```

*Megjegyzés:*

*REDO* esetén azokat az adatokat állítjuk helyre a tranzakció lefutása utáni állapotukra, amelyekre volt *COMMIT*, azaz lefutottak, csak eredményük nem került egészében mentésre (mivel eközben következett be a hiba), azaz nem volt *END*. Így mivel *U* tranzakcióra volt *COMMIT*, de *END* nem, ezért a fenti példában ennek az eredményét állítottuk helyre. *T*-vel nem foglalkozunk, hiszen arra *COMMIT* sem volt, azaz be sem fejeződött, így nincs lenaplózva és nem is tudhatjuk az összes adatváltoztatásának következményét (azaz ha helyreállítanánk a naplóban szereplő eddigi adatváltoztatásait, nem konzisztens állapotú adatbázist kapnánk).

## Az Oracle naplózása

- Az Oracle valójában *REDO* naplózást használ. Így elég csak az utasítást lementenie egy update-ről.
- Az *UNDO* naplózásra külön *UNDO táblateret* használ, mivel ez nagyon memóriaigényes, hiszen ekkor az összes olyan táblaelem korábbi értékét le kell menteni, amelyet az adott update megváltoztatott, azaz előfordulhat, hogy komplett táblákat kell eltárolni. Ezen mentések felhasználásával tudja megoldani az Oracle, hogy amennyiben egyszerre többen használnak egy táblát, akkor a használat kezdeti pillanatbeli állapotát bocsájtja a felhasználó rendelkezésére a táblának.

## 2. Rész

### 2.1. Elméleti kérdések

1. Mit jelent az, hogy két ütemezés konfliktus-ekvivalens ?
  - Azt mondjuk, hogy két ütemezés konfliktus ekvivalens (conflict-equivalent), ha szomszédos műveletek nem konfliktusos cseréinek sorozatával az egyiket átalakíthatjuk a másikká.
  - Megjegyzés: A konfliktus (conflict) vagy konfliktuspár olyan egymást követő műveletpár az ütemezésben, amelyeknek ha a sorrendjét felcseréljük, akkor legalább az egyik tranzakció viselkedése eg változhat.
2. Mit jelent az, hogy egy ütemezés konfliktus-sorbarendezhető ?
  - Azt mondjuk, hogy egy ütemezés konfliktus-sorbarendezhető (conflict-serializable schedule), ha konfliktus-ekvivalens valamely soros ütemezéssel.
3. Mi az a megelőzési gráf és hogyan épül fel ?
  - 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  $T_i <_S T_j$ .
4. Mit állíthatunk konzisztens, kétfázisú tranzakciók jogoszerű ütemezéséről ?
  - Konzisztens, kétfázisú zárolású tranzakciók bármely S jogoszerű ütemezését át lehet alakítani konfliktusekvivalens ütemezéssé.
5. Igaz-e, hogy konzisztens tranzakciók jogoszerű ütemezése konfliktus-sorbarendezhető? (ellenpélda)

| $T_1$          | $T_2$        | $A$ | $B$ |
|----------------|--------------|-----|-----|
|                |              | 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 |

6. Igaz-e, hogy konzisztens, kétfázisú tranzakciók esetén nem alakulhat ki holtpont? (ellenpélda)

| $T_1$                       | $T_2$                       | $A$ | $B$ |
|-----------------------------|-----------------------------|-----|-----|
| $l_1(A); r_1(A);$           | $l_2(B); r_2(B);$           | 25  | 25  |
| $A := A + 100;$             | $B := B * 2;$               |     |     |
| $w_1(A);$                   | $w_2(B);$                   |     |     |
| $l_1(B); \text{elutasítva}$ | $l_2(A); \text{elutasítva}$ | 125 | 50  |

7. Mi az a várakozási gráf és hogyan épül fel?

- Várakozási gráf: csúcsai a tranzakciók és akkor van él  $T_i$ -ból  $T_j$ -be, ha  $T_i$  vár egy olyan zár elengedésére, amit  $T_j$  tart éppen.
- A várakozási gráf változik az ütemezés során, ahogy újabb zárkérések érkeznek vagy zárelengedések történnek, vagy az ütemező abortáltat egy tranzakciót.

## 2.2. Feladatok

### Források

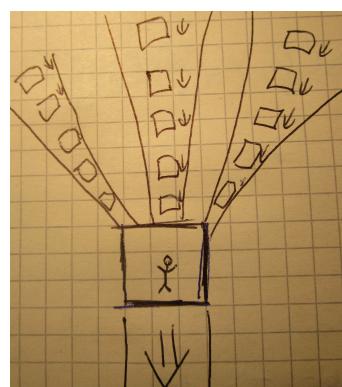
- Az előadás diái közül a *konkurencia.ppt*, mely a 10., 11., 12. előadás anyagánál megtalálható az előadó honlapján: <http://people.inf.elte.hu/kiss/15ab2/13ab2osz.htm>
- A tankönyv ütemezésekre vonatkozó része a gyakorlat honlapján: [http://people.inf.elte.hu/nikovits/AB2/UW\\_utemezesek.doc](http://people.inf.elte.hu/nikovits/AB2/UW_utemezesek.doc).
- A feladat és további elméleti részek NIKOVITS Tibor honlapján: [http://people.inf.elte.hu/nikovits/AB2/ab2\\_feladat10.txt](http://people.inf.elte.hu/nikovits/AB2/ab2_feladat10.txt)

### Soros ütemezések (*konkurencia.ppt 6.dia*)

- "Két tranzakciónak két soros ütemezése van, az egyikben  $T_1$  megelőzi  $T_2$ -t, a másikban  $T_2$  előzi meg  $T_1$ -et."
- Probléma: több tranzakciót egyszerre kéne végrehajtani olyan sorrendben, amely olyan eredményt kapunk, mintha sorasan hajtottuk volna őket végre.
- Mindegy, hogy  $T_1$  és  $T_2$  egyszerre végrehajtott tranzakciók közül melyik hajtódik végre először, a lényeg a konzisztencia megőrzése.

### Ütemezés (*konkurencia.ppt 3. dia*)

- "Az ütemezés (schedule) egy vagy több tranzakció által végrehajtott lényeges műveletek időrendben vett sorozata, amelyben az egy tranzakcióhoz tartozó műveletek sorrendje megegyezik a tranzakcióban megadott sorrenddel"
- Szemléletesen:
  - Képzeljük el, hogy egy kereszteződésben több különböző út találkozik, melyekről egyszerre érkeznek autók melyek minden egy adott útra akarnak besorolni, melyről nem jön senki.
  - Az ütemező egy sorrendet állít fel, mint egy forgalmat irányító rendőr.
  - Elvárások:
    - Az azonos útról érkező autók sorrendje ne változzon!
    - Azaz egy tranzakció műveleteit adott sorrendben kell elvégezni, de attól még más tranzakciók műveleteit csinálhatjuk közben. compactitem



## 9.1.2 Feladat

Adott az alábbi három tranzakció:

```
T1: READ(X,t); t:=t+100; WRITE(X,t);
T2: READ(X,t); t:=t*2; WRITE(X,t);
T3: READ(X,t); t:=t+10; WRITE(X,t);
```

a) Hányfélé különböző ütemezése van a fenti 3 tranzakciónak?

- Ekkor nem törődünk azzal, hogy az egyes műveletek mely tranzakcióhoz tartoznak, de azt szem előtt tartjuk, hogy bár nem sorosan hajtjuk őket végre, de az eredménynek olyannak kell lennie, mintha ezt tettük volna.
- Összesen 9 műveletet szeretnénk végrehajtani, ha semmit sem veszünk figyelembe, ezt  $9!$  féle képpen tehetjük meg.
- Ekkor amit még nem vettünk figyelembe: egy tranzakcióhoz tartozó lépések egymáshoz képesti sorrendje ne változzon.
  - Ezeknek egymáshoz viszonyított sorrendjét  $3!$  féle képpen cserélhetjük fel.
  - A fenti  $9!$ -ban ez az egy tranzakcióhoz tartozó műveletek  $3!$  féle cseréje is benne van.
  - Tehát az egy tranzakcióhoz tartozó műveletek egymáshoz való sorrendjének változtatásával létrehozott verziók számával leosztva a  $9!$ -t kapjuk a kért darabszámot.
- Tehát a megoldás:

$$\frac{9!}{3!3!3!}$$

b) Hányfélé soros ütemezése van a fenti 3 tranzakciónak?

- Soros ütemezés esetén a 3 tranzakció lépései nem cserélgetjük, hanem először végrehajtjuk az egyik tranzakció összes lépését, majd ha azzal minden megvagyunk, akkor veszünk egy másikat, és azzal is így tovább...
- Azaz először 3 tranzakció közül végrehajtunk 1-et, majd a maradék 2 közül választunk 1-et, majd megcsináljuk az utolsót is.
- Ez így összesen: **3! féle soros ütemezés**.

## 9.2.1 Feladat

Adott az alábbi két tranzakció. Igazoljuk, hogy a (T1, T2) és (T2, T1) soros ütemezések ekvivalensek, vagyis az adatbázison való hatásuk azonos.

```
T1: READ(A,t); t:=t+2; WRITE(A,t); READ(B,t); t:=t*3; WRITE(B,t);
T2: READ(B,s); s:=s*2; WRITE(B,s); READ(A,s); s:=s+3; WRITE(A,s);
```

Adjunk példát a fenti 12 művelet sorba rendezhető és nem sorba rendezhető ütemezésére is.

a) Sorba rendezhető ütemezésre példa:

- a) Végrehajtjuk T1 első 3 műveletét.
- b) Végrehajtjuk T2 első 3 műveletét.
- c) Végrehajtjuk T1 maradék, utolsó 3 műveletét.
- d) Végrehajtjuk T2 maradék, utolsó 3 műveletét.

```
READ(A,t)
t:=t+2
WRITE(A,t)
READ(B,s)
s:=s*2
WRITE(B,s)
```

```

||| READ(B,t)
||| t:=t*3
||| WRITE(B,t)
||| READ(A,s)
||| s:=s+3
||| WRITE(A,s)

```

b) Nem sorba rendezhető ütemezésre példa:

- a) Végre hajtjuk T1 első 2 műveletét.
- b) Végre hajtjuk T2 minden 6 műveletét.
- c) Végre hajtjuk T1 maradék 4 műveletét.

```

||| READ(A,t)
||| t:=t+2
||| READ(B,s)
||| s:=s*2
||| WRITE(B,s)
||| READ(A,s) //Az A értékét még korábban elkezdtük változtatni, de még
||| s:=s+3 //nem írtuk vissza, itt A korábbi értékével kezdtünk el dolgozni!!
||| WRITE(A,s)
||| WRITE(A,t) //Felülírja T2 változtatását!
||| //Olyan lesz, mintha T2 nem módosította volna A-t
||| READ(B,t)
||| t:=t*3
||| WRITE(B,t);

```

### Konfliktus-sorbarendezhetőség (*konkurenencia.ppt 11-14. dia*)

- ELV: nem konfliktusos cserékkel az ütemezést megpróbáljuk soros ütemezéssé átalakítani. Ha ezt meg tudjuk tenni, akkor az eredeti ütemezés sorbarendevezhető 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.
- A konfliktus-sorbarendezhetőség elégsgéges feltétel a sorbarendevezhetőségre, vagyis egy konfliktus-sorbarendezhető ütemezés sorbarendevezhető ütemezés is egyben.
- **Megjegyzés:**
  - Egy tranzakcióban belüli műveletek cseréje nem engedélyezett!!
  - Szempont: Több tranzakció azonos elemet egyszerre nem változtathat, de különbözőkét igen.
  - Az ütemező konfliktus - sorbarendevezhető sorrendeket enged meg.

### 9.2.2 Feladat

Az előző feladat tranzakcióiban csak az írási és olvasási műveleteket jelölve a következő két tranzakciót kapjuk:

|| T1: R1(A); W1(A); R1(B); W1(B);  
|| T2: R2(B); W2(B); R2(A); W2(A);

A fenti 8 művelet ütemezései közül hány darab konfliktusekvivalens a (T1,T2) soros sorrenddel?

- (T1,T2) soros sorrend:  
|| R1(A); W1(A); R1(B); W1(B) | R2(B); W2(B); R2(A); W2(A)
- Csak olyan egymás melletti elemeket cserélhetünk, melyek különböző tranzakcióhoz tartoznak (*lásd. 11-14. dia*).
- Tehát az elsőként egyedül felmerülő cserelehetőségünk a két tranzakció "határán" lévő két művelet, de ezeket nem cserélhetjük fel, mivel ugyan azzal a változóval dolgoznak.
- Tehát egyetlen konfliktusekvivalens sorrendet találtunk a (T1, T2)vel: saját magát.
- Azaz a helyes megoldás: **1 db**.

### 9.2.3 Feladat

Adjuk meg a konfliktus-sorbarendezhető ütemezések számát az alábbi két tranzakcióra.

|| T1: R1(A); W1(A); R1(B); W1(B);  
|| T2: R2(A); W2(A); R2(B); W2(B);

- (T1, T2) és (T2, T1) minden soros ütemezés esetén szóba jövő cserékkel keletkezett konfliktus-sorbarendezhető ütemezéseket meg kell nézni.

- **(T1, T2):**

|| //T1:  
|| R1(A); W1(A); //1  
|| R1(B); W1(B); //2  
  
|| //T2:  
|| R2(A); W2(A); //3  
|| R2(B); W2(B); //4

- 4 részre osztottuk fel a tranzakciókat: ezen részek egy adatot használnak fel, és nem kerülhet bele azonos adatra vonatkozó művelet.
- 1.) és 4.) helye fix, mivel a tranzakción belüli sorrendjüket nem cserélhetjük meg, ugyanakkor
  - A 4.) nem kerülhet T1 utolsó művelete elő, mivel ugyan azon a adaton hajtanak végre műveletet.
  - Az 1.) hasonlóan nem cserélhető fel azonos tranzakció műveleteivel, és a T2 első műveletével sem, mivel azonos adaton hajtanak végre műveletet.
- A 2.) és 3.)-at cserélhetjük úgy, hogy azonos tranzakción belüli műveletek sorrendje ne változzon egymáshoz képest.
  - Mivel 1.) és 4.)-et nem cserélhetjük, megállapítottuk, hogy csak 2.) és 3.)-on belül csinálhatunk cseréket.
  - Mintha 2 tranzakciót akarnánk ütemezni:  $\frac{4!}{2!2!} = 6$ .

- **(T2, T1):**

- szimmetrikus ( $T_1$ ,  $T_2$ )-re.
  - Így itt is:  $\frac{4!}{2!2!} = 6$ .
- Tehát a konfliktus-sorbarendezhető ütemezések száma összesen:  $6 + 6 = 12$
- Megjegyzés: ehhez hasonló feladat biztosan lesz a ZH-ban.

## Megelőzési gráf

- Lesz olyan feladat, melyben majd fel kell rajzolni.
- Lásd. *konkurencia.ppt 19.dia*
- Pl. ha (1)-nek (2) előtt kell állnia, akkor (1)-vől megy nyíl (2)-be.
- Ha van irányított kör benne, akkor nem konfliktus-sorba rendezhető.
- Ha nincs benne irányított kör, akkor konfliktus-sorba rendezhető.

## A zárak használata

- A zárolási ütemező a konfliktus-sorbarendezhetőséget követeli meg, (ez erősebb követelmény, mint a sorbarendezhetőség).
- Tranzakciók konzisztenciája (consistency of transactions):
  1. A tranzakció csak akkor olvashat vagy írhat egy elemet, ha már korábban zárolta azt, és még nem oldotta fel a zárat.
  2. Ha egy tranzakció zárol egy elemet, akkor később azt fel kell szabadítania.
- Az ütemezések jogoszerűsége (legality of schedules):
  1. Nem zárolhatja két tranzakció ugyanazt az elemet, csak úgy, ha az egyik előbb már feloldotta a zárat.