

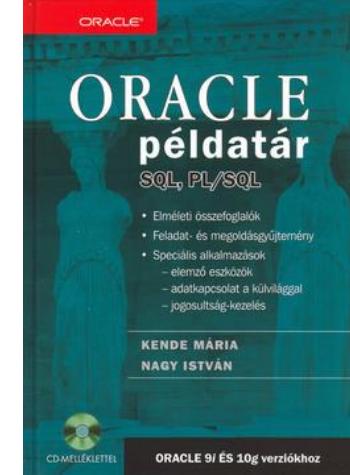
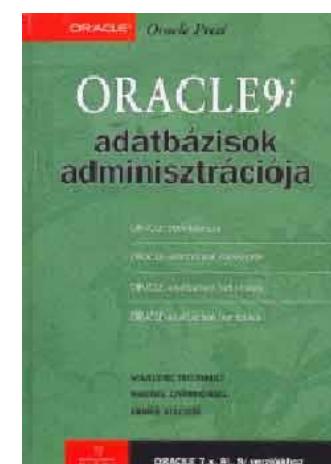
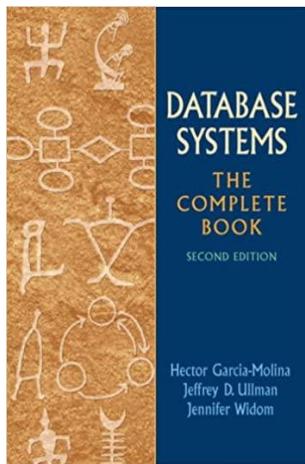
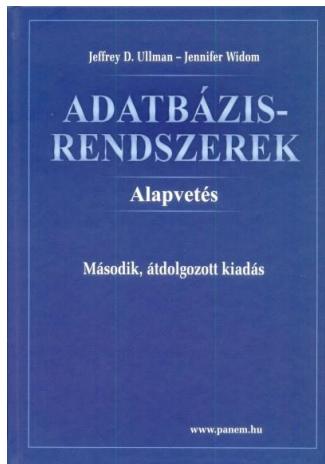
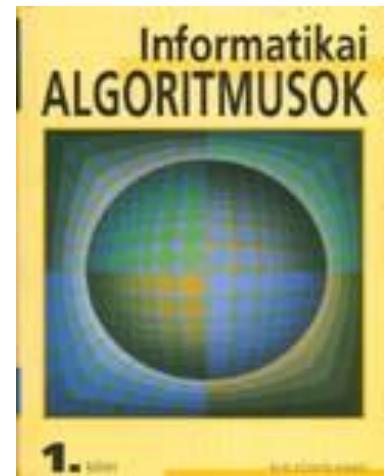
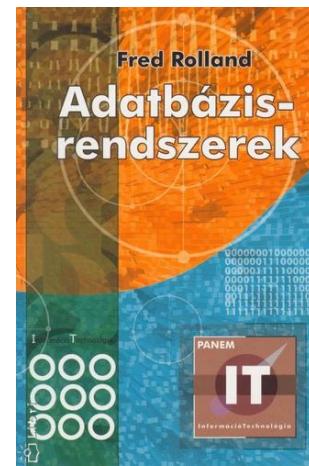
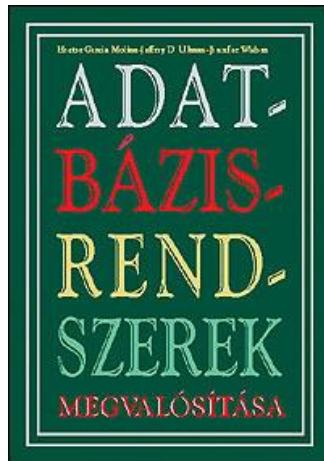
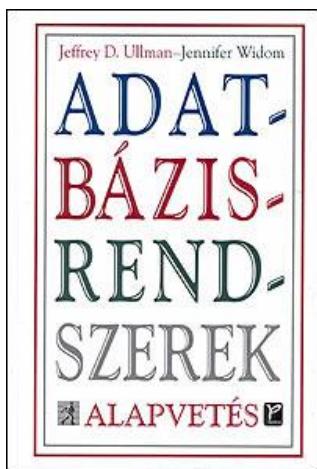
Adatbázisok 2

Kiss Attila kiss@inf.elte.hu

- A kurzus tematikája:
- Lekérdezések optimalizálása
 - relációs algebrai optimalizáció, logikai lekérdező terv
 - fizikai fájlszervezés alapjai
 - fizikai lekérdező terv, operátorok költsége
 - több tábla összekapcsolása
 - Oracle SQL optimalizálása
- Adatbázis rendszerhibák kezelése
 - Naplózás, helyreállítás, ellenőrzőpontok
 - Oracle naplózási technikái
- Adatbázisok konkurens használata
 - Tranzakciók, ütemezése tulajdonságai
 - Szérializálhatóság biztosítása, aktív (zártípusok, időbélyegzők) és passzív módszerek
 - Az Oracle zárolási technológiája



IRODALOM



<https://people.inf.elte.hu/kiss/>



ISMÉTLÉS

(Az Adatbázis 1 tárgyban tanultak)

1. **Adatbázis-kezelő rendszerek** általános jellemzői.
2. **A relációs adatmodell**, a relációs algebra műveletei, használata
3. **Az SQL nyelv részei** (ORACLE specifikusan):
-DDL, DML, QL, triggerek, jogosultságok, PL/SQL, függvények, procedúrák, cursorok használata, programozás,
4. **Adatmodellezés**, egyed-kapcsolat modell, az E/K diagram átalakítása relációs adatmodellé.

Adatbázisrendszerek

ABR1 1. fejezet (19.- 45. oldal)

- **Adatbázis-kezelés:**

- **Háttértárolón tárolt, nagy adatmennyiséget hatékony kezelése (lekérdezése, módosítása)**
- Adatmodell támogatása
- Adatbázis-kezelő nyelvek támogatása
- Több felhasználó támogatása
- Tranzakció-kezelés
- Helyreállíthatóság
- Ügyfél-kiszolgáló felépítés
- Adatvédelem, adatbiztonság



Adatmodellek

- **Az adatmodell a valóság fogalmainak, kapcsolatainak, tevékenységeinek magasabb szintű ábrázolása**
 - Hálós, hierarchikus adatmodell (apa-fiú kapcsolatok gráfja, hatékony keresés)
 - Relációs adatmodell (táblák rendszere, könnyen megfogalmazható műveletek)
 - Objektum-orientált adatmodell (az adatbázis-kezelés funkcionalitásainak biztosítása érdekében gyakran relációs adatmodellre épül)
 - Logikai adatmodell (szakértői rendszerek, tények és következtetési szabályok rendszere)
 - Félig strukturált (XML) adatmodell

Adatbázis-kezelő nyelvek

- **DDL** – adatdefiniáló nyelv (sémák, adatstruktúrák megadása)
- **DML** – adatkezelő nyelv (beszúrás, törlés, módosítás)
- **QL** – lekérdező nyelv
 - **Deklaratív** (SQL, kalkulusok)
 - **Procedurális** (relációs algebra)
- **PL/SQL** – programozási szerkezetek + SQL
- **Programozási nyelvbe ágyazás** (előfordító használata)
- **4GL** nyelvek (alkalmazások generálása)

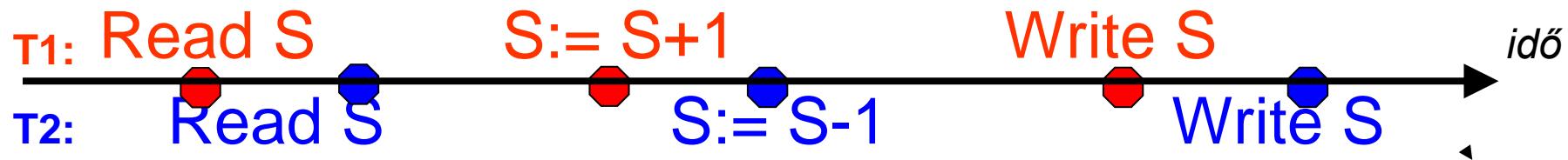


Több felhasználó támogatása

- **Felhasználói csoportok**
- **DBA** – adatbázis-rendszergazda
- **Jogosultságok** (objektumok olvasása, írása, módosítása, készítése, törlése, jogok továbbadása, jogok visszavonása)
- Jogosultságok tárolása rendszertáblákban történik

Tranzakció-kezelés

- **Tranzakció:** adatkezelő műveletekből (adategység írása, olvasása) álló sorozat
- Cél: tranzakciók párhuzamos végrehajtása



- A tranzakció-kezelő biztosítja:
 - **Atomosság** (a tranzakció egységesen lefut vagy nem)
 - **Következetesség** (a tranzakció futása után konzisztens legyen az adatbázis)
 - **Elkülönítés** (párhuzamos végrehajtás eredménye egymás utáni végrehajtással egyezzen meg)
 - **Tartósság** (a befejezett tranzakció eredménye rendszerhiba esetén sem veszhet el)



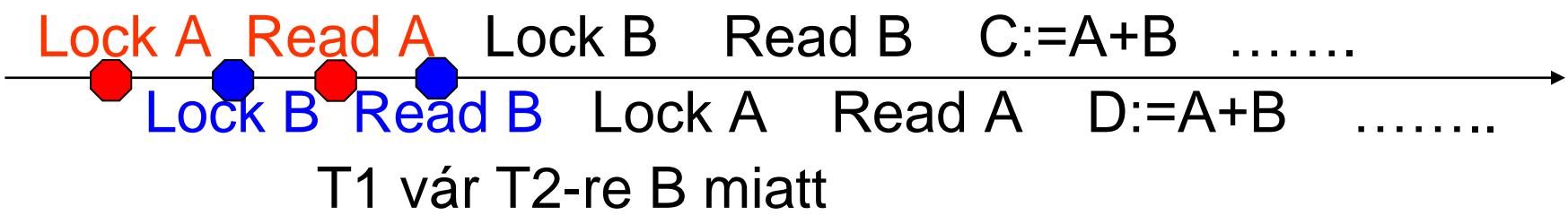
Tranzakció-kezelés

- **Zárolások (Lock, Unlock)**

T1: (Lock S, Read S, $S:=S+1$, Write S, Unlock S)

T2: (Lock S, Read S, $S:=S-1$, Write S, Unlock S)

- A zár kiadásához meg kell várni a zár feloldását.
- Csökken a párhuzamosíthatóság
- Zárak finomsága (zárolt adategység nagysága, zárolás típusa) növeli a párhuzamosíthatóságot
- **Holtpont probléma:**



T1 → T2

T2 vár T1-re A miatt



Tranzakció-kezelés

- **Kétfázisú protokoll** – a tranzakció elején zárolunk minden szükséges adatelemet, a végén minden zárat feloldunk
- **Tranzakciók érvényesítése**, naplózás, Commit, Rollback, Checkpoint
- **Ütemező** (tranzakciók műveleteinek végrehajtási sorrendjét adja meg)
- **Szérializálhatóság** (az ütemezés ekvivalens a tranzakciók egymás utáni végrehajtásával)
- Tranzakciók állapotát, elvégzett műveleteket rendszertáblák tárolják

Helyreállíthatóság

- Szoftver- vagy hardverhiba esetén az **utolsó konzisztens állapot visszaállítása**
- Rendszeres mentések
 - Statikus adatbázis (módosítás nem gyakori)
 - Dinamikus adatbázis (módosítás gyakori)
- Naplóállományok
- Összefügg a tranzakciókezeléssel

Ügyfél-kiszolgáló felépítés

- **Kiszolgáló:**
 - nagy tárhellyel rendelkező, gyors gép
 - adatbázis-műveletek optimalizált, párhuzamos végrehajtása
- **Ügyfél:**
 - adatbázis-művelet megfogalmazása
 - elküldése
 - az eredményadatok fogadása
 - megjelenítése
- Más felépítések is léteznek (például **köztes réteg** az ügyfél és a kiszolgáló között)

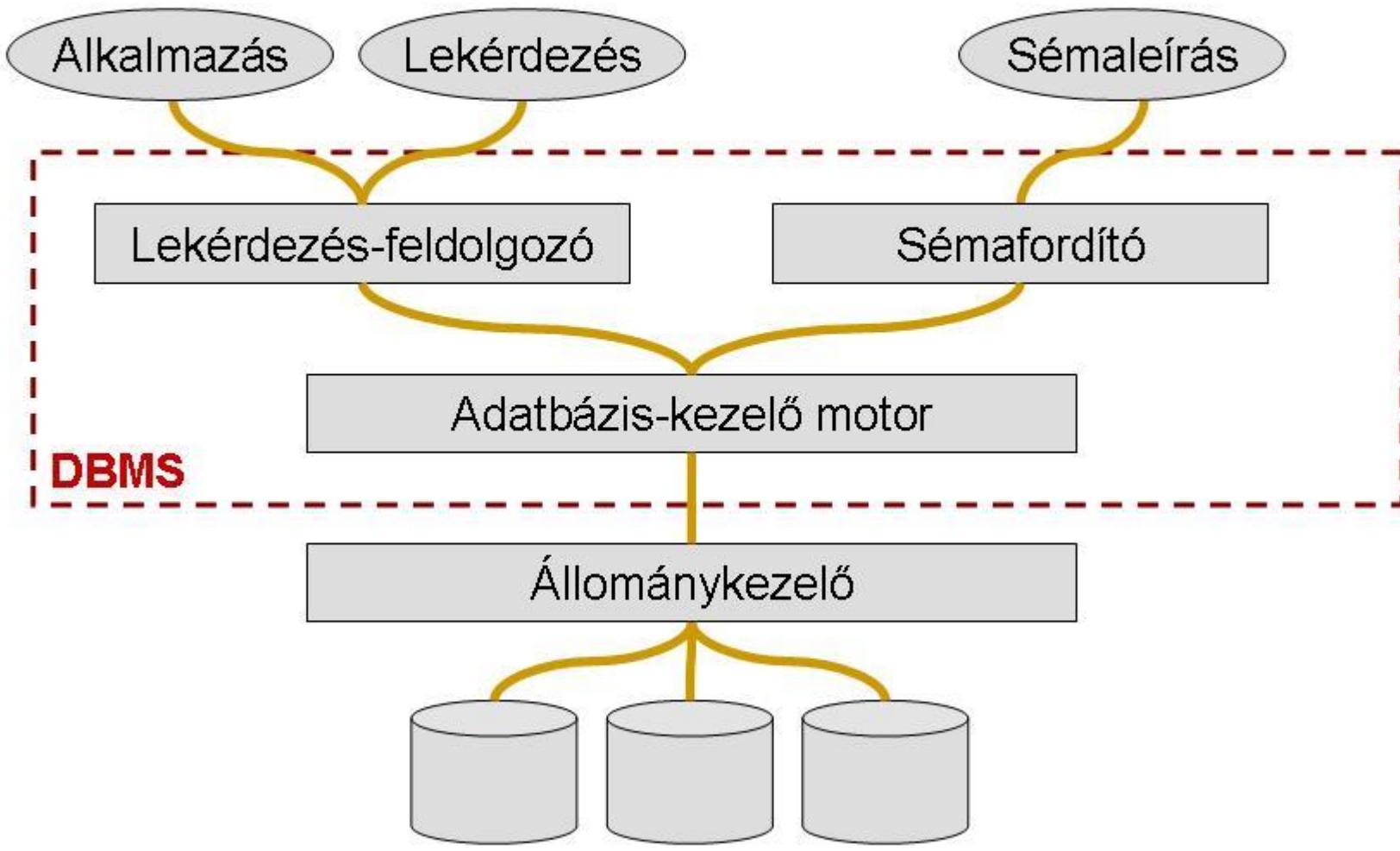
Adatvédelem, adatbiztonság

- **Jogosultságok kezelése**, felhasználók, jelszavak, hozzáférési jogok
- Adatbázissémák korlátozása (virtuális) **nézettáblák** segítségével
- Tárolt adatok, hálózati adatforgalmak **titkosítása** (nagy prímszámok, RSA, DES)

Adatbázis-kezelők felépítése

- **Lekérdezés-feldolgozó**
 - Lekérdezés szintaktikai ellenőrzése
 - Adatbázis-objektumok létezésének, és a hozzáférési jogoknak az ellenőrzése (metaadatbázis, rendszertáblák)
 - Lekérdezés optimális átfogalmazása
 - Végrehajtási tervezés készítése
 - Az adatstruktúrák, méretek statisztikái alapján várhatóan minimális költségű végrehajtási terv kiválasztása
 - Az optimális végrehajtási terv lefuttatása
- **Tranzakció-kezelő:**
 - Tranzakciók párhuzamos végrehajtásának biztosítása (**atomosság**, **következetesség**, **elkülönítés**, **tartósság**)
- **Tárkezelő (**állománykezelő**):**
 - fizikai adatstruktúrák, táblák, **indexek**, **pufferek** kezelése

Adatbázisok ANSI/X3/SPARC modellje



Adatbázisok különböző szintjei

- **Sémák** (tervezek, leírások) és **előfordulások** (konkrét adatok, megvalósulások)
- **Fizikai, logikai, alkalmazói réteg:**

| | Séma | Egy előfordulás | | | | | | |
|--------------------------|--------------------------------------|--|------------|------------|------|----|------|----|
| Alkalmazások | Select sum(fiz) as összfiz from Bér; | 30 | | | | | | |
| Logikai adatbázis | Bér(név, fiz) | <table><thead><tr><th><u>név</u></th><th><u>fiz</u></th></tr></thead><tbody><tr><td>Kiss</td><td>10</td></tr><tr><td>Nagy</td><td>20</td></tr></tbody></table> | <u>név</u> | <u>fiz</u> | Kiss | 10 | Nagy | 20 |
| <u>név</u> | <u>fiz</u> | | | | | | | |
| Kiss | 10 | | | | | | | |
| Nagy | 20 | | | | | | | |
| Fizikai adatbázis | szekvenciális | (Bér,név,fiz,#2,Kiss,10,Nagy,20) | | | | | | |



Adatbázisok különböző szintjei

- **Fizikai adatfüggetlenség**
 - Fizikai adatbázis módosítása (indexek készítése, az adatok más adatstruktúrákban tárolása) nem látszik a felette levő szinteken
 - Hatékonyság növelhető jobb tárolási struktúrákkal
- **Logikai adatfüggetlenség**
 - A logikai adatbázis **bővítése** (új táblák, oszlopok hozzáadása) esetén a régi alkalmazások változtatás nélkül ugyanúgy működjenek

A relációs adatmodell

Edgar Frank Codd 12 szabálya

1. Az egységes megjelenésű információ szabálya

Az adatbázisban szereplő összes információt egy, és csak egy megadott formában (adatmodellben) lehet ábrázolni, nevezetesen táblázatok sorainak oszlopértékeiben.

2. Garantált lokalizálhatóság szabálya

Az adatbázisban minden egyes skaláris értékre logikailag úgy kell hivatkozni, hogy megadjuk az azt tartalmazó táblázat és az oszlop nevét, valamint a megfelelő sor elsődleges kulcsának az értékét.

3. A NULL értékek egységes kezelése

Az adatbázis-kezelő rendszernek (DBMS) olyan egységes módszerrel kell támogatnia a hiányzó vagy nem ismert információ kezelését, amely eltér az összes „rendes” érték kezelésétől, továbbá független az adattípustól.

4. A relációs modell alapján aktív online katalógust kell üzemen tartani

A rendszernek támogatnia kell egy online, beépített katalógust, amelyet a feljogosított felhasználók a lekérdező nyelv segítségével ugyanúgy le tudnak kérdezni, mint a közönséges táblákat.

5. A teljes körű „adatnyelv” szabálya

A rendszernek legalább egy olyan relációs nyelvet kell támogatnia, amelynek

- (a) lineáris a szintaxisa,
- (b) interaktívan és az alkalmazásokhoz készített programokon belül is lehet használni,
- (c) támogatja az adatdefiniáló műveleteket, a visszakereső és adatmódosító (manipulációs) műveleteket, biztonsági és jósági (integritási) korlátokat, valamint a tranzakciókezelési műveleteket (begin, commit, rollback: elkezdés, jóváhagyás és visszatörlesztés).

6. A nézetek frissítésének szabálya

A rendszernek képesnek kell lennie az adatok összes nézetének frissítésére.

A relációs adatmodell

Edgar Frank Codd 12 szabálya

7. Magas szintű beszúrás, frissítés és törlés

A rendszernek támogatnia kell az INSERT, UPDATE, és DELETE (új adat, módosítás, törlés) operátorok halmaz szintű, egyidejű működését.

8. Fizikai szintű adatfüggetlenség

A fizikai adatfüggetlenség akkor áll fenn, ha az alkalmazások (programok) és a felhasználók adatelérési módja független az adatok tényleges (fizikai) tárolási és elérési módjától.

9. Logikai szintű adatfüggetlenség

Logikai adatfüggetlenség akkor áll fenn, ha az adatbázis logikai szerkezetének bővítése nem igényli az adatbázist használó alkalmazások (programok) megváltoztatását.

10. Jóság (integritás) függetlenség

Az adatok jóságának (érvényességének) korlátait az adatfeldolgozási programuktól függetlenül kell tudni meghatározni, és azokat katalógusban kell nyilvántartani. Legyen lehetséges a szóban forgó korlátokat megváltoztatni, anélkül hogy a meglévő alkalmazásokon változtatni kelljen.

11. Elosztástól való függetlenség

A meglévő alkalmazások működése zavartalan kell, hogy maradjon

- (a) amikor sor kerül az adatbázis-kezelő osztott változatának bevezetésére
- (b) amikor a meglévő osztott adatokat a rendszer újra szétosztja.

12. Megkerülhetetlenség szabálya

Ha a rendszernek van egy alacsony szintű (egyszerre egy rekordot érintő) interfésze, akkor ezt az interfészt ne lehessen a rendszer megkerülésére használni, például a relációs biztonsági vagy jósági (integritás védelmi) korlátok megsértésével.

Relációs adatmodell

ABR1 3. fejezet (104.- 110. oldal)

ABR1 4. fejezet (196.- 215. oldal)

- **Relációséma:** $R(A_1, A_2, \dots, A_n)$
 - R – relációnév
 - A_i – attribútum- vagy tulajdonságnevek, oszlopnevek
 - $\text{Dom}(A_i)$ – lehetséges értékek halmaza, típusa
 - Egy sémán belül az attribútumok különbözőek
- **Reláció-előfordulás:** $r \subseteq X^n$
 - r - reláció, tábla, sorhalmaz
 - Egy sor egyszer szerepel
 - Sorok sorrendje lényegtelen
 - Oszlopok sorrendje lényegtelen

Relációs adatmodell

- **Jelölések**
 - $t \in r$ esetén t sor (angolul: tuple – n-es)
 - $t(A_i)$ vagy $t($i)$ – a t sor i -edik komponense
 - $t[A_{i1}, \dots, A_{ik}]$ - a t sor i_1, \dots, i_k -adik komponenseiből álló vektor
- Különböző sémák azonos attribútumai esetén
 - $R.A$ – prefixszel különböztetjük meg
- Egy t sor függvénynek is tekinthető

$$t: \{A_1, \dots, A_n\} \rightarrow \bigcup_{i=1}^n \text{Dom}(A_i) \text{ ahol } t(A_i) \in \text{Dom}(A_i), i=1..n$$

Bér

Példa

| név | fiz | kor | |
|------------|------------|------------|----|
| Kiss | 10 | 35 | t1 |
| Nagy | 20 | 45 | t2 |
| Kovács | 15 | 22 | t3 |

$t1(név)=\text{„Kiss”}$

$t3(\$3)=22$

$t2(név, kor)=(\text{„Nagy”}, 45)$

$t1(\text{Bér.fiz})=10$

SQL lekérdezések felbontása: Relációs algebra

- Az SQL nyelvben összetett, több táblás, alkérdezéseket is tartalmazó lekérdezéseket lehet megfogalmazni.
- Hogyan lehetne egyszerű SQL lekérdezésekből felépíteni az összetett SQL lekérdezéseket?
- Miért jó egy ilyen felbontás?
 - Áttekinthetőbbé válik az összetett lekérdezés.
 - Az egyszerű lekérdezések kiszámítási költségét könnyebb kifejezni, így segít az optimalizálásban.
- Melyek legyenek az egyszerű SQL lekérdezések?
 - Legyenek közöttük egyszerű kiválasztásra épülő SQL lekérdezések.
 - Legyenek közöttük többtáblás lekérdezések.
 - Halmazműveleteket lehessen használni.
 - Lehessen átnevezni táblákat, oszlopokat.
 - Lehessen egy lekérdezés eredményét egy másik lekérdezésben felhasználni (nézettáblák view-k)

Egyesítés, unió

1. **select * from r union select * from s;**

- r, s és $r \cup s$ azonos sémájú
- $r \cup s := \{t \mid t \in r \text{ vagy } t \in s\}$
- $|r \cup s| \leq |r| + |s|$, ahol $|r|$ az r reláció sorainak száma
- azonos sor csak egyszer szerepelhet

| A | B |
|---|---|
| 0 | 0 |
| 0 | 1 |

∪

| A | B |
|---|---|
| 0 | 0 |
| 1 | 0 |

=

| A | B |
|---|---|
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |

Kivonás, különbség

2. **select * from r minus select * from s;**

- r, s és $r - s$ azonos sémájú
- $r - s := \{ t \mid t \in r \text{ és } t \notin s\}$
- $|r - s| \leq |r|$

The diagram shows three tables representing sets. The first table has columns A and B, rows 0 and 1, and values 0 and 1 respectively. The second table has columns A and B, rows 0 and 1, and values 0 and 0 respectively. The third table has columns A and B, rows 0 and 1, and values 0 and 1 respectively. A minus sign between the first two tables indicates set subtraction, resulting in the third table.

| A | B |
|---|---|
| 0 | 0 |
| 0 | 1 |

–

| A | B |
|---|---|
| 0 | 0 |
| 1 | 0 |

=

| A | B |
|---|---|
| 0 | 1 |

select * from r minus select * from s;

VAGY

select * from r where not exists

(select * from s where r.A=s.A and r.B=s.B);



Szorzás, direktszorzat vagy Descartes-szorzat

3. select * from r,s;

- r, s sémáiban nincs közös attribútum
- $r \times s$ sémája a sémkák egyesítése
- $r \times s := \{ t \mid t[R] \in r \text{ és } t[S] \in s \}$
- $|r \times s| = |r| * |s|$

| | |
|---|---|
| A | B |
| 0 | 0 |
| 0 | 1 |

 \times

| | |
|---|---|
| C | D |
| 0 | 0 |
| 1 | 0 |

 $=$

| | | | |
|---|---|---|---|
| A | B | C | D |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |

Vetítés, projekció

4. select distinct A1,...,Ak from r;

- $X \subseteq \{A_1, \dots, A_n\}$
- $\Pi_X(r)$ sémája X
- $\Pi_X(r) := \{ t \mid \text{van olyan } t' \in r, \text{ melyre } t'[X] = t \}$
- $|\Pi_X(r)| \leq |r|$

select distinct B,D from r;

r:

| A | B | C | D |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |

$$\Pi_{BD}(r) =$$

| B | D |
|---|---|
| 0 | 0 |
| 1 | 0 |

select distinct D,A from r;

$$\Pi_{DA}(r) =$$

| D | A |
|---|---|
| 0 | 0 |

Kiválasztások

5. select * from r where A=B;
select * from r where A<B;
select * from r where A>B;
select * from r where A<>B;
select * from r where A<=B;
select * from r where A>=B;

select * from r where A=konstans;
select * from r where A<konstans;
select * from r where A>konstans;
select * from r where A<>konstans;
select * from r where A<=konstans;
select * from r where A>=konstans;

select * from r where feltétel1 **and** feltétel2;
select * from r where feltétel1 **or** feltétel2;
select * from r where **not** (feltétel);

Kiválasztás, szűrés, szelekció

- $\sigma_F(r)$ és r sémája megegyezik
- $\sigma_F(r) := \{ t \mid t \in r \text{ és } F(t) = \text{IGAZ} \}$
- **F feltétel:**
 - atomi, elemi feltétel
 - $A_i \Theta A_j$, ahol $\Theta \in \{ =, \neq, <, >, \leq, \geq \}$
 - $A_i \Theta c, c \Theta A_i$ ahol c egy konstans
 - feltételekből \wedge, \vee, \neg logikai összekapcsolókkal, és zárójelekkel kapható kifejezés

| | A | B | C | D |
|----|---|---|---|---|
| r: | 0 | 0 | 0 | 0 |
| | 0 | 0 | 1 | 0 |
| | 0 | 1 | 0 | 0 |

$$\sigma_{A=C \wedge \neg(B < 1)}(r) =$$

| | A | B | C | D |
|--|---|---|---|---|
| | 0 | 1 | 0 | 0 |
| | | | | |

select * from r where A=B and not (B<1); 

Kiválasztás, szűrés, szelekció

- $|\sigma_F(r)| \leq |r|$
- a feltételben függvények nem használhatók:
 - $\sigma_{A+B < 5}(r)$ nem megengedett
- az összetett feltételek átírhatók elemi feltételeket használó kifejezésekkel a következő szabályok segítségével:
 - $\sigma_{F_1 \wedge F_2}(r) \cong \sigma_{F_1}(\sigma_{F_2}(r)) \cong \sigma_{F_2}(\sigma_{F_1}(r))$
 - $\sigma_{F_1 \vee F_2}(r) \cong \sigma_{F_1}(r) \cup \sigma_{F_2}(r)$
 - A De Morgan azonosság segítségével a negáció beljebb vihető:
 - $\neg(F_1 \wedge F_2)$ helyett $(\neg F_1) \vee (\neg F_2)$
 - $\neg(F_1 \vee F_2)$ helyett $(\neg F_1) \wedge (\neg F_2)$
 - elemi feltétel tagadása helyett a fordított összehasonlítást használjuk:
 - például $\neg(A < B)$ helyett $(A \geq B)$

Kiválasztás, szűrés, szelekció

$$\sigma_{(\neg(A = C) \wedge \neg(B < 1)) \wedge (D < 2)}(r) =$$

$$\sigma_{(\neg(A = C) \vee \neg(\neg(B < 1)) \wedge (D < 2)}(r) =$$

$$\sigma_{A \neq C}(\sigma_{D < 2}(r)) \cup \sigma_{B < 1}(\sigma_{D < 2}(r))$$

- az elemi feltételekhez lekérdezést gyorsító adatszerkezetek, indexek készíthetők

Átnevezés

6. select oszlop [AS] újnév,... from r [AS] újnév;

- A relációnak és az attribútumoknak új nevet adhatunk.
- Ha r sémája $R(A_1, \dots, A_n)$, akkor $\rho_{S(B_1, \dots, B_n)}(r)$ sémája $S(B_1, \dots, B_n)$.
- $|\rho_{S(B_1, \dots, B_n)}(r)| = |r|$ $\rho_{MUNKA(dolg, jöv)}(r) =$.

| BÉR | név | fiz |
|-----|------|-----|
| r: | Kiss | 10 |
| | Nagy | 20 |

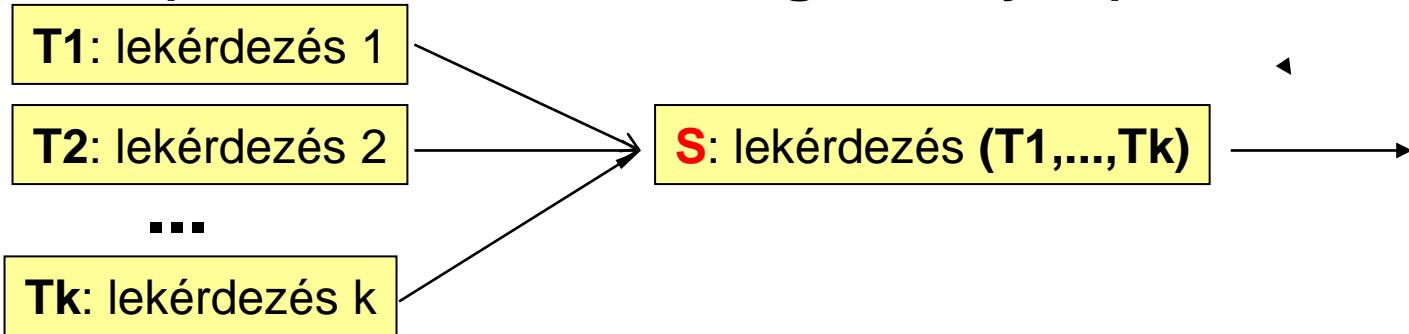
| MUNKA | dolg | jöv |
|-------|------|-----|
| | Kiss | 10 |
| | Nagy | 20 |

select név dolg, fiz jöv from BÉR MUNKA;



Kifejezések kompozíciója

- Az egyszerű SQL lekérdezésekből hogyan lehet felépíteni összetett lekérdezéseket?
- Az SQL lekérdezés eredménye SQL tábla.
- Készítsünk nézettáblát (VIEW) a részlekérdezéshez.
- Az SQL lekérdezés FROM listájában nézettáblák is használhatók. (A nézettábla nem foglal helyet.)



```
create view T1 as select ... from ... where ... ;
```

```
create view T2 as select ... from ... where ... ;
```

```
...
```

```
create view Tk as select ... from ... where ... ;
```

```
create view S as select ... from T1,...,Tk where ... ;
```

Relációs algebra

- **ÖSSZEFOGLALVA:**
 - **Alapoperátorok:**
 1. Egyesítés
 2. Különbség
 3. Szorzat
 4. Vetítés
 5. Kiválasztás
 6. Átnevezés
 - **Kifejezés:**
 - konstans reláció
 - relációs változó
 - alapoperátorok véges sok alkalmazása kifejezésekre
 - ezek és csak ezek
 - **Relációs algebra = kifejezések halmaza**
1. `select * from r union select * from s;`
 2. `select * from r minus select * from s;`
 3. `select * from r,s;`
 4. `select distinct A1,...,Ak from r;`
 5. `select * from r where feltétel;`
 6. `select oszlop [AS] újnév,... from r [AS] újnév;`
- `create view T1 as select ... from ... where ... ;`

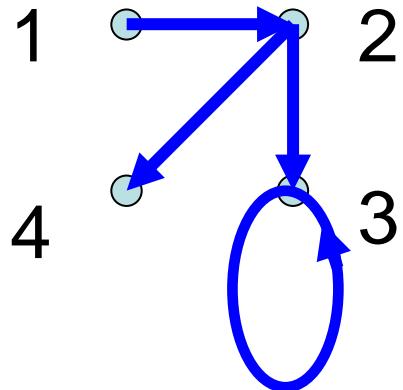
....

`create view Tk as select ... from ... where ... ;`

`create view S as select ... from T1,...,Tk where ... ;`

A relációs algebra kifejezőereje

- Relációs algebrában a legfontosabb lekérdezéseket ki tudjuk fejezni, de nem minden!
- ÉL(honnan, hova)
- ÚT(honnan, hova) – tranzitív lezárás



| honnan | hova |
|--------|------|
| 1 | 2 |
| 2 | 4 |
| 2 | 3 |
| 3 | 3 |

ÚT

| honnan | hova |
|--------|------|
| 1 | 2 |
| 2 | 4 |
| 2 | 3 |
| 3 | 3 |
| 1 | 3 |
| 1 | 4 |

- nem triviális rekurzió
- TÉTEL: Nem létezik olyan relációs algebrai kifejezés, amelyet tetszőleges ÉL táblára alkalmazva a neki megfelelő ÚT táblát eredményezi.

Származtatott műveletek

- A gyakran használt kifejezések helyett új műveleteket vezetünk be.
- Nem alapműveletek, hanem származtatottak
- **Metszet**
 - $r \cap s = \{ t \mid t \in r \text{ és } t \in s \}$ `select * from r intersect select * from s;`
 - többféleképpen kifejezhető relációs algebrában:
 - $r \cap s = r - (r - s) = s - (s - r) = r \cup s - ((r - s) \cup (s - r))$
- **Összekapcsolások (JOIN)**
 - Téta-összekapcsolás (Θ -join)
 - Egyen-összekapcsolás (equi-join)
 - Természetes összekapcsolás (natural join)
 - Félig-összekapcsolás (semi-join)
 - Külső összekapcsolás (outer join)
- A szorzáshoz hasonlóan költséges műveletek, nagy méretű táblákat eredményezhetnek, kivételt képez a félig-összekapcsolás.

Téta-összekapcsolás

select * from r,s where r.Ai összehasonlítás s.Bj;

- r, s sémáiban ($R(A_1, \dots, A_n)$, $S(B_1, \dots, B_n)$) nincs közös attribútum
- $r \bowtie s = \sigma_{\begin{array}{c} Ai \Theta Bj \\ Ai \Theta Bj \end{array}}(r \times s)$

select * from r,s where r.B=s.C;

| A | B |
|---|---|
| 0 | 0 |
| 0 | 1 |

\bowtie
 $B=C$

| C | D |
|---|---|
| 0 | 0 |
| 0 | 1 |

=

| A | B | C | D |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |

- $Ai=Bj$ feltétel esetén **egyen-összekapcsolásnak** hívjuk.

Természetes összekapcsolás

```
select distinct R.A1,...,R.An,R.B1,...,R.Bk,S.C1,...,S.Cm from r,s  
where R.B1=S.B1 and R.B2=S.B2 and ... and R.Bk=S.Bk;
```

- r, s sémái $R(A_1, \dots, A_n, B_1, \dots, B_k)$, illetve $S(B_1, \dots, B_k, C_1, \dots, C_m)$
- $r \bowtie s =$

$$\rho_{P(A_1, \dots, A_n, B_1, \dots, B_k, C_1, \dots, C_m)} \Pi_{A_1, \dots, A_n, R.B_1, \dots, R.B_k, C_1, \dots, C_m} \sigma_{R.B_1=S.B_1 \wedge \dots \wedge R.B_k=S.B_k} (r \times s)$$

| A | B |
|---|---|
| 0 | 0 |
| 2 | 1 |
| 1 | 2 |



| B | C |
|---|---|
| 0 | 0 |
| 0 | 2 |
| 1 | 3 |
| 4 | 3 |

 $=$

| A | B | C |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 2 |
| 2 | 1 | 3 |

```
select distinct A,R.B,C  
from r,s where R.B=S.B;
```

Félig-összekapcsolás

select distinct R.A1,...,R.An,R.B1,...,R.Bk from r,s

where R.B1=S.B1 and R.B2=S.B2 and ... and R.Bk=S.Bk;

- r, s sémái R(A₁,...,A_n,B₁,...,B_k), illetve S(B₁,...,B_k,C₁,...,C_m)
- $r \times s = \rho_{P(A_1, \dots, A_n, B_1, \dots, B_k)} \Pi_{A_1, \dots, A_n, R.B_1, \dots, R.B_k} (r| \times |s)$
- Az első relációban mely sorokhoz létezik kapcsolható sor a második táblából

| A | B |
|---|---|
| 0 | 0 |
| 2 | 1 |
| 1 | 2 |

$$\begin{array}{c} \text{B} \quad \text{C} \\ \hline 0 & 0 \\ 0 & 2 \\ 1 & 3 \\ 4 & 3 \end{array} \times \begin{array}{c} \text{A} \quad \text{B} \\ \hline 0 & 0 \\ 2 & 1 \end{array} = \begin{array}{c} \text{A} \quad \text{B} \\ \hline 0 & 0 \\ 2 & 1 \end{array}$$

select distinct A,R.B
from r,s where R.B=S.B;

Külső összekapcsolás

```
select A,r.B,C from r outer join s on r.B=s.B;
```

- Nem relációs algebrai művelet, mert kilép a modellből
- r, s sémái $R(A_1, \dots, A_n, B_1, \dots, B_k)$, illetve $S(B_1, \dots, B_k, C_1, \dots, C_m)$
- $r \bowtie^o s = r \bowtie s$ relációt kiegészítjük az r és s soraival, a hiányzó helyekre NULL értéket írva

| A | B |
|---|---|
| 0 | 0 |
| 2 | 1 |
| 1 | 2 |

\bowtie^o

| B | C |
|---|---|
| 0 | 0 |
| 0 | 2 |
| 1 | 3 |
| 4 | 3 |

=

| A | B | C |
|------|---|------|
| 0 | 0 | 0 |
| 0 | 0 | 2 |
| 2 | 1 | 3 |
| 1 | 2 | NULL |
| NULL | 4 | 3 |

Összekapcsolások

- Ha r, s **sémáit megegyeznek**, akkor $r| \times |s = r \cap s$.
- Ha r, s sémáiban **nincs közös attribútum**, akkor $r| \times |s = r \times s$.
- Ha $r = \emptyset$, akkor $r \times s = \emptyset$ és $r| \times |s = \emptyset$.
- A külső összekapcsolás lehet bal oldali, ha csak r sorait vesszük hozzá a természetes összekapcsoláshoz: $r| \overset{o}{\times} |_B s$. Hasonlóan értelmezhetjük a jobb oldali összekapcsolást is $r| \overset{o}{\times} |_J s$.

```
select A,r.B,C from r left outer join s on r.B=s.B;  
vagy select A,r.B,C from r,s where r.B = s.B(+);
```

```
select A,r.B,C from r right outer join s on r.B=s.B;  
vagy select A,r.B,C from r,s where r.B(+) = s.B;
```

Osztás, hányados

- Maradékos osztás: $7 \div 3 = 2$, mert 2 a legnagyobb egész, amelyre még $2 * 3 \leq 7$.
- Relációk szorzata esetén \leq helyett tartalmazás.
- r és s sémája $R(A_1, \dots, A_n, B_1, \dots, B_m)$, illetve $S(B_1, \dots, B_m)$, $r \div s$ sémája $R(A_1, \dots, A_n)$
- $r \div s$ a legnagyobb (**legtöbb sort tartalmazó**) reláció, amelyre $(r \div s) \times s \subseteq r$.
- Kifejezhető relációs algebrában:
- $\Pi_{A_1, \dots, A_n}(r) - \Pi_{A_1, \dots, A_n}(\Pi_{A_1, \dots, A_n}(r) \times s - r)$
- Lehetséges értékekből kivonjuk a rossz értékeket.
- $(p \times r) \div r = p$

Osztás, hányados

- Ki szereti legalább azokat, mint Micimackó?

| KI | MIT |
|-----------|--------|
| Füles | málna |
| Füles | méz |
| Füles | alma |
| Micimackó | málna |
| Micimackó | méz |
| Kanga | málna |
| Kanga | körte |
| Nyuszi | lekvár |

÷

| | | |
|-----|-------|-----|
| MIT | málna | méz |
|-----|-------|-----|

=

| | | |
|----|-------|-----------|
| KI | Füles | Micimackó |
|----|-------|-----------|

szeret $\div \prod_{\text{MIT}} (\sigma_{\text{KI}=\text{'Micimackó'}}(\text{szeret}))$



$r(a,b) \div s(b)$ hányados kifejezése SQL-ben (MINUS segítségével):

- $r(a,b) \div s(b) = \Pi_a(r) - \Pi_a(\Pi_a(r) \times s - r)$

- $\Pi_a(r) \times s = \Pi_{r,a,s,b}(r \times s)$

- **select distinct r.a,s.b from r,s;**

- $\Pi_a(r) \times s - r$

- **create view rsz as**
select distinct r.a,s.b from r, s
 minus

- select * from r;**

- $\Pi_a(\Pi_a(r) \times s - r)$

- **select distinct a from rsz;**

- $\Pi_a(r) - \Pi_a(\Pi_a(r) \times s - r)$

- **select distinct a from r**
 minus

- select distinct a from rsz;**

- **$r(a,b) \div s(b):$**

- $\Pi_a(r) - \Pi_a(\Pi_a(r) \times s - r)$

- **create view rsz as**

- select distinct r.a,s.b from r, s**
 minus

- select * from r;**

- **select distinct a from r**
 minus

- select distinct a from rsz;**

$r(a,b) \div s(b)$ hányados kifejezése SQL-ben (NOT EXISTS segítségével):

- $r(a,b) \div s(b) = \Pi_a(r) - \Pi_a(\Pi_a(r) \times s - r)$
- $\Pi_a(r) \times s = \Pi_{r,a,s,b}(r \times s)$
- **select distinct r.a,s.b from r,s;**
- $\Pi_a(r) \times s - r$
- **select distinct r.a,s.b from r r1, s s1
where not exists**
**(select * from r r2
where r2.a=r1.a and s1.b=r2.b);**
- $\Pi_a(\Pi_a(r) \times s - r)$
- **select distinct r.a from r r1, s s1
where not exists**
**(select * from r r2
where r2.a=r1.a and s1.b=r2.b);**

• $\Pi_a(r) - \Pi_a(\Pi_a(r) \times s - r)$

• **select distinct r2.a from r r2
where not exists**

**(select * from r r1, s s1
where r2.a=r1.a and
not exists**

**(select * from r r3
where r3.a=r1.a
and s1.b=r3.b));**



Monotonitás

- **Monoton nem csökkenő** (röviden **monoton**)
kifejezés: bővebb relációra alkalmazva az eredmény is bővebb:
Ha $R_i \subseteq S_i$, $i=1, \dots, n$, akkor $E(R_1, \dots, R_n) \subseteq E(S_1, \dots, S_n)$.
- A kivonás kivétel az alapműveletek monoton műveletek (monoton relációs algebra).

| A | B |
|---|---|
| 0 | 1 |
| 0 | 0 |

-

| A | B |
|---|---|
| 0 | 1 |
| 0 | 1 |

~~\subseteq~~

| A | B |
|---|---|
| 0 | 1 |
| 0 | 0 |

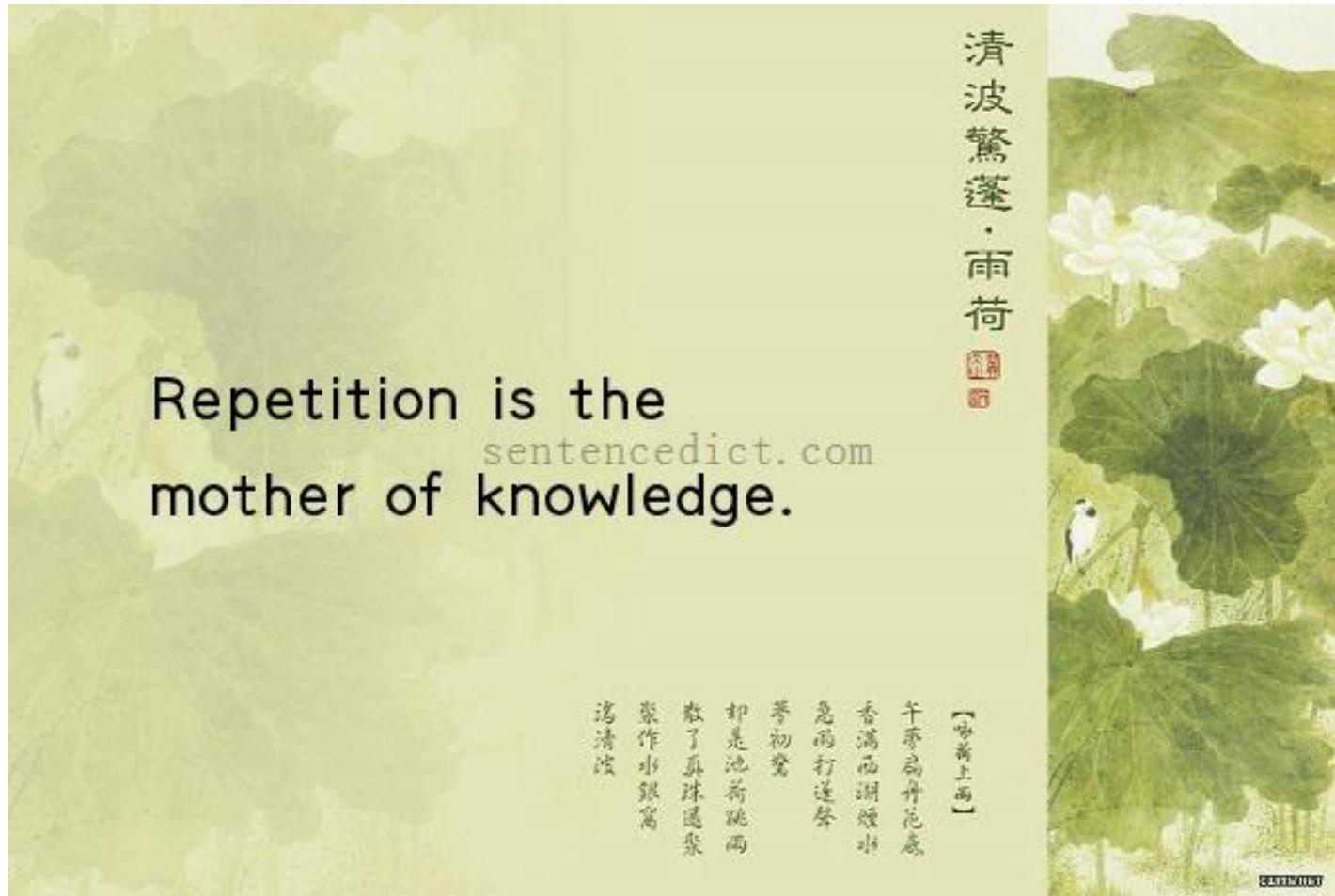
-

| A | B |
|---|---|
| 0 | 1 |
| 0 | 0 |

Monotonitás

- **DE:** Monoton kifejezésben is szerepelhet kivonás: $r \cap s = r - (r - s)$ monoton.
- Ha E, E_1, E_k monoton kifejezések, és $E(E_1(\dots), \dots, E_k(\dots))$ helyes kifejezés, akkor monoton is.
- **Következmény:** A kivonás nem fejezhető ki a többi alapművelettel.

Ismétlés vége



Relációs algebrai optimalizálás

- A lekérdezések optimalizálásának folyamata
- Az algebrai optimalizálás szerepe, célja
- 11 ekvivalencia szabálytípus
- Az algebrai optimalizálás heurisztikus megadása
- Az algoritmus formálisan
- Példa

Lekérdezések optimalizálása

CÉL: A lekérdezéseket gyorsabbá akarjuk tenni a táblákra vonatkozó paraméterek, statisztikák, indexek ismeretében és általános érvényű tulajdonságok, heurisztikák segítségével.

Például, hogyan, milyen procedúrával értékeljük ki az alábbi SQL (deklaratív) lekérdezést?

Legyen adott R(A,B,C) és S(C,D,E). Melyek azok az R.B és S.D értékek azokban az R, illetve S táblabeli sorokban, amely sorokban R.A='c' és S.E=2 és R.C=S.C?

Ugyanez SQL-ben:

Select B,D

From R,S

Where R.A = 'c' and S.E = 2 and R.C=S.C;

Lekérdezések optimalizálása



Lekérdezések optimalizálása

| R | A | B | C | S | C | D | E |
|---|---|----|---|----|---|---|---|
| a | 1 | 10 | | 10 | x | 2 | |
| b | 1 | 20 | | 20 | y | 2 | |
| c | 2 | 10 | | 30 | z | 2 | |
| d | 2 | 35 | | 40 | x | 1 | |
| e | 3 | 45 | | 50 | y | 3 | |

A lekérdezés eredménye:

| B | D |
|---|---|
| 2 | x |

Select B,D From R,S
Where R.A = 'c' and
S.E = 2 and
R.C=S.C;

Lekérdezések optimalizálása

Hogy számoljuk ki tetszőleges tábla esetén az eredményt?

Egy lehetséges terv

- Vegyük a két tábla szorzatát!
- Válasszuk ki a megfelelő sorokat!
- Hajtsuk végre a vetítést!

$\Pi_{B,D} [\sigma_{R.A='c' \wedge S.E=2 \wedge R.C = S.C} (RXS)]$

- Ez a direktszorzaton alapuló összekapcsolás.
- Oracleben: NESTED LOOP.
- Nagyon költséges!

RXS

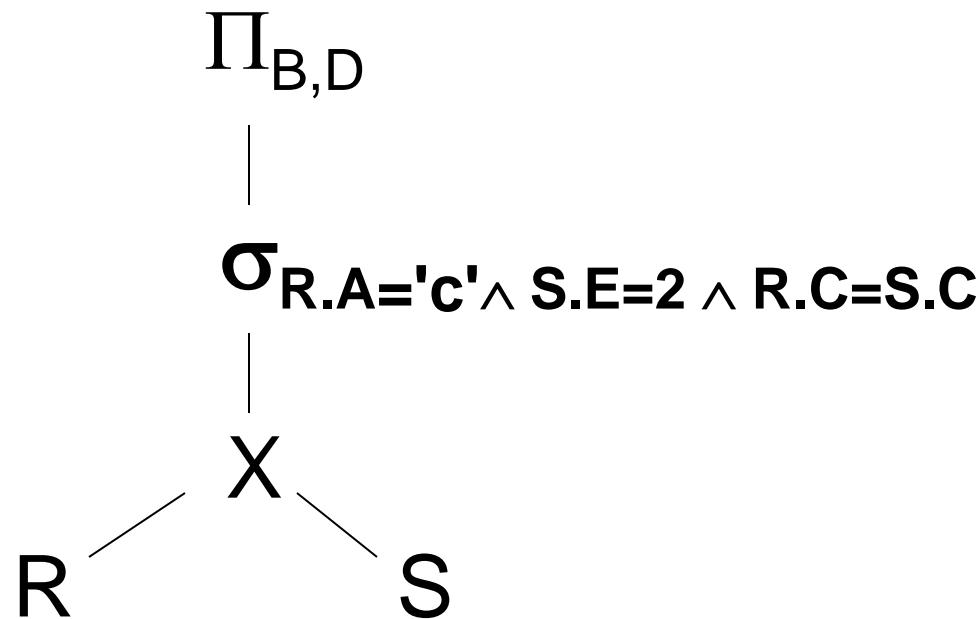
| | R.A | R.B | R.C | S.C | S.D | S.E |
|-------------------|-----|-----|-----|-----|-----|-----|
| | a | 1 | 10 | 10 | x | 2 |
| | a | 1 | 10 | 20 | y | 2 |
| | . | . | . | . | . | . |
| Ez a sor kell! | c | 2 | 10 | 10 | x | 2 |
| | . | . | . | . | . | . |

$$\frac{B}{2} \quad D \quad x$$

$$\Pi_{B,D} [\sigma_{R.A='c' \wedge S.E=2 \wedge R.C = S.C} (RXS)]$$

Lekérdezések optimalizálása

Ugyanez a terv relációs algebrában:



$\Pi_{B,D} [\sigma_{R.A='c' \wedge S.E=2 \wedge R.C = S.C} (RXS)]$

R

Lekérdezések optimalizálása

S

| A | B | C |
|---|---|----|
| a | 1 | 10 |
| b | 1 | 20 |
| c | 2 | 10 |
| d | 2 | 35 |
| e | 3 | 45 |

 $\sigma(R)$

| A | B | C |
|---|---|----|
| c | 2 | 10 |

 $\sigma(S)$

| C | D | E |
|----|---|---|
| 10 | x | 2 |
| 20 | y | 2 |
| 30 | z | 2 |

| C | D | E |
|----|---|---|
| 10 | x | 2 |
| 20 | y | 2 |
| 30 | z | 2 |
| 40 | x | 1 |
| 50 | y | 3 |

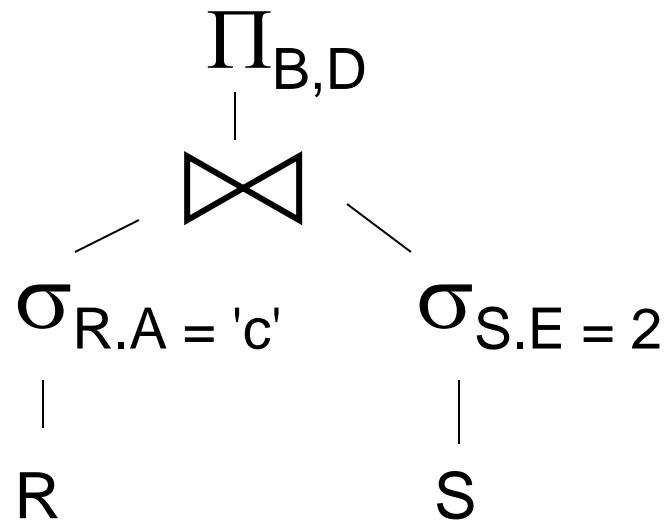
 $\Pi_{B,D}$

Ugyanazt
számolja ki!

| | |
|---|---|
| B | D |
| 2 | x |

Lekérdezések optimalizálása

Egy másik lehetséges kiszámítási javaslat:



$\Pi_{B,D} [\sigma_{R.A='c'} \wedge S.E=2 \wedge R.C = S.C \text{ (RXS)}]$
helyett

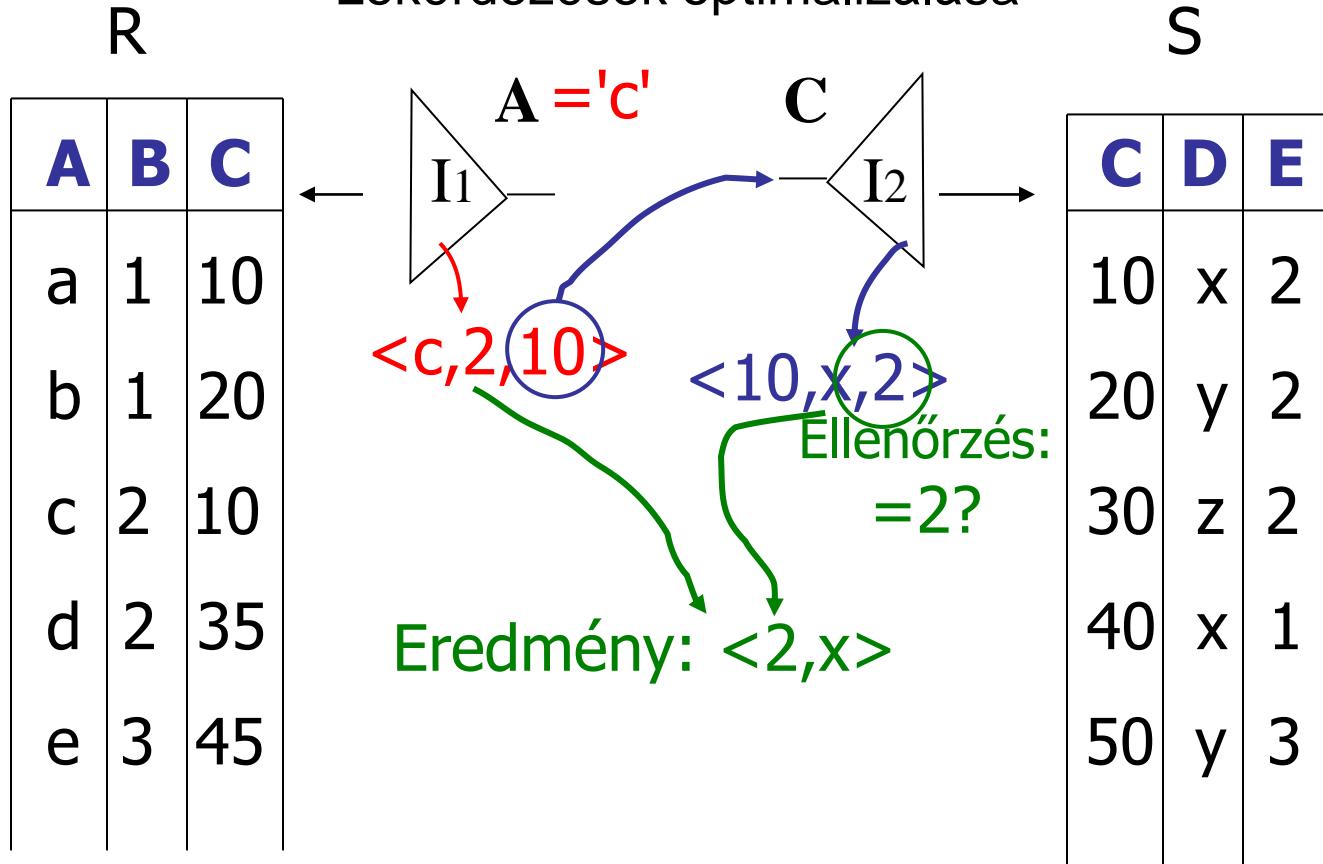
$\Pi_{B,D} [\sigma_{R.A='c'}(R) \times \sigma_{S.E=2}(S)]$

Lekérdezések optimalizálása

Használjuk ki az R.A és S.C oszlopokra készített **indexeket**:

- (1) Az **R.A index alapján keressük meg az R** azon sorait, amelyekre $R.A = 'c'$!
- (2) minden megtalált **R.C értékhez az S.C index alapján keressük meg az S-ből** az ilyen értékű sorokat!
- (3) **Válasszuk ki** a kapott S-beli sorok közül azokat, amelyekre $S.E = 2$!
- (4) **Kapcsoljuk össze** az R és S így kapott sorait, és végül **vetítsünk** a B és D oszlopokra.

Lekérdezések optimalizálása



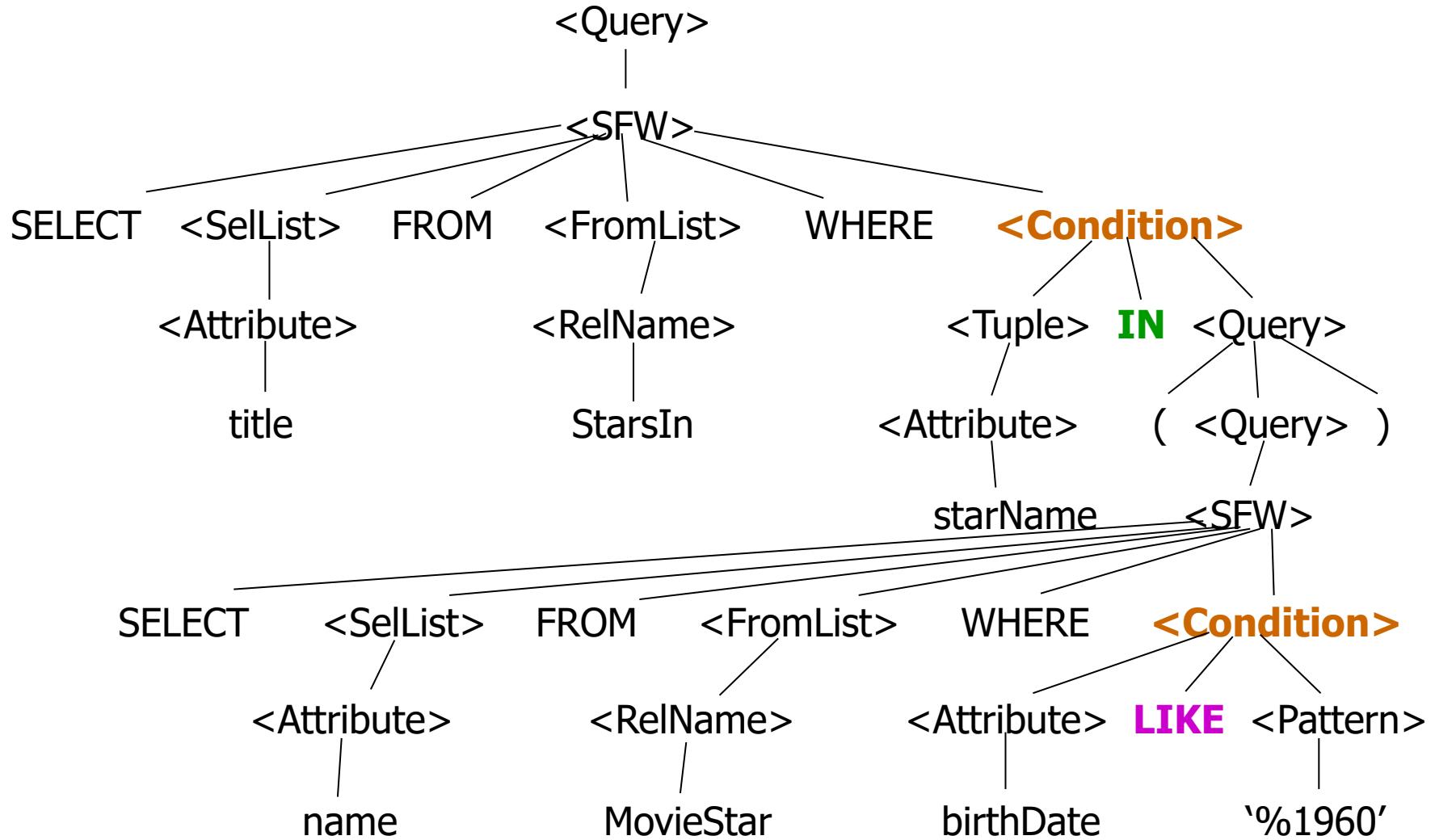
INDEXES ÖSSZEKAPCSOLÁS

Példa: SQL lekérdezés

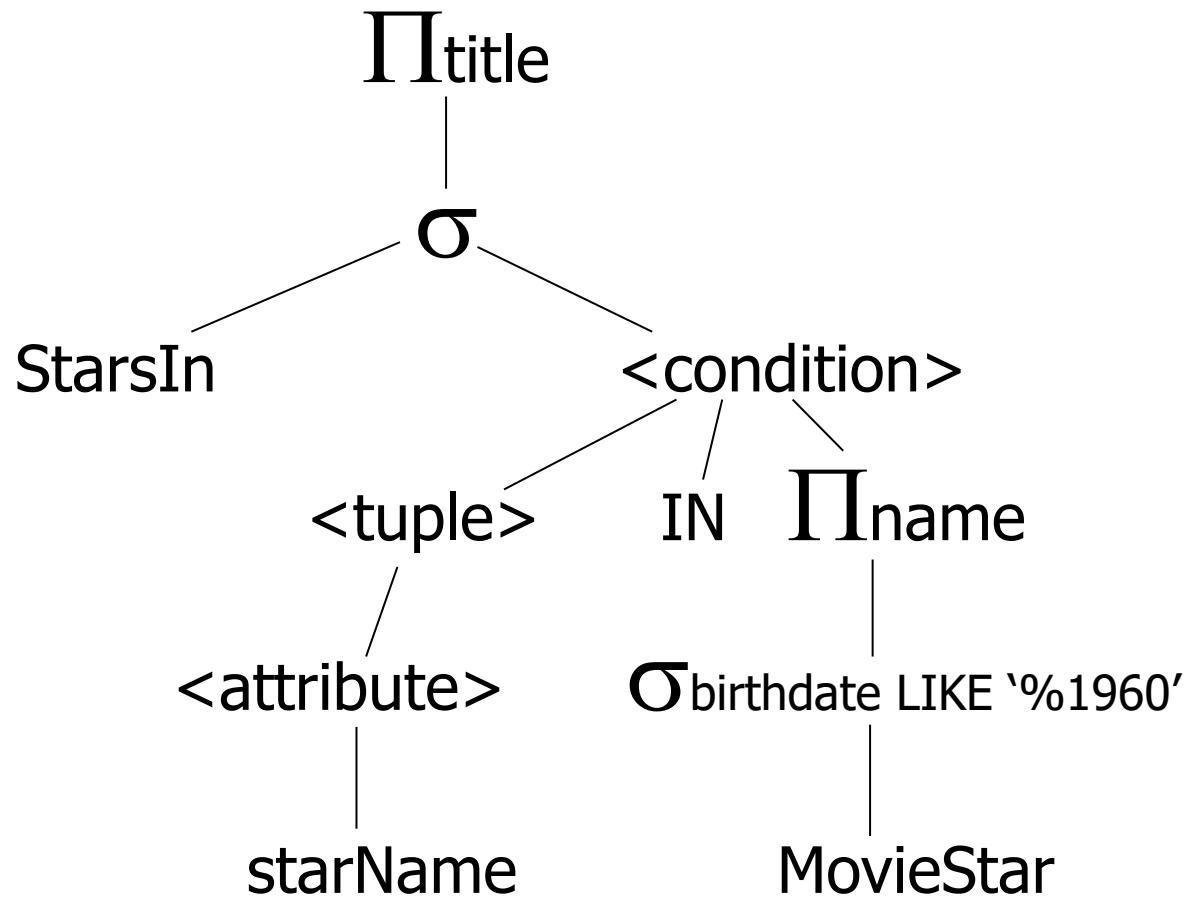
```
SELECT title  
FROM StarsIn  
WHERE starName IN (  
    SELECT name  
    FROM MovieStar  
    WHERE birthdate LIKE '%1960'  
);
```

Milyen filmekben szerepeltek 1960-as születésű színészek?

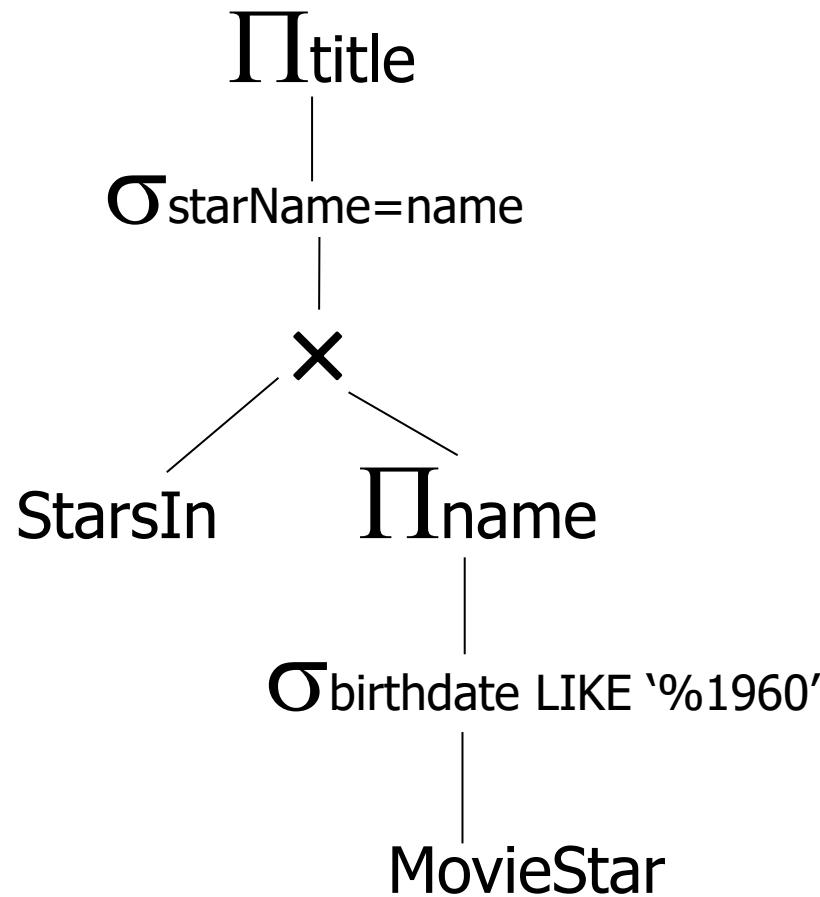
Elemzőfa: Parse Tree



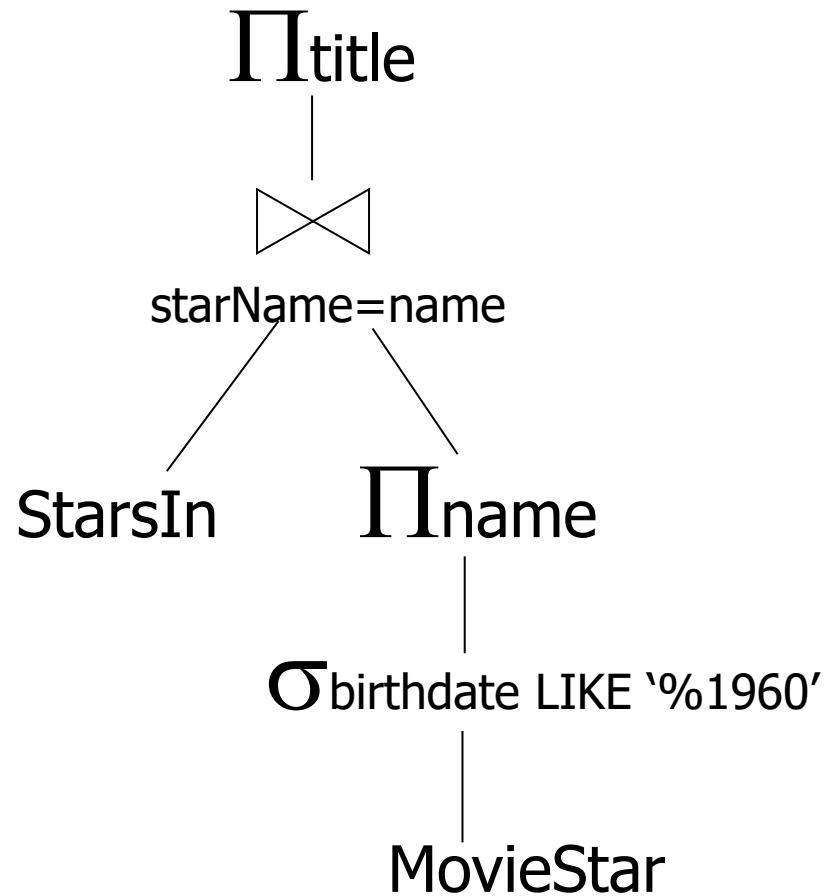
Ugyanez relációs algebrában:



Átalakított logikai lekérdező terv



Továbbjavított logika lekérdező terv



Algebrai optimalizáció

- **Cél:** a relációs algebrai kifejezéseket minél gyorsabban akarjuk kiszámolni.
- **Költségmodell:** a kiszámítás költsége arányos a relációs algebrai kifejezés részkifejezéseinak megfelelő relációk tárolási méreteinek összegével.
- **Módszer:** a műveleti tulajdonságokon alapuló ekvivalens átalakításokat alkalmazunk, hogy várhatóan kisebb méretű relációk keletkezzenek.
- **Az eljárás heurisztikus,** tehát nem az argumentum relációk valódi méretével számol.
- **Az eredmény nem egyértelmű:** Az átalakítások sorrendje nem determinisztikus, így más sorrendben végrehajtva az átalakításokat más végeredményt kaphatunk, de mindegyik általában jobb költségű, mint amiből kiindultunk.
- **Megjegyzés:** Mivel az SQL bővebb, mint a relációs algebra, ezért az optimalizálást bővített relációs algebrára is meg kell adni, de először a hagyományos algebrai kifejezéseket vizsgáljuk.

Algebrai optimalizáció

- A relációs algebrai kifejezést **gráf** ábrázoljuk.
- **Kifejezésfa:**
 - a **nem levél csúcsok**: a relációs algebrai műveletek:
 - unáris (σ, Π, ρ) – egy gyereke van
 - bináris ($,$, \cup , \times) – két gyereke van (bal oldali az első, jobb oldali a második argumentumnak felel meg)
 - a **levél csúcsok**: konstans relációk vagy relációs változók

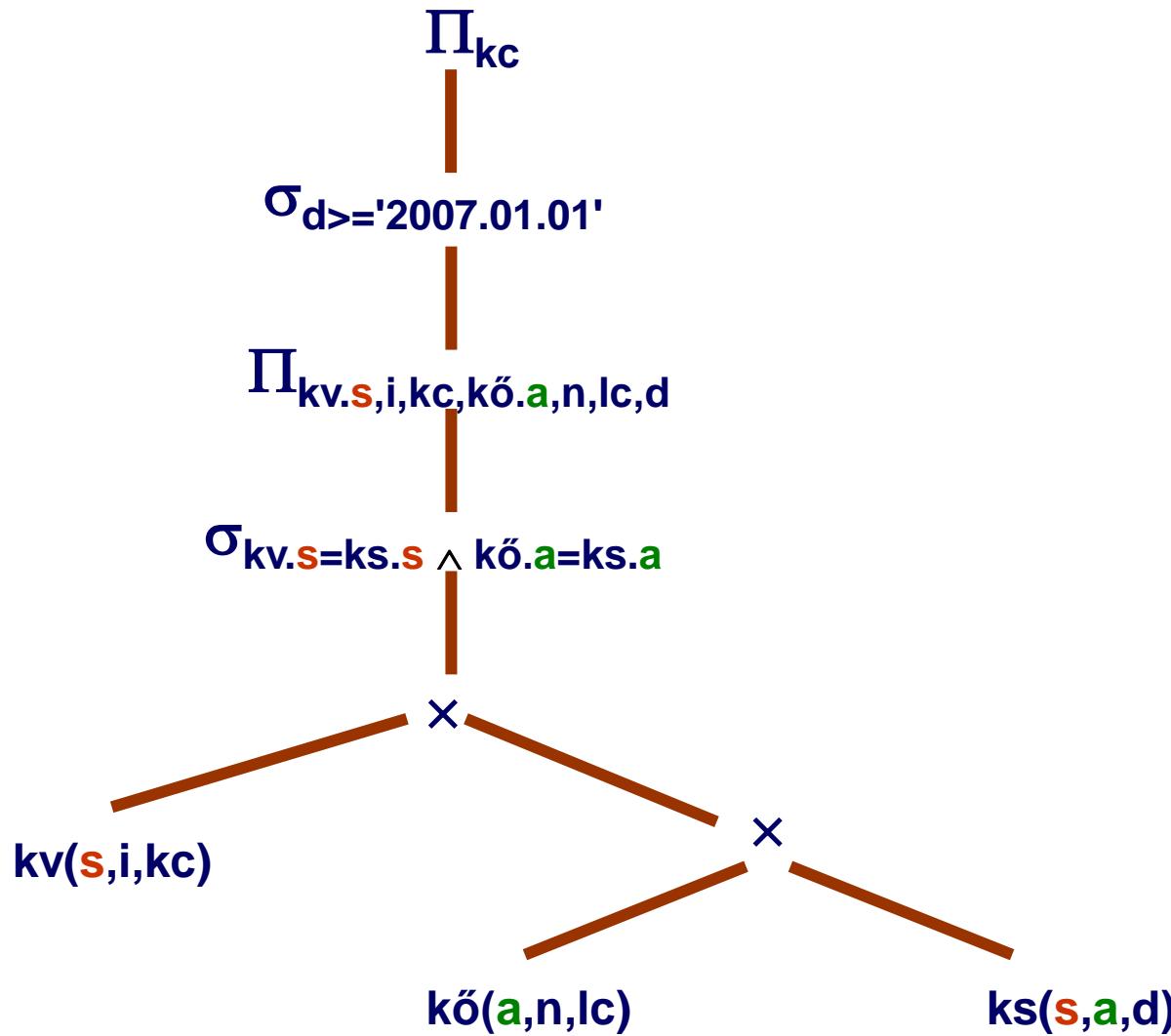
Algebrai optimalizáció

- könyv(sorszám,író,könyvcím)
 - $kv(s,i,kc)$
- kölcsönző(azonosító,név,lakcím)
 - $kő(a,n,lc)$
- kölcsönzés(sorszám,azonosító,dátum)
 - $ks(s,a,d)$
- Milyen című könyveket kölcsönöztek ki 2007-től kezdve?
- $\Pi_{kc}(\sigma_{d \geq '2007.01.01'}(kv \times |kő| \times |ks)))$
- Az összekapcsolásokat valamelyen sorrendben kifejezzük az alapműveletekkel:

$$\Pi_{kc}(\sigma_{d \geq '2007.01.01'}(\Pi_{kv,s,i,kc,kő,a,n,lc,d}(\sigma_{kv.s=ks.s \wedge kő.a=ks.a}(kv \times (kő \times ks))))))$$

Algebrai optimalizáció

$\Pi_{kc}(\sigma_{d \geq '2007.01.01'}(\Pi_{kv.s,i,kc,kō.a,n,lc,d}(\sigma_{kv.s=ks.s \wedge kō.a=ks.a}(kv \times (kō \times ks))))))$



Algebrai optimalizáció

- $E_1(r_1, \dots, r_k)$ és $E_2(r_1, \dots, r_k)$ **relációs algebrai kifejezések ekvivalensek** ($E_1 \cong E_2$), ha tetszőleges r_1, \dots, r_k relációkat véve $E_1(r_1, \dots, r_k) = E_2(r_1, \dots, r_k)$.
- **11 szabályt** adunk meg. A szabályok olyan állítások, amelyek kifejezések ekvivalenciáját fogalmazzák meg. Bizonyításuk könnyen végiggondolható.
- Az állítások egy részében a kifejezések szintaktikus helyessége egyben elégséges feltétele is az ekvivalenciának.

1. Kommutativitás (szorzás, természetes összekapcsolás, téta-összekapcsolás)

- $E_1 \times E_2 \cong E_2 \times E_1$
- $E_1 | \times | E_2 \cong E_2 | \times | E_1$
- $E_1 | \times | E_2 \cong E_2 | \times | E_1$
 Θ Θ

Algebrai optimalizáció

2. Asszociativitás (szorzás, természetes összekapcsolás, téta-összekapcsolás)

- $(E_1 \times E_2) \times E_3 \cong E_1 \times (E_2 \times E_3)$
- $(E_1 | \times | E_2) | \times | E_3 \cong E_1 | \times | (E_2 | \times | E_3)$
- $(E_1 | \underset{\Theta}{\times} | E_2) | \underset{\Theta}{\times} | E_3 \cong E_1 | \underset{\Theta}{\times} | (E_2 | \underset{\Theta}{\times} | E_3)$

3. Vetítések összevonása, bővítése

- Legyen A és B két részhalmaza az E reláció oszlopainak úgy, hogy A \subseteq B.
- Ekkor $\Pi_{\underline{A}}(\Pi_{\underline{B}}(E)) \cong \Pi_{\underline{A}}(E)$.

4. Kiválasztások felcserélhetősége, felbontása

- Legyen F1 és F2 az E reláció oszlopain értelmezett kiválasztási feltétel.
- Ekkor $\sigma_{F_1 \wedge F_2}(E) \cong \sigma_{F_1}(\sigma_{F_2}(E)) \cong \sigma_{F_2}(\sigma_{F_1}(E))$.

Algebrai optimalizáció

5. Kiválasztás és vetítés felcserélhetősége

- Legyen F az E relációnak csak az \underline{A} oszlopain értelmezett kiválasztási feltétel.

- a) • Ekkor $\Pi_{\underline{A}}(\sigma_F(E)) \cong \sigma_F(\Pi_{\underline{A}}(E))$.
- Általánosabban: Legyen F az E relációnak csak az $\underline{A} \cup \underline{B}$ oszlopain értelmezett kiválasztási feltétel, ahol $\underline{A} \cap \underline{B} = \emptyset$.
- b) • Ekkor $\Pi_{\underline{A}}(\sigma_F(E)) \cong \Pi_{\underline{A}}(\sigma_F(\Pi_{\underline{A} \cup \underline{B}}(E)))$.

6. Kiválasztás és szorzás felcserélhetősége

- Legyen F az E_1 reláció oszlopainak egy részhalmazán értelmezett kiválasztási feltétel.

- a) • Ekkor $\sigma_F(E_1 \times E_2) \cong \sigma_F(E_1) \times E_2$.
- Speciálisan: Legyen $i=1,2$ esetén F_i az E_i reláció oszlopainak egy részhalmazán értelmezett kiválasztási feltétel, legyen továbbá $F=F_1 \wedge F_2$.
- b) • Ekkor $\sigma_F(E_1 \times E_2) \cong \sigma_{F_1}(E_1) \times \sigma_{F_2}(E_2)$.
- Általánosabban: Legyen F_1 az E_1 reláció oszlopainak egy részhalmazán értelmezett kiválasztási feltétel, legyen F_2 az $E_1 \times E_2$ reláció oszlopainak egy részhalmazán értelmezett kiválasztási feltétel, úgy hogy mindenkét sémből legalább egy oszlop szerepel benne, legyen továbbá $F=F_1 \wedge F_2$.
- c) • Ekkor $\sigma_F(E_1 \times E_2) \cong \sigma_{F_2}(\sigma_{F_1}(E_1) \times E_2)$.

Algebrai optimalizáció

7. Kiválasztás és egyesítés felcserélhetősége

- Legyen E1, E2 relációk sémája megegyező, és F a közös sémán értelmezett kiválasztási feltétel.
- Ekkor $\sigma_F(E1 \cup E2) \cong \sigma_F(E1) \cup \sigma_F(E2)$.

8. Kiválasztás és kivonás felcserélhetősége

- Legyen E1, E2 relációk sémája megegyező, és F a közös sémán értelmezett kiválasztási feltétel.
- Ekkor $\sigma_F(E1 - E2) \cong \sigma_F(E1) - \sigma_F(E2)$.

9. Kiválasztás és természetes összekapcsolás felcserélhetősége

- Legyen F az E1 és E2 közös oszlopainak egy részhalmazán értelmezett kiválasztási feltétel.
- Ekkor $\sigma_F(E1 | \times | E2) \cong \sigma_F(E1) | \times | \sigma_F(E2)$.

Algebrai optimalizáció

10. Vetítés és szorzás felcserélhetősége

- Legyen $i=1,2$ esetén \underline{A}_i az E_i reláció oszlopainak egy halmaza, valamint legyen $\underline{A}=\underline{A}_1 \cup \underline{A}_2$.
- Ekkor $\Pi_{\underline{A}}(E_1 \times E_2) \cong \Pi_{\underline{A}_1}(E_1) \times \Pi_{\underline{A}_2}(E_2)$.

11. Vetítés és egyesítés felcserélhetősége

- Legyen E_1 és E_2 relációk sémája megegyező, és legyen \underline{A} a sémában szereplő oszlopok egy részhalmaza.
- Ekkor $\Pi_{\underline{A}}(E_1 \cup E_2) \cong \Pi_{\underline{A}}(E_1) \cup \Pi_{\underline{A}}(E_2)$.
- Megjegyzés: **A vetítés és kivonás nem cserélhető fel**, azaz $\Pi_{\underline{A}}(E_1 - E_2) \neq \Pi_{\underline{A}}(E_1) - \Pi_{\underline{A}}(E_2)$. Például:

| | | | | | | | |
|-----|---|---|---|---|---|---|---|
| E1: | <table border="1"><tr><td>A</td><td>B</td></tr><tr><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td></tr></table> | A | B | 0 | 0 | 0 | 1 |
| A | B | | | | | | |
| 0 | 0 | | | | | | |
| 0 | 1 | | | | | | |

| | | | | | |
|-----|--|---|---|---|---|
| E2: | <table border="1"><tr><td>A</td><td>B</td></tr><tr><td>0</td><td>0</td></tr></table> | A | B | 0 | 0 |
| A | B | | | | |
| 0 | 0 | | | | |

esetén

míg

$$\Pi_{\underline{A}}(E_1 - E_2) = \emptyset$$

| | | | |
|-----------------------------------|--|---|---|
| $\Pi_{\underline{A}}(E_1 - E_2):$ | <table border="1"><tr><td>A</td></tr><tr><td>0</td></tr></table> | A | 0 |
| A | | | |
| 0 | | | |

Algebrai optimalizáció

- Az optimalizáló algoritmus a következő **heurisztikus elveken** alapul:
 1. **Minél hamarabb szelektáljunk**, hogy a részkifejezések várhatóan kisebb relációk legyenek.
 2. A szorzás utáni kiválasztásokból **próbálunk természetes összekapcsolásokat képezni**, mert az összekapcsolás hatékonyabban kiszámolható, mint a szorzatból történő kiválasztás.
 3. **Vonjuk össze az egymás utáni unáris műveleteket** (kiválasztásokat és vetítéseket), és ezekből lehetőleg egy kiválasztást, vagy vetítést, vagy kiválasztás utáni vetítést képezzünk. Így csökken a műveletek száma, és általában a kiválasztás kisebb relációt eredményez, mint a vetítés.
 4. **Keressünk közös részkifejezéseket**, amiket így elég csak egyszer kiszámolni a kifejezés kiértékelése során.

Algebrai optimalizáció

- **Algebrai optimalizációs algoritmus:**
- **INPUT:** relációs algebrai kifejezés kifejezésfája
- **OUTPUT:** optimalizált kifejezésfa optimalizált kiértékelése

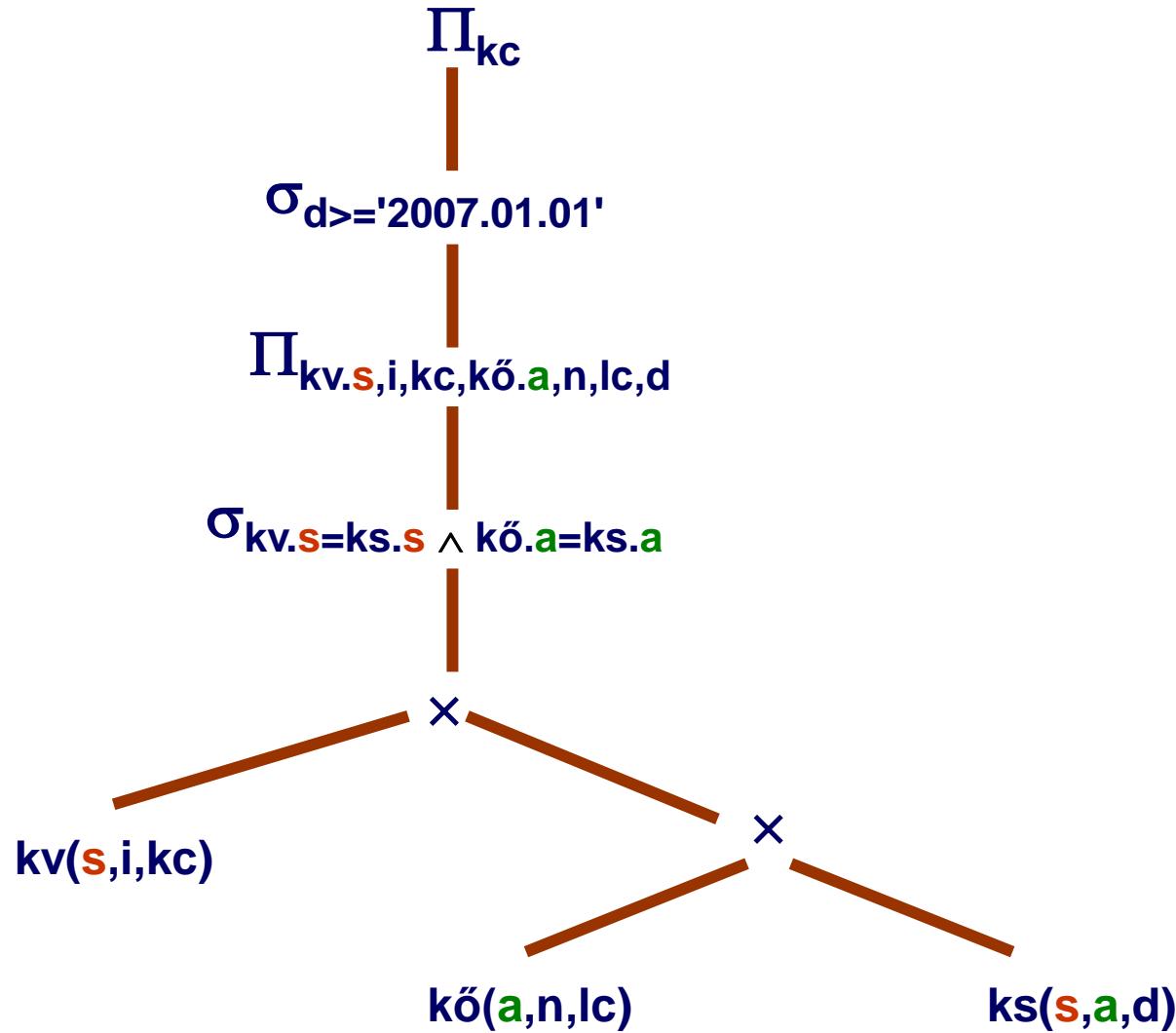
Hajtsuk végre az alábbi lépéseket a megadott sorrendben:

1. **A kiválasztásokat bontsuk fel** a **4. szabály** segítségével:
 - $\sigma_{F_1 \wedge \dots \wedge F_n}(E) \cong \sigma_{F_1}(\dots(\sigma_{F_n}(E)))$
2. **A kiválasztásokat** a **4., 5., 6., 7., 8., 9. szabályok** segítségével **vigyük** olyan **mélyre** a kifejezésfában, amilyen mélyre csak lehet.
3. **A vetítéseket** a **3., 5., 10., 11. szabályok** segítségével **vigyük** olyan **mélyre** a kifejezésfában, amilyen mélyre csak lehet. Hagyjuk el a triviális vetítéseket, azaz az olyanokat, amelyek az argumentum reláció összes attribútumára vetítenek.
4. Ha egy relációs változóra vagy konstans relációra közvetlenül egymás után kiválasztásokat vagy vetítéseket alkalmazunk, akkor ezeket a **3., 4., 5. szabályok** segítségével **vonjuk össze egy kiválasztássá, vagy egy vetítéssé, vagy egy kiválasztás utáni vetítéssé, ha lehet** (azaz egy $\Pi.(\sigma.)()$ alakú kifejezéssé). **Ezzel megkaptuk az optimalizált kifejezésfát.**
5. A gráfot **a bináris műveletek alapján bontsuk részgráfokra**. minden részgráf egy bináris műveletnek feleljen meg. A részgráf csúcsai legyenek: a bináris műveletnek (\cup , $-$, \times) megfelelő csúcs és a csúcs felett a következő bináris műveletig szereplő kiválasztások (σ) és vetítések (Π). Ha a bináris művelet szorzás (\times), és a részgráf equi-joinnak felel meg, és a szorzás valamelyik ága nem tartalmaz bináris műveletet, akkor ezt az ágat is vegyük hozzá a részgráfhoz.
6. Az előző lépésben kapott részgráfok is fát képeznek. **Az optimális kiértékeléshez** ezt a fát értékeljük ki alulról felfelé haladva, tetszőleges sorrendben.

Megjegyzés. Az **equi-join** azt jelenti, hogy a kiválasztás feltétele egyenlőség, amely a szorzás két ágának egy-egy oszlopát hasonlítja össze.

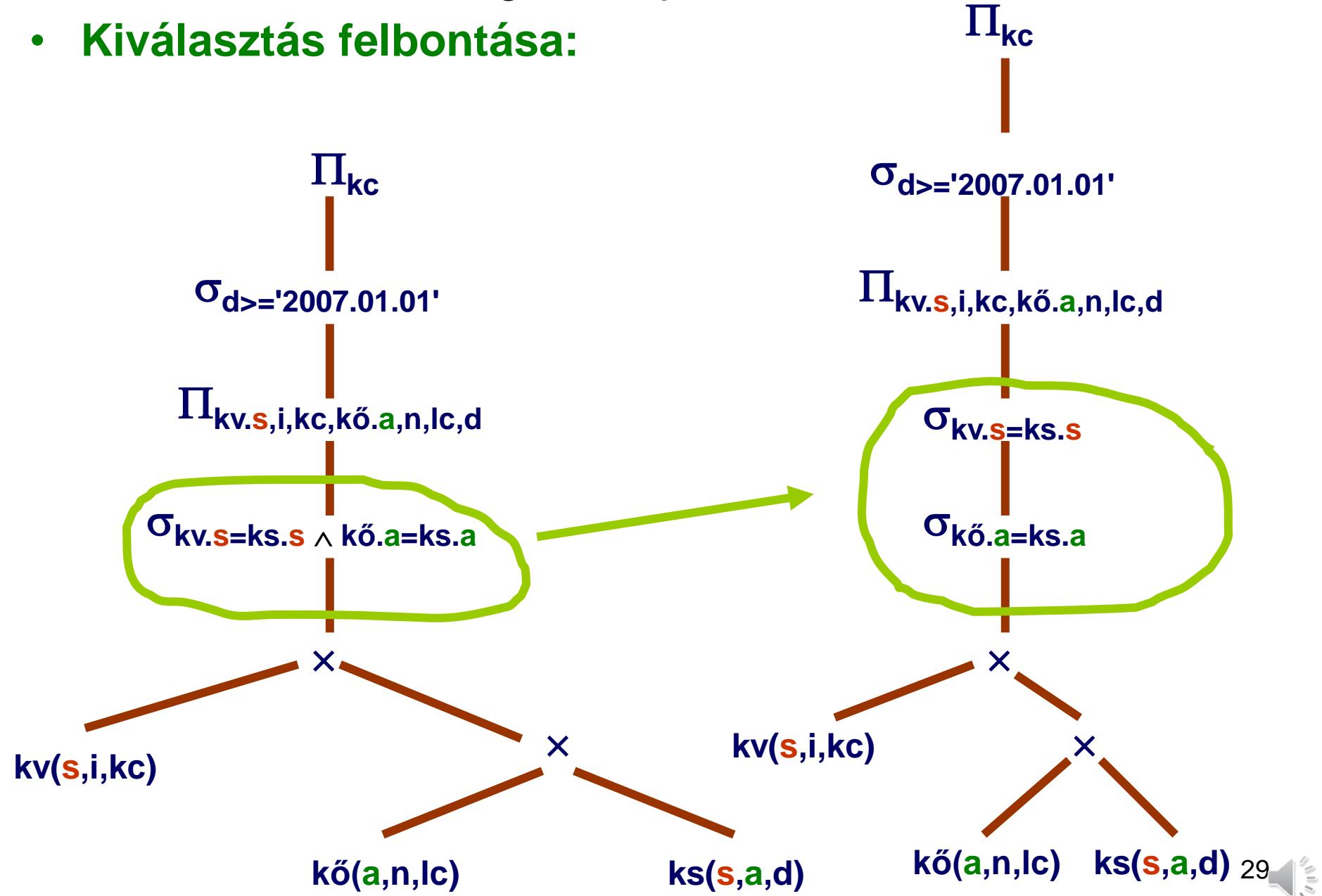
Algebrai optimalizáció

- Optimalizáljuk a következő kifejezést:

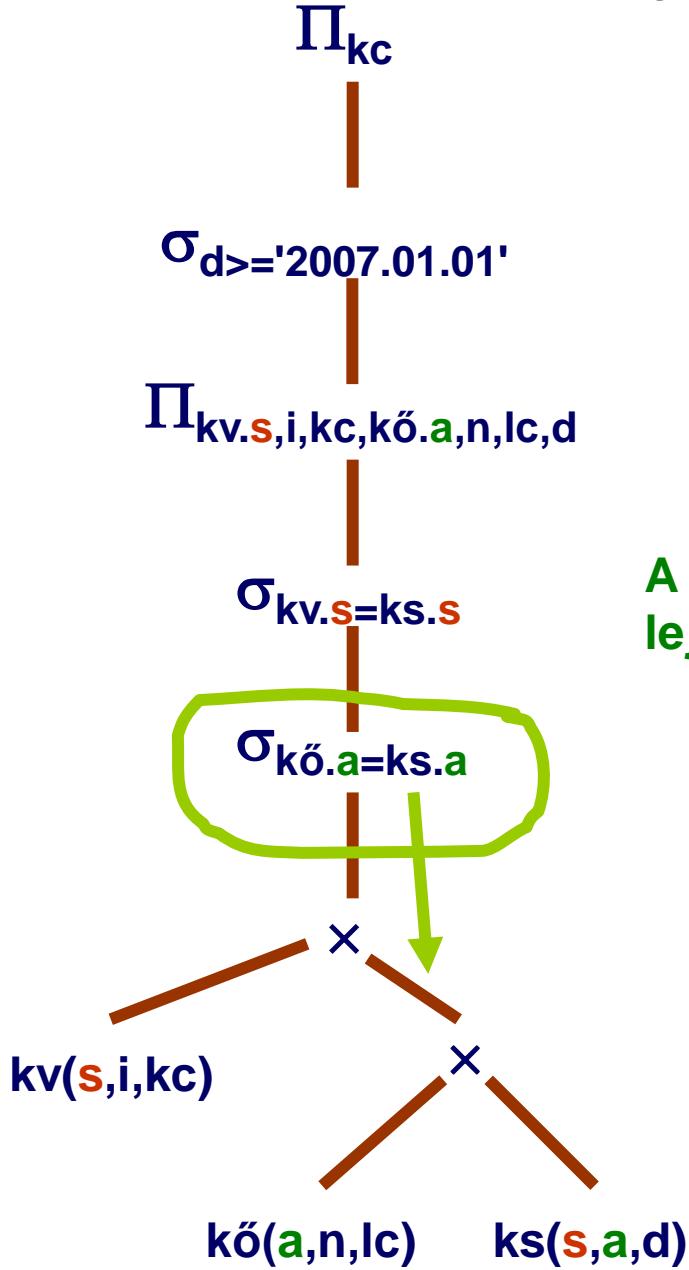
$$\Pi_{kc}(\sigma_{d \geq '2007.01.01'}(\Pi_{kv.s,i,kc,kō.a,n,lc,d}(\sigma_{kv.s=ks.s \wedge kō.a=ks.a}(kv \times (kō \times ks)))))$$


Algebrai optimalizáció

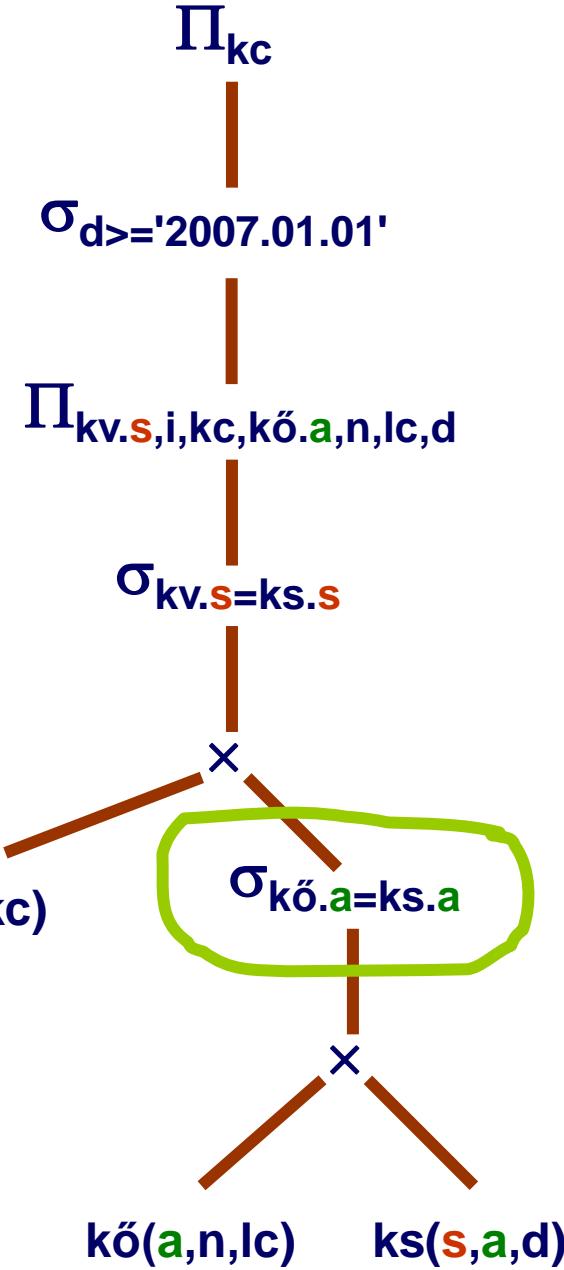
- Kiválasztás felbontása:



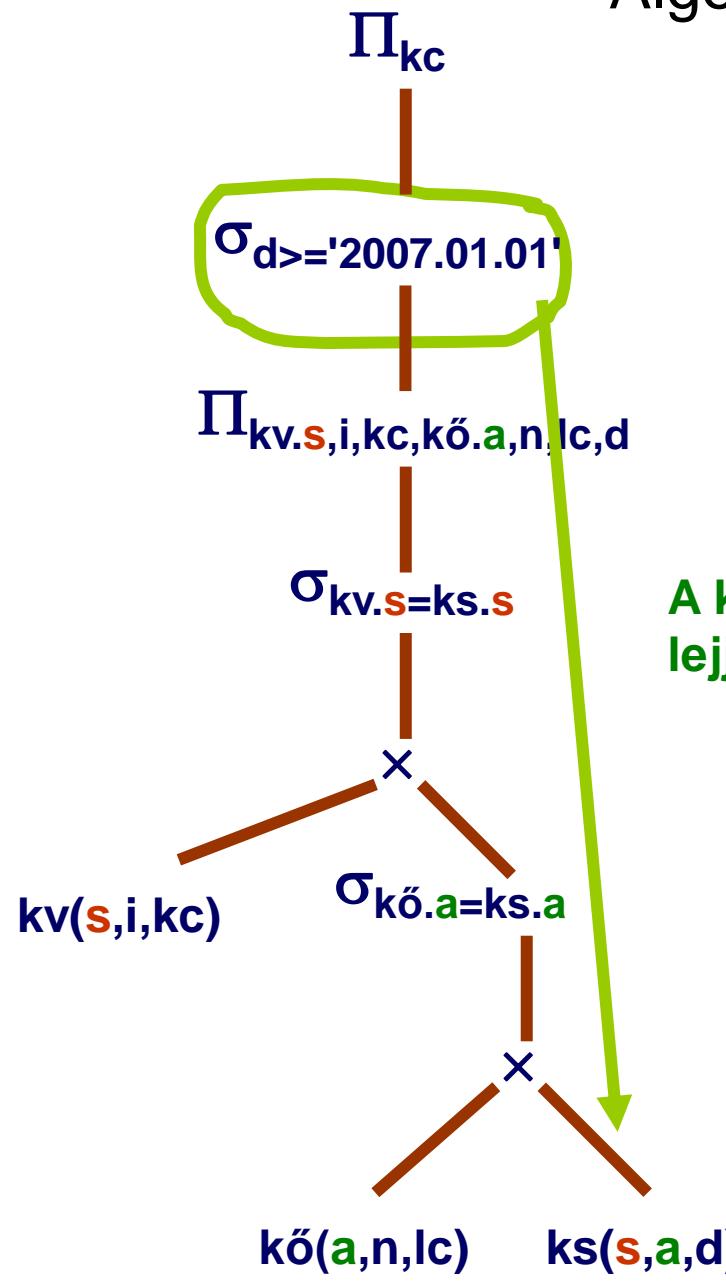
Algebrai optimalizáció



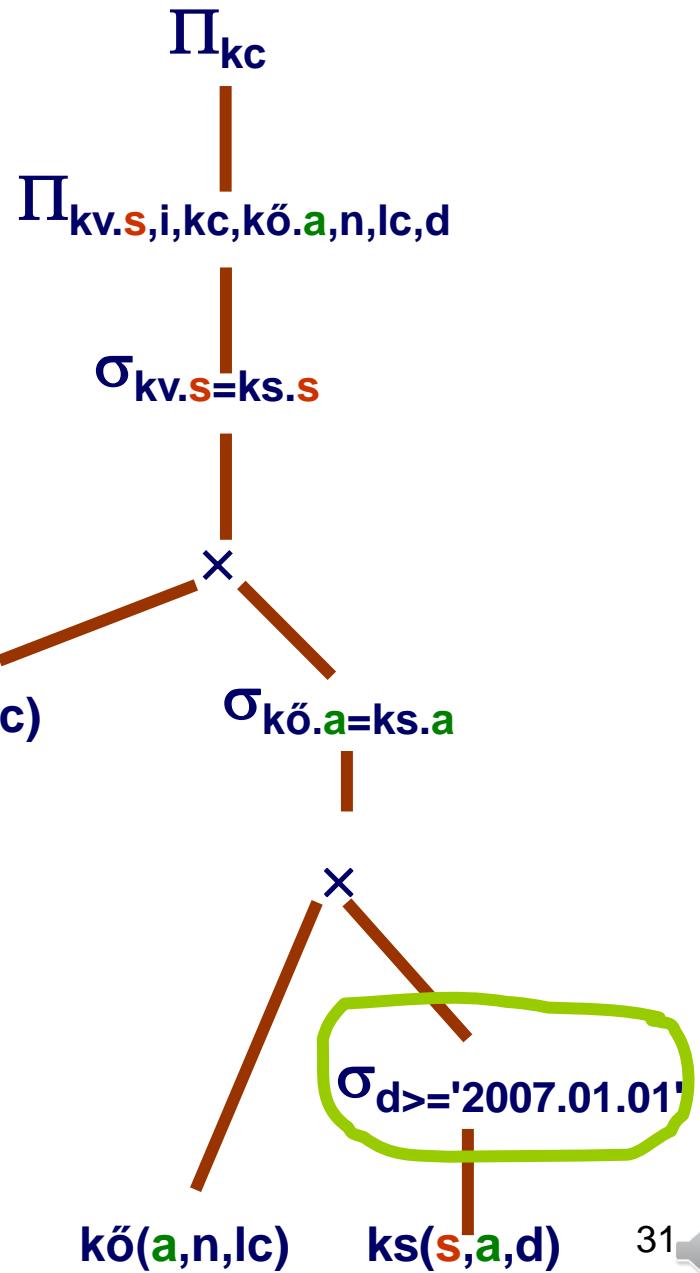
A kiválasztást
lejjebb visszük



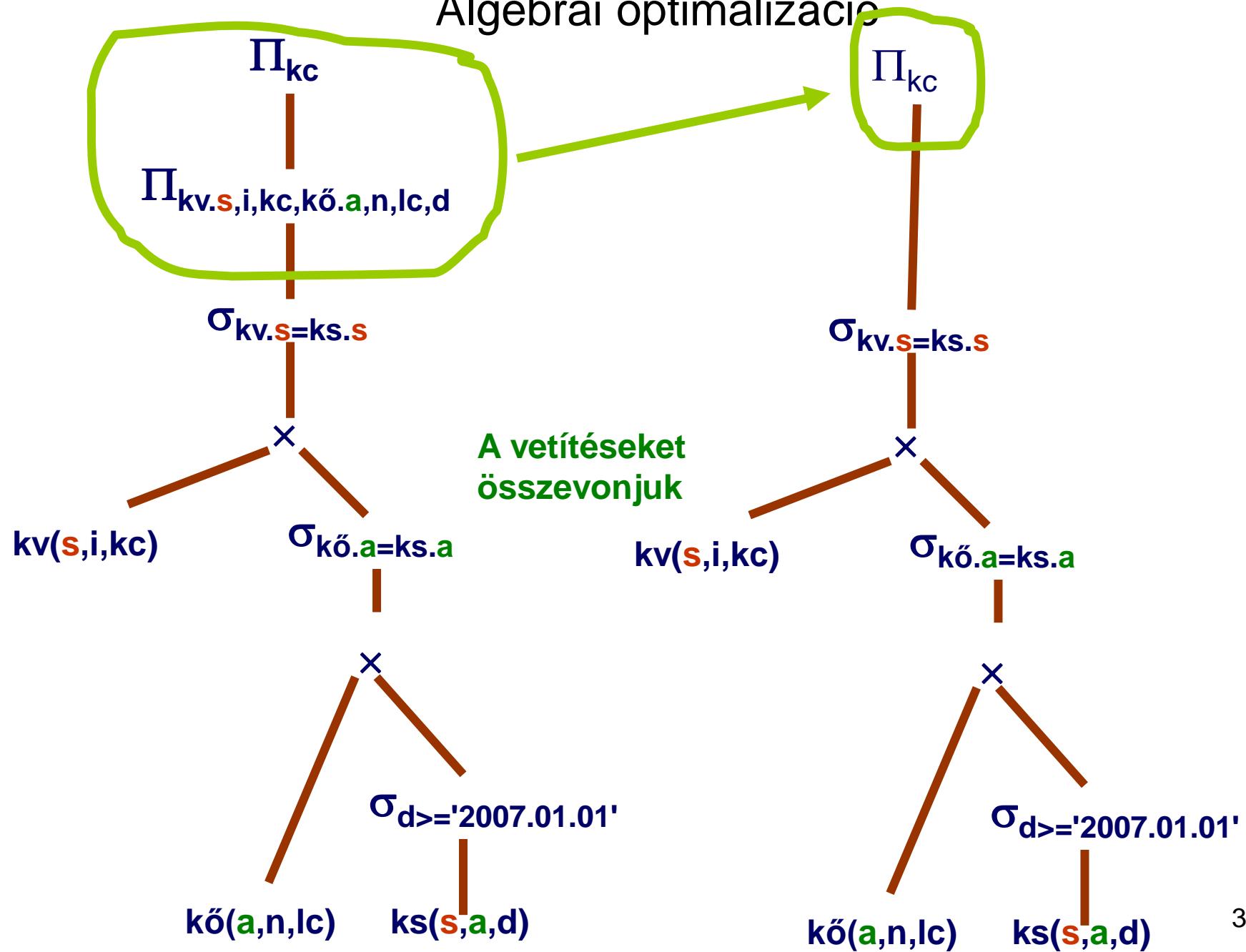
Algebrai optimalizáció



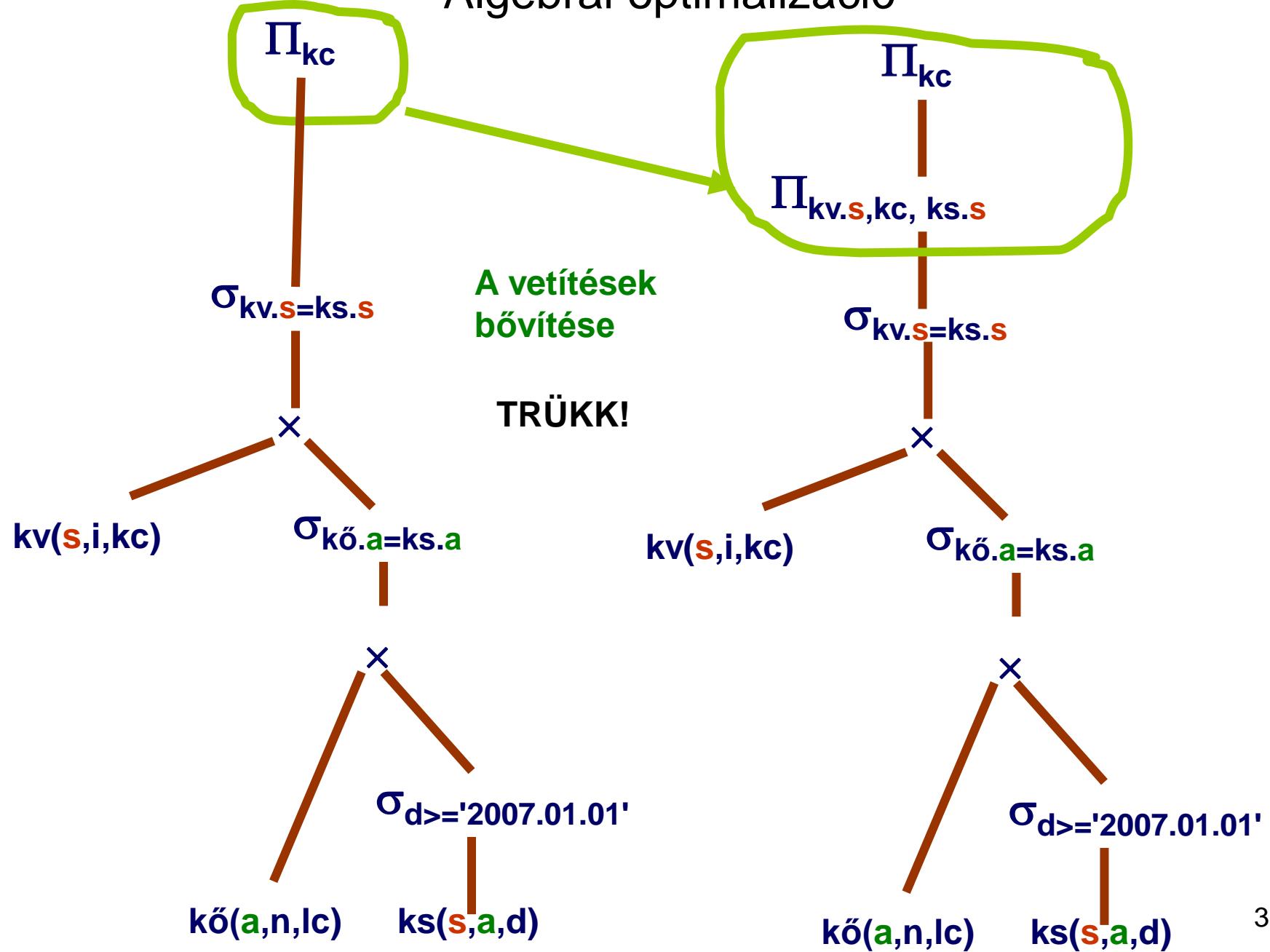
A kiválasztást
lejjebb visszük



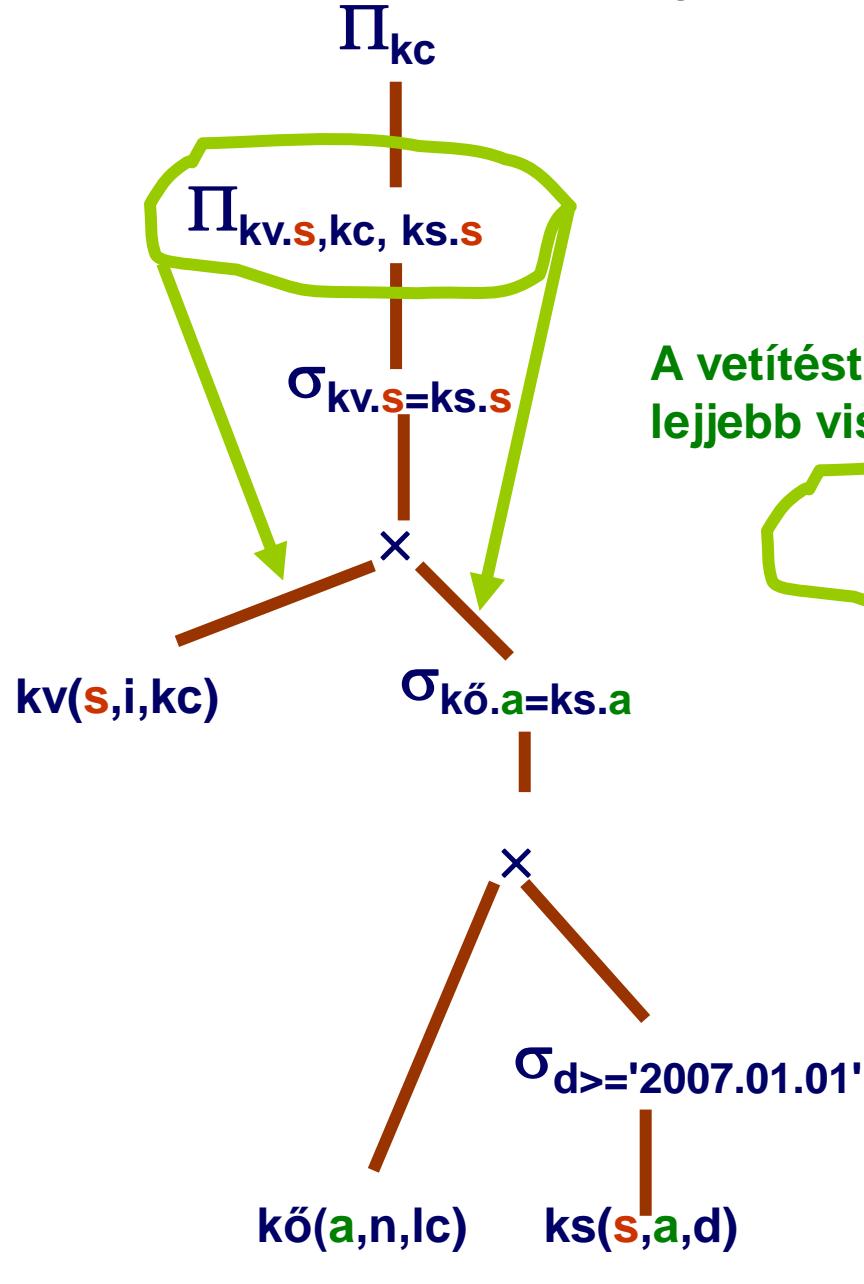
Algebrai optimalizáció



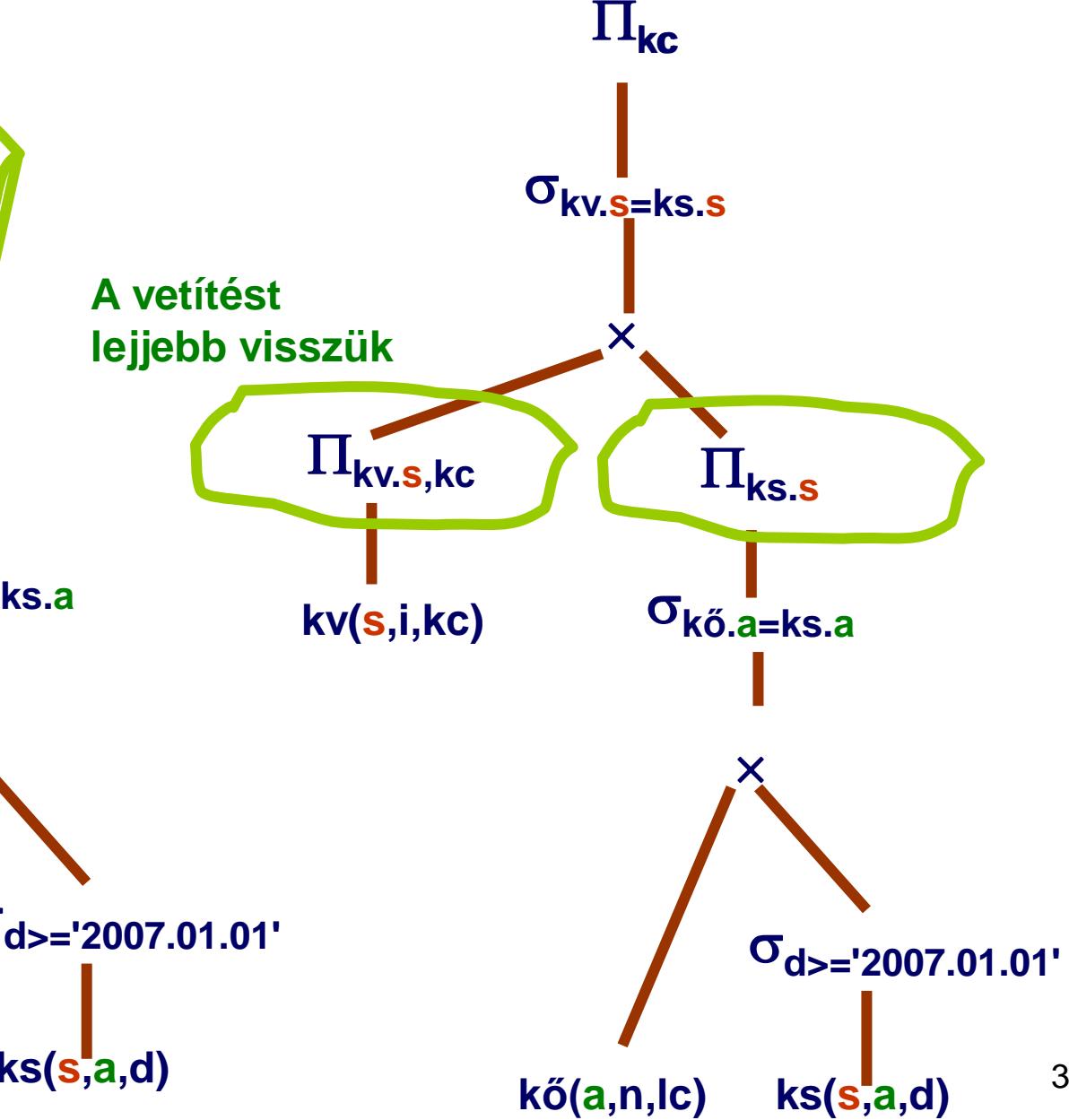
Algebrai optimalizáció



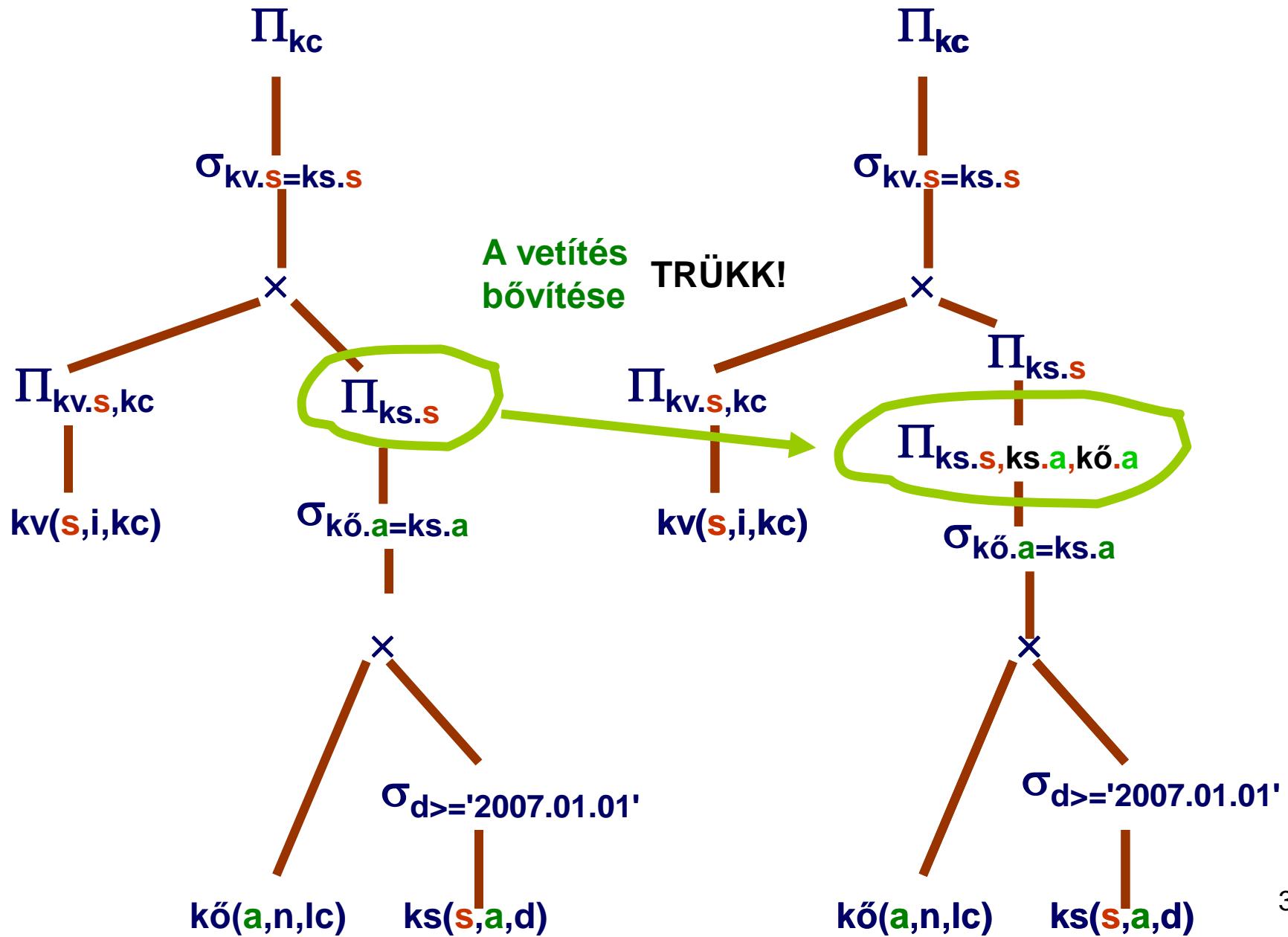
Algebrai optimalizáció



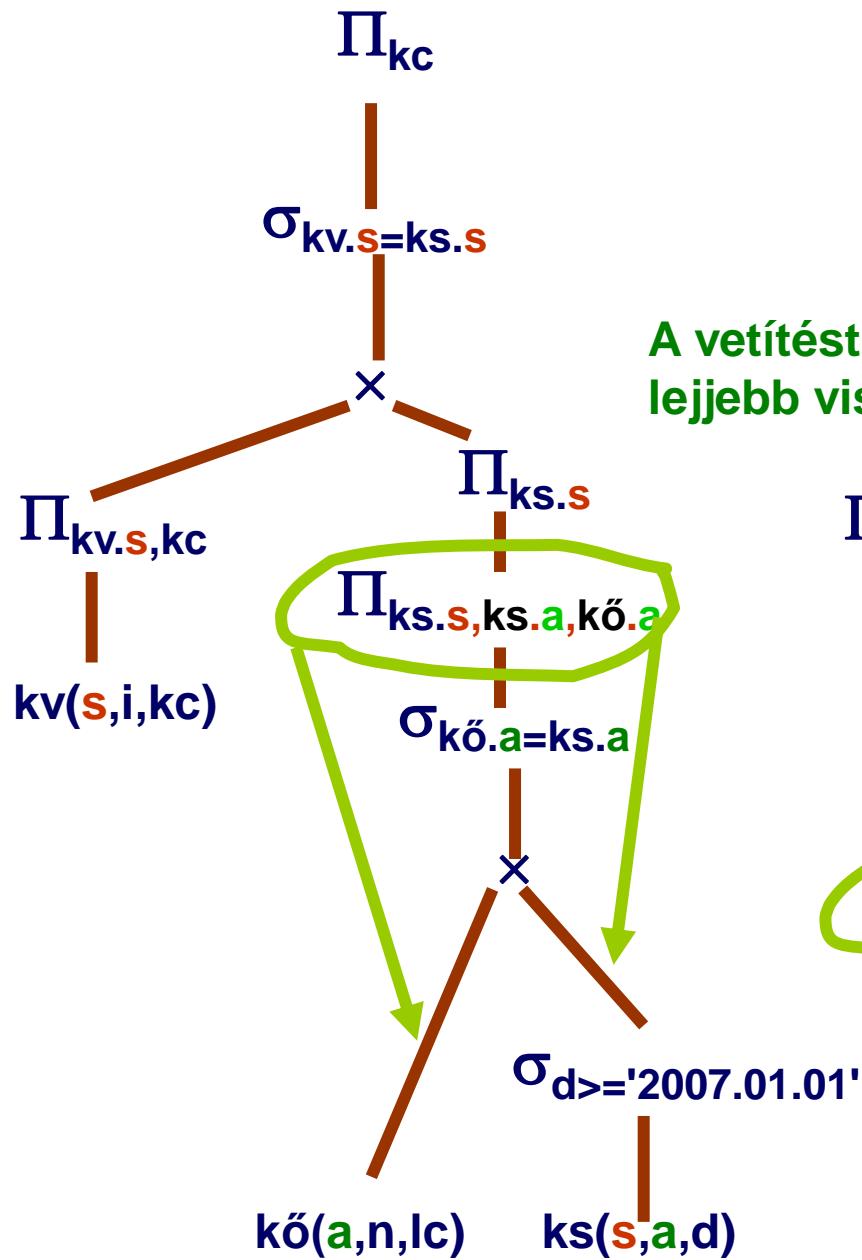
A vetítést
lejjebb visszük



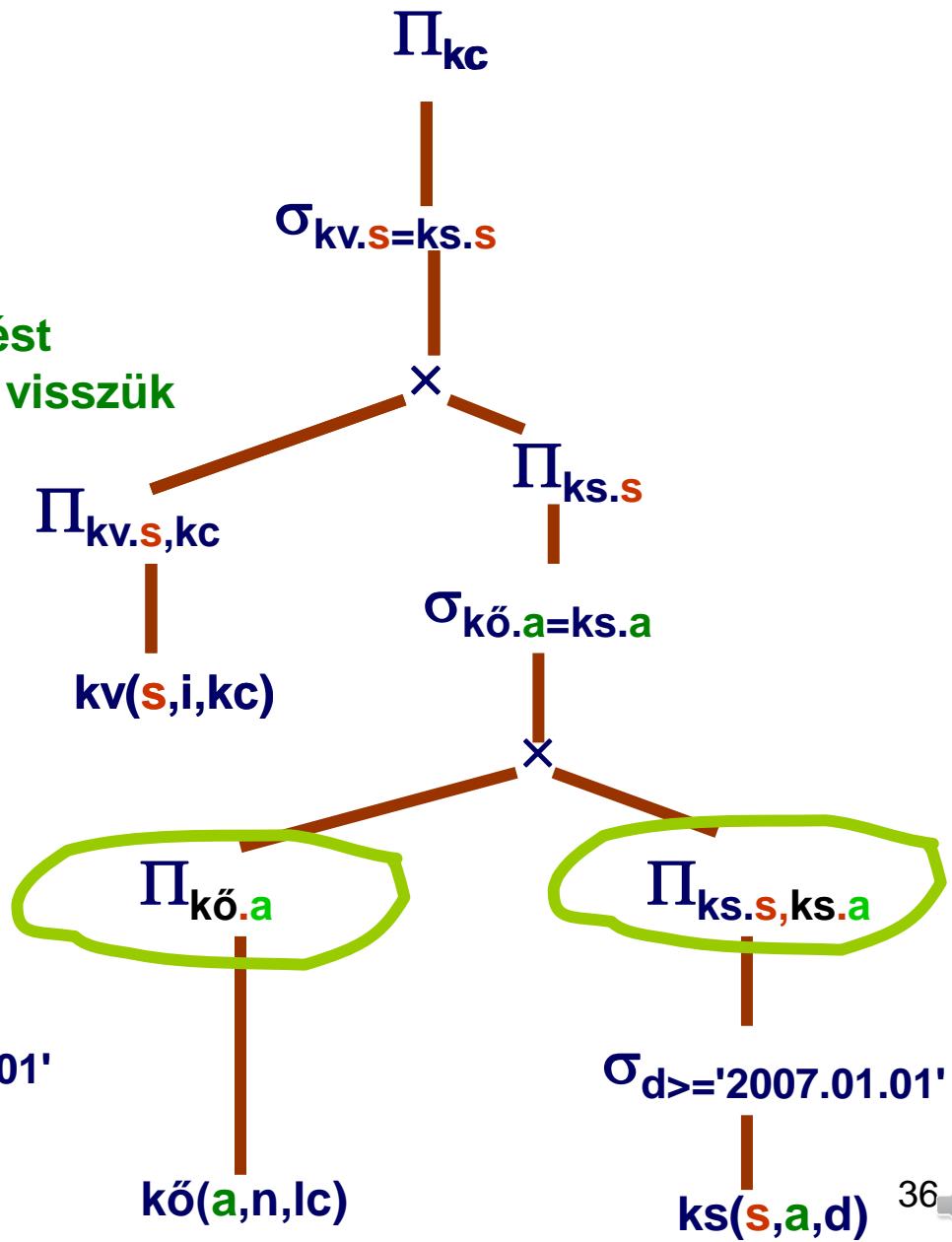
Algebrai optimalizáció



Algebrai optimalizáció



A vetítést
lejjebb visszük



Algebrai optimalizáció

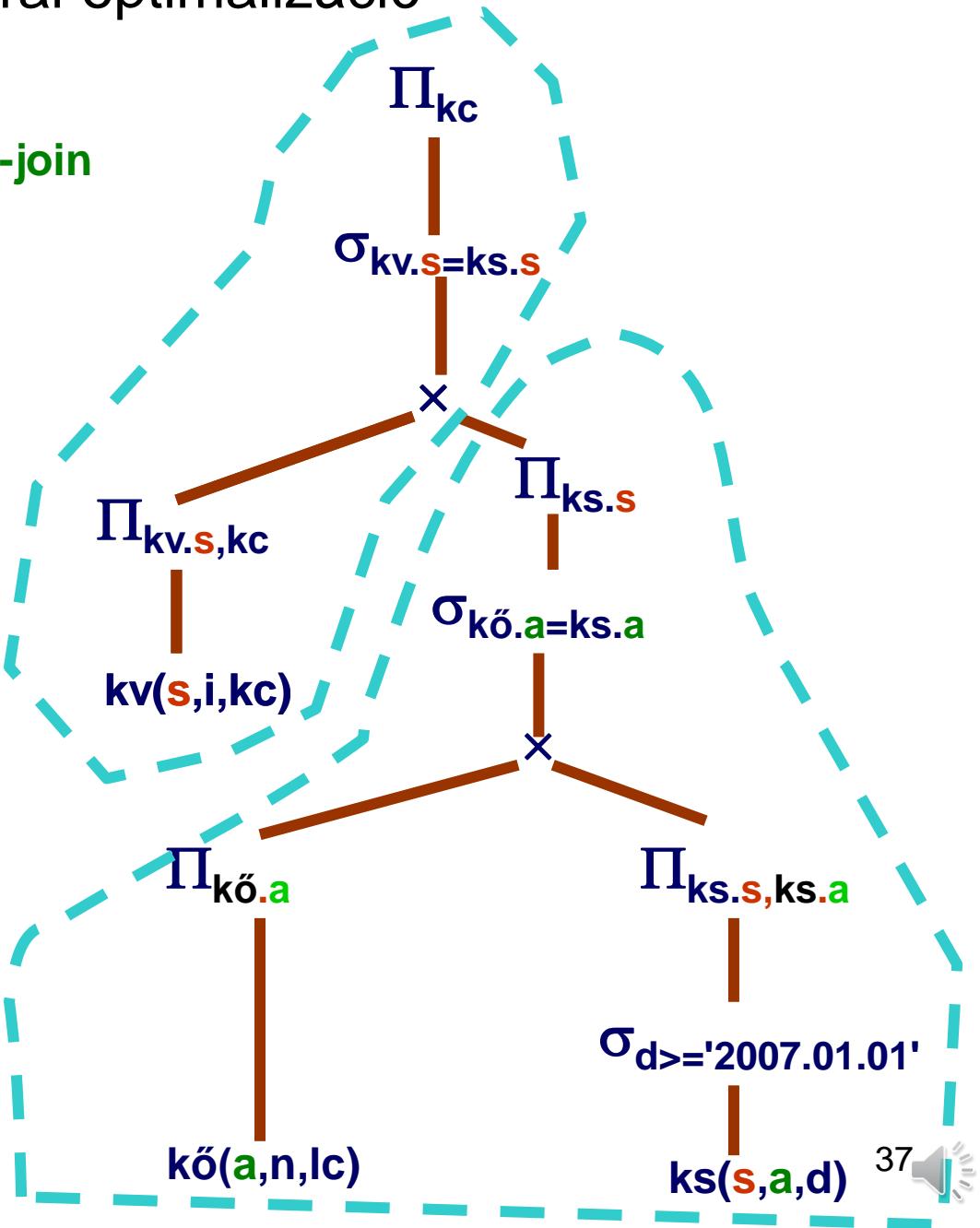
Részgráfokat képezünk (az equi-join miatt a levelekig kiegészítjük a csoportokat)

2. részgráf

Az algebrai optimalizáció eredménye:

Először az 1. részgráfnek megfelelő kifejezést számoljuk ki, és utána a 2. részgráfnak megfelelő kifejezést.

1. részgráf





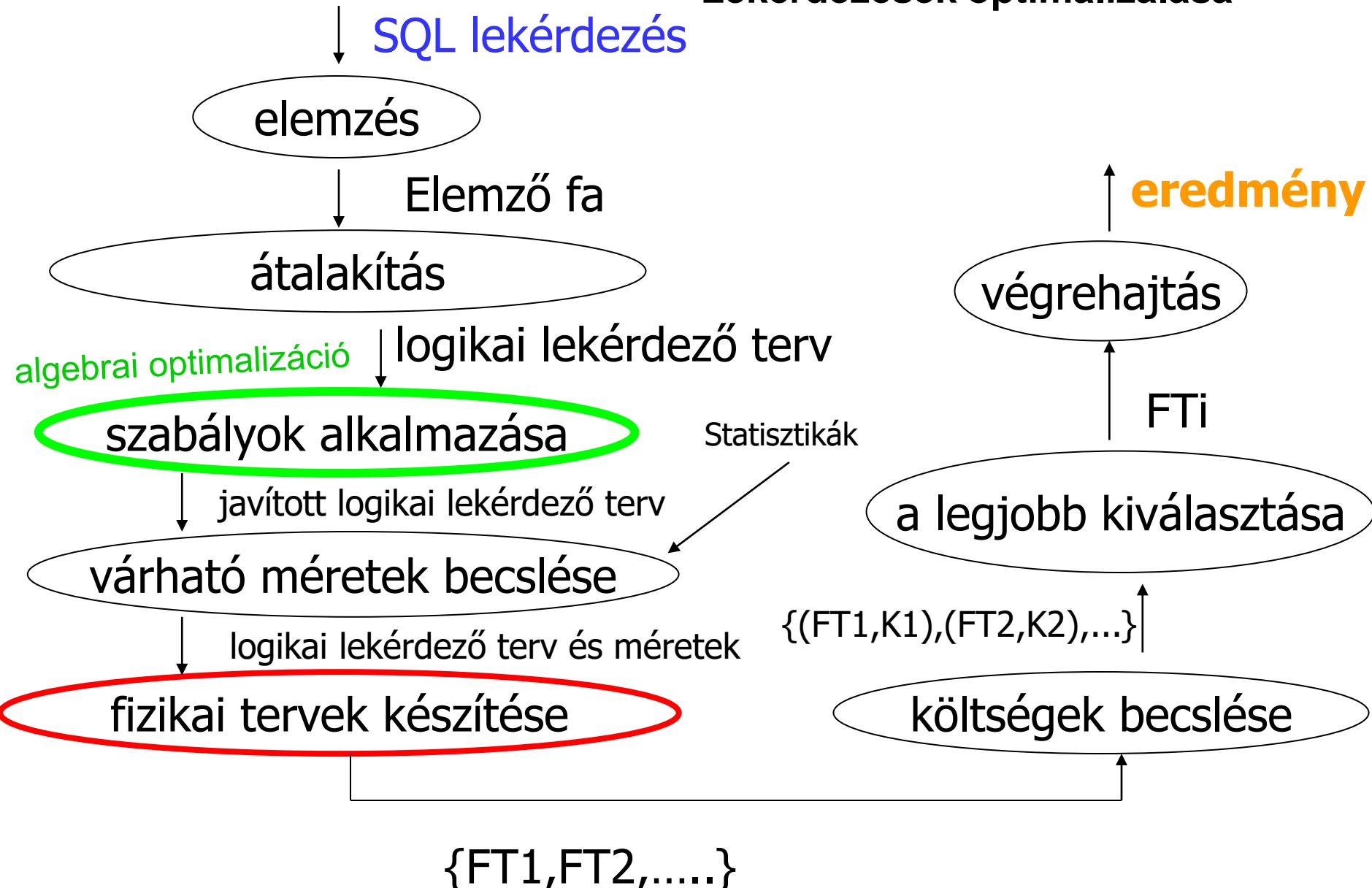
Köszönöm a figyelmet!

Fizikai tervezek

- Paraméterek, költségek
- Fizikai fájlszervezés, indexek
- Műveletek megvalósítása, kiszámítási költség, outputméret
- Optimális fizikai terv meghatározása

The screenshot shows the Oracle SQL Developer interface. The top window is a 'Worksheet' containing a SQL query:SELECT d.deptno,
 d.dname,
 e.ename,
 e.job
 FROM dept d
 , emp e
 WHERE d.deptno = e.deptno
 AND d.dname = 'SALES'
 ORDER BY
 e.enameThe bottom window is the 'Explain Plan' tab, which displays the execution plan for the query. The plan details the operations and costs:| OPERATION | OBJECT_NAME | OPTIONS | COST |
| --- | --- | --- | --- |
| SELECT STATEMENT | | | 6 |
| SORT | | ORDER BY | 6 |
| HASH JOIN | | | 5 |
| Access Predicates | D.DEPTNO=E.DEPTNO | | |
| TABLE ACCESS | DEPT | FULL | 2 |
| Filter Predicates | D.DNAME='SALES' | | |
| TABLE ACCESS | EMP | FULL | 2 |

Lekérdezések optimalizálása



Indexelés

- Célok:
 - gyors lekérdezés,
 - gyors adatmódosítás,
 - minél kisebb tárolási terület.
- Nincs általánosan legjobb optimalizáció. Az egyik cél a másik rovására javítható (például indexek használatával csökken a keresési idő, nő a tárméret, és nő a módosítási idő).
- Az adatbázis-alkalmazások alapján az adatbázis lehet:
 - statikus (ritkán módosul, a lekérdezések gyorsasága a fontosabb),
 - dinamikus (gyakran módosul, ritkán végünk lekérdezést).
- Hogyan mérjük a költségeket?
- Memória műveletek nagyságrenddel gyorsabbak, mint a háttértárolóról beolvasás, kiírás.
- Az író-olvasó fej nagyobb adategységeket (**blokkokat**) olvas be.
- A blokkméret függhet az operációs rendszertől, hardvertől, adatbáziskezelőtől.
- A blokkméretet fixnek tekintjük. Oracle esetén 8K az alapértelmezés.
- **Feltételezzük, hogy a beolvasás, kiírás költsége arányos a háttértároló és memória között mozgatott blokkok számával.**

Indexelés

- Célszerű a fájlokat blokkokba szervezni.
- A fájl rekordokból áll.
- A **rekordok szerkezete** eltérő is lehet.
- A rekord tartalmaz:
 - **leíró fejlécet** (rekordstruktúra leírása, belső/külső mutatók, (hol kezdődik egy mező, melyek a kitöltetlen mezők, melyik a következő rekord, melyik az előző rekord), törlési bit, statisztikák),
 - **mezőket**, melyek üresek, vagy adatot tartalmaznak.
- A rekordhossz lehet:
 - **állandó**,
 - **változó** (változó hosszú mezők, ismétlődő mezők miatt).
- Az egyszerűség kedvéért feltezzük, hogy állandó hosszú rekordokból áll a fájl, melyek hossza az átlagos rekordméretnek felel.
- A **blokkok** tartalmaznak:
 - **leíró fejlécet** (rekordok száma, struktúrája, fájlok leírása, belső/külső mutatók (hol kezdődik a rekord, hol vannak üres helyek, melyik a következő blokk, melyik az előző blokk, statisztikák (melyik fájlból hány rekord szerepel a blokkban)),
 - **rekordokat** (egy vagy több fájlból),
 - **üres helyeket**.

Indexelés

- A költségek méréséhez paramétereket vezetünk be:
- **I** - (length) **rekordméret** (bájtokban)
- **b** - **blokkméret** (bájtokban)
- **T** - (tuple) **rekordok száma**
- **B** - a **fájl mérete blokkokban**
- **bf** – **blokkolási faktor** (mennyi rekord fér el egy blokkban:
 $bf = \lfloor b/I \rfloor$ - alsó egészrész)
- **B=** $\lceil T/bf \rceil$
- **M** – **memória mérete blokkokban**
- Például **R×S** mérete mekkora:
 - $I(R \times S) = I(R) + I(S)$
 - $T(R \times S) = T(R) * T(S)$
 - $bf(R \times S) = b / (I(R) + I(S))$
 - $B(R \times S) = (T(R) * T(S)) * (I(R) + I(S)) / b$
 $= (T(S) * T(R) * I(R) / b) + (T(R) * T(S) * I(S) / b) =$
 $= T(S) * B(R) + T(R) * B(S)$

Indexelés

- Milyen lekérdezéseket vizsgálunk?
- A relációs algebrai kiválasztás felbontható atomi kiválasztásokra, így elég ezek költségét vizsgálni.
- A legegyszerűbb kiválasztás:
 - **A=a** (**A** egy keresési mező, **a** egy konstans)
- Kétféle **bonyolultság**ot szokás vizsgálni:
 - **átlagos**,
 - **legrosszabb** eset.
- Az esetek vizsgálatánál az is számít, hogy az **A=a** feltételnek megfelelő rekordokból lehet-e több, vagy biztos, hogy csak egy lehet.
- Fel szoktuk tenni, hogy az **A=a** feltételnek eleget tevő rekordokból nagyjából egyforma számú rekord szerepel. (Ez az **egyenletességi feltétel**.)

Indexelés

- Az A oszlopban szereplő különböző értékek számát **képméretnek** hívjuk és **I(A)**-val jelöljük.
- **I(A)=|Π_A(R)|**
- Egyenletességi feltétel esetén:
 - $T(\sigma_{A=a}(R)) = T(R) / I(A)$
 - $B(\sigma_{A=a}(R)) = B(R) / I(A)$
- A következő fájlszervezési módszereket fogjuk megvizsgálni:
 - **kupac (heap)**
 - **hasító index (hash)**
 - **rendezett állomány**
 - **elsődleges index (ritka index)**
 - **másodlagos index (sűrű index)**
 - **többszintű index**
 - **B⁺-fa, B^{*}-fa**
- Azt az esetet vizsgáljuk, mikor az A=a feltételű rekordok közül elég az elsőt megkeresni.
- **Módosítási műveletek:**
 - **beszúrás (insert)**
 - **frissítés (update)**
 - **törlés (delete)**
- Az egyszerűsített esetben nem foglalkozunk azzal, hogy a beolvasott rekordokat bent lehet tartani a memóriában, későbbi keresések céljára.

Indexelés

- **Kupac szervezés:**

- a rekordokat a blokk első üres helyre tesszük a beérkezés sorrendjében.

- **Tárméret: B**

- A=a **keresési idő:**

- **B** (a legrosszabb esetben),
 - $B/2$ (átlagos esetben egyenletességi feltétel esetén).

- **Beszúrás:**

- utolsó blokkba tesszük a rekordot, 1 olvasás + 1 írás
 - módosítás: 1 keresés + 1 írás
 - törlés: 1 keresés + 1 írás (üres hely marad, vagy a törlési bitet állítják át)

Indexelés

- **Hasítóindex-szervezés** (Hashelés):
 - a rekordokat **blokklánc**okba (**bucket – kosár**) soroljuk és a blokklánc utolsó blokkjának első üres helyére tesszük a rekordot a beérkezés sorrendjében.
 - a blokkláncok száma
 - előre adott: K (**statikus hasítás**)
 - a tárolt adatok alapján változhat (**dinamikus hasítás**)
- A besorolás az indexmező értékei alapján történik.
- Egy $h(x) \in \{1, \dots, K\}$ **hasító függvény** értéke mondja meg, hogy melyik kosárba tartozik a rekord, ha x volt az indexmező értéke a rekordban.
- A hasító függvény általában maradékos osztáson alapul. Például **mod(K)**.
- Akkor **jó egy hasító függvény**, ha nagyjából egyforma hosszú blokkláncok keletkeznek, azaz egyenletesen sorolja be a rekordokat.
- Jó hasító függvény esetén **a blokklánc B/K blokból áll**.

Indexelés

- **Keresés ($A=a$)**
 - ha az indexmező és keresési mező eltér, akkor kupac szervezést jelent,
 - ha az indexmező és keresési mező megegyezik, akkor csak elég a **h(a)** sorszámu kosarat végignézni, amely **B/K blokkból** álló kupacnak felel meg, azaz **B/K** legrosszabb esetben. **A keresés K-szorosára gyorsul.**
- Miért nem érdemes nagyon nagy K-t választani?
- **Tárméret:** **B**, ha minden blokk nagyjából tele.
- Nagy K esetén sok olyan blokklánc lehet, amely egy blokkból fog állni, és a blokkban is csak 1 rekord lesz. Ekkor a keresési idő: 1 blokkbeolvasás, de B helyett T számú blokkban tároljuk az adatokat.
- **Módosítás:** B/K blokkból álló kupac szervezésű kosarat kell módosítani.
- **Intervallumos ($a < A < b$) típusú keresésre nem jó.**

Indexelés

Tegyük fel, hogy 1 blokkba 2 rekord fér el.

Szűrjuk be a következő hasító értékkel rendelkező rekordokat!

INSERT:

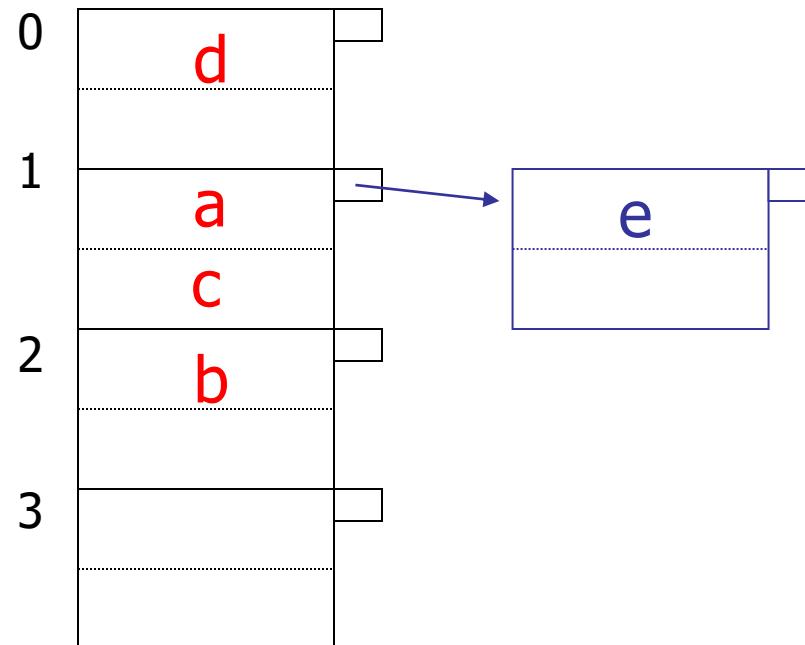
$$h(a) = 1$$

$$h(b) = 2$$

$$h(c) = 1$$

$$h(d) = 0$$

$$h(e) = 1$$

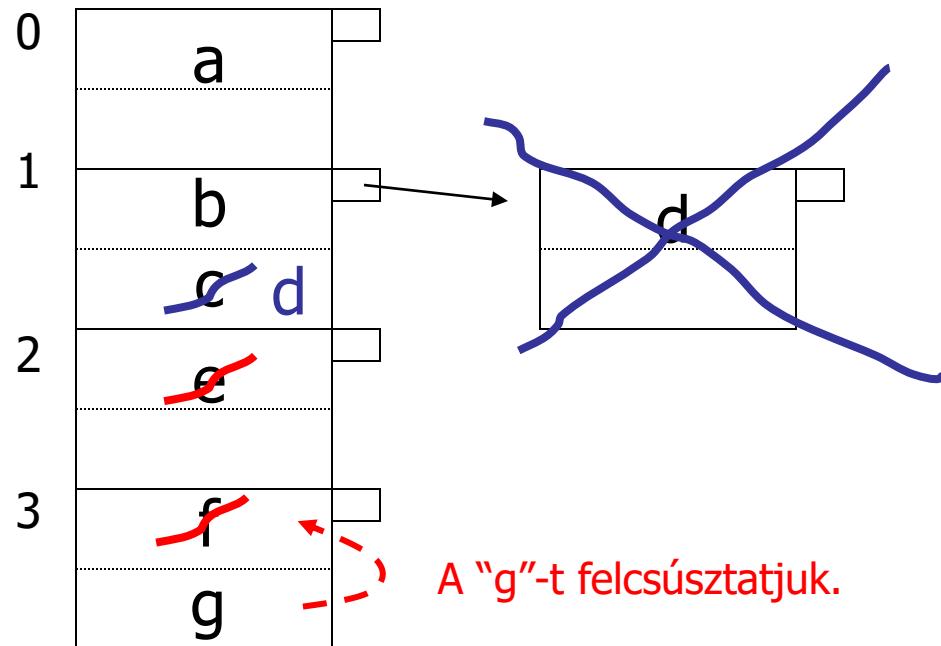


Indexelés

Töröljük a következő hasító értékkel rendelkező rekordokat!
(A megüresedett túlcordulási blokkokat megszüntetjük.)

Delete:

e
f
c



Indexelés

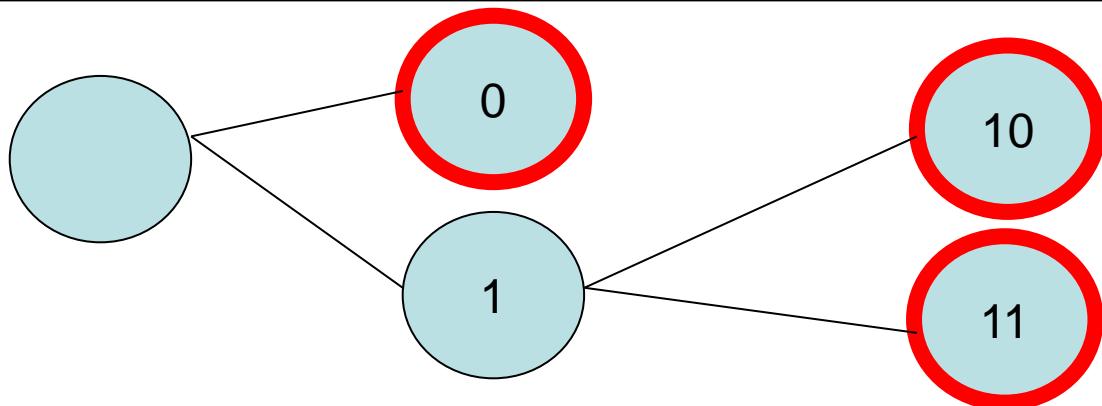
- Dinamikus hasító indexek:
 - kiterjeszthető (expandable)
 - lineáris
- Előre nem rögzítjük a kosarak számát, a kosarak száma beszúráskor, törléskor változhat.
- Kiterjeszthető hasító index:
- minden kosár 1 blokkból áll. Keresési költség: 1.
- Legyen **$k > \log$ (a rekordok várható számának felső korlátja)**,
 - azaz k hosszú bináris sorozatból több van, mint ahány rekord
- A **h hasító függvény értéke egy k hosszú bináris sorozat**.
- minden kosárhoz tartozik egy legfeljebb k hosszú bináris sorozat (kódszó).
- A kosarakhoz rendelt kód **prefix kód**. A maximális kód hossza legyen i .
- A $h(x)$ k hosszú kódnak vegyük az **i hosszú elejét**, és **azt kosarat, amelynek kódja a $h(x)$ kezdő szelete**. Ha van hely a kosárban, tegyük bele a rekordot, ha nincs, akkor nyissunk egy új kosarat, és a következő bit alapján osszuk ketté a telített kosár rekordjait. Ha ez a bit mindegyikre megegyezik, akkor a következő bitet vesszük a szétosztáshoz, és így tovább.

Indexelés

| | |
|---|------|
| 0 | 0010 |
| | |
| 1 | 1010 |
| | 1011 |

1101 beszúrása
2. bit alapján

| | |
|---|------|
| 0 | 0010 |
| | |
| 1 | 1010 |
| | 1011 |
| 0 | 1101 |
| | |
| 1 | |



Indexelés

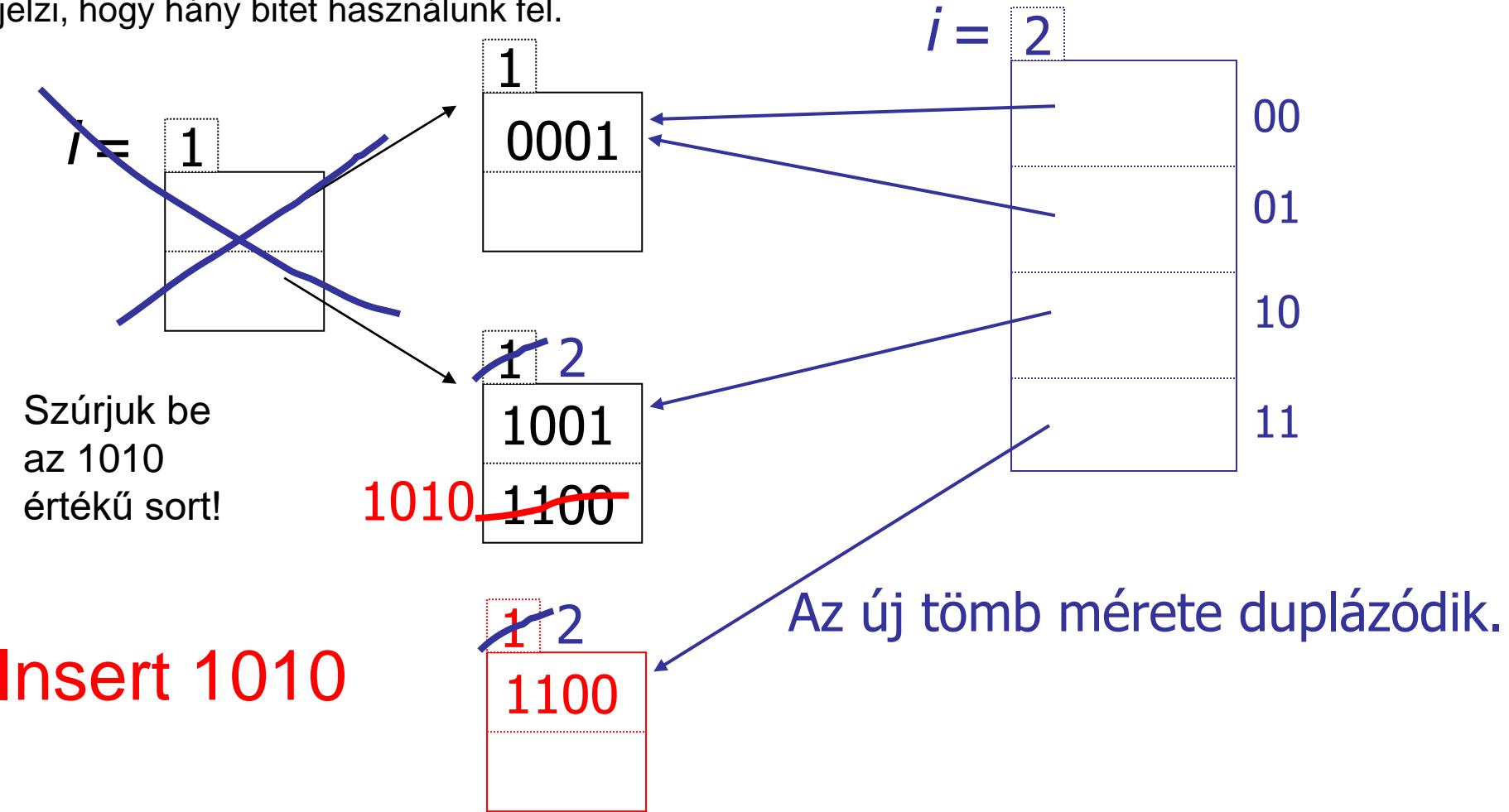
- A bináris fa levelei a kosárblökkök kódszavai. A hasító függvény értékéből annyi bitet használunk, ahányadik szinten szerepel a levél.
- A gráfot a memóriában tárolhatjuk.

Probléma: Ha az új sorok hasító értékének eleje sok bitben megegyezik, akkor **hosszú ágak** keletkezhetnek.

(Nincs kiegyensúlyozva a fa.)

Indexelés

A bináris gráfot teljessé is tehetjük. A gráfot egy tömbbel ábrázolhatjuk. Ekkor minden kosár azonos szinten lesz, de közös blokkjai is lehetnek a kosaraknak. Túlcsordulás esetén a kosarak száma duplázódik. Legyen például $h(x)$ 4 bites és 2 rekord férjen egy blokkba. Az i jelzi, hogy hány bitet használunk fel.



Indexelés

Szűrjuk be most a
0111 és 0000 értékű
sorokat!

$i =$

| |
|----|
| 00 |
| 01 |
| 10 |
| 11 |

| |
|------|
| 2 |
| 0000 |
| 0001 |

| |
|------|
| 12 |
| 0001 |
| 0111 |

0111

| |
|------|
| 2 |
| 1001 |
| 1010 |

| |
|------|
| 2 |
| 1100 |

Insert:

0111

0000

Indexelés

Szúrjuk be az
1001 sort!

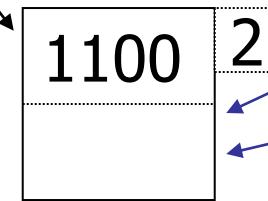
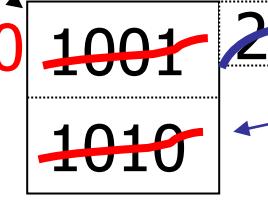
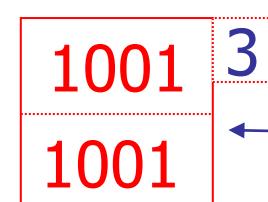
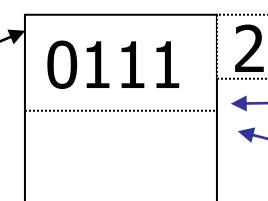
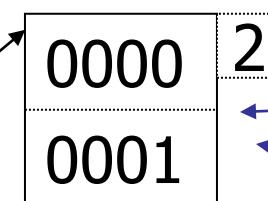
$i = 2$

00

01

10

11



000
001
010
011
100
101
110
111

Insert:

1001

Indexelés

- **Lineáris hasító index:**

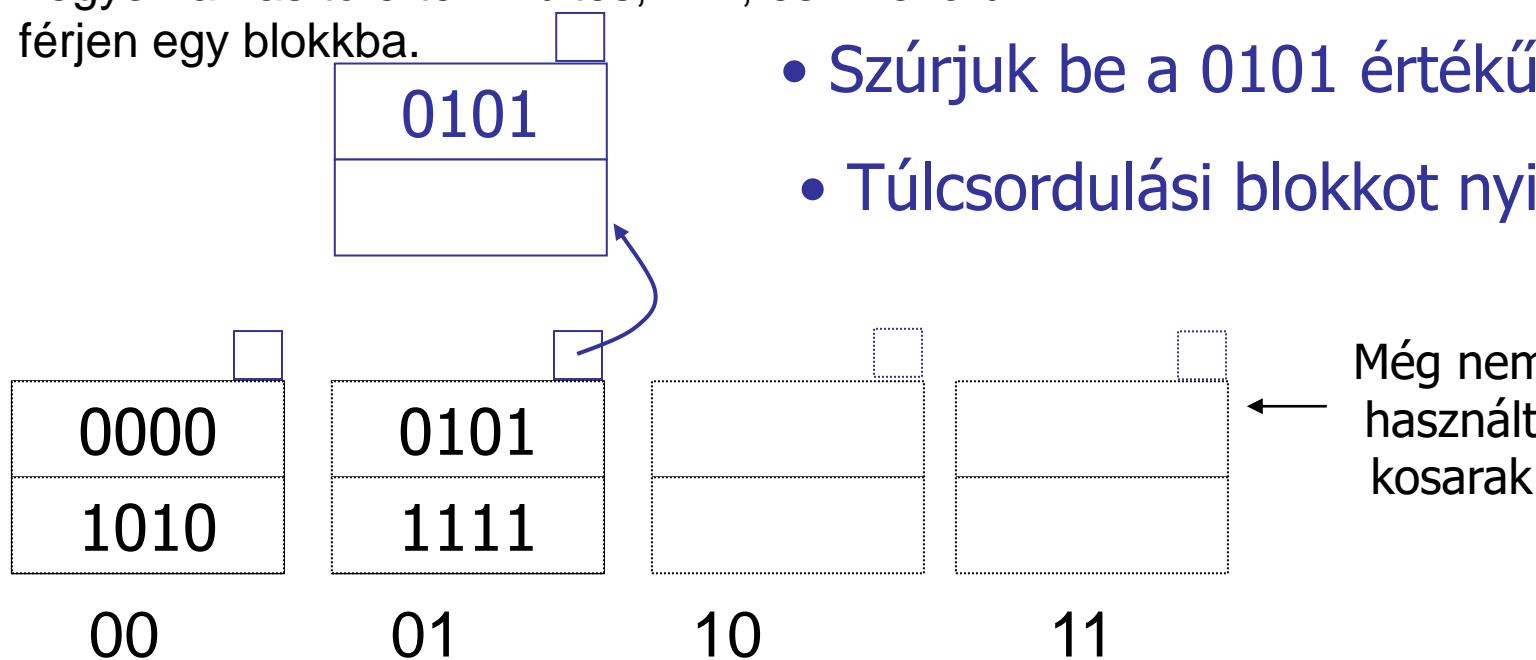
- A kosarak 1 vagy több blokkból is állhatnak.
- Új kosarat akkor nyitunk meg, ha egy előre megadott értéket elér a kosarakra jutó átlagos rekordszám.

(rekordok száma/kosarak száma > küszöb)

- **A kosarakat 0-tól kezdve sorszámozzuk, és a sorszámot binárisan ábrázoljuk.**
- Ha n kosarunk van, akkor a hasító függvény értékének **utolsó log(n) bitjével** megegyező sorszámú kosárba tesszük, ha van benne hely. Ha nincs, akkor hozzáláncolunk egy új blokkot és abba tesszük.
- Ha nincs megfelelő sorszámú kosár, akkor abba a sorszámú kosárba tesszük, amely csak az első bitjében különbözik a keresett sorszámtól.

Indexelés

Legyen a hasító érték 4 bites, $i=2$, és 2 rekord férjen egy blokkba.



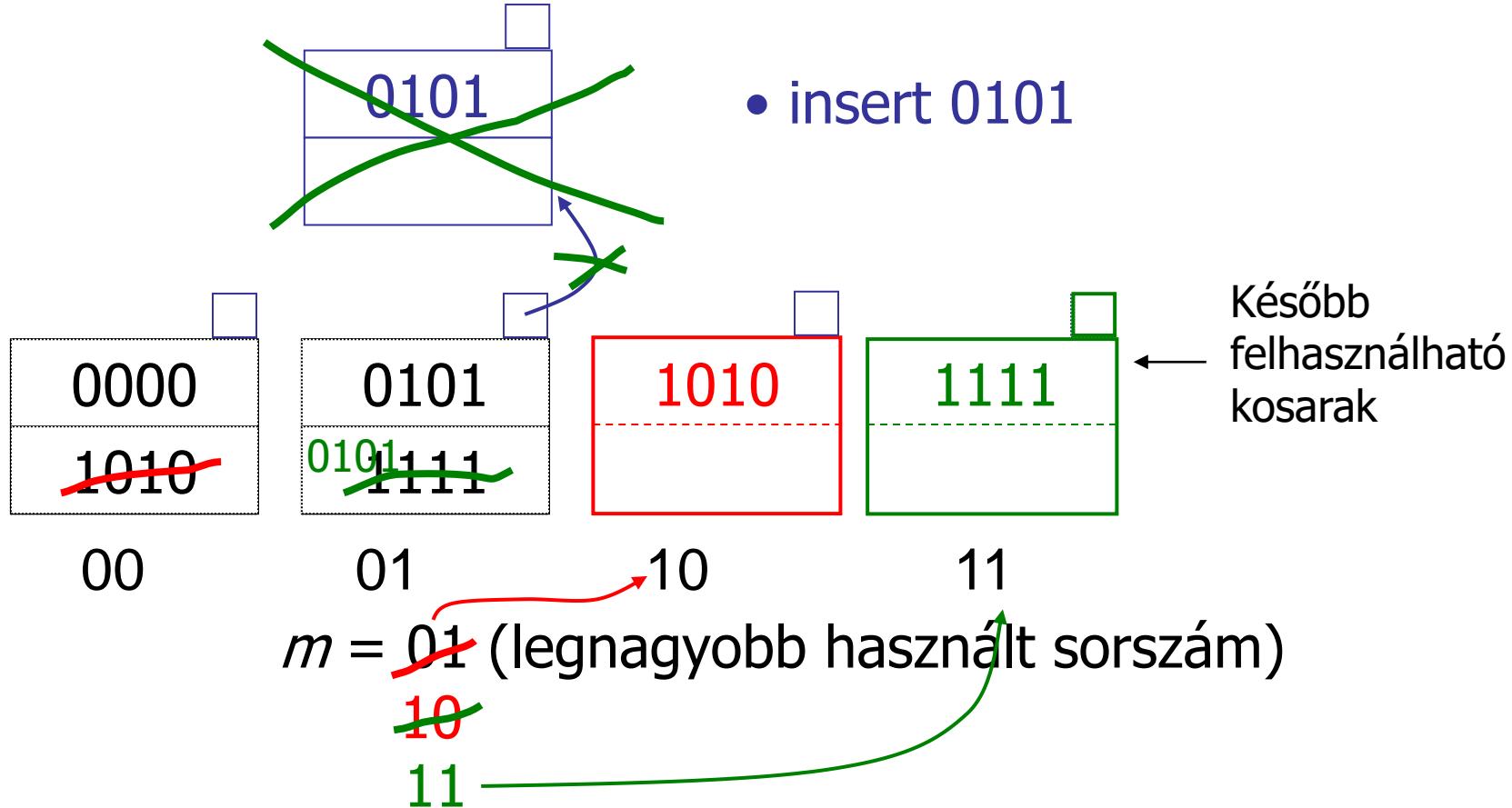
- Szűrjuk be a 0101 értékű rekordot!
- Túlcordulási blokkot nyitunk.

$m = 01$ (a legnagyobb sorszámú) blokk sorszáma

Szabály: Ha $h(x)[i] \leq m$, akkor a rekordot tegyük a $h(x)[i]$ kosárba,
különben pedig a $h(x)[i] - 2^{i-1}$ kosárba!

Megjegyzés: $h(x)[i]$ és $h(x)[i] - 2^{i-1}$ csak az első bitben különbözik!

Indexelés

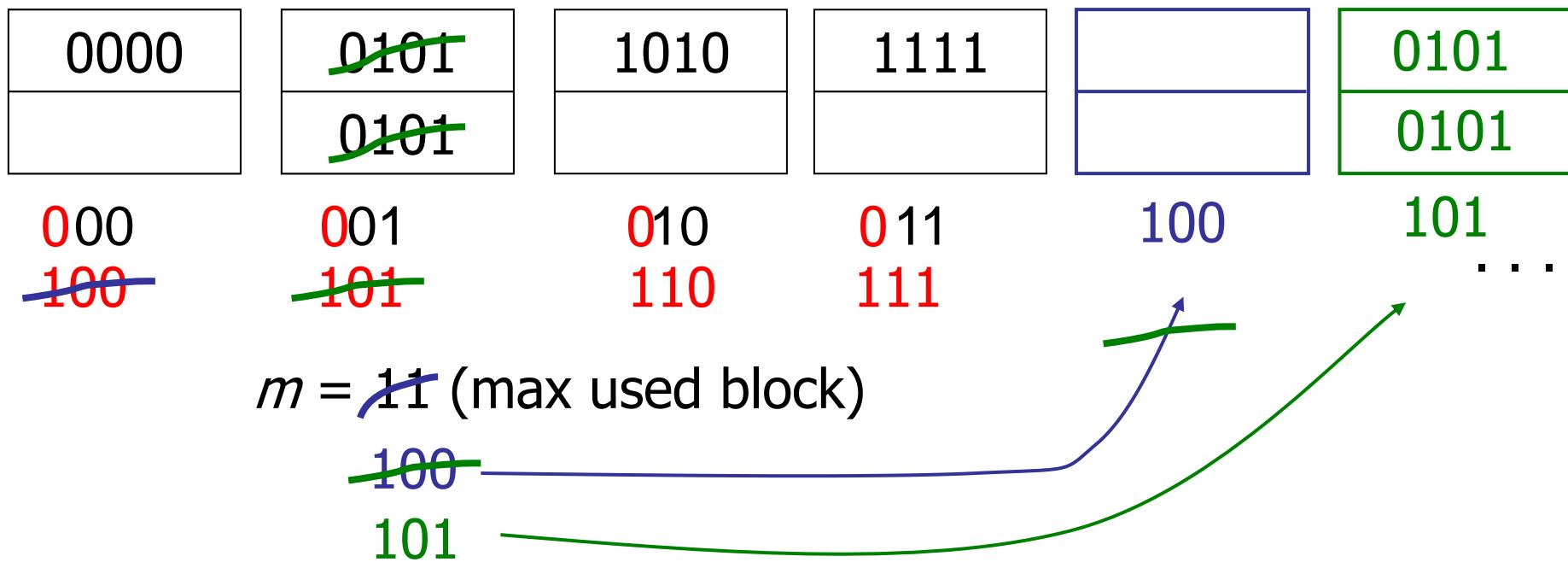


Tegyük fel, hogy átléptük a küszöbszámot, és ezért új kosarat kell nyitni, majd az első bitben különböző sorszámú kosárból át kell tenni ebbe az egyező végződésű rekordokat.

Indexelés

Ha i bitet használunk és 2^i kosarunk van, akkor a következő kosárnyitás előtt i-t megnöveljük 1-gyel, és az első bitben különböző sorszámú kosárból áttöljük a szükséges rekordokat és így tovább.

$$i = \cancel{2} \ 3$$





Összefoglalás:

Fizikai tervezés, alapfogalmak, jelölések, költségek, kupac (szekvenciális) és hasításon alapuló szervezés

Köszönöm a figyelmet!

Fizikai tervezek (folytatás)

- Paraméterek, költségek
- Fizikai fájlszervezés,
 - szekvenciális (kupac), hasító indexek (statikus, dinamikus (kiterjeszthető, lineáris))
 - Rendezett állomány, elsődleges, másodlagos indexek, többszintű indexek, B^+ -fák, B^* -fák
- Műveletek megvalósítása, kiszámítási költség, outputmérétek
- Optimális fizikai terv meghatározása

Indexelés

- **Rendezett állomány**
- Egy **rendező mező** alapján rendezett, azaz a blokkok láncolva vannak, és a következő blokkban nagyobb értékű rekordok szerepelnek, mint az előzőben.
- Ha a rendező mező és **kereső mező** nem esik egybe, akkor kupac szervezést jelent.
- Ha a rendező mező és kereső mező egybeesik, akkor **bináris (logaritmikus) keresést** lehet alkalmazni:
 - **beolvassuk a középső blokkot,**
 - ha nincs benne az A=a értékű rekord, akkor eldöntjük, hogy a blokklánc második felében, vagy az első felében szerepelhet-e egyáltalán,
 - beolvassuk a felezett blokklánc középső blokkját,
 - addig folytatjuk, amíg megtaláljuk a rekordot, vagy a vizsgálandó maradék blokklánc már csak 1 blokkból áll.
- **Keresési idő:** $\log_2(B)$

Indexelés

- **Beszúrás:**
 - keresés + üres hely készítés miatt a rekordok eltolása az összes blokkban, az adott találati blokktól kezdve ($B/2$ blokkot be kell olvasni, majd az eltolások után visszaírni=B művelet)
- **Szokásos megoldások:**
- **Gyűjtő (túlcsordulási) blokk használata:**
 - az új rekordok számára nyitunk egy blokkot, ha betelik hozzáláncunk egy újabb blokkokat,
 - keresést 2 helyen végezzük: **$\log_2(B-G)$** költséggel keresünk a rendezett részben, és ha nem találjuk, akkor a gyűjtőben is megnézzük (**G** blokkművelet, ahol **G a gyűjtő mérete**), azaz az összköltség: **$\log_2(B-G)+G$**
 - ha a G túl nagy a $\log_2(B)$ – hez képest, akkor újrarendezzük a teljes fájlt (a rendezés költsége **$B*\log_2(B)$**)

Indexelés

- **Üres helyeket** hagyunk a blokkokban:
 - például félig üresek a blokkok:
 - a keresés után 1 blokkművelettel visszaírjuk a blokkot, amibe beírtuk az új rekordot,
 - tárméret **2*B** lesz
 - keresési idő: $\log_2(2^*B) = \mathbf{1+\log_2(B)}$
 - ha betelik egy blokk, vagy elér egy határt a telítettsége, akkor 2 blokkba osztjuk szét a rekordjait, a rendezettség fenntartásával.
- **Törlés:**
 - keresés + a törlés elvégzése, vagy a törlési bit beállítása után visszaírás (1 blokkírás)
 - túl sok törlés után újraszervezés
- **Frissítés: törlés + beszúrás**

Indexelés

- **Indexek használata:**

- keresést gyorsító segédstruktúra
- több mezőre is lehet indexet készíteni
- az index tárolása növeli a tárméretet
- **nem csak a főfájlt, hanem az indexet is karban kell tartani, ami plusz költséget jelent**
- ha a keresési mező egyik indexmezővel sem esik egybe, akkor kupac szervezést jelent

- **Az indexrekordok szerkezete:**

- **(a,p)**, ahol a egy érték az indexelt oszlopból, p egy **blokkmutató**, arra a blokkra mutat, amelyben az **A=a** értékű rekordot tároljuk.
- az **index minden sorrendben** az indexértékek szerint
- Oracle SQL-ben:
- **egyszerű index:**
 - **CREATE INDEX supplier_idx
ON supplier (supplier_name);**
- **összetett index:**
 - **CREATE INDEX supplier_idx
ON supplier (supplier_name, city)
COMPUTE STATISTICS;**
 - -- az optimalizáláshoz szükséges statisztikák elkészítésével --

Indexelés

- Elsődleges index:
 - főfájl is rendezett
 - csak 1 elsődleges indexet lehet megadni (mert csak egyik mező szerint lehet rendezett a főfájl).
 - elég a főfájl minden blokkjának legkisebb rekordjához készíteni indexrekordot
 - indexrekordok száma: $T(I)=B$ (**ritka index**)
 - indexrekordból sokkal több fér egy blokkba, mint a főfájl rekordjaiból:
bf(I)>>bf, azaz az indexfájl sokkal kisebb rendezett fájl, mint a főfájl:
B(I) = B / bf(I) << B=T / bf
- Keresési idő:
 - az indexfájlban nem szerepel minden érték, ezért csak **fedő értéket kereshetünk, a legnagyobb olyan indexértéket, amely a keresett értéknél kisebb vagy egyenlő**
 - fedő érték keresése az index rendezettsége miatt bináris kereséssel történik:
log₂(B(I))
 - a fedő indexrekordban szereplő blokkmutatónak megfelelő blokkot még be kell olvasni
 - **1+log₂(B(I)) << log₂(B) (rendezett eset)**
- Módosítás:
 - rendezett fájlba kell beszúrni
 - ha az első rekord változik a blokkban, akkor az indexfájlba is be kell szúrni, ami szintén rendezett
 - megoldás: **üres helyeket hagyunk a főfájl, és az indexfájl blokkjaiban is.**
Ezzel a tárméret duplázódhat, de a beszúrás legfeljebb egy főrekord és egy indexrekord visszaírását jelenti.



Indexelés

Elsődleges index

Az adatfájl rendezett, ezért elég a blokkok első rekordjaihoz indexrekordokat tárolni.

Ritka index

| | |
|----|--|
| 10 | |
| 30 | |
| 50 | |
| 70 | |

| | |
|-----|--|
| 90 | |
| 110 | |
| 130 | |
| 150 | |

| | |
|-----|--|
| 170 | |
| 190 | |
| 210 | |
| 230 | |

Adatállomány

| | |
|----|--|
| 10 | |
| 20 | |

| | |
|----|--|
| 30 | |
| 40 | |

| | |
|----|--|
| 50 | |
| 60 | |

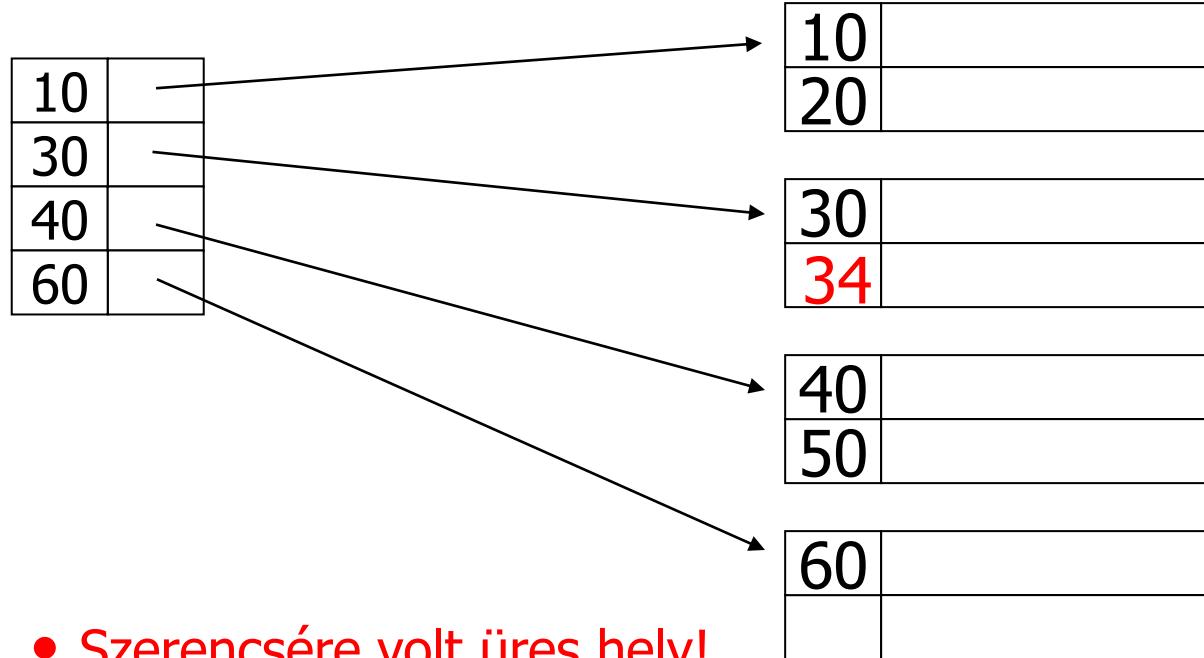
| | |
|----|--|
| 70 | |
| 80 | |

| | |
|-----|--|
| 90 | |
| 100 | |

Indexelés

Beszúrás ritka index esetén:

Vigyük be a 34-es rekordot!

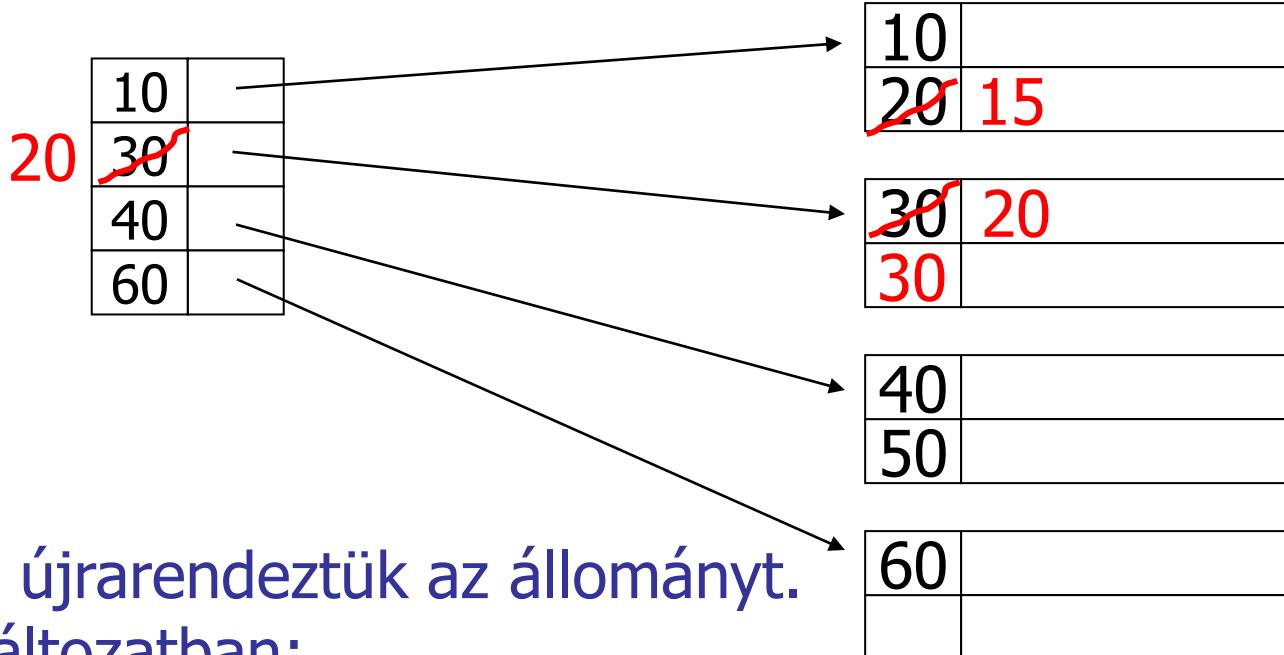


- Szerencsére volt üres hely!

Indexelés

Beszúrás ritka index esetén:

Vigyük be a 15-ös rekordot!

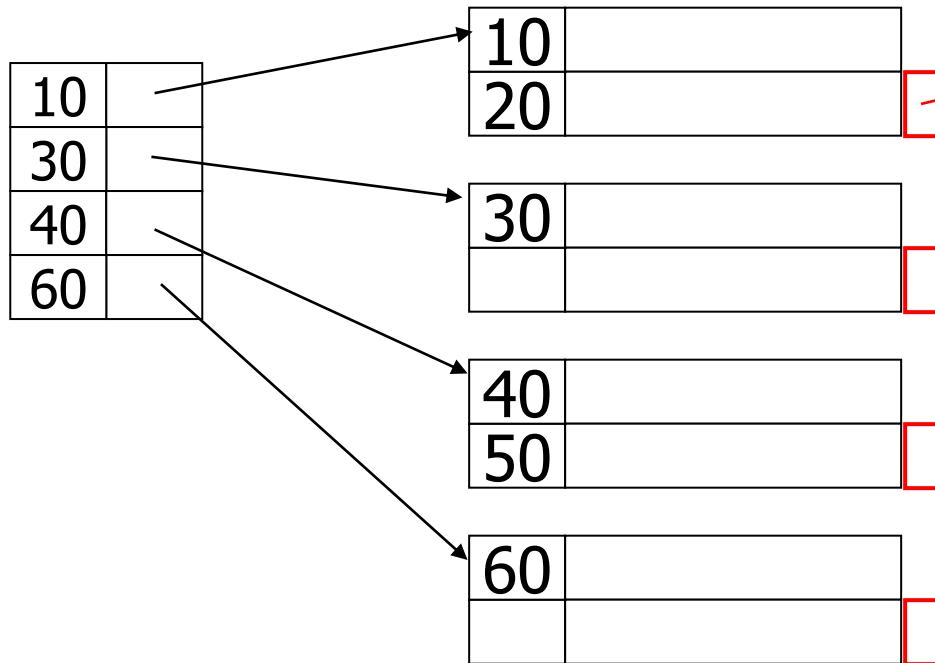


- Azonnal újrarendeztük az állományt.
- Másik változatban:
 - túlcsordulási blokkot láncolunk a blokkhoz

Indexelés

Beszúrás ritka index esetén:

Vigyük be a 25-ös rekordot!

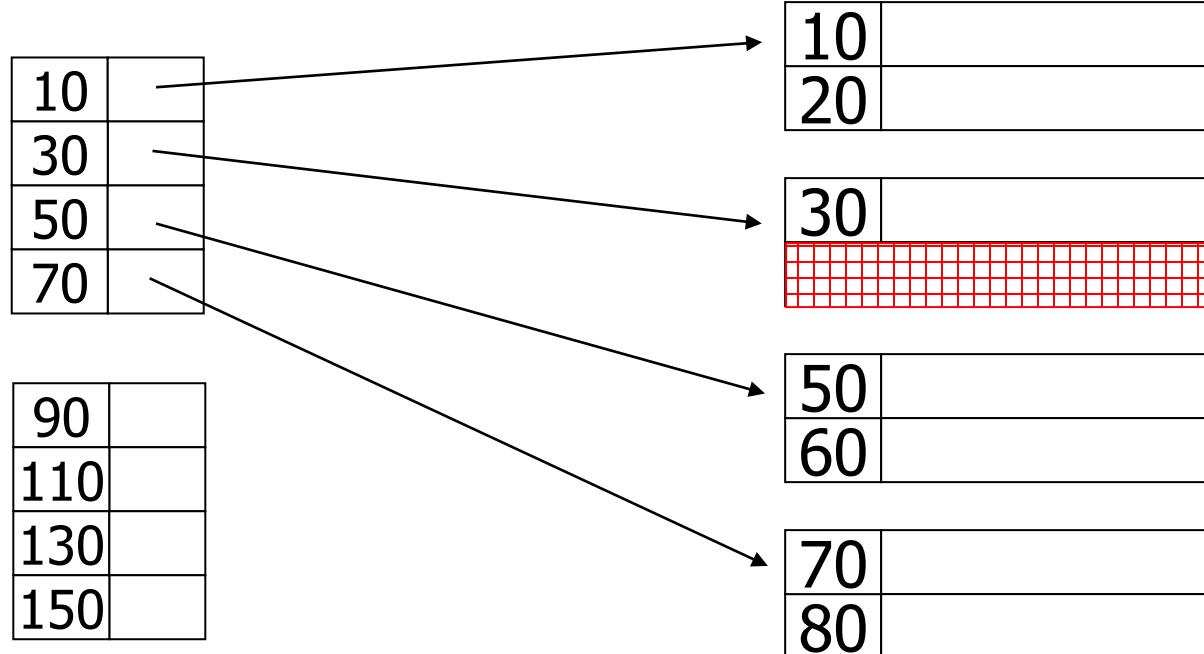


Túlcordulási blokkot nyitunk,
és későbbre halasztjuk az
újrarendezést.

Indexelés

Törlés ritka indexből:

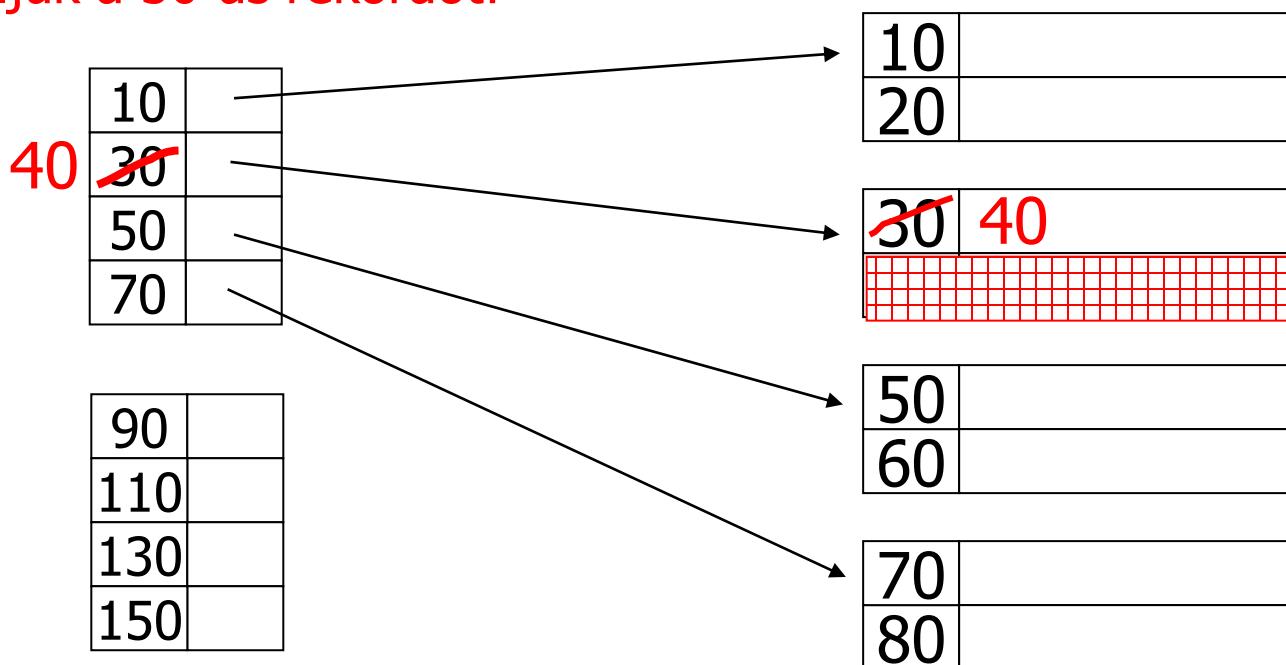
Töröljük a 40-es rekordot!



Indexelés

Törlés ritka indexből:

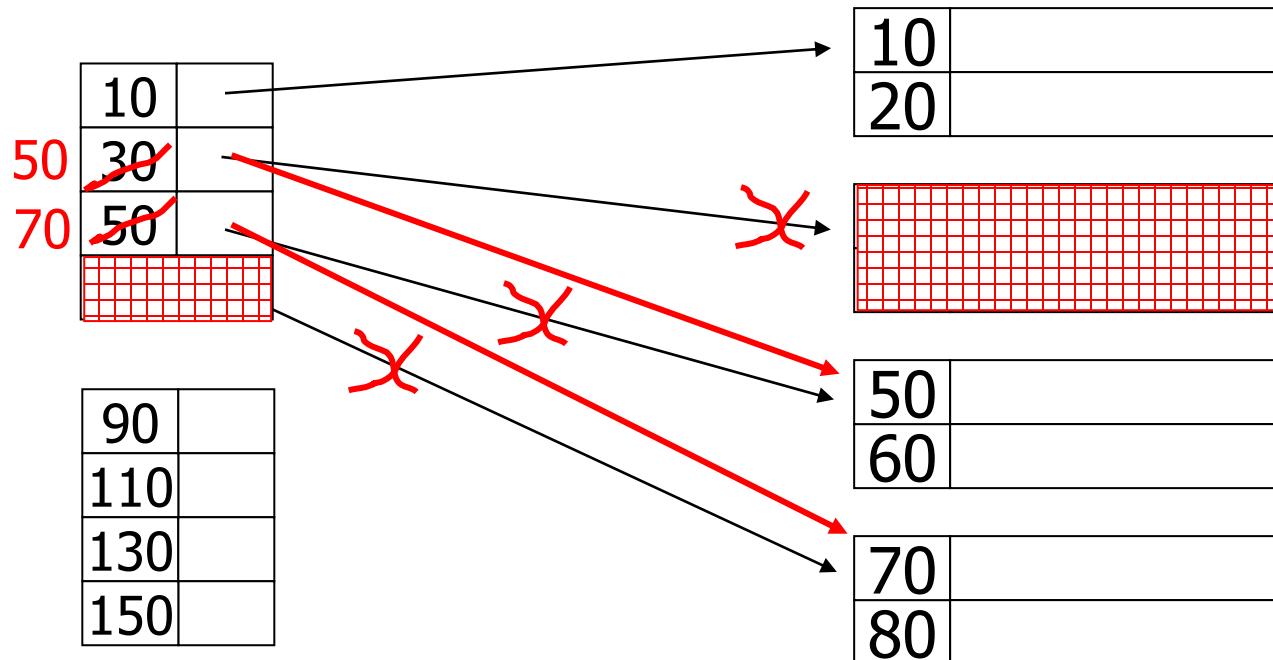
Töröljük a 30-as rekordot!



Indexelés

Törlés ritka indexből:

Töröljük a 30-as és 40-es rekordot!



Indexelés

- **Másodlagos index:**

- főfájl rendezetlen (az indexfájl minden rekordjához kell készíteni indexrekordot)
- több másodlagos indexet is meg lehet adni
- a főfájl minden rekordjához kell készíteni indexrekordot
- indexrekordok száma: $T(I)=T$ (sűrű index)
- indexrekordból sokkal több fér egy blokkba, mint a főfájl rekordjaiból: $bf(I) >> b$, azaz az indexfájl sokkal kisebb rendezett fájl, mint a főfájl:
- $B(I) = T/bf(I) << B=T/b$

- **Keresési idő:**

- az indexben keresés az index rendezettsége miatt bináris kereséssel történik: $\log_2(B(I))$
- a talált indexrekordban szereplő blokkmutatónak megfelelő blokkot még be kell olvasni
- $1+\log_2(B(I)) << \log_2(B)$ (rendezett eset)
- az elsődleges indexnél rosszabb a keresési idő, mert több az indexrekord

- **Módosítás:**

- a főfájl kupac szervezésű
- rendezett fájlba kell beszúrni
- ha az első rekord változik a blokkban, akkor az indexfájlba is be kell szúrni, ami szintén rendezett
- megoldás: üres helyeket hagyunk a főfájl, és az indexfájl blokkjaiban is. Ezzel a tárméret duplázódhat, de a beszúrás legfeljebb egy főrekord és egy indexrekord visszaírását jelenti.

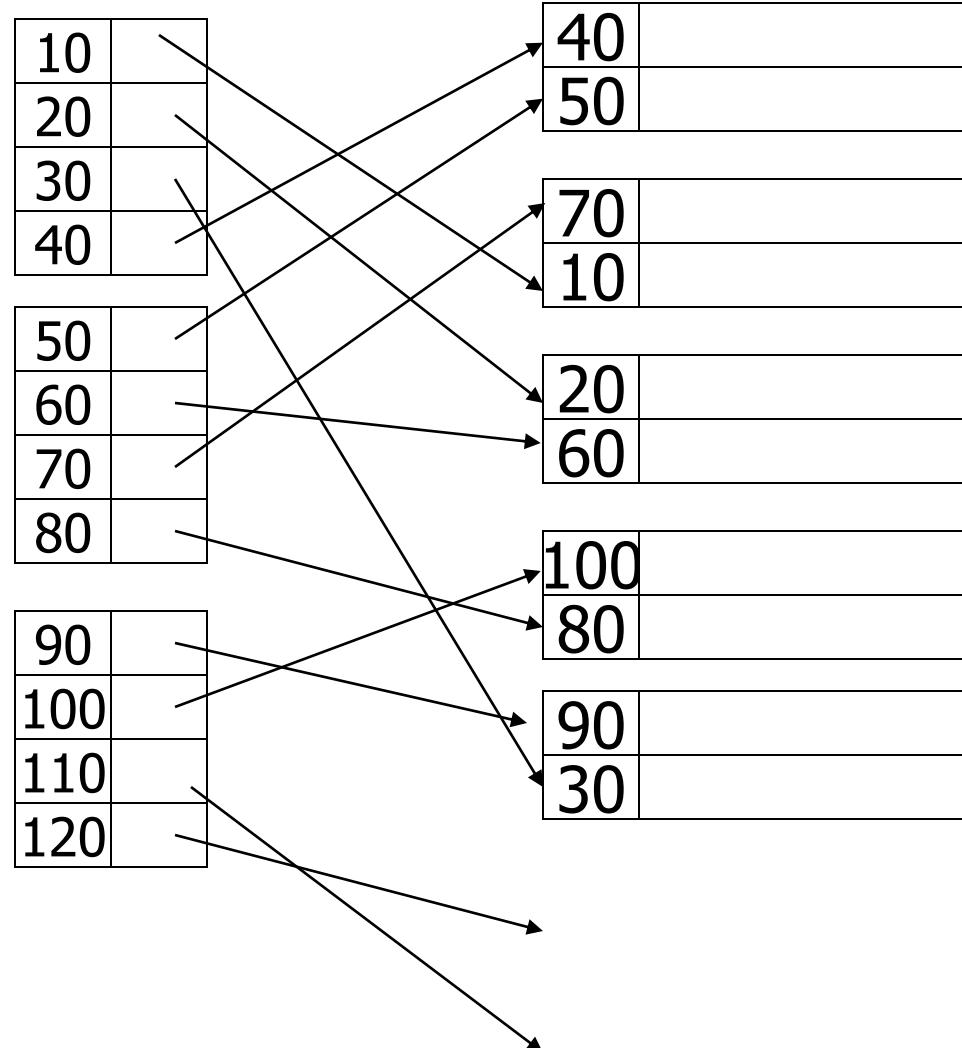


Indexelés

Másodlagos
index

Minden
rekordhoz
tartozik
indexrekord.

Sűrű index

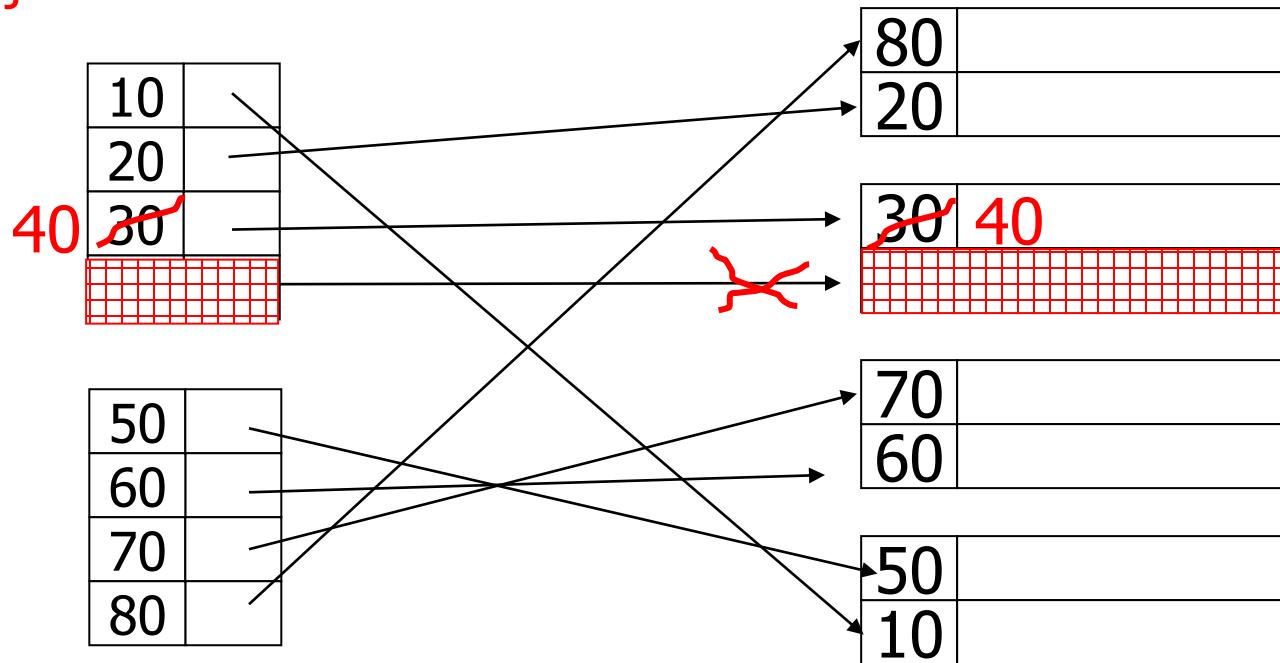


Adatállomány

Indexelés

Törlés sűrű indexből:

Töröljük a 30-as rekordot!



Indexelés

Mi történik, ha egy érték többször is előfordulhat?

Több megoldás is lehetséges. Először tegyük fel, hogy rendezett az állomány.

Sűrű index

| | |
|----|--|
| 10 | |
| 10 | |
| 10 | |
| 20 | |

| | |
|----|--|
| 10 | |
| 10 | |

| | |
|----|--|
| 10 | |
| 20 | |

| | |
|----|--|
| 20 | |
| 30 | |
| 30 | |
| 30 | |

| | |
|----|--|
| 20 | |
| 30 | |

| | |
|----|--|
| 30 | |
| 30 | |

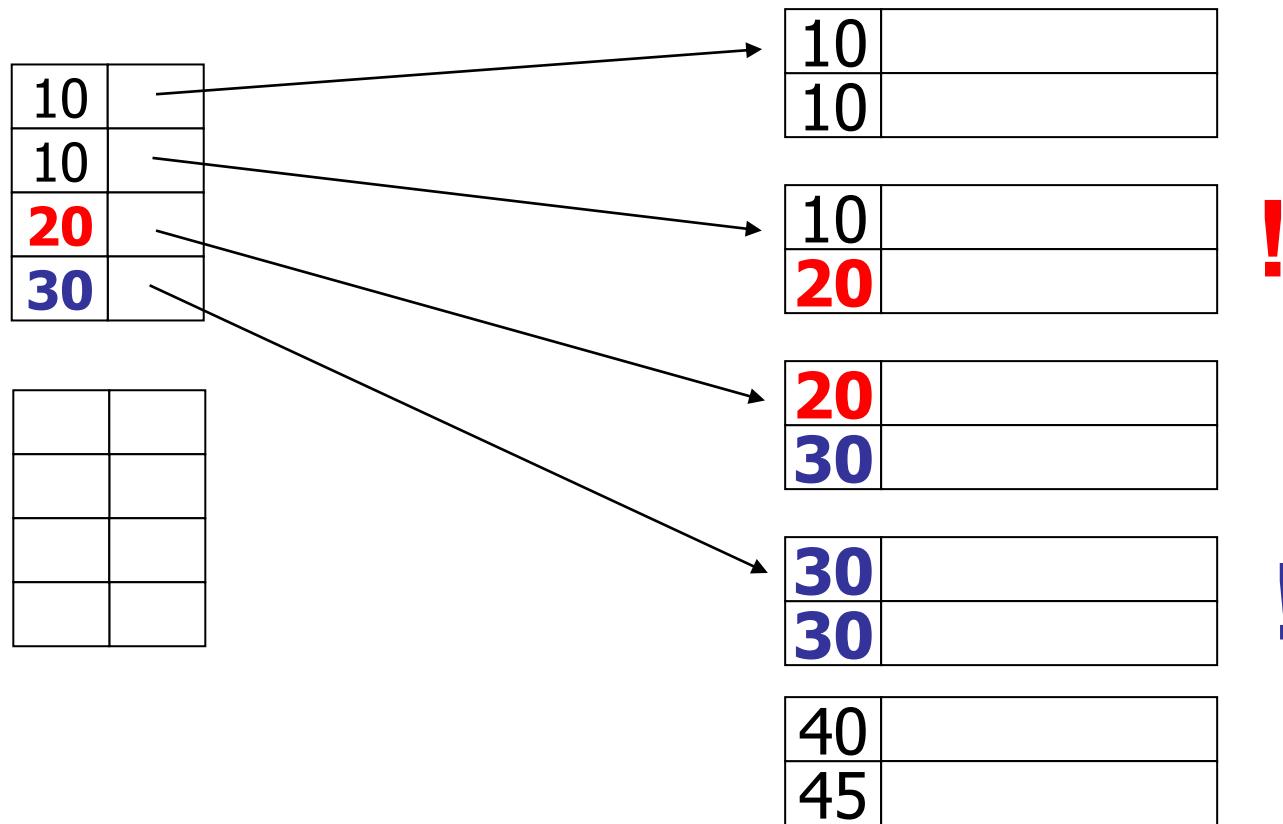
| | |
|----|--|
| 40 | |
| 45 | |

1. megoldás:

Minden rekordhoz tárolunk egy indexrekordot.

Indexelés

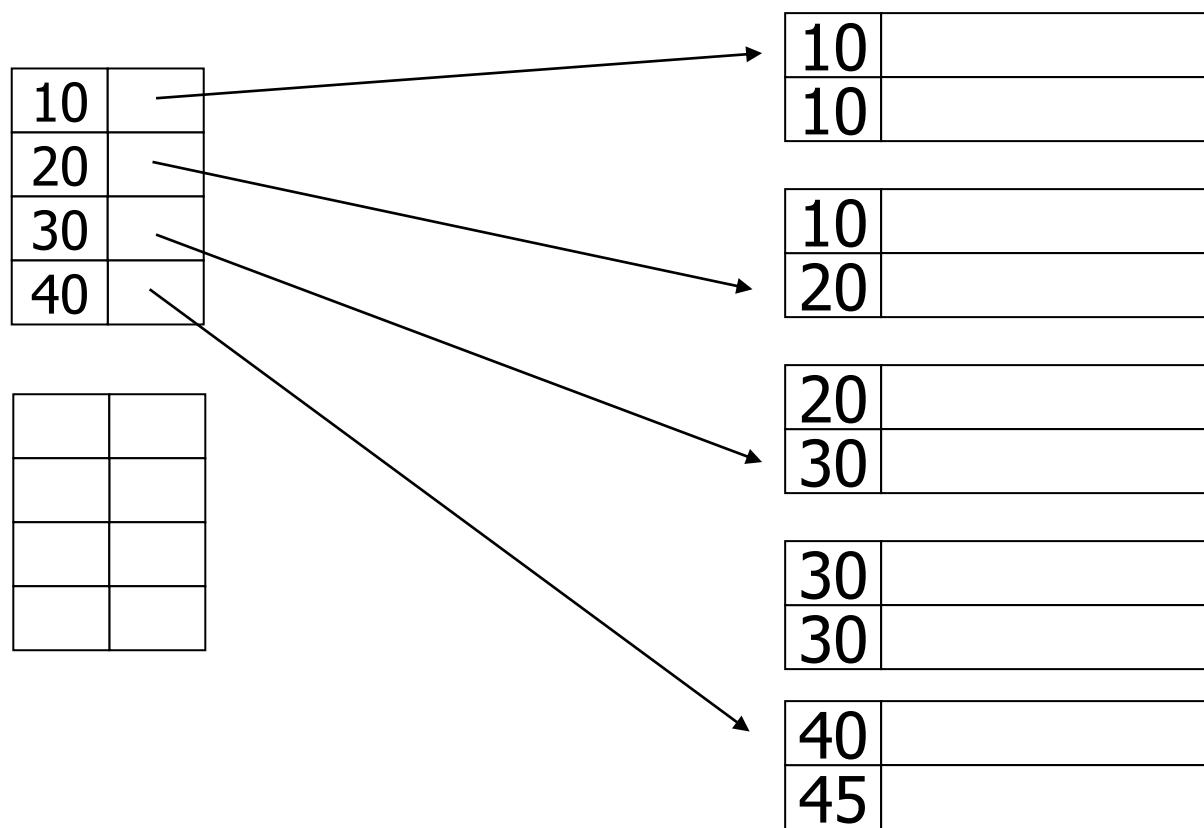
Vigyázat! Ritka index nem jó. A fedőértéknek megfelelő blokk előtti és utáni blokkokban is lehetnek találatok. Például, ha a 20-ast vagy a 30-ast keressük.



Indexelés

Rendezett állomány

Ritka index

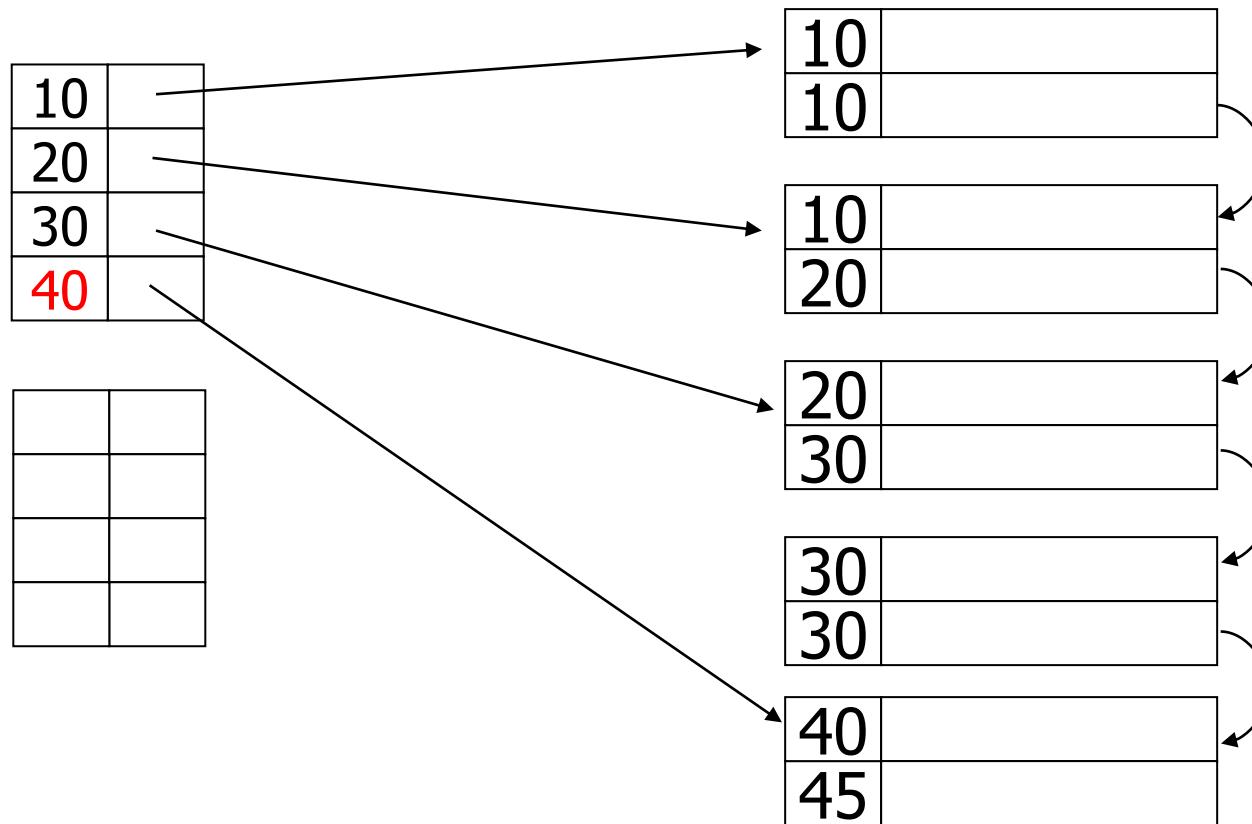


2. megoldás:

Rendezett állomány esetén csak az első előforduláshoz tárolunk egy indexrekordot.

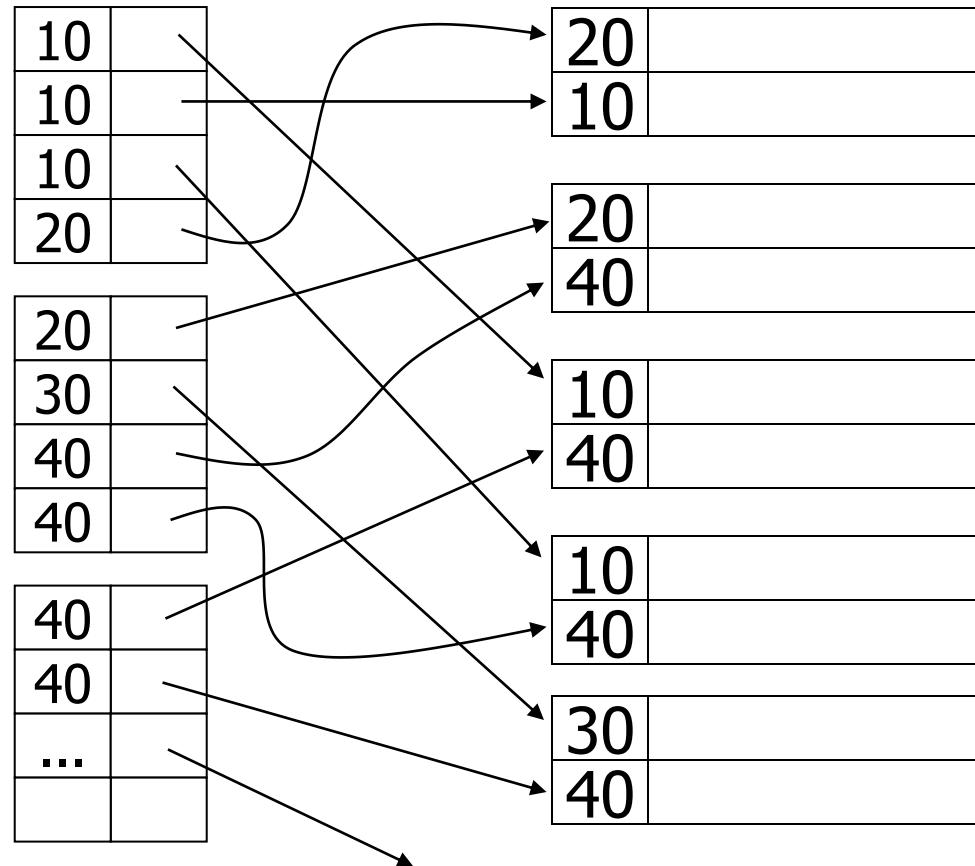
Indexelés

Rendezett állomány esetén nem az első rekordhoz, hanem az értékhez tartozó első előforduláshoz készítünk indexrekordot. Az adatállomány blokkjait láncoljuk.



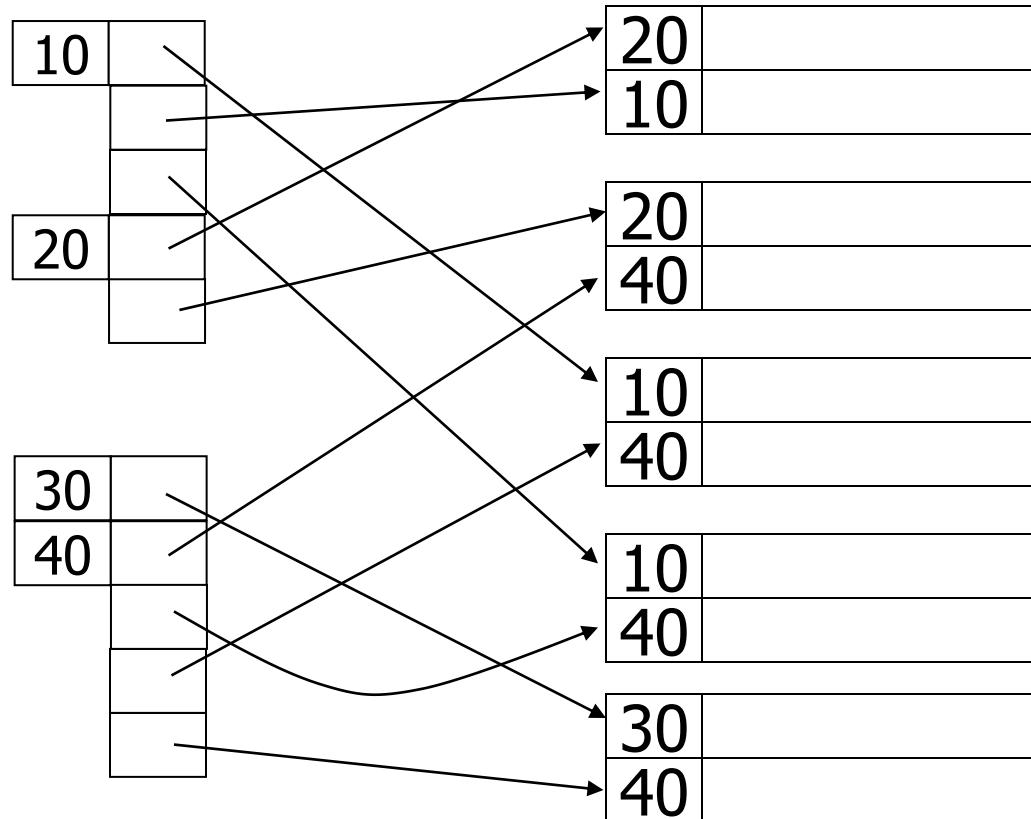
Indexelés

Ha nem rendezett az állomány, akkor nagy lehet a tárolási és keresési költség is:



Indexelés

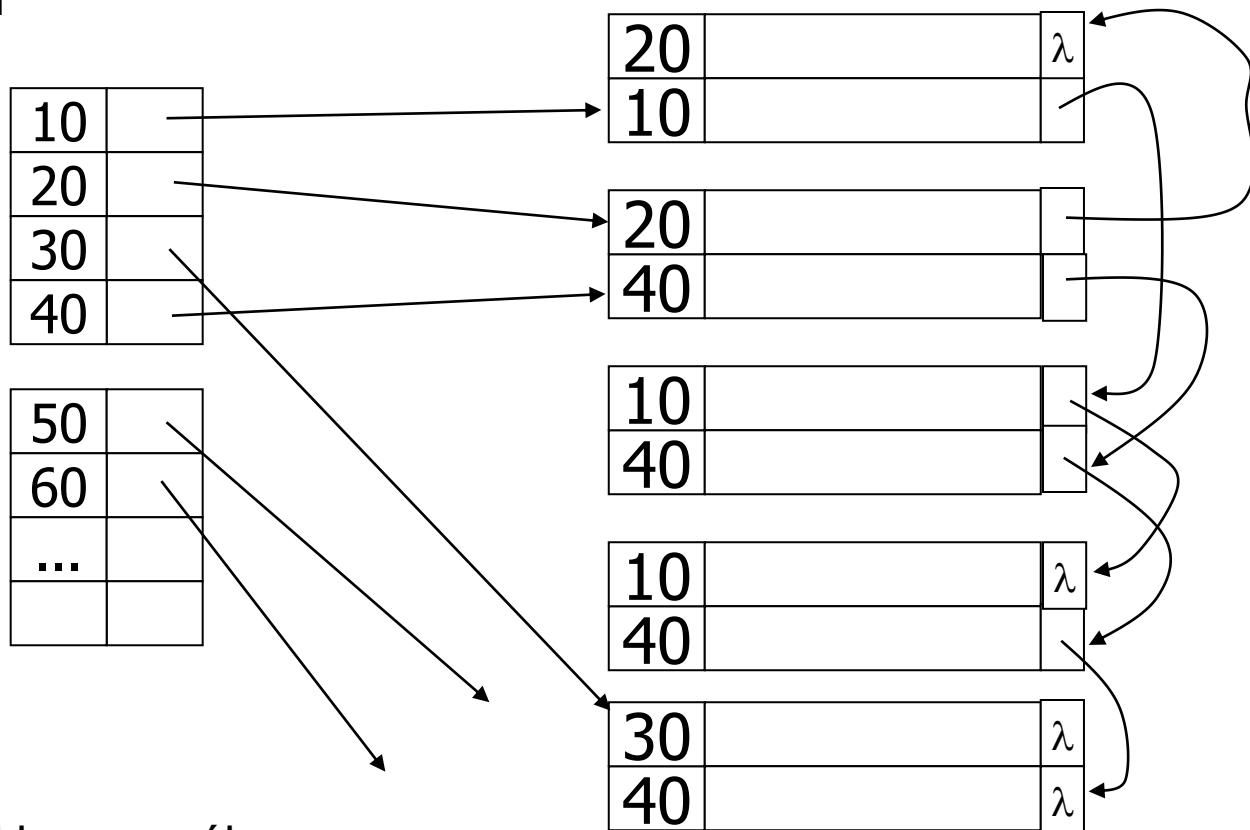
Egy lehetséges megoldás,
hogy az indexrekordok
szerkezetét módosítjuk:
(indexérték, mutatóhalmaz)



Probléma: változó
hosszú indexrekordok
keletkeznek

Indexelés

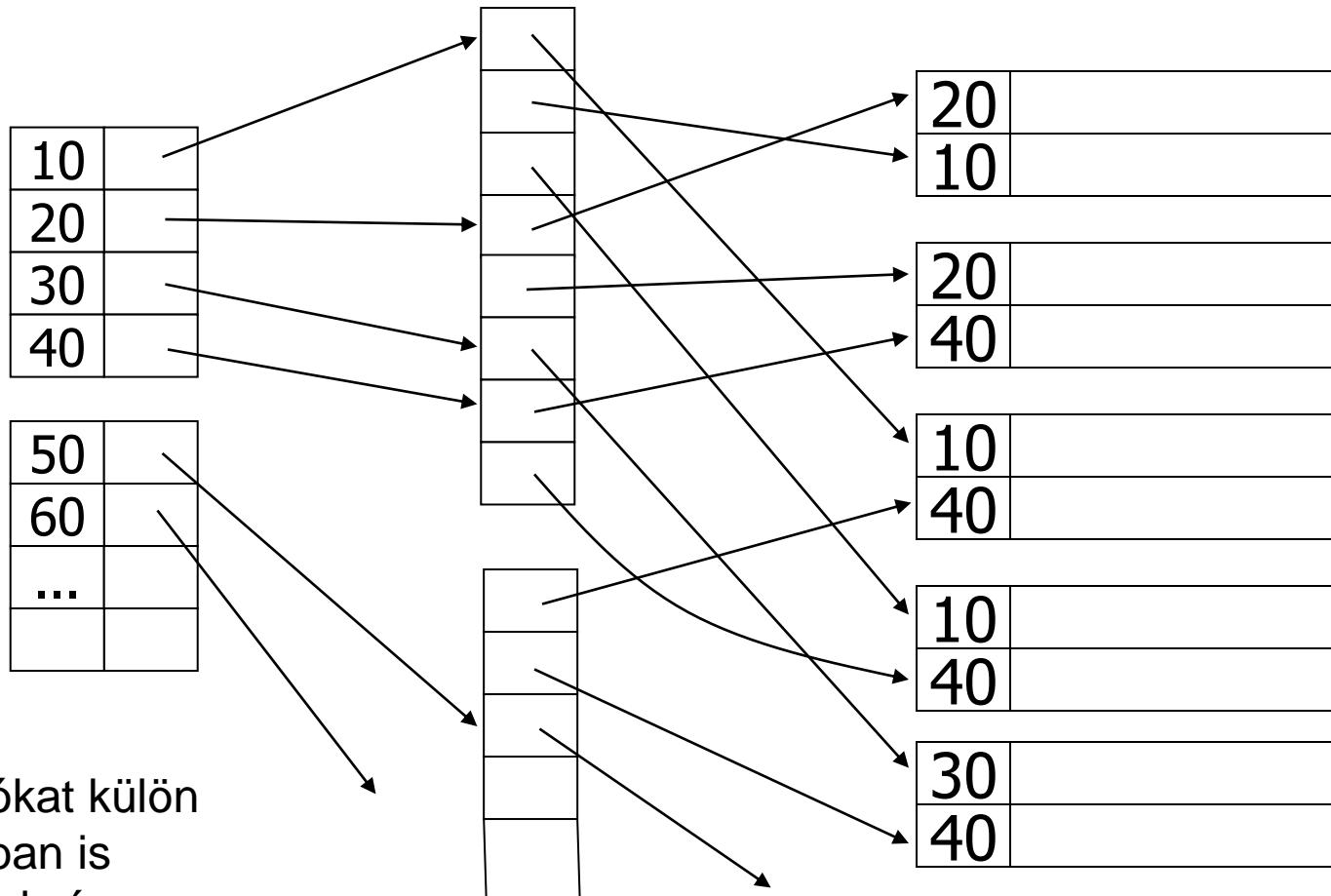
Összeláncolhatjuk az egyforma értékű rekordokat.



Probléma:

- a rekordokhoz egy új, mutató típusú mezőt kell adnunk
- követni kell a láncot

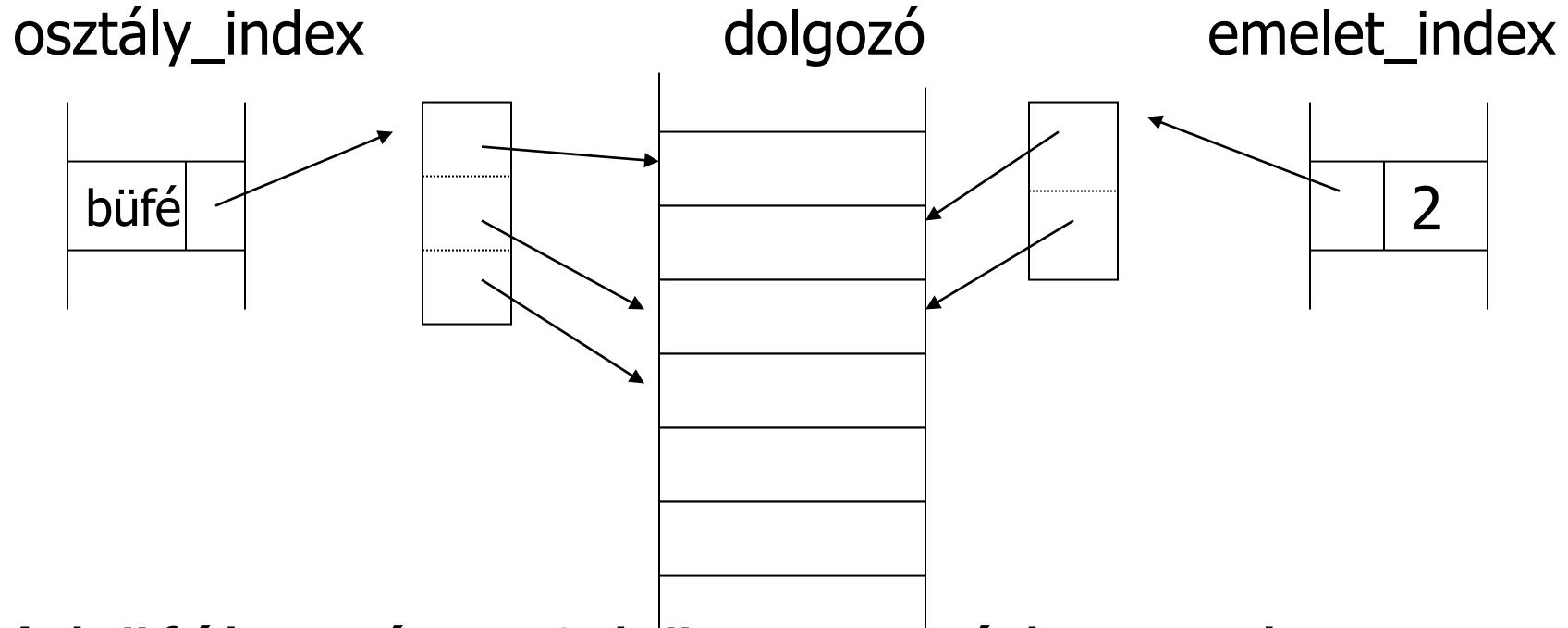
Indexelés



ELŐNY: több index esetén a logikai feltételek halmazműveletekkel kiszámolhatók.

Indexelés

select * from dolgozó where osztály='büfé' and emelet=2;



A büföhöz és a 2-höz tartozó kosarak metszetét kell képezni, hogy megkapjuk a keresett mutatókat.

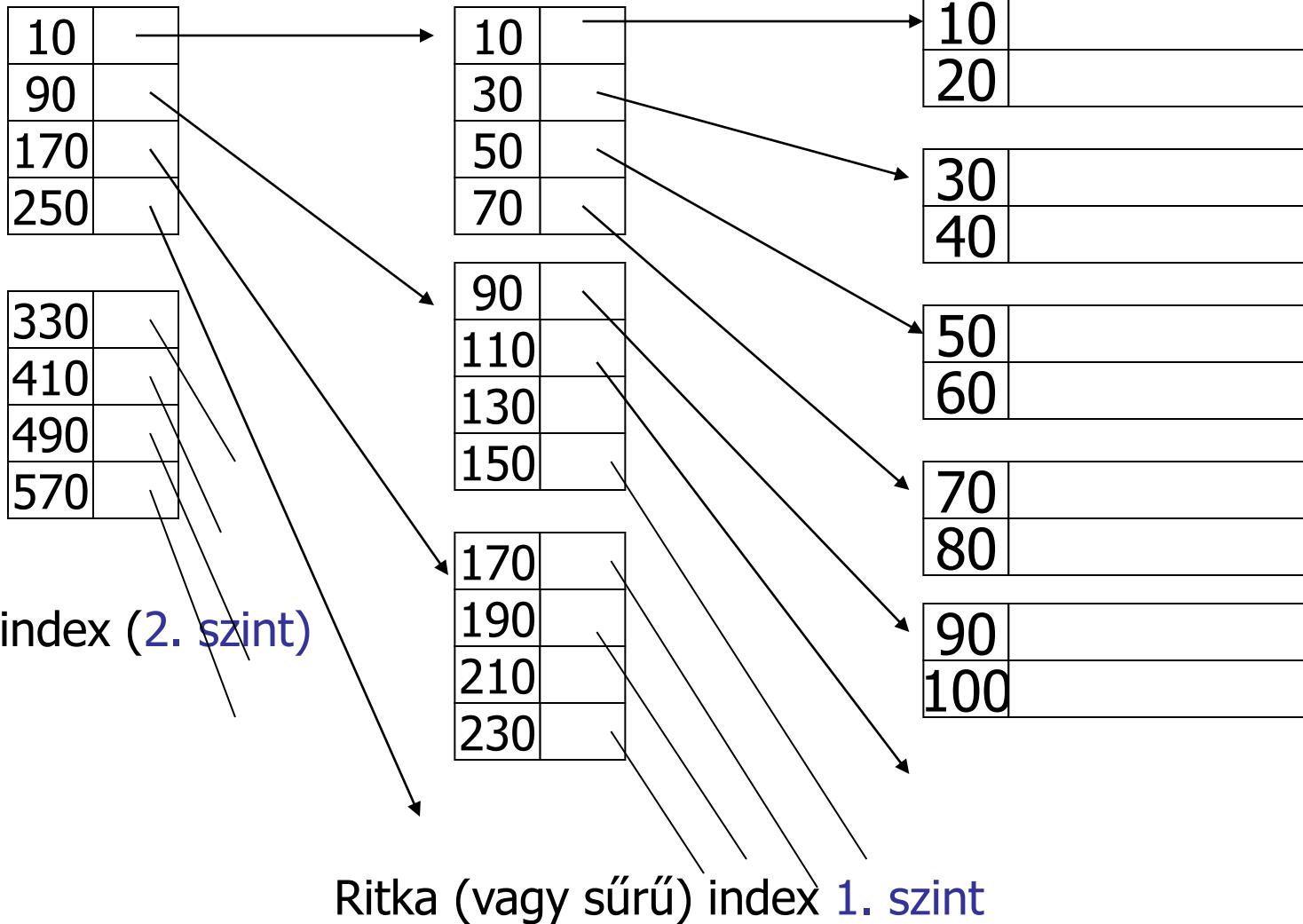
Indexelés

- **Klaszter (nyaláb, fürt)**
- Klaszterszervezés **egy tábla** esetén egy A oszlopra:
 - az azonos A-értékű sorok fizikailag egymás után blokkokban helyezkednek el.
 - **CÉL:** az első találat után az összes találatot megkapjuk soros beolvasással.
- **Klaszterindex:**
 - klaszterszervezésű fájl esetén index az A oszlopra
- Klaszterszervezés **két tábla** esetén az összes közös oszlopra:
 - a közös oszlopokon egyező sorok egy blokkban, vagy fizikailag egymás utáni blokkokban helyezkednek el.
 - **CÉL:** összekapcsolás esetén az összetartozó sorokat soros beolvasással megkaphatjuk.

Indexelés

Ha nagy az index, akkor az indexet is indexelhetjük.

Adatállomány



Indexelés

- **Többszintű index:**

- az indexfájl (1. indexszint) is fájl, ráadásul rendezett, így ezt is meg lehet indexelni, elsődleges indexsel.
- a főfájl lehet rendezett vagy rendezetlen (az indexfájl mindenkor rendezett)
- **t-szintű index:** az indexszinteket is indexeljük, összesen t szintig

Keresési idő:

- a t-ik szinten ($I^{(t)}$) bináris kereséssel **keressük meg a fedő indexrekordot**
- követjük a mutatót, minden szinten, és végül a főfájlban: **$\log_2(B(I^{(t)})) + t$ blokkolvasás**
- ha a legfelső szint 1 blokkból áll, akkor **t+1** blokkolvasást jelent. (**t=?**)
- **minden szint blokkolási faktora megegyezik**, mert egyforma hosszúak az indexrekordok.

Indexelés

| | FŐFÁJL | 1. szint | 2. szint | ... | t. szint |
|-------------------|--------|----------|----------------------|-----|--------------------------|
| blokkok száma | B | B/bf(I) | B/bf(I) ² | ... | B/bf(I) ^t |
| rekordok száma | T | B | B/bf(I) | ... | B/bf(I) ^(t-1) |
| blokkolási faktor | bf | bf(I) | bf(I) | ... | bf(I) |

- t-ik szinten 1 blokk: $1 = B/bf(I)^t$

azaz $t = \log_{bf(I)} B < \log_2(B)$ azaz jobb a rendezett fájlszervezésnél.

- $\log_{bf(I)} B < \log_2(B)$ is teljesül általában, így az egyszintű indexeknél is gyorsabb

Indexelés

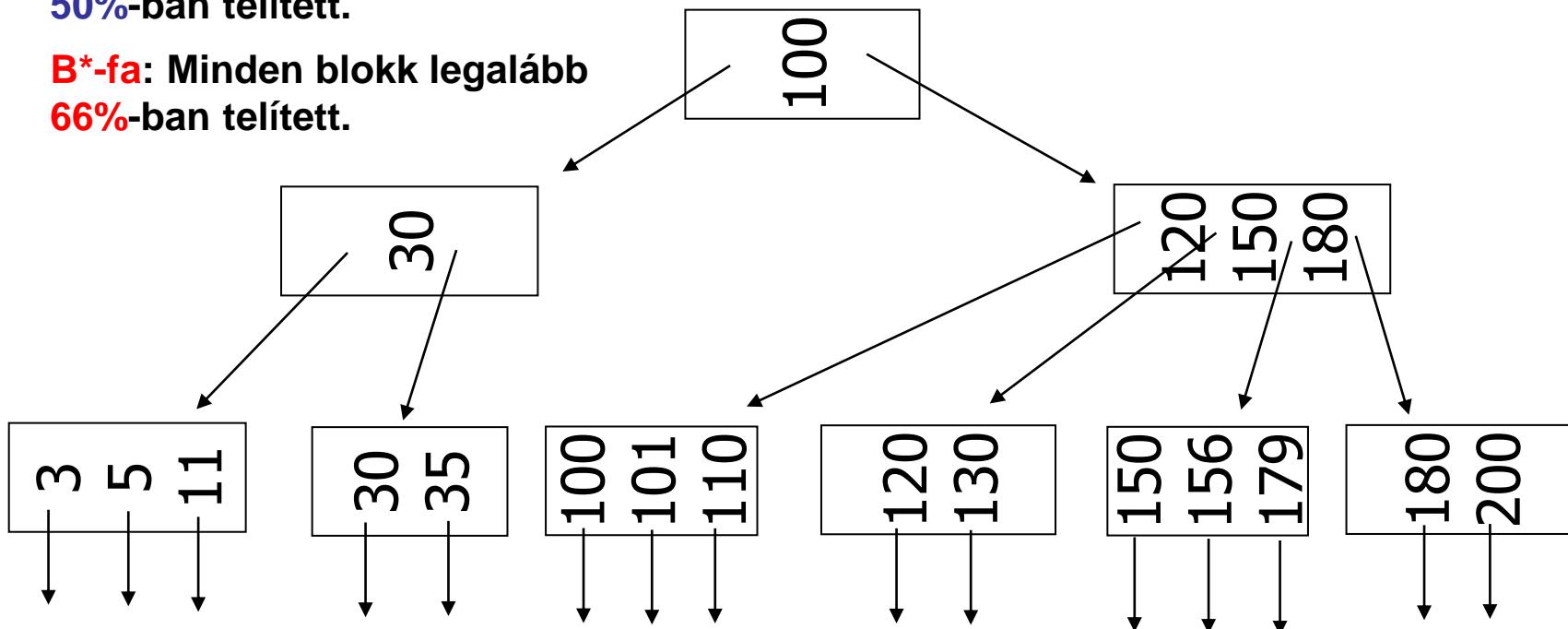
A többszintű indexek közül a **B⁺-fák**, **B^{*}-fák** a legelterjedtebbek.

B⁺-fa: minden blokk legalább 50%-ban telített.

B^{*}-fa: minden blokk legalább 66%-ban telített.

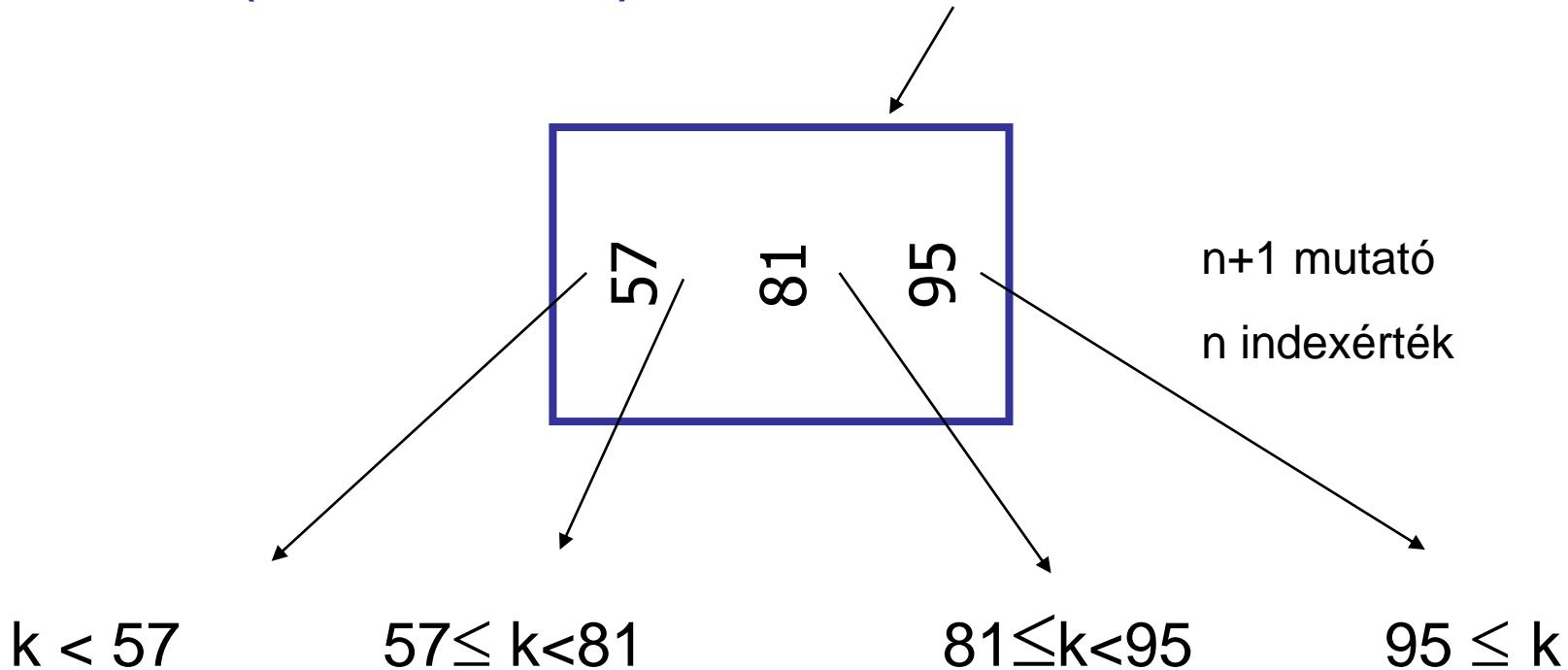
B⁺-fa: a szerkezeten kívül a telítettséget biztosító karbantartó algoritmusokat is beleértjük

Gyökér



Indexelés

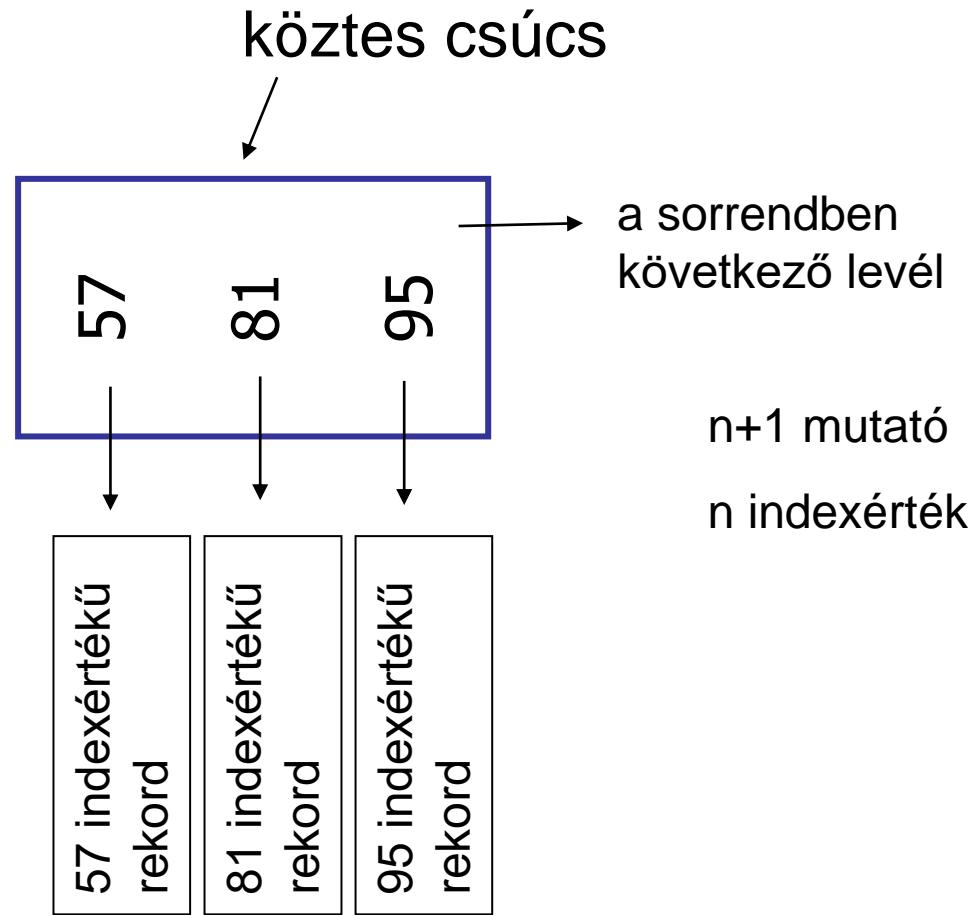
Köztes (nem-levél) csúcs szerkezete



Ahol k a mutató által meghatározott részben
(részgráfban) szereplő tetszőleges indexérték

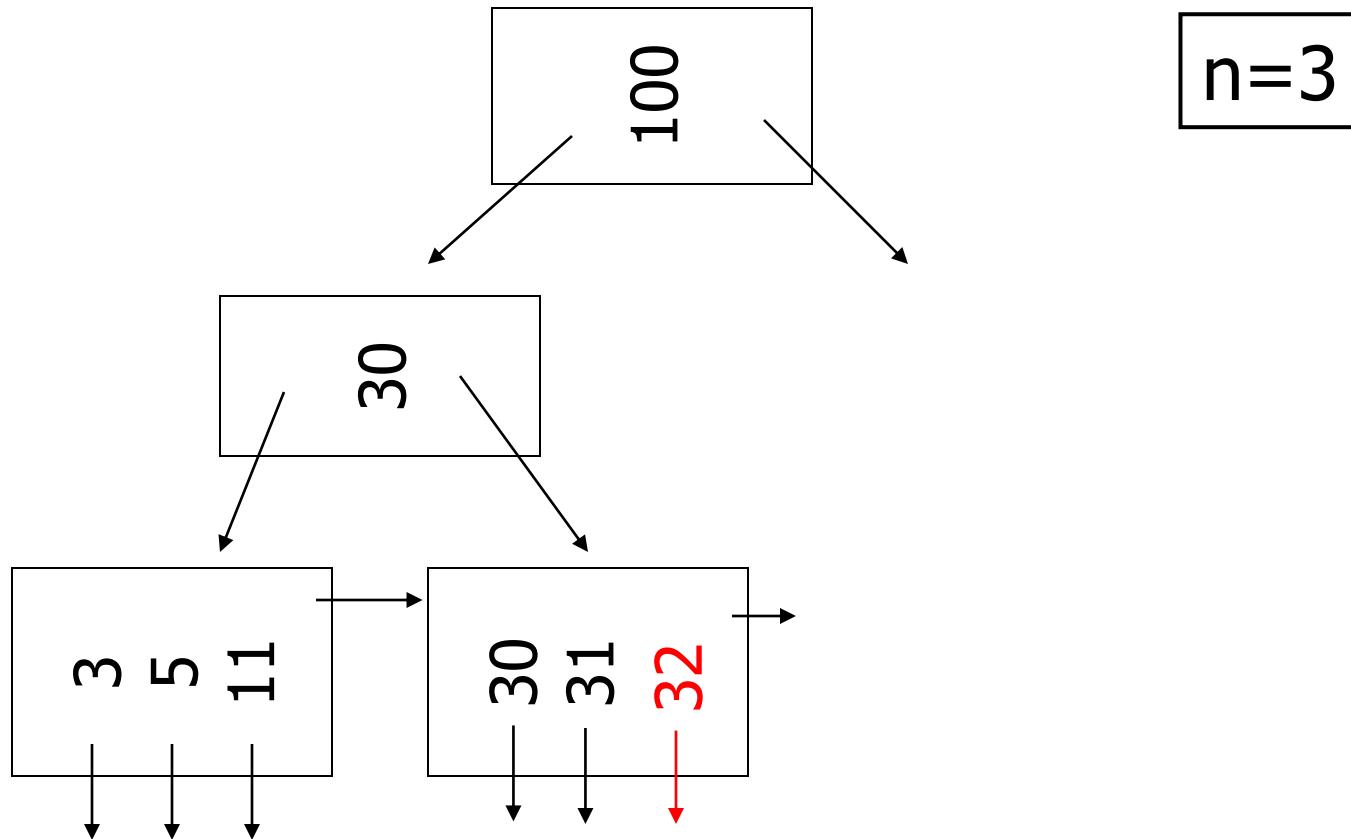
Indexelés

Levél csúcs szerkezete



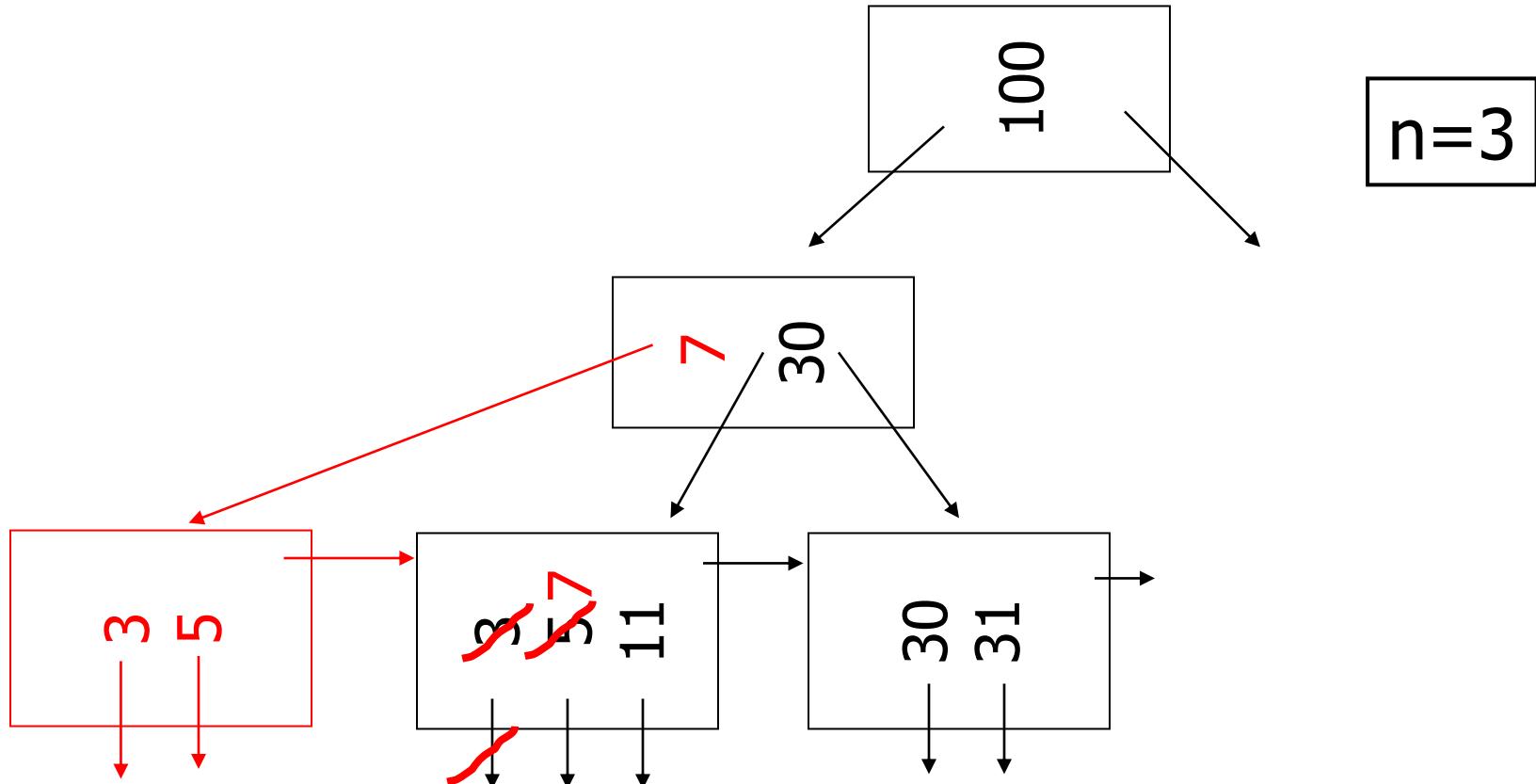
Indexelés

Szúrjuk be a 32-es indexértékű rekordot!



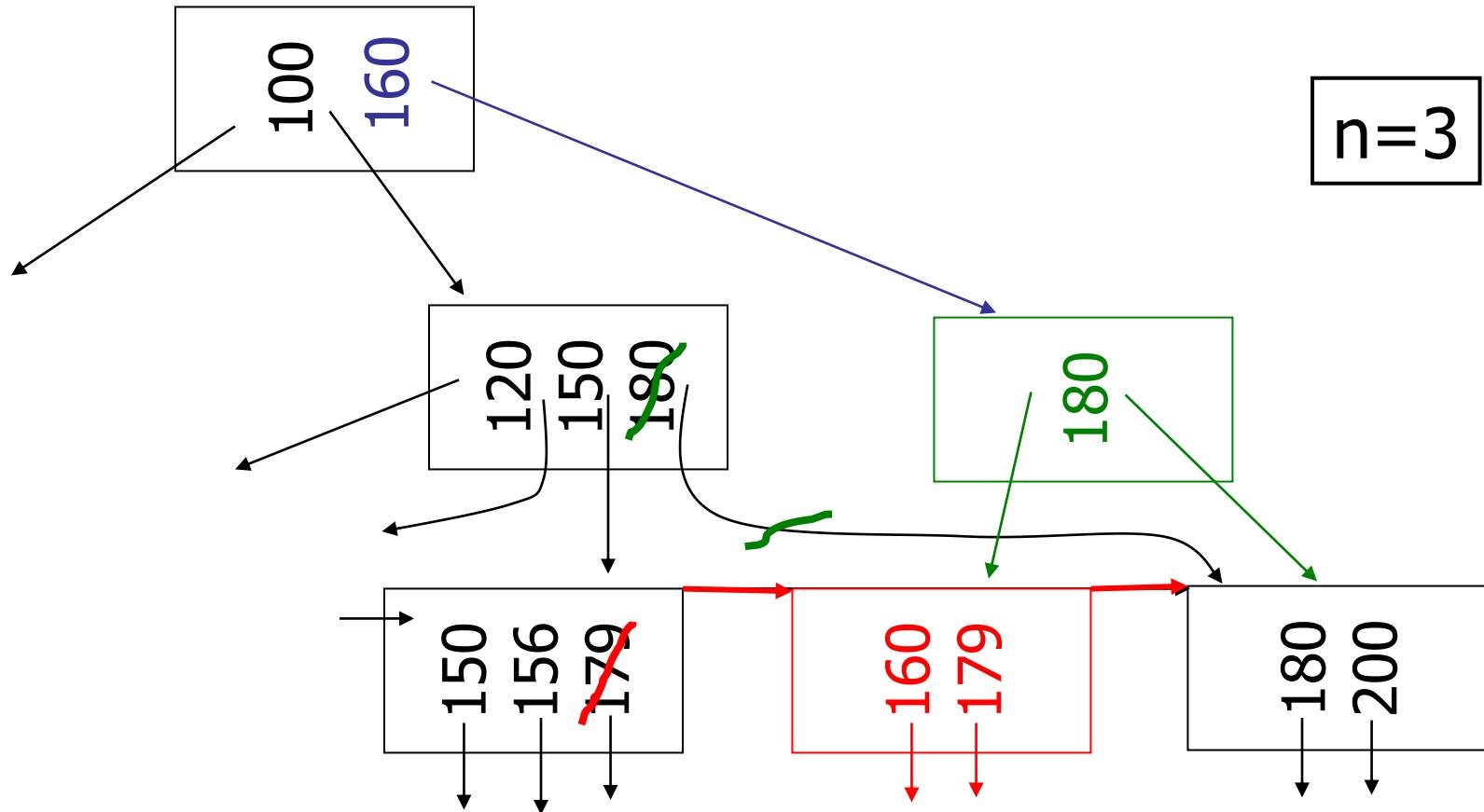
Indexelés

Szűrjuk be a 7-es indexértékű rekordot!



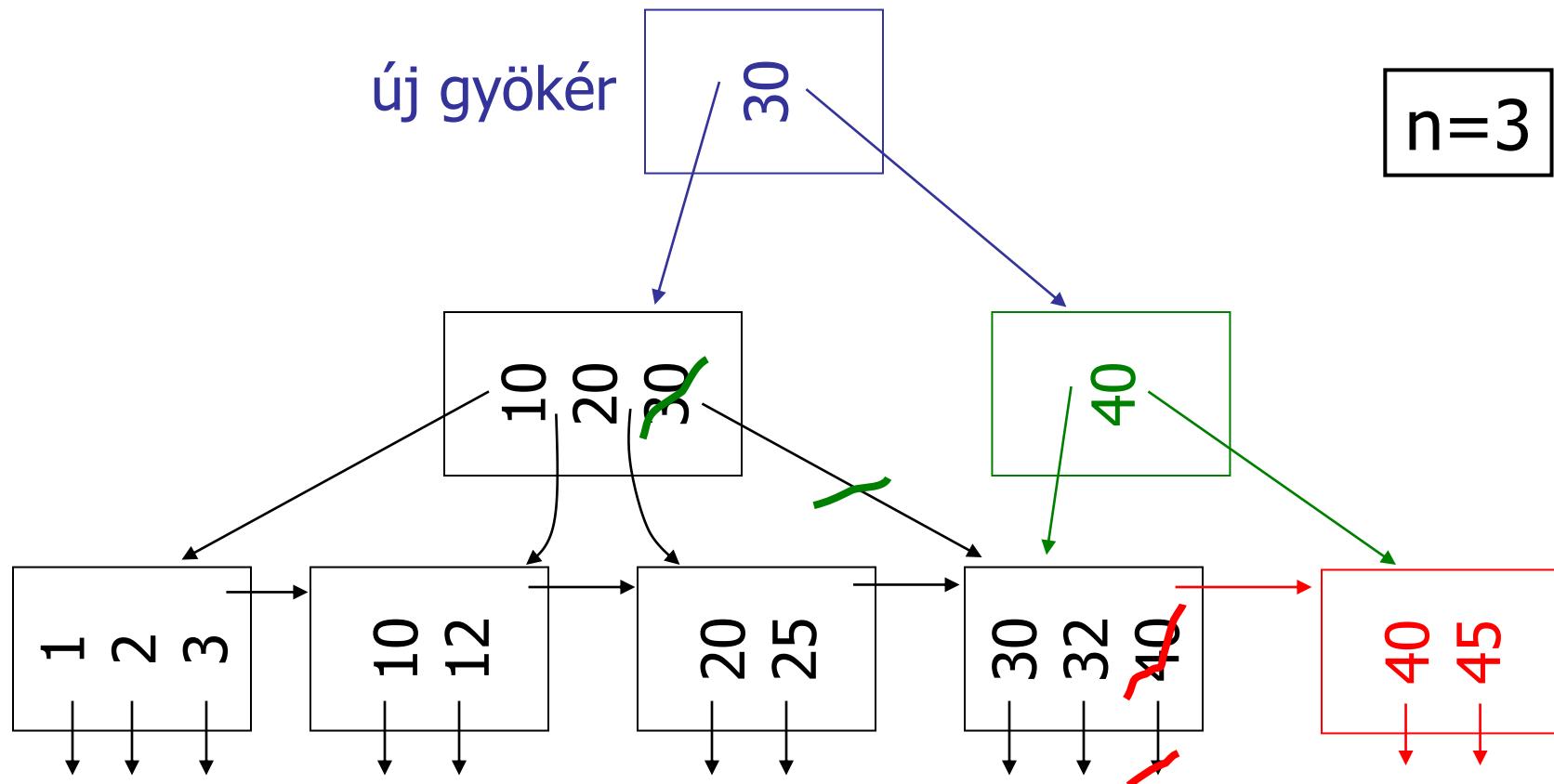
Indexelés

Szűrjuk be a 160-as indexértékű rekordot!



Indexelés

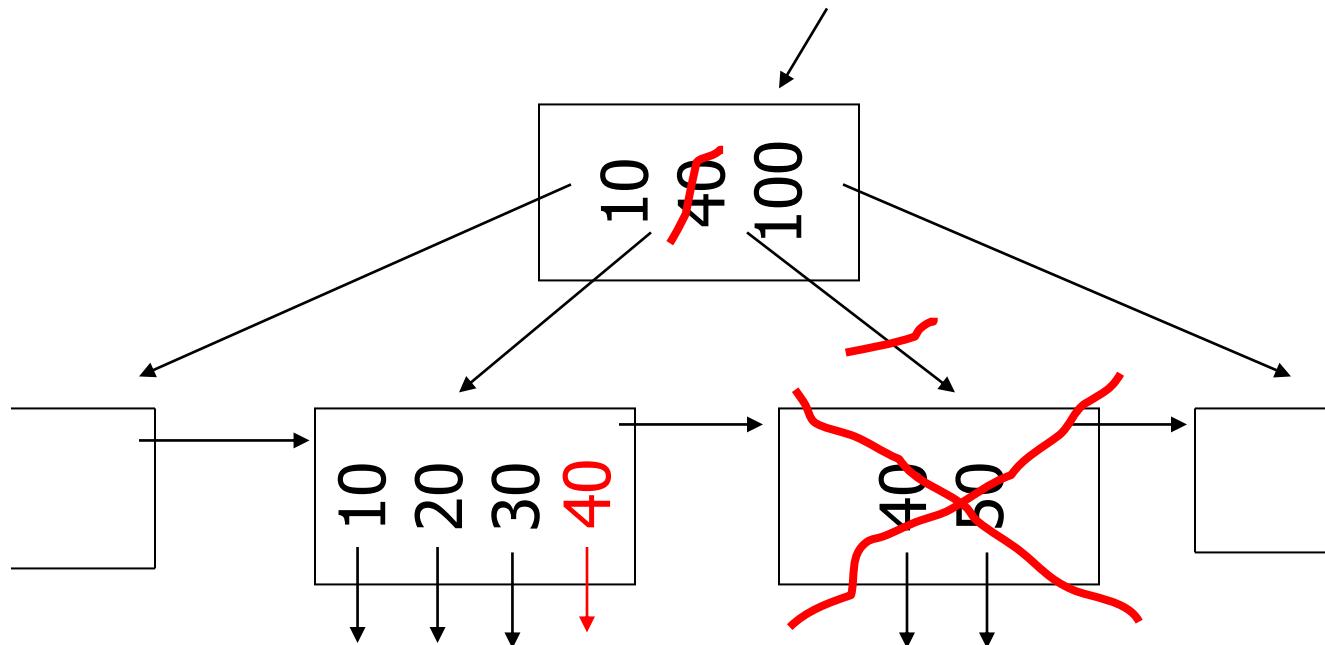
Szúrjuk be a 45-ös indexértékű rekordot!



Indexelés

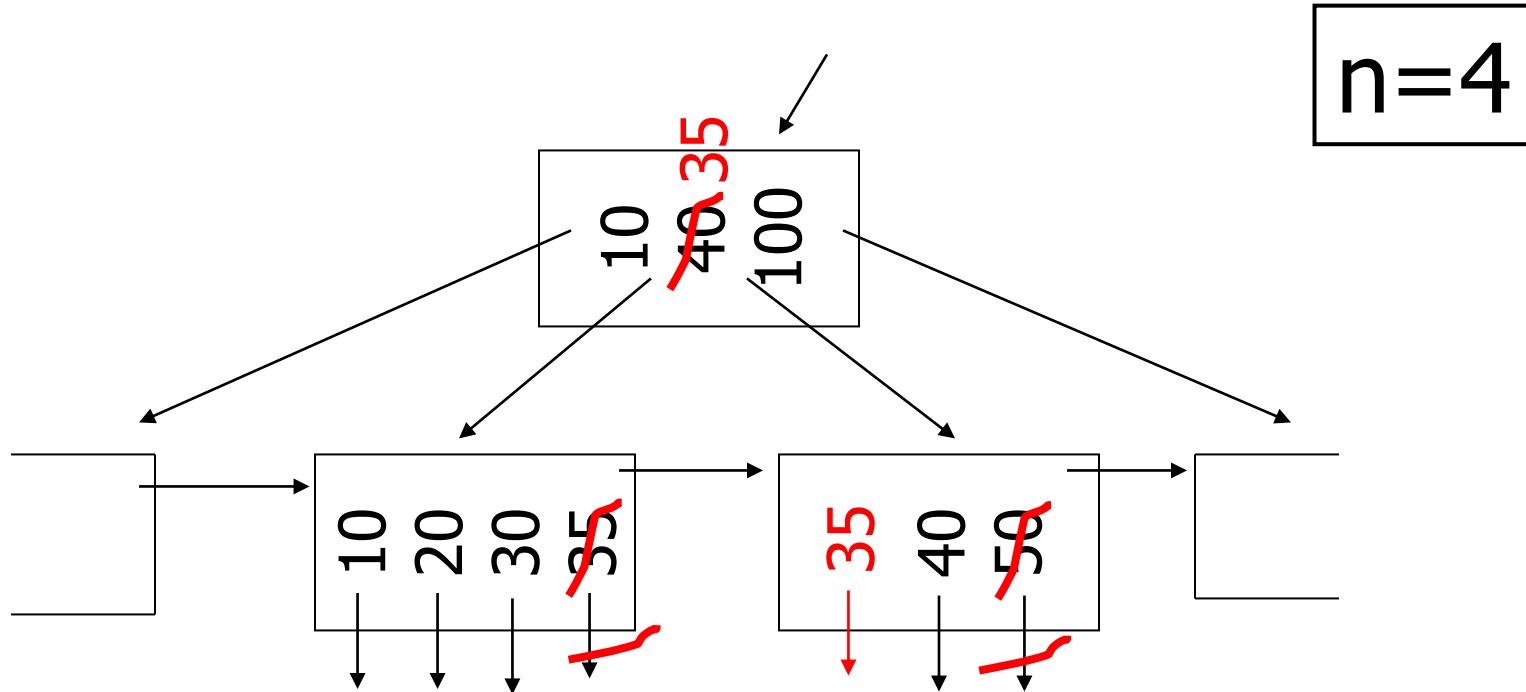
Töröljük az 50-es indexértékű rekordot!

n=4



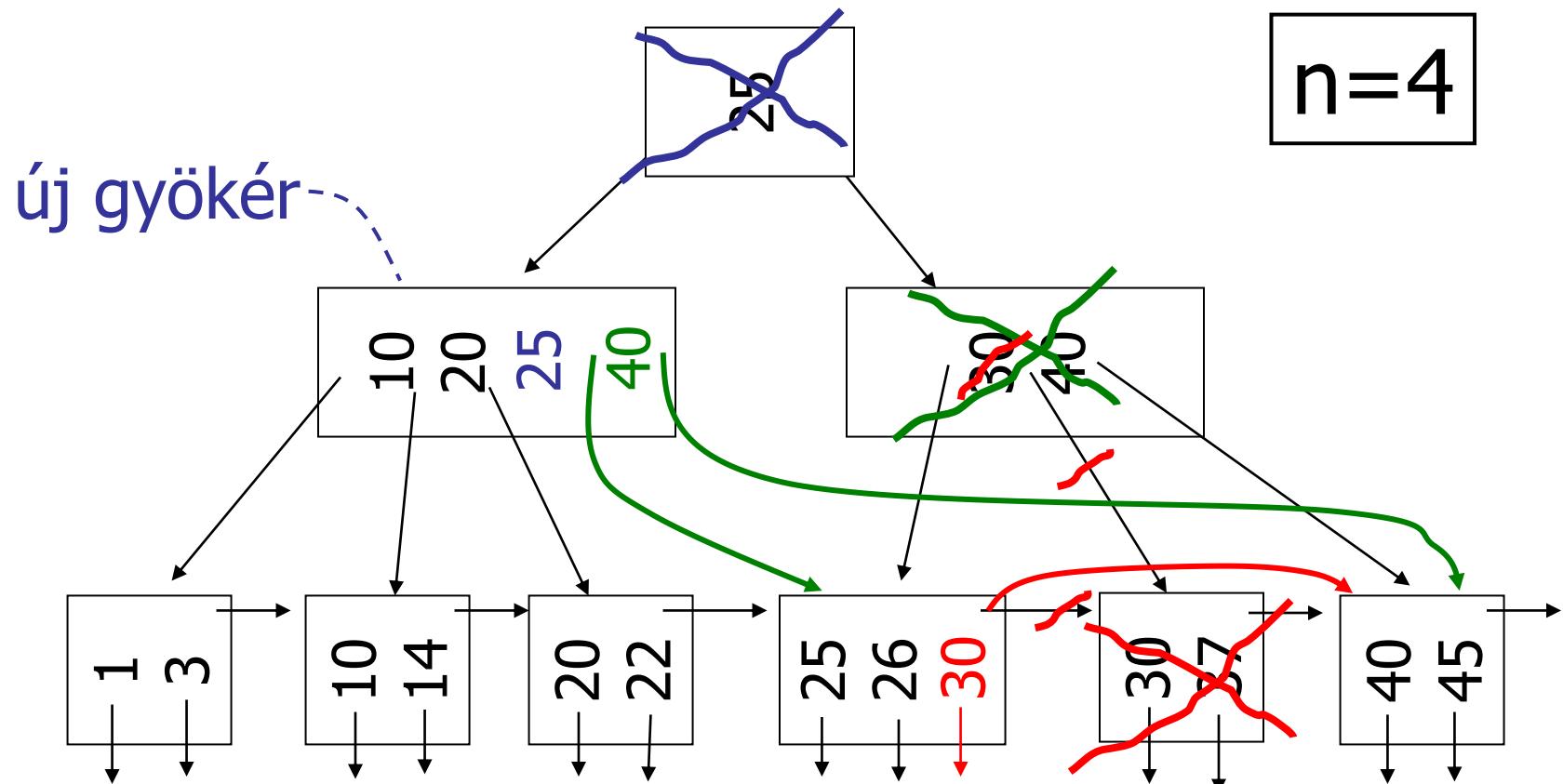
Indexelés

Töröljük az 50-es indexértékű rekordot!



Indexelés

Töröljük a 37-es indexértékű rekordot!





Összefoglalás:

Keresés rendezett állományban,
elsődleges és másodlagos indexekben,
többszintű indexekben, B⁺-fákban, B^{*}-fákban

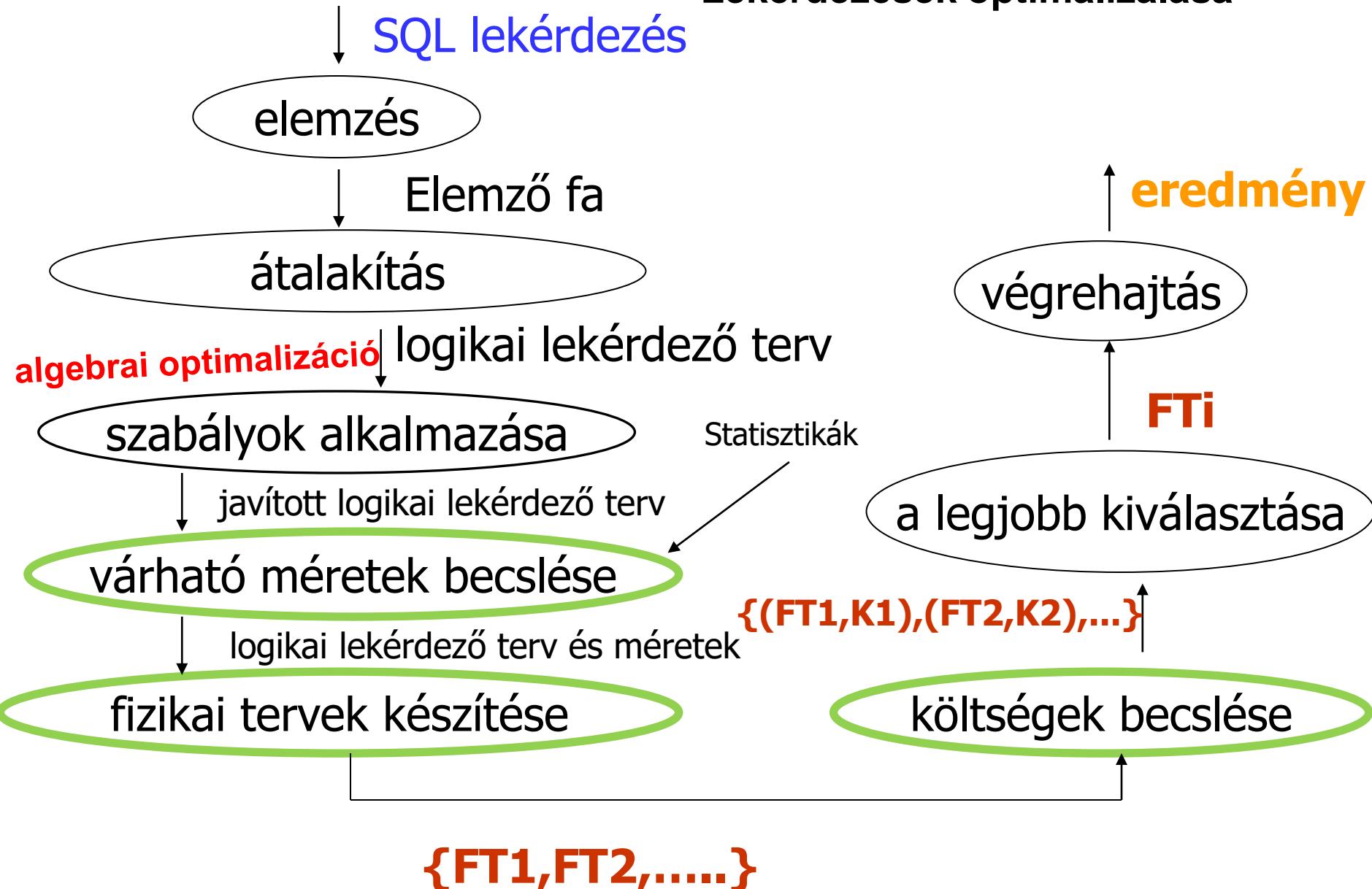
Köszönöm a figyelmet!

Fizikai tervezek (folytatás)

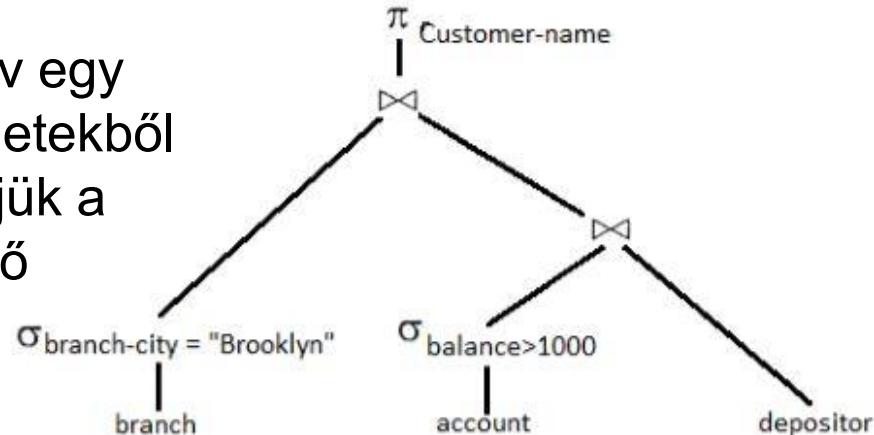
- Paraméterek, költségek
- Fizikai fájlszervezés,
 - szekvenciális (kupac), hasító indexek (statikus, dinamikus (kiterjeszthető, lineáris))
 - Rendezett állomány, elsődleges, másodlagos indexek, többszintű indexek, B^+ -fák, B^* -fák
- Műveletek megvalósítása, kiszámítási költség, outputmérő
- Optimális fizikai terv meghatározása



Lekérdezések optimalizálása



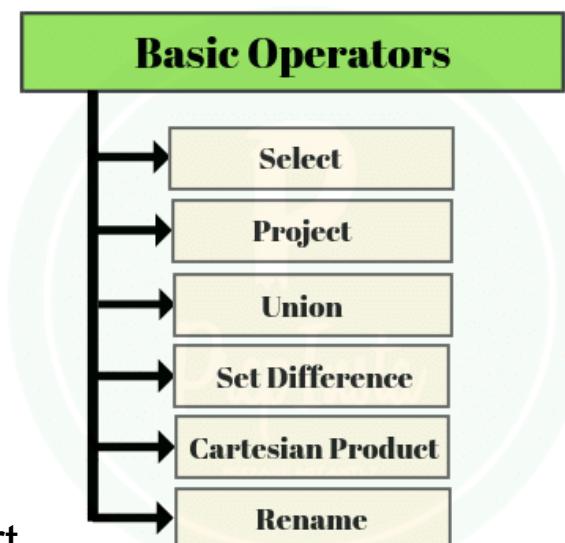
- Az optimális logikai lekérdező terv egy relációs algebrai kifejezés. Műveletekből áll. Ha minden műveletnek ismerjük a költségét, akkor a fának megfelelő költséget is ki tudjuk számolni.



- Hogyan végezzük el az egyes műveletek? Adott inputméret esetén mekkora a várható output méret.

- Műveletek:

- kiválasztás: σ (szekvenciális, index, rendezés)
- vetítés: π
- unió: \cup
- különbség: $-$
- szorzat: \times
- átnevezés: ρ (ezzel nem foglalkozunk, mert méretet nem változtat, költsége nincs)
- összekapcsolás: \bowtie (származtatott, de fontos művelet)



- $\text{SELECT * FROM student WHERE name=Paul}$
 - $\sigma_{\text{name}=\text{Paul}}(\text{student})$
- $\pi_{\text{name}}(\sigma_{\text{cid}<00112235}(\text{student}))$
- $\pi_{\text{name}}(\sigma_{\text{coursename}=\text{Advanced DBs}}((\text{student} \bowtie_{\text{cid}} \text{takes}) \bowtie_{\text{courseid}} \text{course}))$
- Sokféle lehetőségünk van egy lekérdezés kiértékelésére
 $\pi_{\text{name}}(\sigma_{\text{coursename}=\text{Advanced DBs}}((\text{student} \bowtie_{\text{cid}} \text{takes}) \bowtie_{\text{courseid}} \text{course}))$
 - $\pi_{\text{name}}((\text{student} \bowtie_{\text{cid}} \text{takes}) \bowtie_{\text{courseid}} \sigma_{\text{coursename}=\text{Advanced DBs}}(\text{course}))$
 - (relációs algebrai optimalizálás – várhatóan jobb költség - heurisztika)

Most viszont számokkal kifejezett költséget tudunk összehasonlítani.

| student | |
|------------|-------------|
| <u>cid</u> | <u>name</u> |
| 00112233 | Paul |
| 00112238 | Rob |
| 00112235 | Matt |

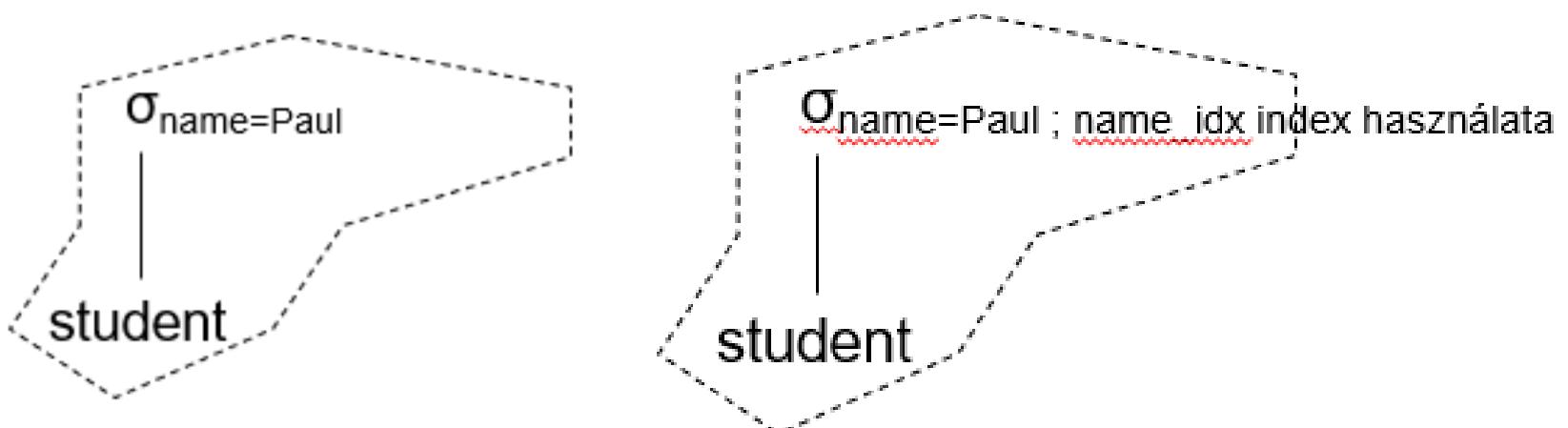
| takes | |
|------------|-----------------|
| <u>cid</u> | <u>courseid</u> |
| 00112233 | 312 |
| 00112233 | 395 |
| 00112235 | 312 |

| course | |
|-----------------|-------------------|
| <u>courseid</u> | <u>coursename</u> |
| 312 | Advanced DBs |
| 395 | Machine Learning |



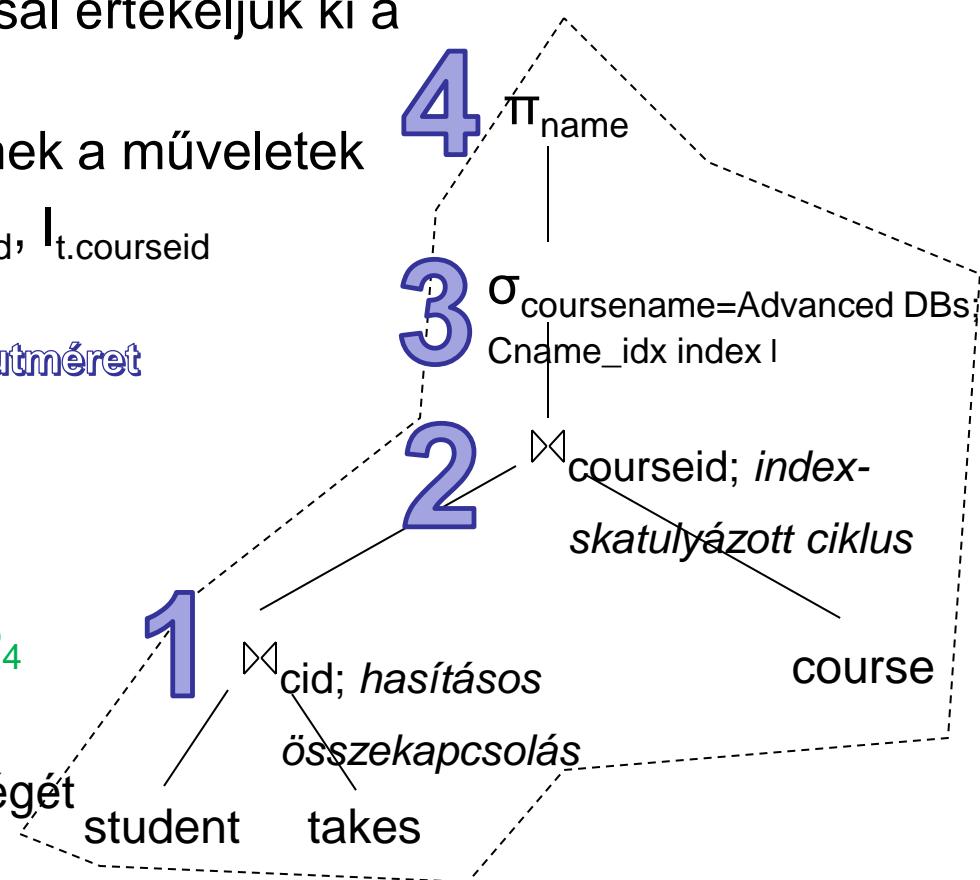
Műveletek kiértékelésének meghatározása

- Több lehetőség egy művelet elvégzésére
 - $\sigma_{\text{name}=\text{Paul}}(\text{student})$
 - fájlban szekvenciális keresés
 - másodlagos index a student.name mezőn
- Több elérési útvonal
 - elérési útvonal: mely módon érhetjük el a rekordokat (például a name_idx index alapján)



$$\Pi_{\text{name}}(\sigma_{\text{coursename}=\text{Advanced DBs}}((\text{student} \bowtie_{\text{cid}} \text{takes}) \bowtie_{\text{courseid}} \text{course}))$$

- Adjuk meg, melyik elérési útvonalat használjuk
- Adjuk meg, milyen algoritmussal értékeljük ki a műveleteket
- Adjuk meg, hogyan következnek a műveletek
- **INPUT:** $T_s, T_t, T_c, bf_s, bf_t, bf_c, I_{s.cid}, I_{t.courseid}$
- K_1, O_1
- K_2, O_2 Számítási költség, outputméret
- K_3, O_3
- K_4, O_4
- **OUTPUT:** $K = K_1 + K_2 + K_3 + K_4 + O_4$
- Optimalizáció:
 - becsüljük meg a tervezet költségét (nem minden)
 - válasszuk a legalacsonyabb becsült költségűt

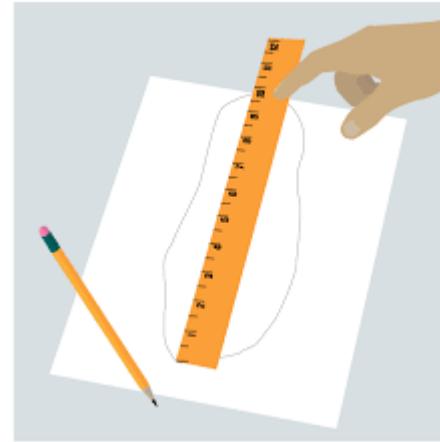


Költségbecslés

- Mit kell számításba venni:
 - **Lemez I/O**
 - szekvenciális
 - Indexelt elérés
 - CPU idő (elhanyagolható)
 - Hálózati kommunikáció (csak osztott adatbázisok esetén)
- Mit fogunk figyelembe venni:
 - Lemez I/O
 - Lapok (blokkok) olvasása, írása
 - Elhanyagoljuk a végeredmény kiírásának költségét (mivel amiket összehasonlítunk, azoknál mindenél ugyanaz a végeredmény mérete) az előző példában $K = K_1 + K_2 + K_3 + K_4 + O_4$ helyett $K = K_1 + K_2 + K_3 + K_4$ –t elég vizsgálnunk a fizikai tervezet összehasonlításánál.



Műveletek és költségek



SQL | 0.052 seconds

| OPERATION | OBJECT_NAME | CARDINALITY | COST |
|---------------------------------|-------------|-------------|------|
| SELECT STATEMENT | | 106 | 6 |
| MERGE JOIN | | 106 | 6 |
| TABLE ACCESS (BY INDEX ROWID) | DEPARTMENTS | 27 | 2 |
| INDEX (FULL SCAN) | DEPT_ID_PK | 27 | 1 |
| SORT (JOIN) | | 107 | 4 |
| Access Predicates | | | |
| E.DEPARTMENT_ID=D.DEPARTMENT_ID | | | |
| Filter Predicates | | | |
| E.DEPARTMENT_ID=D.DEPARTMENT_ID | EMPLOYEES | 107 | 3 |
| TABLE ACCESS (FULL) | | | |



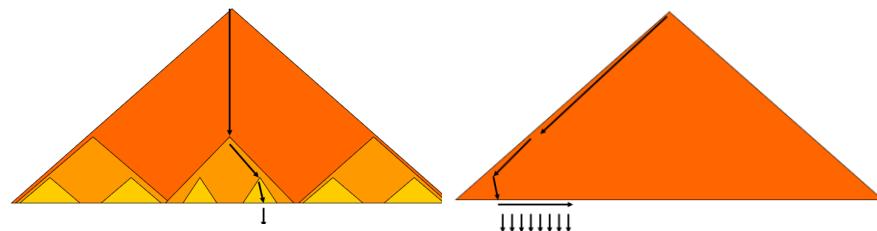
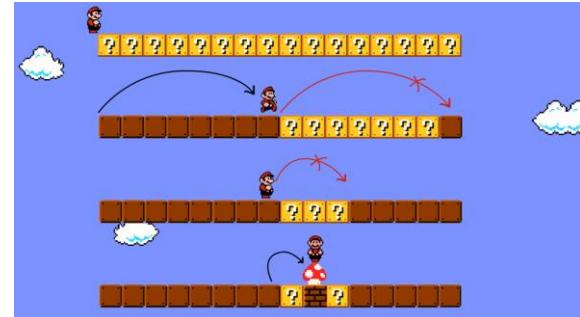
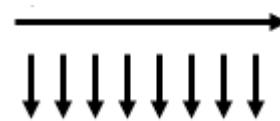
Műveletek és költségek

- Műveletek: σ , π , \cup , $-$, \times , \bowtie , \cap ,
- Költségek:
 - N_R : R rekordjainak száma (ezt korábban T_R -rel jelöltük)
 - L_R : R egy rekordjának mérete (ezt korábban I_R -rel jelöltük)
 - F_R : blokkolási tényező (ezt korábban bf_R -rel jelöltük)
 - egy lapon, blokkban levő rekordok száma
 - B_R : az R reláció tárolásához szükséges lapok, blokkok száma
 - $V(A,R)$: az A mező különböző értékeinek száma R-ben
(Képméret) (ezt korábban $I_{R,A}$ -val jelöltük)
 - $SC(A,R)$: az A mező kiválasztási számossága R-ben
 - Szelektivitás – hány darab $A=a$ értékű rekord van (egyenletességi feltétel esetén)
 - A kulcs: $SC(A,R)=1$
 - A nem kulcs: $SC(A,R)= N_R / V(A,R)$
 - HT_i : az i index szintjeinek száma
 - a törteket és logaritmusokat felfelé kerekítjük



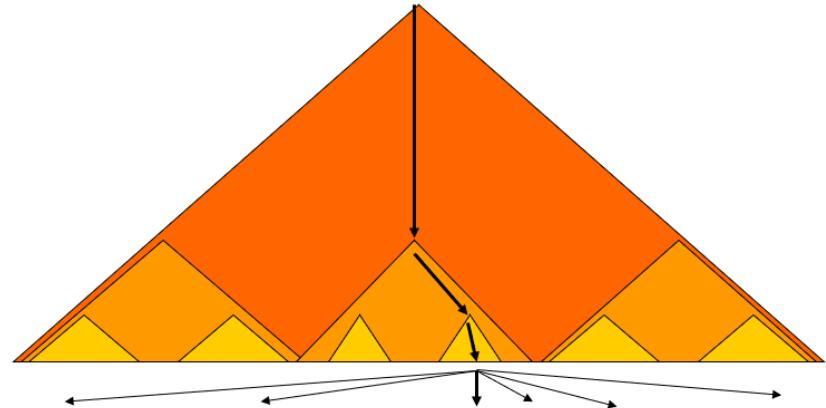
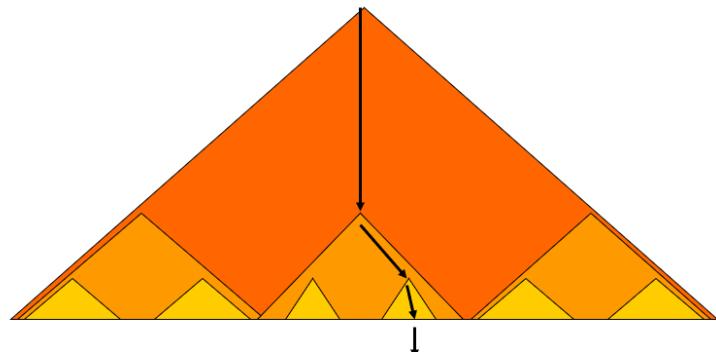
Kiválasztás σ

- Lineáris keresés
 - olvassunk be minden lapot és keressük az egyezéseket (egyenlőség vizsgálata esetén)
 - átlagos költség:
 - nem kulcs esetén B_R , kulcs esetén $0.5 * B_R$
- Logaritmikus keresés
 - rendezett mező esetén $\lceil \log_2 B_R \rceil + m$
 - átlagos költség:
 - m további lapot kell beolvasni
 - $m = \lceil \text{SC}(A, R) / F_R \rceil - 1$
- Elsődleges/cluster index
 - átlagos költség:
 - egyetlen rekord $HT_i + 1$
 - több rekord $HT_i + \lceil \text{SC}(A, R) / F_R \rceil$



Kiválasztás σ

- Másodlagos index
 - átlagos költség:
 - kulcs mező $HT_i + 1$
 - nem kulcs mező
 - legrosszabb eset $HT_i + SC(A, R)$
 - a lineáris keresés kedvezőbb, ha sok a megfelelő rekord



Összetett kiválasztás σ_{kif}

- konjunkciós kiválasztás: $\sigma_{\theta_1 \wedge \theta_2 \dots \wedge \theta_n}$
 - végezzünk egyszerű kiválasztást a legkisebb költségű θ_i -re
 - pl. a θ_i -hez tartozó index felhasználásával
 - a fennmaradó θ feltételek szerint szűrjük az eredményt
 - $\sigma_{cid>00112233 \wedge courseid=312}(\text{takes})$
 - költség: az egyszerű kiválasztás költsége a kiválasztott θ -ra
 - több index
 - válasszuk ki a θ_i -hez tartozó indexeket
 - keressünk az indexekben és adjuk vissza a RID-ket
 - válasz: RID-k metszete
 - költség: a költségek összege + rekordok beolvasása
- diszjunkciós kiválasztás: $\sigma_{\theta_1 \vee \theta_2 \dots \vee \theta_n}$
 - több index
 - RID-k uniója
 - költség: a költségek összege + rekordok beolvasása
 - lineáris keresés



Méretbecslés - kiválasztás

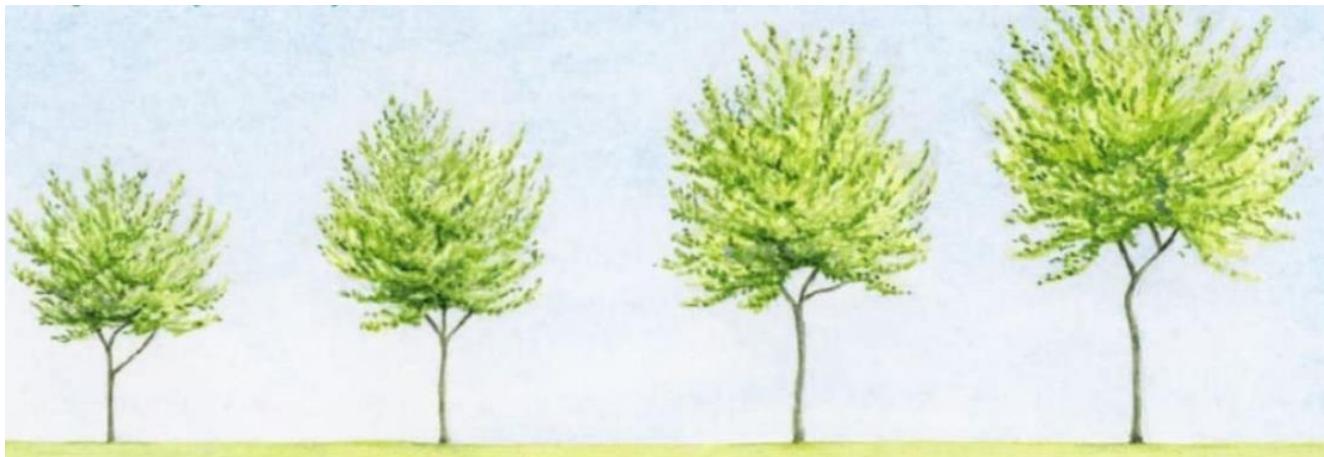
- $\sigma_{A=v}(R)$
 - Sorok száma: SC(A,R)
Blokkok száma: SC(A,R)/F_R
- $\sigma_{A \leq v}(R)$
 - Sorok száma: $N_R * \frac{v - \min(A, R)}{\max(A, R) - \min(A, R)}$
Blokkok száma: Sorok száma/ F_R
- $\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(R)$
 - szorzódó valószínűségek (s_i az i-ik feltétel szelektivitása:hány rekord elégíti ki)
– Sorok száma: $N_R * [(s_1/N_R) * (s_2/N_R) * \dots * (s_n/N_R)]$ Blokkok száma: Sorok száma/ F_R
- $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(R)$
 - annak valószínűsége, hogy egy rekordra egy θ se igaz:
$$[(1 - s_1/N_R) * (1 - s_2/N_R) * \dots * (1 - s_n/N_R)]$$
 s_i az i-ik feltétel szelektivitása
 - Sorok száma: $N_R * (1 - [(1 - s_1/N_R) * (1 - s_2/N_R) * \dots * (1 - s_n/N_R)])$
Blokkok száma: Sorok száma/ F_R

A keresési feltételekre, azaz a bennük szereplő oszlopokra függetlenségi feltételt tettünk fel. Ezzel felső korlátokat kaptunk. Ha nem teljesülne a függetlenség, akkor több egybeesés lehetne, kisebb lenne a mérezez. Ezenkívül egyenletességi feltételt is tettünk.



Vetítés és halmazműveletek visszavezetése rendezésre

- SELECT DISTINCT cid FROM takes
 - π-hez szükséges a duplikált értékek kiszűrése
 - Rendezéssel tudjuk a duplikátumokat kiszűrni
- Halmazműveletek
 - $R \cap S$ ki kell szűrni a duplikált értékeket
 - $R \cup S$ ki kell szűrni a duplikált értékeket
 - $R - S$ (ha minden tábla rendezett, akkor elég egyszerre végigolvasni minden táblát.)
 - Mindegyikhez kell a rendezés



Rendezés

- sok művelet hatékony kiértékelése
- a lekérdezés is igényelheti:
 - $\text{SELECT cid, name FROM student ORDER BY name}$
- megvalósítás
 - belső rendezés (ha a rekordok beférnek a memóriába)
betölthetjük: B_R művelet,
memóriában rendezzük: 0 művelet
kiírjuk: B_R művelet
összesen $2*B_R$
 - **külső rendezés** (ha nem fér be a teljes tábla memóriába):
SORT-MERGE



FreakingNews.com



Külső összefésüléses rendezés (1/3)

- Rendező lépés: rendezett futamok létrehozása

M // a memória mérete blokkokban

i=0;

ismétlés

M lap beolvasása az R relációból a memóriába

az M lap rendezése

kiírás az R_i fájlba (futamba)

i növelése

amíg el nem fogynak a lapok

$N = i$ // futamok száma



Külső összefésüléses rendezés (2/3)

- **Összevonási lépés:** rendezett futamok összefésülése

//feltéve, hogy $N < M$

minden R_i fájlhoz egy lap lefoglalása
 // N lap lefoglalása

minden R_i -ből egy-egy P_i lap
 beolvasása

ismétlés

az N lap közül a (rendezés szerint) első rekord kiválasztása,
 legyen ez a P_j lapon

a rekord kiírása a kimenetre és
 törlése a P_j lapról

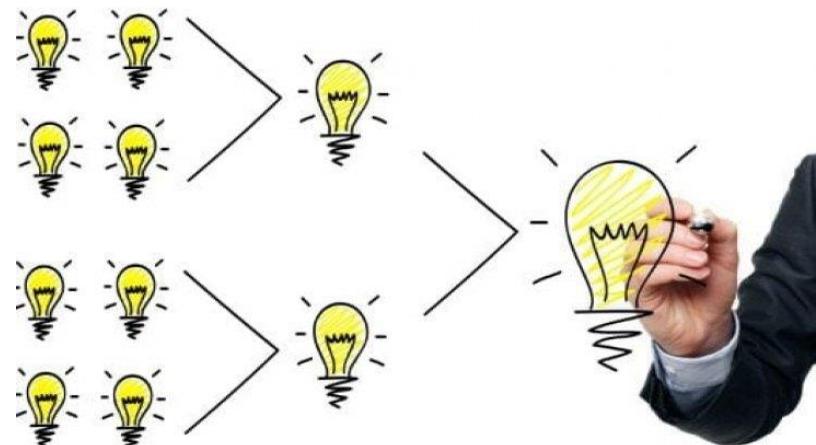
ha üres a lap, a következő P_j'
 beolvasása R_j -ből

amíg minden lap ki nem ürül



Külső összefésüléses rendezés (3/3)

- Összevonási lépés: rendezett futamok összefésülése
- Mi van, ha $N > M$?
 - több *menet*
 - minden *menet* $M-1$ futamot von össze, amíg nincs feldolgozva a reláció
 - a következő menetben a futamok száma kisebb és az összevonást ismételjük rekurzívan
 - a végső *menetben* keletkezik a végső kimenet



Összefésüléses rendezés - példa

| | |
|---|----|
| d | 95 |
| a | 12 |
| x | 44 |
| s | 95 |
| f | 12 |
| o | 73 |
| t | 45 |
| n | 67 |
| e | 87 |
| z | 11 |
| v | 22 |
| b | 38 |

memória

| | | |
|----------------|---|----|
| R ₁ | a | 12 |
| | d | 95 |
| | x | 44 |

| | | |
|----------------|---|----|
| R ₂ | f | 12 |
| | o | 73 |
| | s | 95 |

| | | |
|----------------|---|----|
| R ₃ | e | 87 |
| | n | 67 |
| | t | 45 |

| | | |
|----------------|---|----|
| R ₄ | b | 38 |
| | v | 22 |
| | z | 11 |

1. menet

| | |
|---|----|
| d | 95 |
| f | 12 |
| d | 95 |

futam

| | |
|---|----|
| a | 12 |
| d | 95 |
| f | 12 |
| o | 73 |
| s | 95 |

2.menet

| | |
|---|----|
| b | 38 |
| e | 87 |
| n | 67 |
| t | 45 |
| v | 22 |

| | |
|---|----|
| a | 12 |
| b | 38 |
| d | 95 |
| e | 87 |
| f | 12 |
| n | 67 |
| o | 73 |
| s | 95 |
| t | 45 |
| v | 22 |
| x | 44 |
| z | 11 |



Összefésüléses rendezés költsége

- B_R : R lapjainak száma
- **Rendezési lépés (rendezett futamok előállítása)**: $2 * B_R$
 - reláció olvasása/írása
- **Összevonási lépés (rendezett futamból M-1 darab összefésülése)**:
 - Hány futamot hajtunk végre?
 - kezdetben $\left\lfloor \frac{B_R}{M} \right\rfloor$ összevonandó futam
 - minden *menet* M-1 futamot rendez, vagyis a futamok számát mindig M-1-gyel kell osztani
 - tehát az összes menet száma: $\left\lceil \log_{M-1} \left(\frac{B_R}{M} \right) \right\rceil$
 - minden menetben $2 * B_R$ lapot olvasunk
 - reláció olvasása/írása
 - kivéve az utolsó kiírást
- **Teljes számolási költség** (az utolsó lépés, vagyis a rendezett tábla kiírása nélkül)
 - $2 * B_R + 2 * B_R * \left\lceil \log_{M-1} \left(\frac{B_R}{M} \right) \right\rceil - B_R$



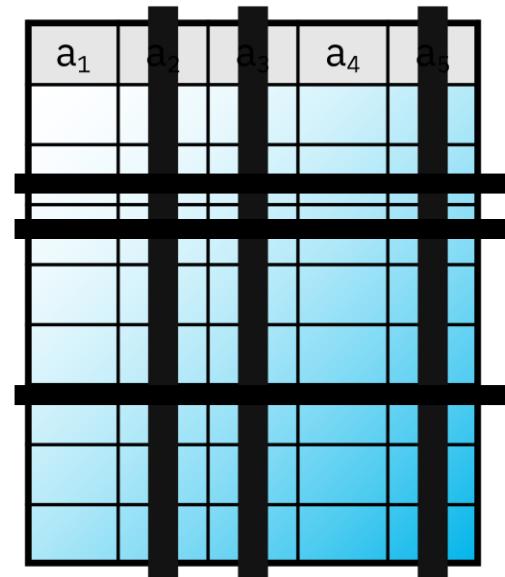
Vetítés

- $\Pi_{A1, A2\dots}(R)$
- felesleges mezők törlése
 - Végigolvassuk a táblát és a felesleges mezőket elhagyjuk, így kapjuk az R₁ táblát
 - Műveletigény: $B_R + B_{R1}$
- duplikált rekordok törlése
 - az R1 rendezése az összes mező alapján
 - Műveletigény kiírással együtt: (például külső összefésülés esetén):
 - $2 * B_{R1} + 2 * B_{R1} * \left\lceil \log_{M-1} \left(\frac{B_{R1}}{M} \right) \right\rceil$
 - a rendezett eredményt egyszer végigolvassuk, duplikált (szomszédos) rekordokat töröljük, így kapjuk az R2 táblát, azaz végeredményt
 - Műveletigény: $B_{R1} + B_{R2}$
- Teljes számítási költség
 - $B_R + B_{R1} + 2 * B_{R1} + 2 * B_{R1} * \left\lceil \log_{M-1} \left(\frac{B_{R1}}{M} \right) \right\rceil + B_{R1} + B_{R2}$
 - Felső becslés: $6 * B_R + 2 * B_R * \left\lceil \log_{M-1} \left(\frac{B_R}{M} \right) \right\rceil$



Méretbecslés - Vetítés

- $S := \pi_{A_1, A_2 \dots A_k}(R)$
- Felső becslés: B_R
- Ha csak egy oszlopra vetítünk, akkor $V(A, R)$ sor van a vetületben
 - Sorok száma: $V(A, R)$ Blokkok száma: $V(A, R)/F_S$
- Több oszlop esetén különböző értékekből kapható különböző sorok maximális száma:
 - $V(A_1, R) * V(A_2, R) * \dots * V(A_k, R)$
 - A vetületben nem lehet több sor mint a táblában:
$$\min(V(A_1, R) * V(A_2, R) * \dots * V(A_k, R), N_R)$$
 - **Sorok száma:**
$$\min(V(A_1, R) * V(A_2, R) * \dots * V(A_k, R), N_R) / F_S$$
 - **Blokkok száma:**
$$\min(V(A_1, R) * V(A_2, R) * \dots * V(A_k, R), N_R) / F_S$$



Unió

- $P := R \cup S$
- P esetén felső becslés: **sorok száma**: $N_R + N_S$ **blokkok száma**: $B_R + B_S$
- duplikált rekordok törlése
 - az P rendezése az összes mező alapján
 - **Műveletigény szerint kiírással együtt**: (például külső összefésülés esetén):
 - $2 * B_P + 2 * B_P * \lceil \log_{M-1}(B_P/M) \rceil$
 - a rendezett eredményt egyszer megolvassuk, duplikált (szomszédos) rekordokat töröljük, így kapjuk az P_1 tablát, azaz végeredményt
 - **Műveletigény**: $B_P + B_{P1}$
- **Teljes számítási költség**
 - $2 * B_P + 2 * B_P * \lceil \log_{M-1}(B_P/M) \rceil + B_P + B_{P1}$

Felső becslés:

$$4 * (B_R + B_S) + 2 * (B_R + B_S) * \lceil \log_{M-1}((B_R + B_S)/M) \rceil$$

Különbség, Metszet, Szorzat visszavezetése az összekapcsolásra

- $P := R \times S$ ($= R \bowtie S$ speciális esete, ahol az **R** és **S** sémáknak nincs közös attribútuma)

sorok száma: $N_R \times N_S$

blokkok száma: $B_R * N_S + B_S * N_R$

- $P := R \cap S$ ($= R \bowtie S$ speciális esete, ahol az **R** és **S** sémák teljesen megegyeznek)

sorok száma: $\min(N_R, N_S)$

blokkok száma: $\min(B_R, B_S)$

- $P := R - S$

sorok száma: N_R

blokkok száma: B_R

A kiszámítást visszavezetjük a rendezéses-összefésüléses összekapcsolás kiszámítására, mivel az összes ($t_{R,i}, t_{S,j}$) sorpárnak egyszer be kell kerülni a memóriába, ahogy az összekapcsolások esetén is. (2 blokkot nyitunk a rendezett R és rendezett S számára. Ha R-beli sorhoz találunk S-beli sort, akkor kihagyjuk. Ha feldolgoztunk egy blokkoárt, akkor annak a helyére töltünk be új blokkot, amelyiknek kisebb volt a legnagyobb értéke, mint a másik blok legnagyobb értéke.

Összekapcsolás

- $\Pi_{\text{name}}(\sigma_{\text{coursename}=\text{Advanced DBs}}((\text{student} \bowtie_{\text{cid}} \text{takes}) \bowtie_{\text{courseid}} \text{course}))$
- megvalósítások
 - skatulyázott ciklusos összekapcsolás ([nested loop join](#))
 - blokk-skatulyázott ciklusos összekapcsolás ([block-nested loop join](#))
 - indexelt skatulyázott ciklusos összekapcsolás ([index nested loop join](#))
 - összefésüléses rendező összekapcsolás ([sort-merge join](#))
 - hasításos összekapcsolás ([hash join](#))



Skatulyázott ciklusos összekapcsolás(1/2)

- $R \bowtie S$

R minden t_R rekordján

S minden t_S rekordján

ha (t_R t_S illeszkedik) $t_R \bowtie t_S$ kiírása

vége

vége

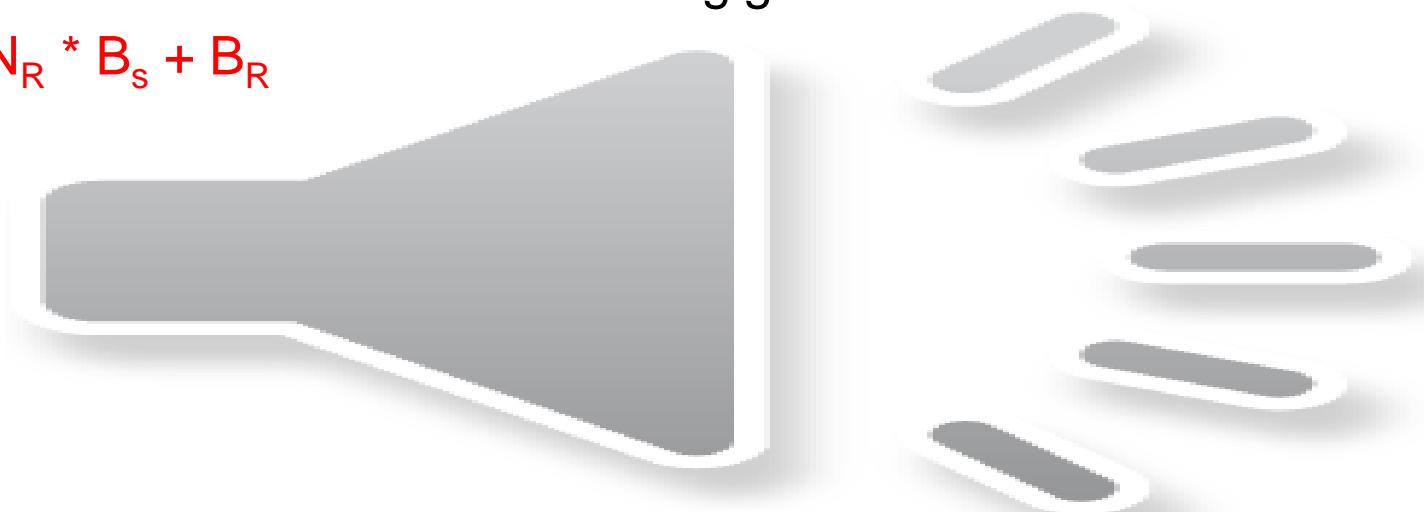
- Bármilyen összekapcsolási feltétnél működik
 - Ha nincs illesztési feltétel, akkor a direkt szorzatot adja vissza
- S belső reláció
- R külső reláció

ási feltétnél működik

– Ha nincs illesztési feltétel, akkor a direkt szorzatot adja vissza

Skatulyázott ciklusos összekapcsolás(2/2)

- Költség:
 - legjobb eset, ha a kisebb reláció elfér a memóriában
 - ezt használjuk belső relációnak
 - $B_R + B_S$
 - legrosszabb eset, ha minden R-beli rekordnál végig kell olvasni
 - $N_R * B_s + B_R$



Blokk-skatulyázott ciklusos összekapcsolás (1/2)

R minden X_R lapján

S minden X_S lapján

X_R minden t_R rekordján

X_S minden t_S rekordján

ha (t_R t_S illeszkedik) $t_R \bowtie t_S$ kiírása

vége

vége

vége

vége



Blokk-skatulyázott ciklusos összekapcsolás (2/2)

- Költség:
 - legjobb eset, ha a kisebb reláció elfér a memóriában
 - ezt használjuk belső relációnak
 - $B_R + B_S$
 - legrosszabb eset, ha minden R-beli lapnál végig kell olvasni
 - S-t minden R-beli lapnál végig kell olvasni
 - $B_{R^-} * B_s + B_R$ az előző esetben nagyobb volt: $N_R * B_s + B_R$

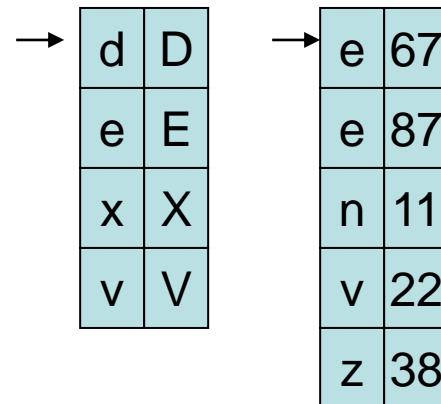


Indexelt skatulyázott ciklusos összekapcsolás

- $R \bowtie S$
- Index a belső relációt (S) – lehetőleg klaszterindex
- a külső reláció (R) minden rekordjánál *keresünk* a belső reláció indexe alapján illeszkedő sorokat az S -ből.
- Költség:
 - $B_R + N_R * c$
 - c a belső relációból index szerinti kiválasztás költsége
 - $c = \text{indexelt keresési költség} + \text{találat mérete blokkokban}$
 - Klaszterindex esetén $c = HTi + \lceil SC(A,R)/F_S \rceil =$
 $= \log(\text{index méret}) + \lceil (N_S / V(A,S)) / F_S \rceil =$
 $= \log(\text{index méret}) + \lceil (B_S / V(A,S)) \rceil \approx \lceil (B_S / V(A,S)) \rceil$ mert a logaritmusos tag sokkal kisebb
 - $B_R + N_R * B_S / V(A,S)$ klaszterindex és egyenletességi feltétel esetén
 - a kevesebb rekordot tartalmazó reláció legyen a külső

Összefésüléses rendező összekapcsolás

- $R \bowtie S$
- A relációk rendezettek az összekapcsolási mezők szerint
- Összefésüljük a rendezett relációkat
 - mutatók az első rekordra minden két relációban
 - beolvassunk S -ből egy rekordcsoportot, ahol az összekapcsolási attribútum értéke megegyezik
 - beolvassunk rekordokat R -ből és feldolgozzuk
- A rendezett relációkat csak egyszer kell végigolvasni
- Költség:
 - **rendezés költsége + $B_S + B_R$**

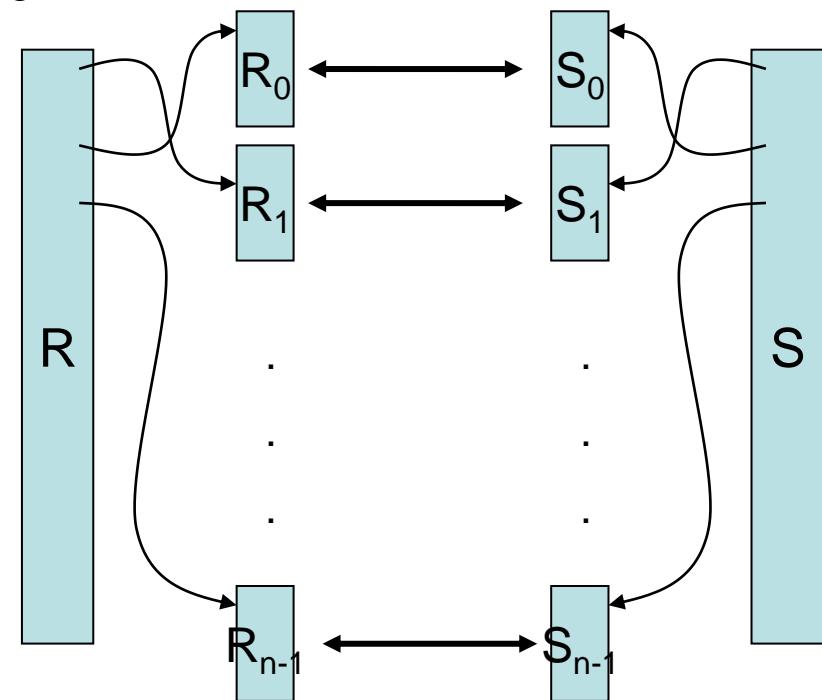


| | |
|---|----|
| d | D |
| e | E |
| x | X |
| v | V |
| z | 38 |



Hasításos összekapcsolás

- $R \bowtie S$
- R és S -ben alkalmazzuk ugyanazt a *h hasító függvényt* az összekapcsolási mezőre és felosztjuk a rekordokat a memoriában elférő részekre (R_i és S_i férjen be a memoriába egyszerre)
 - R rekordjainak felosztása $R_0 \dots R_{n-1}$
 - S rekordjainak felosztása $S_0 \dots S_{n-1}$
- az egymáshoz illő, ugyanolyan indexű partíciók (kosarak) rekordjait összekapcsoljuk (mert ha két érték megegyezik, akkor a hasítófüggvény értékük is megegyezik, tehát csak azonos indexű partícióban lehetnek)
 - hasítófüggvény alapján indexelt blokk-skatulyázott ciklusos összekapcsolással
- Költség: $2*(B_R+B_S) + (B_R+B_S)$



Méretbecslés - összekapcsolás

- $R \bowtie S$
 - $R \cap S = \emptyset$ esetén $R \bowtie S = R \times S$
sorok száma: $N_R * N_S$ **blokkok száma:** $B_R * N_S + B_S * N_R$
 - $R \cap S = \{A\}$, sem R-nek, sem S-nek nem kulcsa
 - $N_R * N_S / V(A, S)$ mert minden R-beli sorhoz $N_S / V(A, S)$ különböző érték illeszkedhet
 - $N_S * N_R / V(A, R)$ mert minden S-beli sorhoz $N_R / V(A, R)$ különböző érték illeszkedhet
- **sorok száma:** $N_S * N_R / \text{Max}(V(A, R), V(A, S))$
blokkok száma: $(B_R * N_S + B_S * N_R) / \text{Max}(V(A, R), V(A, S))$
Speciálisan például, ha $R.A \subseteq S.A$
 - **sorok száma:** $N_S * N_R / V(A, S)$
 - **blokkok száma:** $(B_R * N_S + B_S * N_R) / V(A, S)$
- $R \cap S$ kulcs R-en esetén (S-nek idegen kulcsa)
sorok száma: N_S **blokkok száma:** $(B_R * N_S + B_S * N_R) / N_R$

Összefoglalás

- minden egyes művelet költségét és az eredmény méretét megadtuk
- A lekérdezési terv költségének becslését a költségek összegeként értelmeztük.
- Tovább lehetne javítani a költségeken materializálással, csövezetékesítés (pipeline), kommutatív, asszociatív egymás utáni műveletek optimális sorrendben történő elvégzésével, párhuzamos kiértékeléssel.



Köszönöm a figyelmet!



Több tábla összekapcsolása

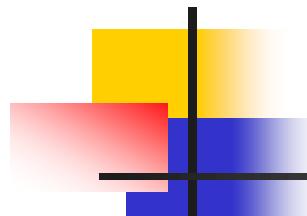
Lekérdezések optimalizálása



Algebrai optimalizáció

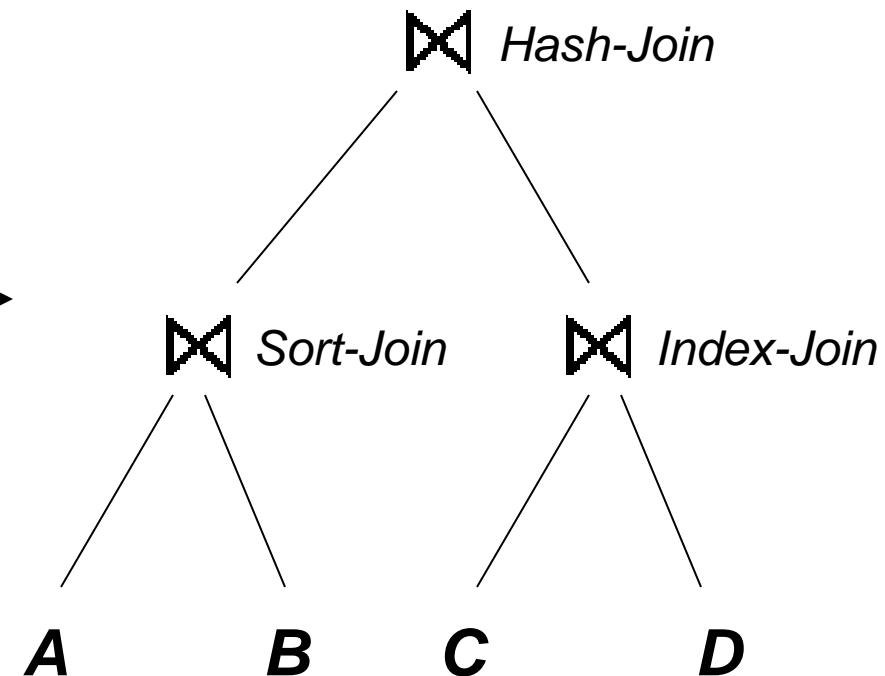
- könyv(sorszám,író,könyvcím)
 - $\text{kv}(\text{s},\text{i},\text{kc})$
- kölcsönző(azonosító,név,lakcím)
 - $\text{kő}(\text{a},\text{n},\text{lc})$
- kölcsönzés(sorszám,azonosító,dátum)
 - $\text{ks}(\text{s},\text{a},\text{d})$
- Milyen című könyveket kölcsönöztek ki 2007-től kezdve?
- $\Pi_{\text{kc}}(\sigma_{\text{d} \geq '2007.01.01'}(\text{kv} \times | \text{kő} | \times | \text{ks}))$
- Az összekapcsolásokat valamelyen sorrendben kifejezzük az alapműveletekkel:

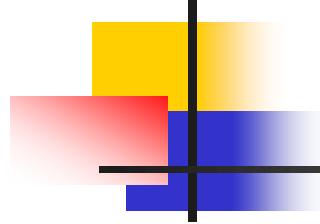
$\Pi_{\text{kc}}(\sigma_{\text{d} \geq '2007.01.01'}(\Pi_{\text{kv},\text{i},\text{kc},\text{kő},\text{a},\text{n},\text{lc},\text{d}}(\sigma_{\text{kv}.\text{s} = \text{ks}.\text{s} \wedge \text{kő}.\text{a} = \text{ks}.\text{a}}(\text{kv} \times (\text{kő} \times \text{ks}))))))$



What is the best way to join n relations?

SELECT ...
FROM A, B, C, D
WHERE A.x = B.y →
AND C.z = D.z



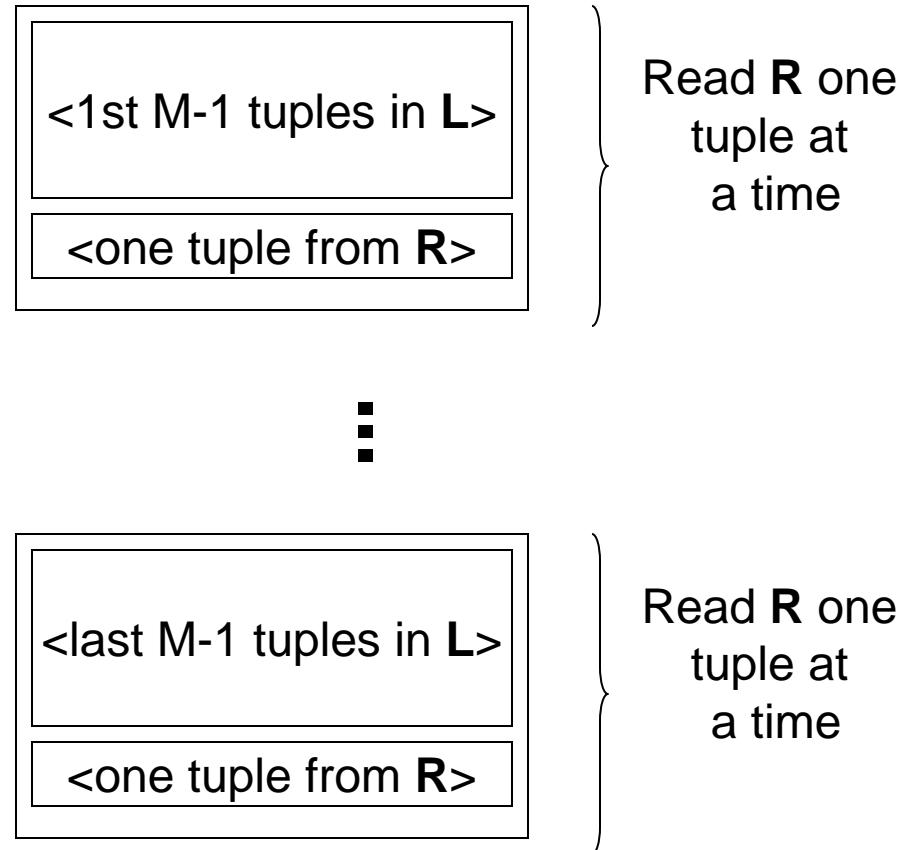


Issues to consider for 2-way Joins

- Join attributes sorted or indexed?
- Can either relation fit into memory?
 - Yes, evaluate in a single pass
 - No, require $\log_M B$ passes. M is #of buffer pages and B is # of pages occupied by smaller of the two relations
- Algorithms (nested loop, sort, hash, index)

Nested Loop Join

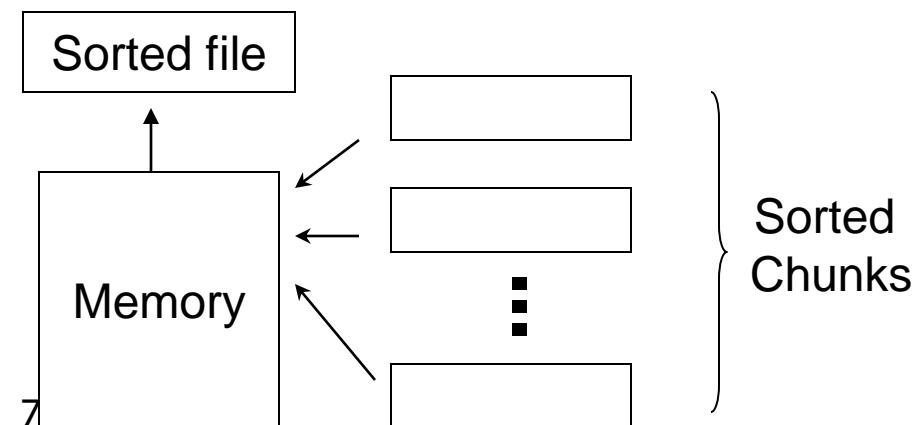
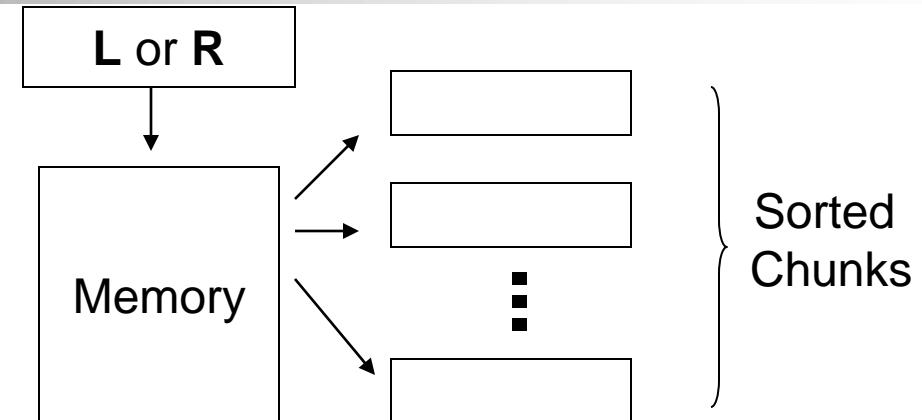
- L is left/outer relation
- Useful if no index, and not sorted on the join attribute
- Read as many pages of L as possible into memory
- Single pass over L, $B(L)/(M-1)$ passes over R



$L \bowtie R$

Sort Merge Join

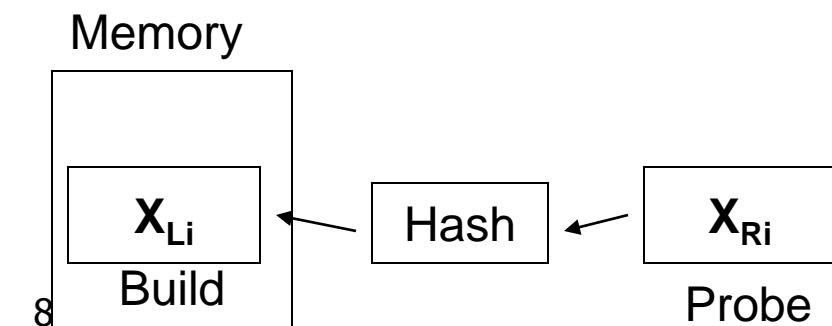
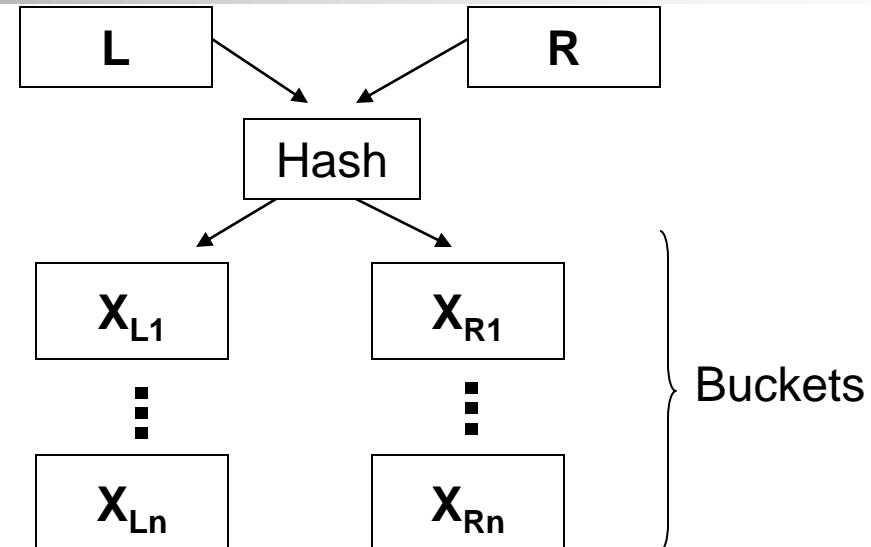
- Useful if either relation is sorted. Result sorted on join attribute.
- Divide relation into M sized chunks
- Sort the each chunk in memory and write to disk
- Merge sorted chunks



$L \bowtie R$

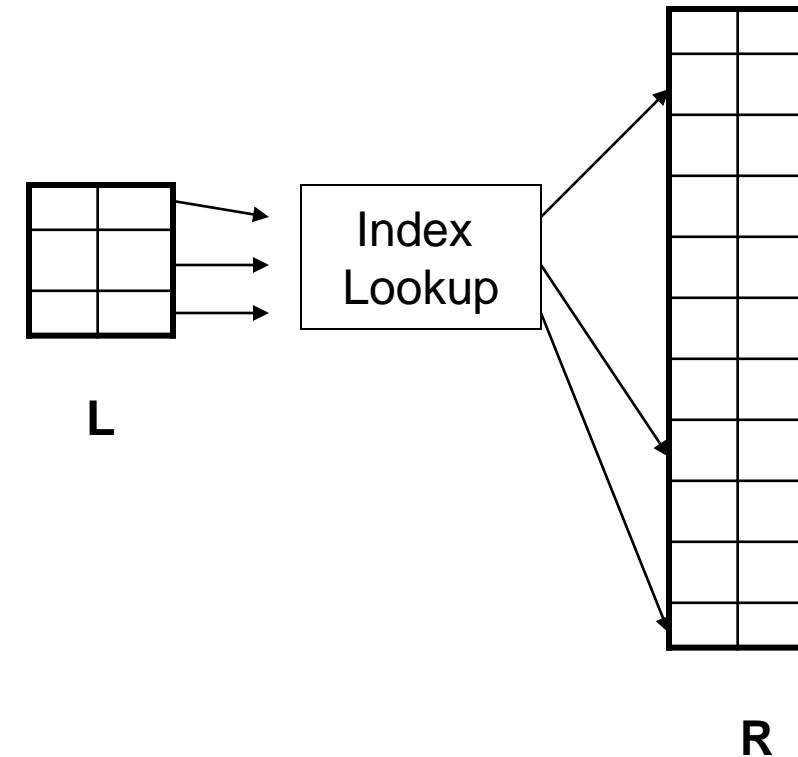
Hash Join

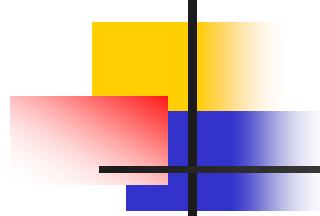
- Hash (using join attribute as key) tuples in **L** and **R** into respective buckets
- Join by matching tuples in the corresponding buckets of **L** and **R**
- Use **L** as build relation and **R** as probe relation



Index Join

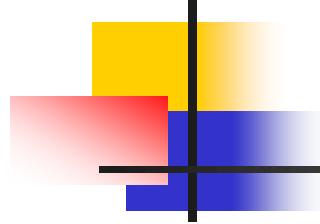
- Join attribute is indexed in R
- Match tuples in R by performing an index lookup for each tuple in L
- If clustered b-tree index, can perform sort-join using only the index
- Costs:
 - $B_L + T_L * C$
 - C the cost of index-based selection of inner relation R
 - $C = B_R/I$ If R is clustered
 - $C = T_R/I$ If R is not clustered





Ordering N-way Joins

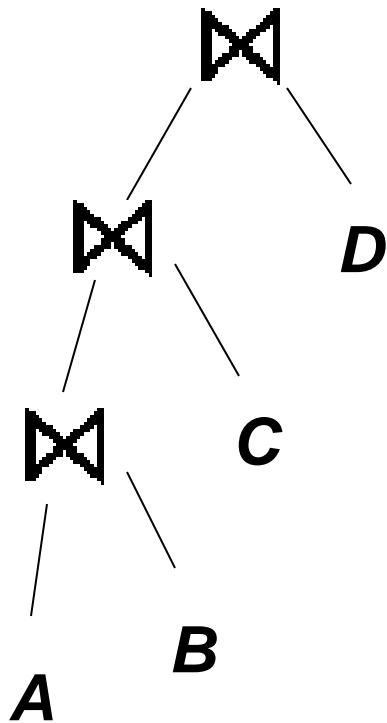
- Joins are commutative and associative
 - $(A \bowtie B) \bowtie C = A \bowtie (B \bowtie C)$
- Choose order which minimizes the sum of the sizes of intermediate results
 - Likely I/O and computationally efficient
 - If the result of $A \bowtie B$ is smaller than $B \bowtie C$, then choose $(A \bowtie B) \bowtie C$
- Alternative criteria: disk accesses, CPU, response time, network



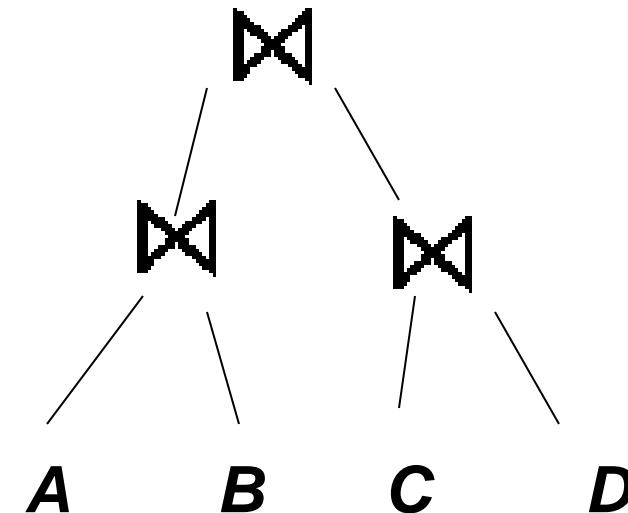
Ordering N-way Joins

- Choose the shape of the join tree
 - Equivalent to number of ways to parenthesize n-way joins
 - Recurrence: $T(1) = 1$
$$T(n) = \sum T(i)T(n-i), T(6) = 42$$
- Permutation of the leaves
 - $n!$
- For $n = 6$, the number of join trees is $42*6!$
Or 30,240

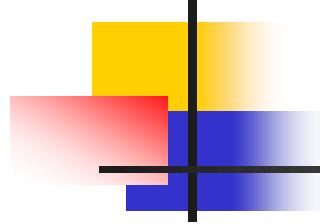
Shape of Join Tree



Left-deep Tree

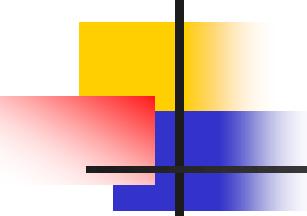


Bushy Tree



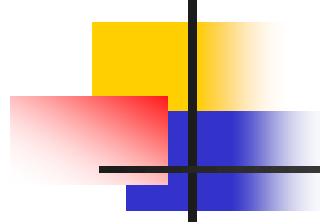
Shape of Join Tree

- A left-deep (right-deep) tree is a join tree in which the right-hand-side (left-hand-side) is a relation, not an intermediate join result
- Bushy tree is neither left nor right-deep
- Considering only left-deep trees is usually good enough
 - Smaller search space
 - Tend to be efficient for certain join algorithms (order relations from smallest to largest)
 - Allows for pipelining
 - Avoid materializing intermediate results on disk



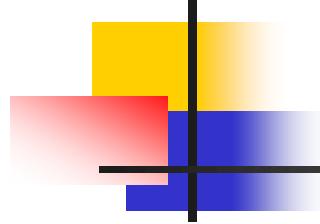
Searching for the best join plan

- Exhaustively enumerating all possible join order is not feasible ($n!$)
- Dynamic programming
 - use the best plan for $(k-1)$ -way join to compute the best k -way join
- Greedy heuristic algorithm
 - Iterative dynamic programming



Dynamic Programming

- The best way to join k relations is drawn from k plans in which the left argument is the least cost plan for joining $k-1$ relations
- $\text{BestPlan}(A,B,C,D,E) = \min \text{ of } ($
 $\text{BestPlan}(A,B,C,D) \bowtie E,$
 $\text{BestPlan}(A,B,C,E) \bowtie D,$
 $\text{BestPlan}(A,B,D,E) \bowtie C,$
 $\text{BestPlan}(A,C,D,E) \bowtie B,$
 $\text{BestPlan}(B,C,D,E) \bowtie A)$

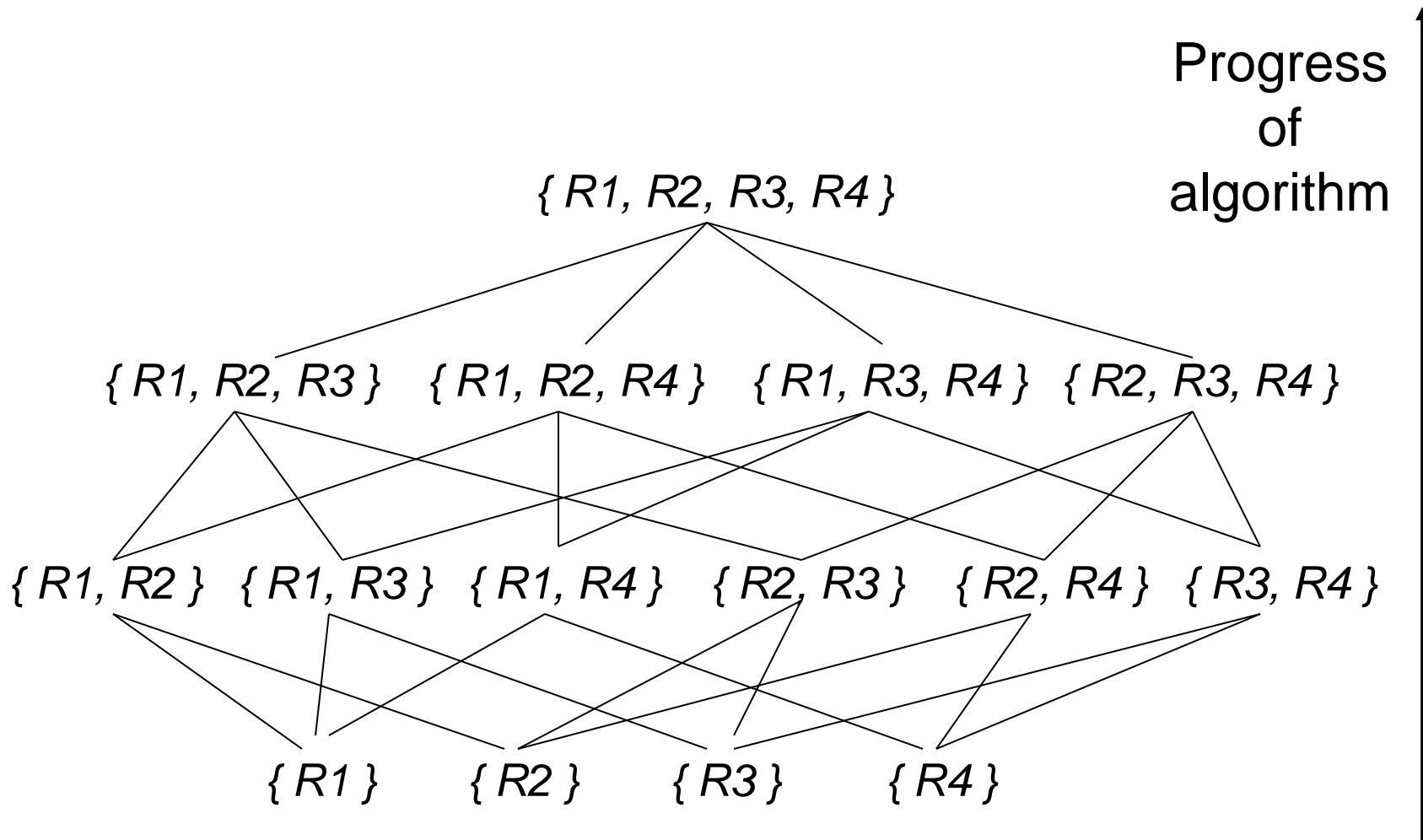


Complexity

- Finds optimal join order but must evaluate all 2-way, 3-way, ..., n-way joins ($n \text{ choose } k$)
- Time $O(n*2^n)$, Space $O(2^n)$
- Exponential complexity, but joins on > 10 relations rare

Selinger Algorithm:

Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4$



Notation

$\text{OPT}(\{R1, R2, R3\})$:

Cost of optimal plan to join $R1, R2, R3$

$\text{T}(\{R1, R2, R3\})$:

Number of tuples in $R1 \bowtie R2 \bowtie R3$

Selinger Algorithm:

$\text{OPT}(\{R_1, R_2, R_3\})$:

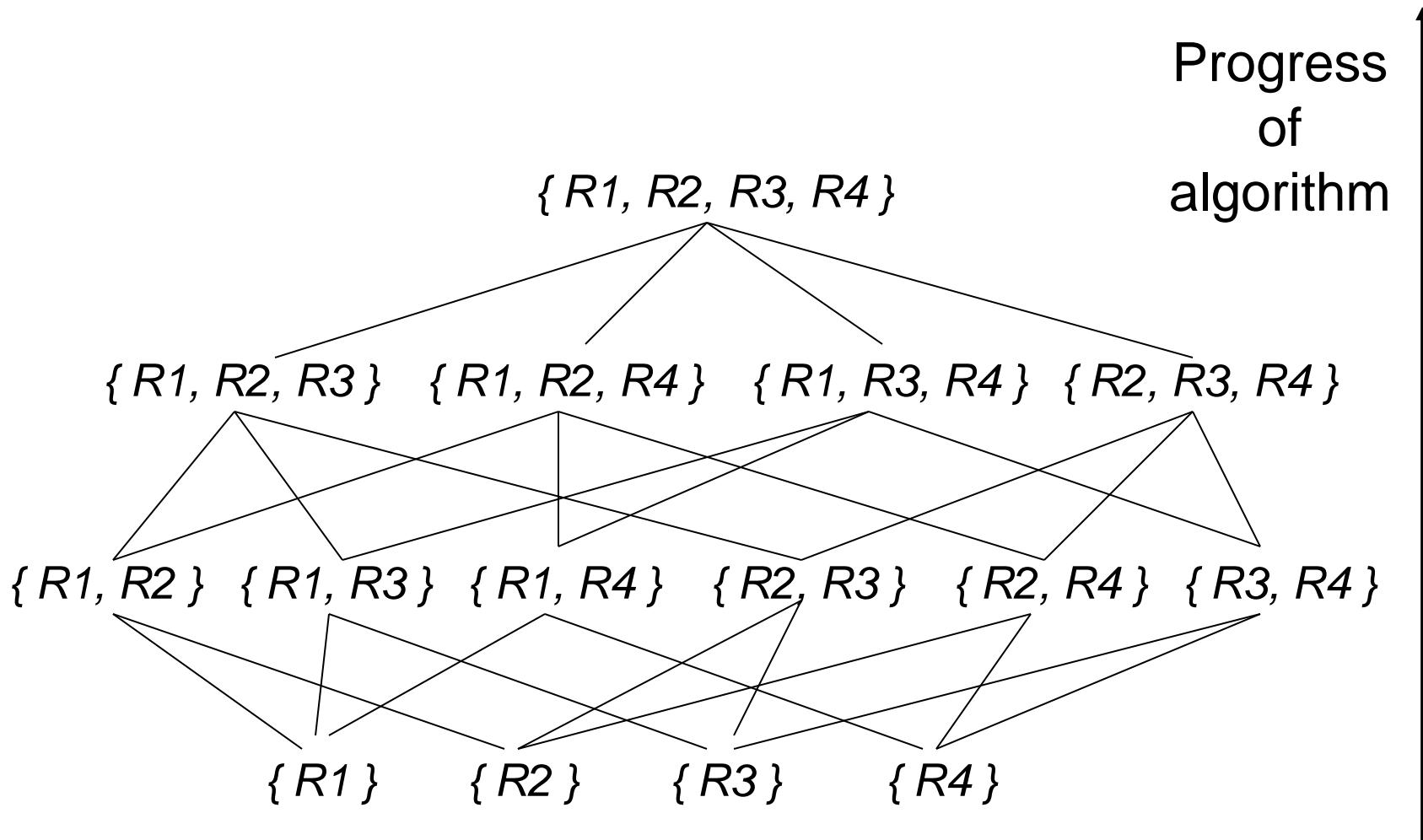
Min {

$$\begin{aligned} & \text{OPT}(\{R_1, R_2\}) + T(\{R_1, R_2\}) + T(R_3) \\ & \text{OPT}(\{R_2, R_3\}) + T(\{R_2, R_3\}) + T(R_1) \\ & \text{OPT}(\{R_1, R_3\}) + T(\{R_1, R_3\}) + T(R_2) \end{aligned}$$

Note: Valid only for the simple cost model

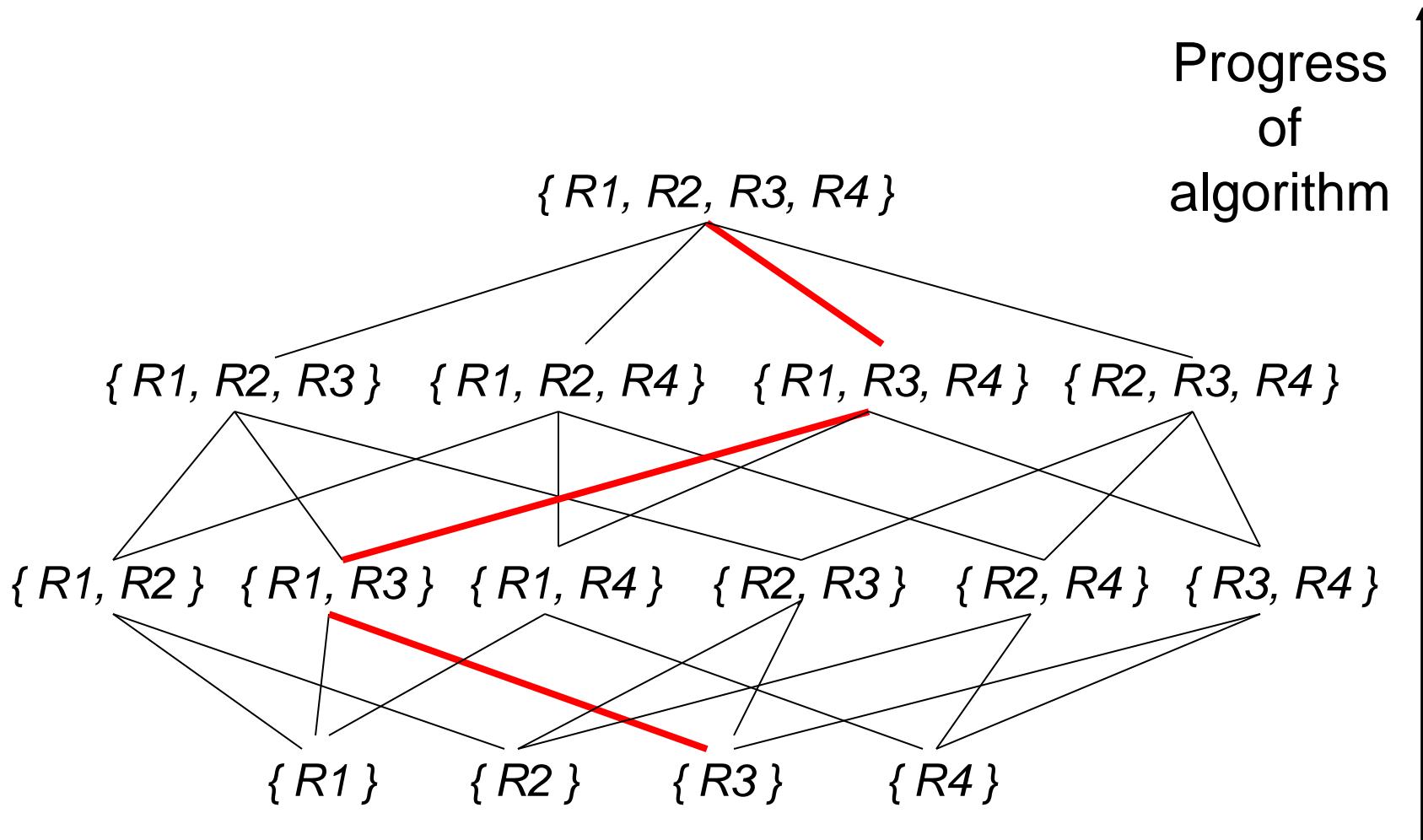
Selinger Algorithm:

Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4$



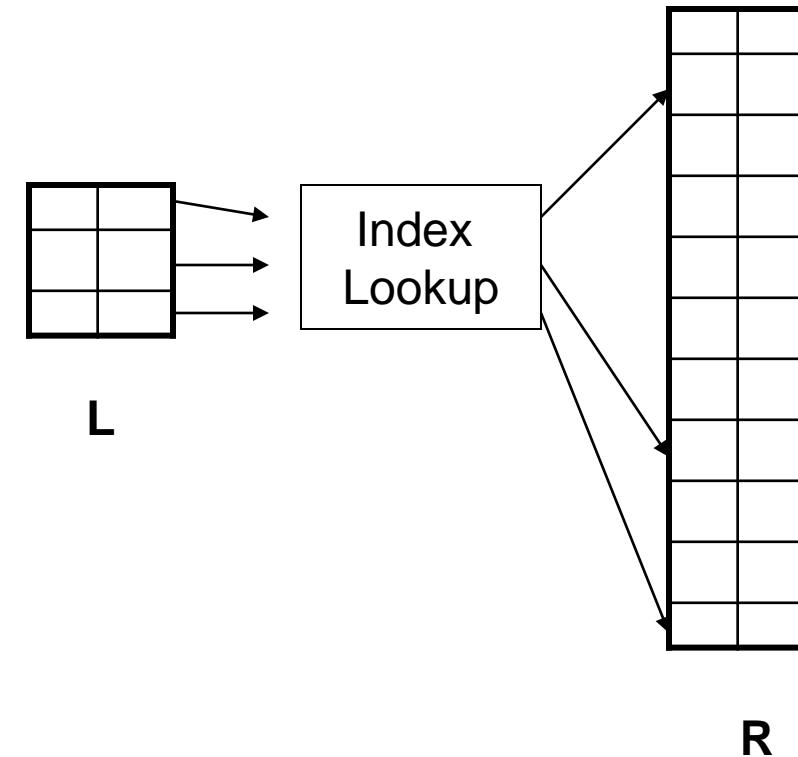
Selinger Algorithm:

Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4$



Index Join

- Join attribute is indexed in R
- Match tuples in R by performing an index lookup for each tuple in L
- If clustered b-tree index, can perform sort-join using only the index
- **Costs:**
 - $B_L + T_L * C$
 - C the cost of index-based selection of inner relation R
 - $C = B_R/I$ If R is clustered
 - $C = T_R/I$ If R is not clustered



9. téte

A $Q(A,B) JOIN R(B,C) JOIN S(C,D)$ háromféle kiszámítási módja és költsége, (feltéve, hogy Q,R,S paraméterei megegyeznek, Q.B-re és S.C-re klaszterindexünk van).

- a) balról jobbra,
- b) balról jobbra és a memóriában összekapcsolva a harmadik táblával,
- c) a középső ténytábla soraihoz kapcsolva a szélső dimenziótáblákat.

Feltevések:

$$T_Q = T_R = T_S = T \quad (\text{ugyanannyi soruk van})$$

$$B_Q = B_R = B_S = B \quad (\text{ugyanannyi helyet foglalnak})$$

$$I_{Q.B} = I_{R.B} = I_{R.C} = I_{S.C} = I \quad (\text{a képméretek, vagyis az előforduló értékek száma azonos})$$

Előzetes számítások

Az alábbiakban kiszámolt értékeket fel fogjuk használni a későbbiekben.

Először nézzük meg, hogyan lehetne előállítani két tábla összekapcsolását

$R(A,B) JOIN S(B,C)$ -t, ha minden táblán van index a közös oszlopra. Az azonos értékekhez tartozó sorokat az indexek alapján olvassuk be a táblákból, majd a memóriában összekapcsoljuk őket. Feltesszük, hogy az összekapcsolandó sorok beférnek a memóriába, vagyis $B_R/I + B_S/I \leq M$, valamint, hogy R.B részhalmaza S.B-nek. Egy index segítségével történő beolvasás költsége \approx a beolvasott blokkok száma, vagyis $B_R/I_{R,B}$ illetve $B_S/I_{S,S}$.

A teljes **JOIN művelet I/O költsége** (beolvassuk R-et, majd minden sorához index segítségével S-et. Az alábbi képlet az output kiírásának költségét nem tartalmazza.)
 $B_R + T_R * B_S/I_{S.B}$ $I_{R.B}=I_{S.B} = I$ esetén:

(1) $B_R + T_R * B_S/I$

Hány sora lesz a JOIN-nak?

$$T_{R|><|S} = I_{R.B} * (T_R/I_{R.B} * T_S/I_{S.B}) \quad (\text{az egyes értékekhez tartozó részek direkt szorzata})$$

Ha feltesszük, hogy $I_{R.B}=I_{S.B} = I$, akkor **a JOIN sorainak száma**:

(2) $T_{R|><|S} = T_R * T_S/I$

Mekkora méretű lesz az output? ($R \times S$ esetén $T_R * B_S + T_S * B_R$ lenne)

Az output mérete:

(3) $(T_R * B_S + T_S * B_R)/I$

A fenti 3 képletet fogjuk felhasználni a $Q(A,B) JOIN R(B,C) JOIN S(C,D)$ kiszámításához.

a) balról jobbra történő kiszámítás

$Q(A,B) JOIN R(B,C)$ -re

Output mérete: $2*T*B/I$ lásd (3)

Sorok száma: T^2/I lásd (2)

I/O költség: $B + T*B/I$ lásd (1)

Használjuk fel a fentieket $Q(A,B) JOIN R(B,C) JOIN S(C,D)$ esetén az output és az I/O költség kiszámításához.

Output mérete (3)-ba helyettesítve: $[(T^2/I)*B + (2*T*B/I)*T]/I = 3*T^2*B/I^2$

A teljes JOIN I/O költsége:

Az 1. join költsége $B + T*B/I$ plusz

Az 1. join kiírása (output mérete): $2*T*B/I$ plusz

A 2. join költsége $2*T*B/I + [(T^2/I)*B]/I$ plusz

A teljes output kiírása: $3*T^2*B/I^2$

összesen:

a) végeredménye: $B + 5*T*B/I + 4 *T^2*B/I^2$

b) balról jobbra és a memóriában összekapcsolva a harmadik táblával,
Megspórolhatjuk az 1. join eredményének kiírását majd újbóli beolvasását, vagyis
 $2 * (2*T*B/I)$ -t. Az eredmény ekkor:

b) végeredménye: $B + T*B/I + 4 *T^2*B/I^2$

c) a középső ténytábla soraihoz kapcsolva a szélső dimenziótáblákat.

Beolvassuk R-et, majd R minden sorára index alapján olvassuk be Q és S sorait. A költség ekkor:

Q beolvasása

B plusz

Q és S olvasása R minden sorára: **T*(B/I + B/I)** plusz

A teljes output kiírása: **3*T^2*B/I^2**

összesen:

c) végeredménye: B + 2*T*B/I + 3 *T^2*B/I^2

Nézzük meg, hogy a b) és c) esetek közül melyik a kisebb költségű. A két költség közötti különbség (b-c): **T^2*B/I^2 - T*B/I**

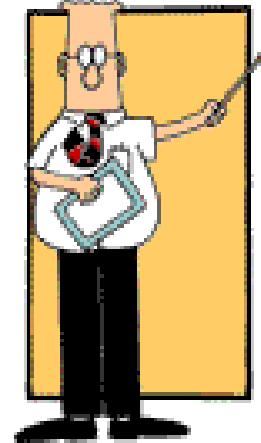
Nagyméretű táblák esetén a T/I hányados nagy szám lesz, ezért a négyzetes tag jóval nagyobb lesz, mint a lineáris tag, vagyis a c) módszer a leghatékonyabb.

Ha a c/b arányt tekintjük, akkor azt mondhatjuk, hogy ez az arány $\frac{3}{4}$ -hez tart, ha T/I tart a végtelenbe. Vagyis ha T/I elég nagy, akkor a c költsége nagyjából $\frac{3}{4}$ -e a b-nek.

Oracle SQL Tuning

Áttekintés

- Alapozás
 - Optimalizáló, költség vs. szabály, adattárolás, SQL végrehajtási fázisok, ...
- Végrehajtási tervezek létrehozása és olvasása
 - Elérési utak, egyetlen tábla, összekapcsolás, ...
- Eszközök
 - Követőfájlok, SQL tippek, analyze/dbms_stat
- Adattárház jellemzők
 - Csillag lekérdezés és bittérkép indexelés
 - ETL



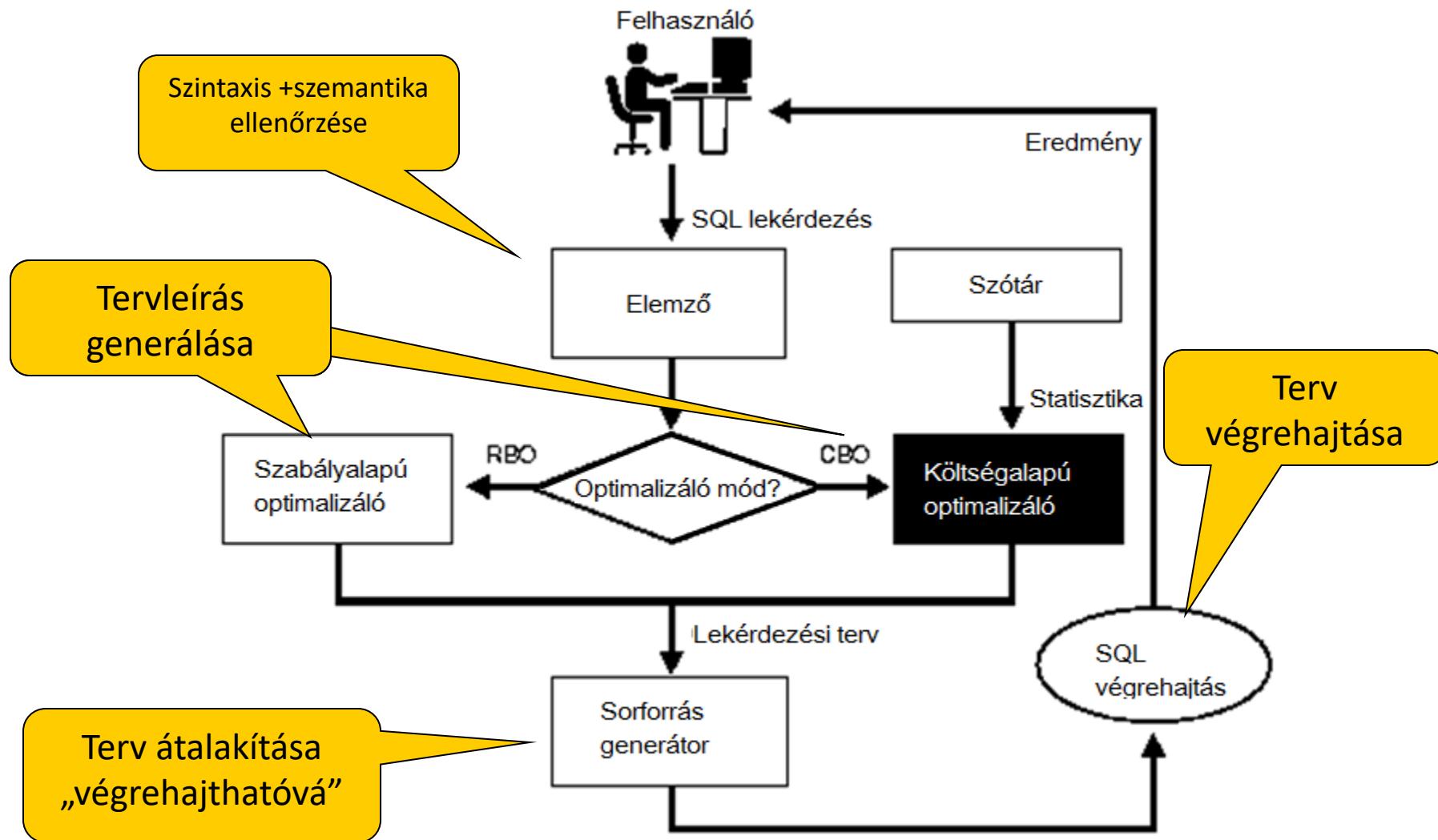
Célok

- Végrehajtási tervezés olvasása
 - Táblaelérés
 - Indexelérés
 - Összekapcsolás
 - Allekérdezések
- Végrehajtási tervezés megértése
 - Teljesítmény megértése
 - SQL optimalizáció alapjainak megértése
- Úgy gondolkodunk, hogy mi hogy hajtanánk végre

Következik...

- Alapfogalmak
 - Háttérinformáció
- SQL végrehajtás
 - Olvasás + értés

Optimalizáló áttekintés



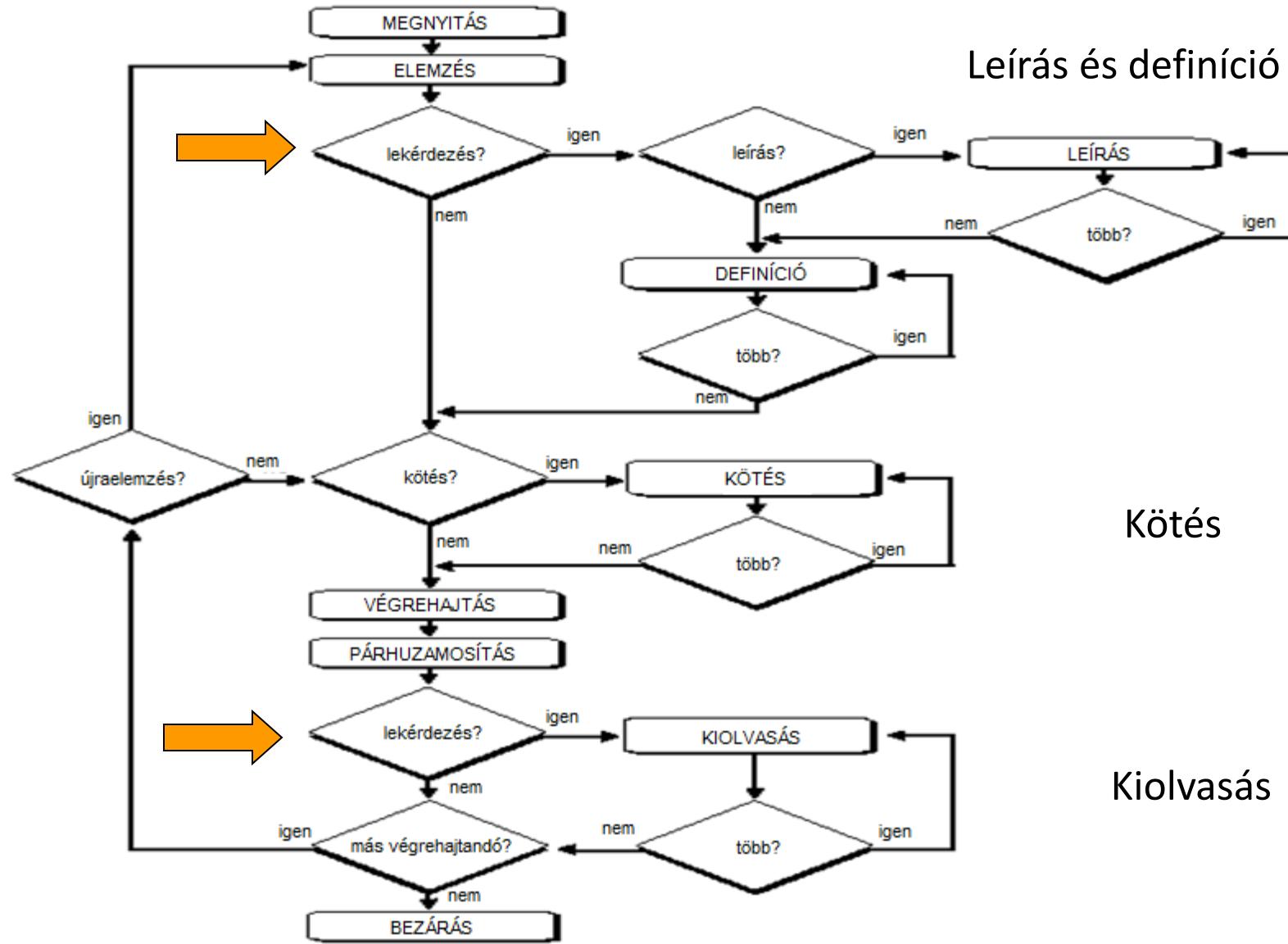
Költség vs. Szabály

- Szabály
 - Rögzített heurisztikus szabályok határozzák meg a tervet
 - „Indexen keresztül elérés gyorsabb, mint az egész tábla átnézése”
 - „teljesen megegyező index jobb, mint a részben megegyező index”
 - ...
- Költség (2 mód)
 - Az adatstatisztikák szerepet játszanak a terv meghatározásában
 - Legjobb átfutás: **minden sort** minél hamarabb
 - Először számoljon, aztán gyorsan térjen vissza
 - Legjobb válaszidő: az **első sort** minél hamarabb
 - Számítás közben már térjen vissza (ha lehetséges)

Melyiket hogyan állítjuk be?

- Példány szinten: Optimizer_Mode paraméter
 - Szabály
 - Választás
 - statisztikánál CBO (all_rows), egyébként RBO
 - First_rows, First_rows_n (1, 10, 100, 1000)
 - All_rows
- Munkamenet szinten:
 - Alter session set optimizer_mode=<mode>;
- Utasítás szinten:
 - SQL szövegben elhelyezett tippek mutatják a használandó módot

SQL végrehajtás: DML vs. lekérdezések



DML vs. Lekérdezések

- Megnyitás => Elemzés => Végrehajtás (=> Kiolvasásⁿ)

```
SELECT ename, salary  
FROM emp  
WHERE salary>100000
```

Kliens általi
kiolvasás

```
UPDATE emp  
SET commission='N'  
WHERE salary>100000
```

Ugyanaz az SQL
optimalizáció

Minden beolvasást belsőleg
az SQL végrehajtó végez el

KLIENS

=> SQL =>
<= Adat vagy visszatérési kód<=

SZERVER

Adattárolás: Táblák

- Az Oracle az összes adatot adatfájlokban tárolja
 - Hely és méret DBA által meghatározott
 - Logikailag táblaterekbe csoportosítva
 - minden fájlt egy relatív fájlszám (fno) azonosít
- Az adatfájl adatblokkokból áll
 - Mérete egyenlő a *db_block_size* paraméterrel
 - minden blokkot a fájlbeli eltolása azonosít
- Az adatblokkok sorokat tartalmaznak
 - minden sort a blokkban elfoglalt helye azonosít

ROWID: <Blokk>.<Sor>.<Fájl>

Adattárolás: Táblák

x. fájl

| 1. blokk | 2. blokk | 3. blokk | 4. blokk |
|----------|-----------|---|----------|
| 5. blokk | ... blokk | <Rec1><Rec2><Rec3> <Rec4><Rec5><Rec6> <Rec7><Rec8><Rec9> ... | |
| | | | |
| | | | |



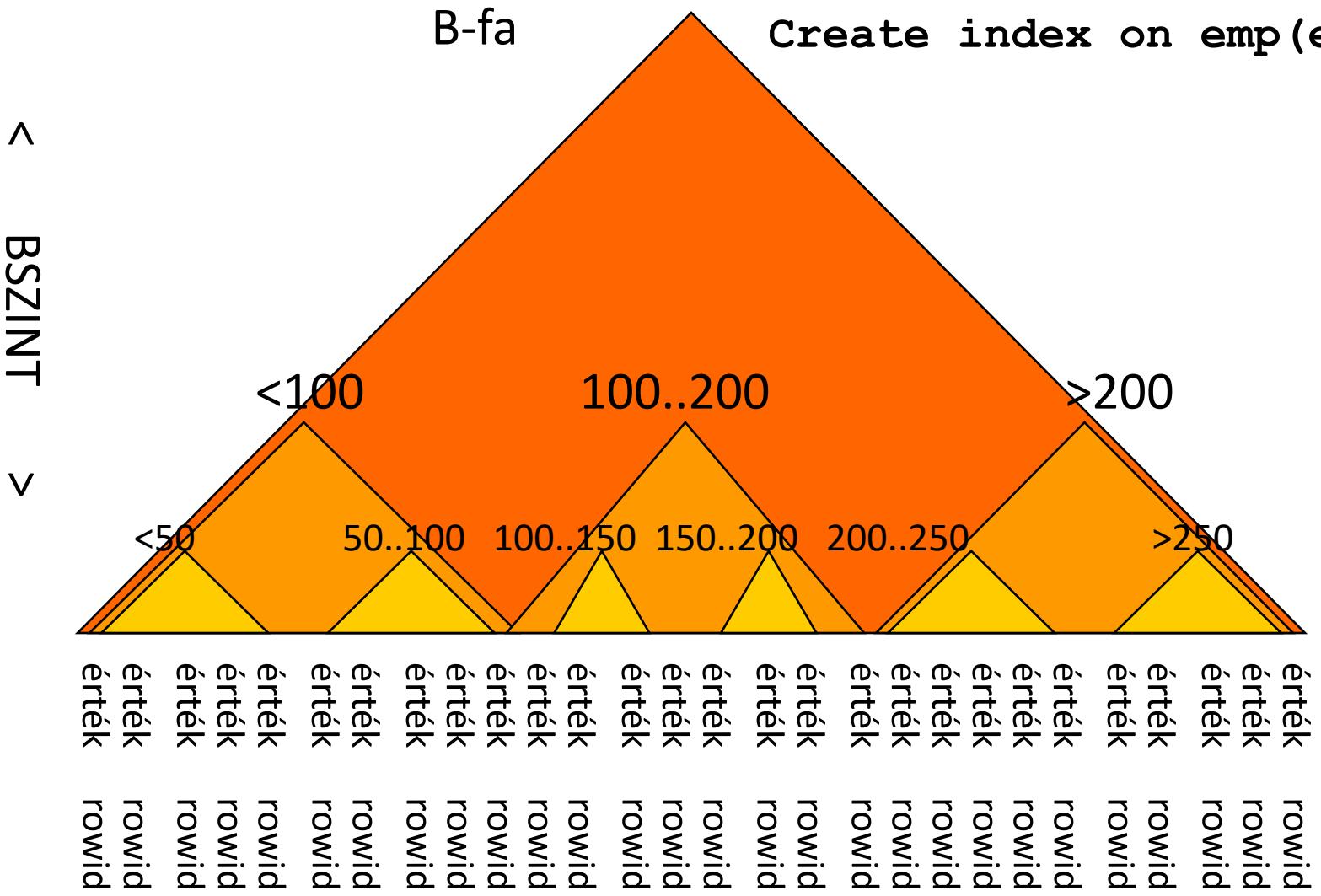
Rowid: 00000006.0000.000X

Adattárolás: Indexek

- Kiegyensúlyozott fák
 - Indexelt oszlop(ok) rendezett tárolása külön
 - a NULL érték kimarad az indexből
 - A mutatószerkezet logaritmikus keresést tesz lehetővé
 - Először az indexet érjük el, megkeressük a táblamutatót, aztán elérjük a táblát
- B-fa tartalma:
 - Csomópont blokkok
 - Más csomópontokhoz vagy levelekhez tartalmaz mutatókat
 - Levélblokkok
 - A tényleges indexelt adatot tartalmazzák
 - Tartalmaznak rowid-ket (sormutatókat)
- Szintén blokkokban tárolódik az adatfájlokban
 - Szabadalmazott formátum

CSOMÓPONTOK LEVELEK

Adattárolás: Indexek



Adattárolás: Indexek

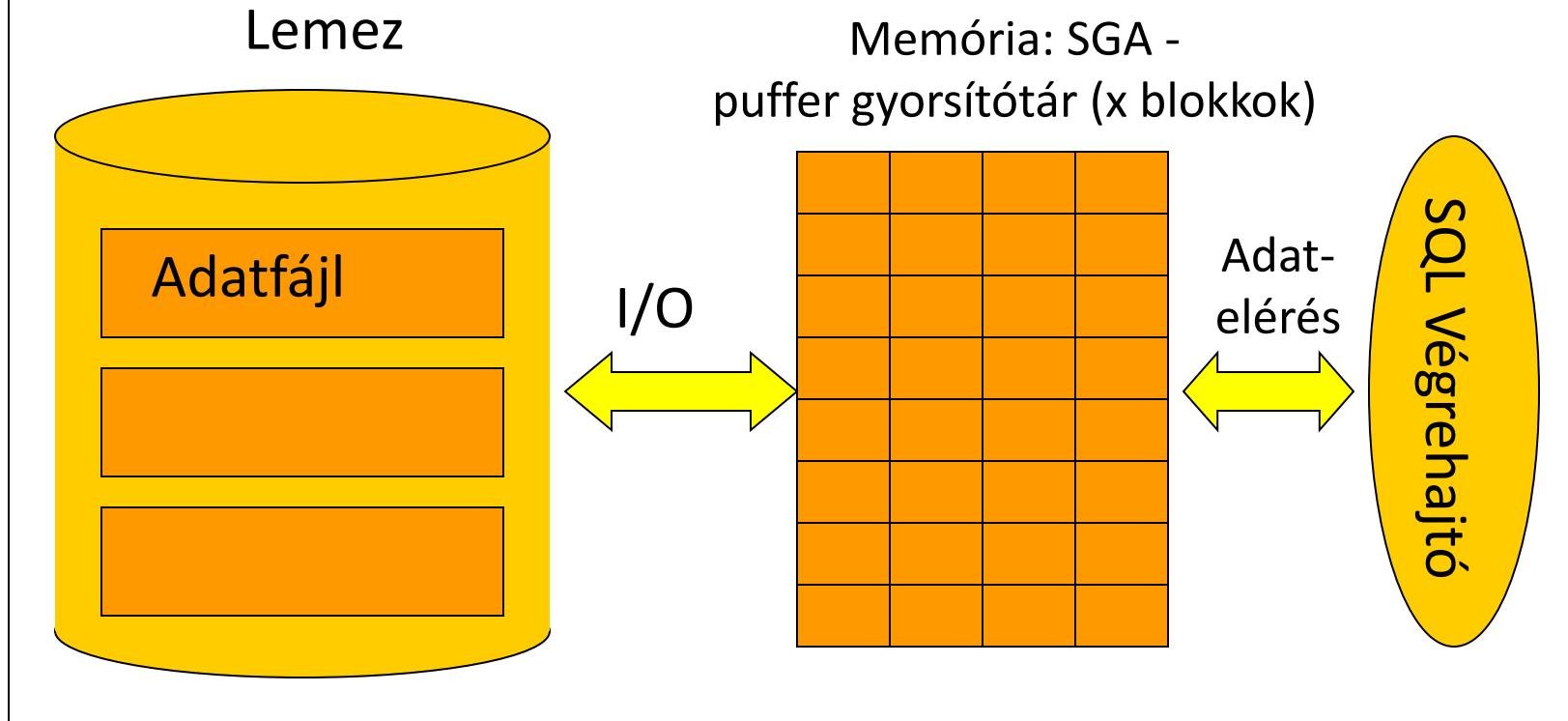
Adatfájl

| | | | |
|-------------------------|----------|-----------------------------|-------------------------|
| 1. blokk | 2. blokk | 3. blokk | 4. blokk |
| 5. blokk | ...blokk | Index csomópont blokk | Index levél blokk |
| Index levél blokk | | | |
| | | | |

Nincs kitüntetett sorrendje a csomópont és levél blokkoknak

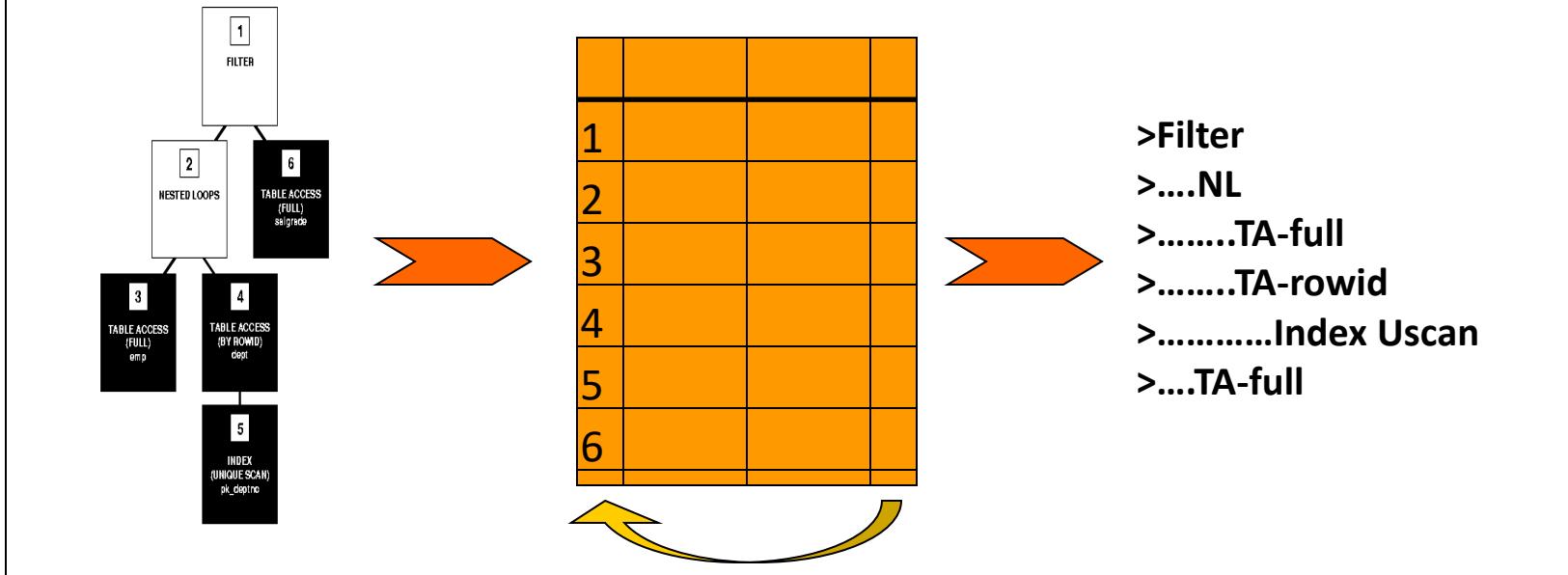
Tábla és Index I/O

- Az I/O blokk szinten történik
 - LRU lista vezérli, kinek „jut hely” a gyorsítótárban



Tervmagyarázó eszköz

- “Explain plan for <SQL-utasítás>”
 - Elmenti a tervet (sorforrások + műveletek) Plan_Table-be
 - Plan_Table nézete (vagy külső eszköz) formázza olvasható tervvé



Tervmagyarázó eszköz

```
create table PLAN_TABLE (
    statement_id      varchar2(30),      operation        varchar2(30),
    options          varchar2(30),      object_owner     varchar2(30),
    object_name       varchar2(30),      id               numeric,
    parent_id         numeric,          position         numeric,
    cost              numeric,          bytes            numeric);
```

```
create or replace view PLANS(STATEMENT_ID,PLAN,POSITION) as
select statement_id,
       rpad('>',2*level,'.')||operation||
       decode(options,NULL,' ',' (')||nvl(options,' ') ||
       decode(options,NULL,' ',' )') ||
       decode(object_owner,NULL,' ',object_owner||'.')||object_name plan,
       position
  from plan_table
 start with id=0
 connect by prior id=parent_id
        and prior nvl(statement_id,'NULL')=nvl(statement_id,'NULL')
```

Végrehajtási tervezek

1. Egyetlen tábla index nélkül
2. Egyetlen tábla indexsel
3. Összekapcsolások
 1. Skatulyázott ciklusok
 2. Összefésüléses rendezés
 3. Hasítás1 (kicsi/nagy), hasítás2 (nagy/nagy)
4. Speciális műveletek

Egyetlen tábla, nincs index (1.1)

```
SELECT *  
FROM emp;
```

```
>.SELECT STATEMENT  
>...TABLE ACCESS full emp
```

- Teljes táblabeolvasás (FTS)
 - minden blokk beolvasása sorozatban a puffer gyorsítótárba
 - másik neve “buffer-gets”
 - többszörös blokk I/O-val (db_file_multiblock_read_count)
 - amíg a magas vízsintjelzőt el nem érjük (truncate újraindítja, delete nem)
 - blokkonként: kiolvasás + minden sor visszaadása
 - aztán a blokk visszarakása a LRU-végen az LRU listába (!)
 - minden más művelet a blokkot az MRU-végre rakja

Egyetlen tábla, nincs index(1.2)

```
SELECT *  
FROM emp  
WHERE sal > 100000;
```

```
>.SELECT STATEMENT  
>...TABLE ACCESS full emp
```

- Teljes táblabeolvasás szűréssel
 - minden blokk beolvasása
 - Blokkonként beolvasás, szűrés, aztán sor visszaadása
 - **Az egyszerű where-feltételek nem látszanak a tervben**
 - FTS-nél: sorok-be < sorok-ki

Egyetlen tábla, nincs index (1.3)

```
SELECT *  
FROM emp  
ORDER BY ename;
```

```
>.SELECT STATEMENT  
>...SORT order by  
>....TABLE ACCESS full emp
```

- FTS, aztán rendezés a rendezendő mező(kö)n
 - „Aztán”, tehát a rendezés addig nem ad vissza adatot, amíg a szülő sorforrás nem teljes
 - SORT order by: sorok-be = sorok-ki
 - Kis rendezések a memóriában (SORT_AREA_SIZE)
 - Nagy rendezések a TEMPORARY táblatéren
 - Lehet, hogy nagy mennyiségű I/O

Egyetlen tábla, nincs index (1.3)

```
SELECT *  
FROM emp  
ORDER BY ename;
```

Emp(ename)

```
>.SELECT STATEMENT  
>...TABLE ACCESS full emp  
>.....INDEX full scan i_emp_ename
```

- Ha a rendezendő mező(kö)n van index
 - Index Full Scan
 - CBO használja az indexet, ha a mód = First_Rows
 - Ha használja az indexet => nem kell rendezni

Egyetlen tábla, nincs index(1.4)

```
SELECT job,sum(sal)  
FROM emp  
GROUP BY job;
```

```
>.SELECT STATEMENT  
>...SORT group by  
>....TABLE ACCESS full emp
```

- FTS , aztán rendezés a csoportosító mező(kö)n
 - FTS csak a job és sal mezőket olvassa ki
 - Kis köztes sorméret => gyakrabban rendezhető a memóriában
 - SORT group by: sorok-be >> sorok-ki
 - A rendezés kiszámolja az aggregátumokat is

Egyetlen tábla, nincs index (1.5)

```
SELECT job,sum(sal)  
FROM emp  
GROUP BY job  
HAVING sum(sal)>200000;
```

```
>.SELECT STATEMENT  
>...FILTER  
>.....SORT group by  
>.....TABLE ACCESS full emp
```

- HAVING szűrés
 - Csak a having feltételnek megfelelő sorokat hagyja meg

Egyetlen tábla, nincs index(1.6)

```
SELECT *  
FROM emp  
WHERE rowid=  
'00004F2A.00A2.000C'
```

```
>.SELECT STATEMENT  
>...TABLE ACCESS by rowid emp
```

- Táblaelérés rowid alapján
 - Egy sor megkeresése
 - Azonnal a blokkra megy és kiszűri a sort
 - A leggyorsabb módszer egy sor kinyerésére
 - Ha tudjuk a rowid-t

Egyetlen tábla, index(2.1)

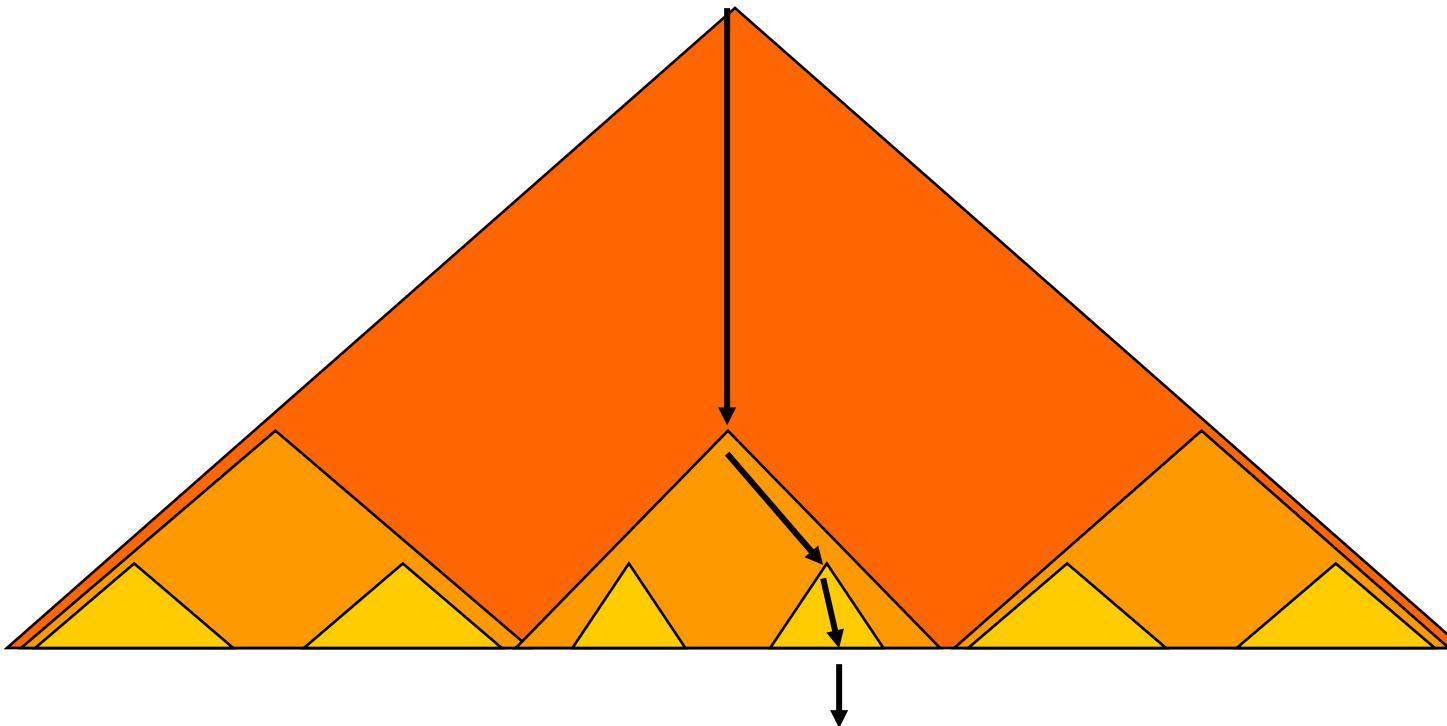
```
SELECT *  
FROM emp  
WHERE empno=174;
```

Unique emp(empno)

```
>.SELECT STATEMENT  
>...TABLE ACCESS by rowid emp  
>....INDEX unique scan i_emp_pk
```

- Index egyedi keresés
 - Bejárja a csomópont blokkokat, hogy megtalálja a megfelelő levélblokkot
 - Megkeresi az értéket a levélblokkban (ha nem találja => kész)
 - Visszaadja a rowid-t a szülő sorforrásnak
 - Szülő: eléri a fájl+blokkot és visszaadja a sort

Index egyedi keresés (2.1)



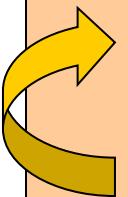
Táblaelrés rowid alapján

Egyetlen tábla, index(2.2)

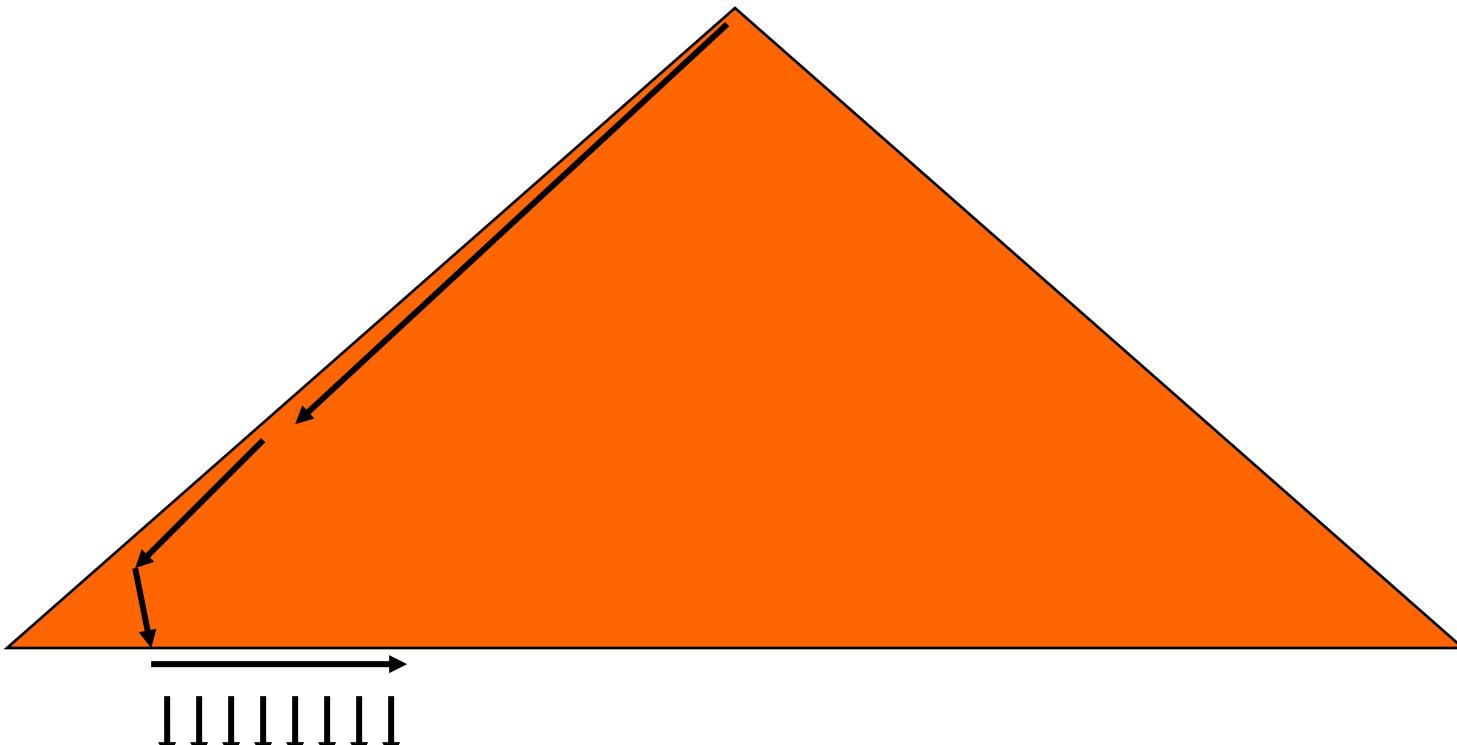
```
SELECT *  
FROM emp  
WHERE job='manager';
```

emp(job)

```
>.SELECT STATEMENT  
>...TABLE ACCESS by rowid emp  
>.....INDEX range scan i_emp_job
```

- (Nem egyedi) index intervallum keresés
 - Bejárja a csomópont blokkokat, hogy megtalálja a bal szélső levélblokkot
 - Megkeresi az érték első előfordulását
 - Visszaadja a rowid-t a szülő sorforrásnak
 - Szülő: eléri a fájl+blokkot és visszaadja a sort
 - Folytatja az érték minden előfordulására
 - Amíg van még előfordulás
- 

Index intervallum keresés (2.2)



Táblaelrés rowid alapján

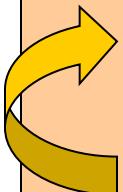
Egyetlen tábla, index(2.3)

```
SELECT *  
FROM emp  
WHERE empno>100;
```

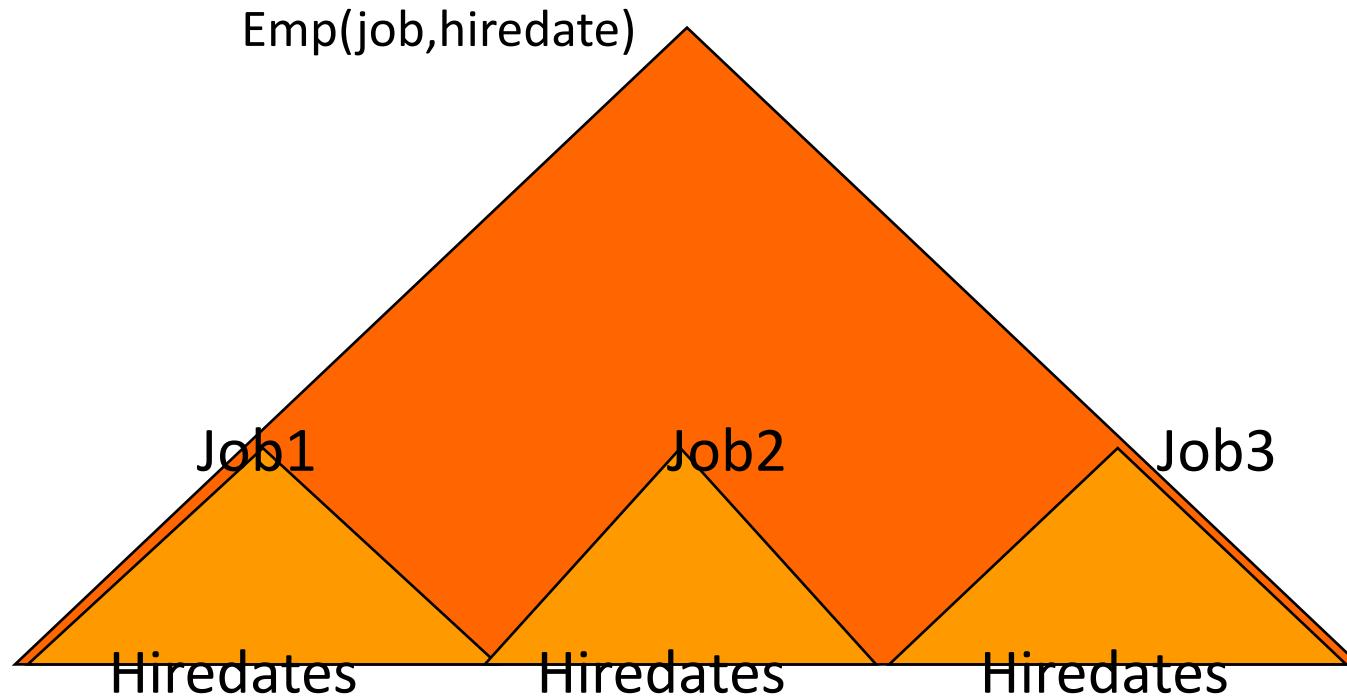
Unique emp(empno)

```
>.SELECT STATEMENT  
>...TABLE ACCESS by rowid emp  
>.....INDEX range scan i_emp_pk
```

- Egyedi index intervallum keresés
 - Bejárja a csomópont blokkokat, hogy megtalálja a bal szélső levélblokkot a kezdőértékkel
 - Megkeresi az intervallumbeli első előforduló értéket
 - Visszaadja a rowid-t a szülő sorforrásnak
 - Szülő: eléri a fájl+blokkot és visszaadja a sort
 - Folytatja a következő érvényes előfordulással
 - Amíg van előfordulás az intervallumban



Összefűzött indexek



Többszintű B-fa, mezők szerinti sorrendben

Egyetlen tábla, index(2.4)

```
SELECT *
FROM emp
WHERE job='manager'
AND hiredate='01-01-2001';
```

Emp(job,hiredate)

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>.....INDEX range scan i_emp_j_h
```

- Teljes összefűzött index
 - Felhasználja a job értékét az al-B-fához navigálásra
 - Aztán megkeresi az alkalmas hiredate-eket

Egyetlen tábla, index(2.5)

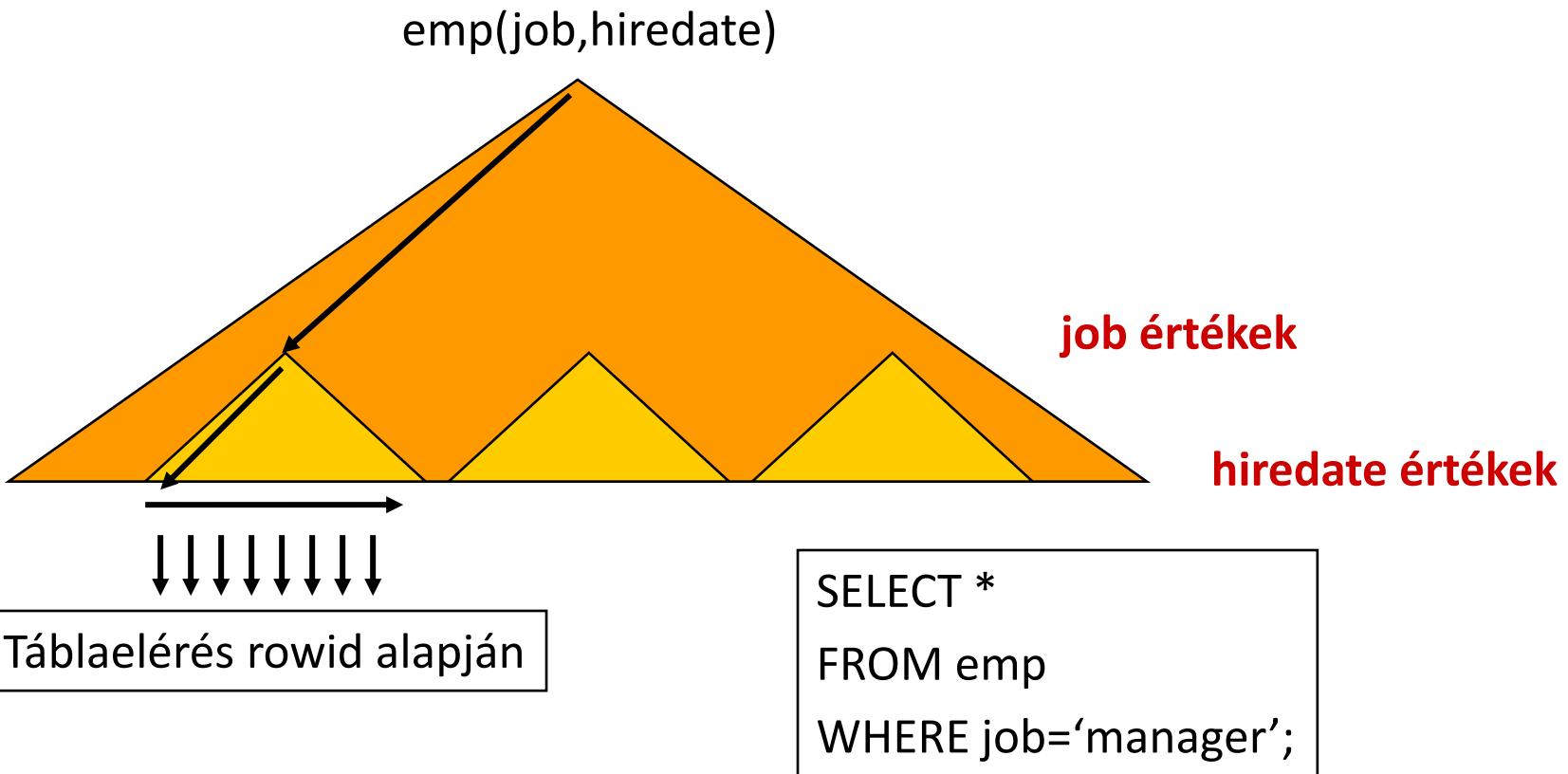
```
SELECT *  
FROM emp  
WHERE job='manager';
```

Emp(job,hiredate)

```
>.SELECT STATEMENT  
>...TABLE ACCESS by rowid emp  
>....INDEX range scan i_emp_j_h
```

- (Bevezető) Összefűzött index prefixe
 - Végignézi a teljes al-B-fát a nagy B-fán belül

Index intervallumkeresés (2.5)



Egyetlen tábla, index(2.6)

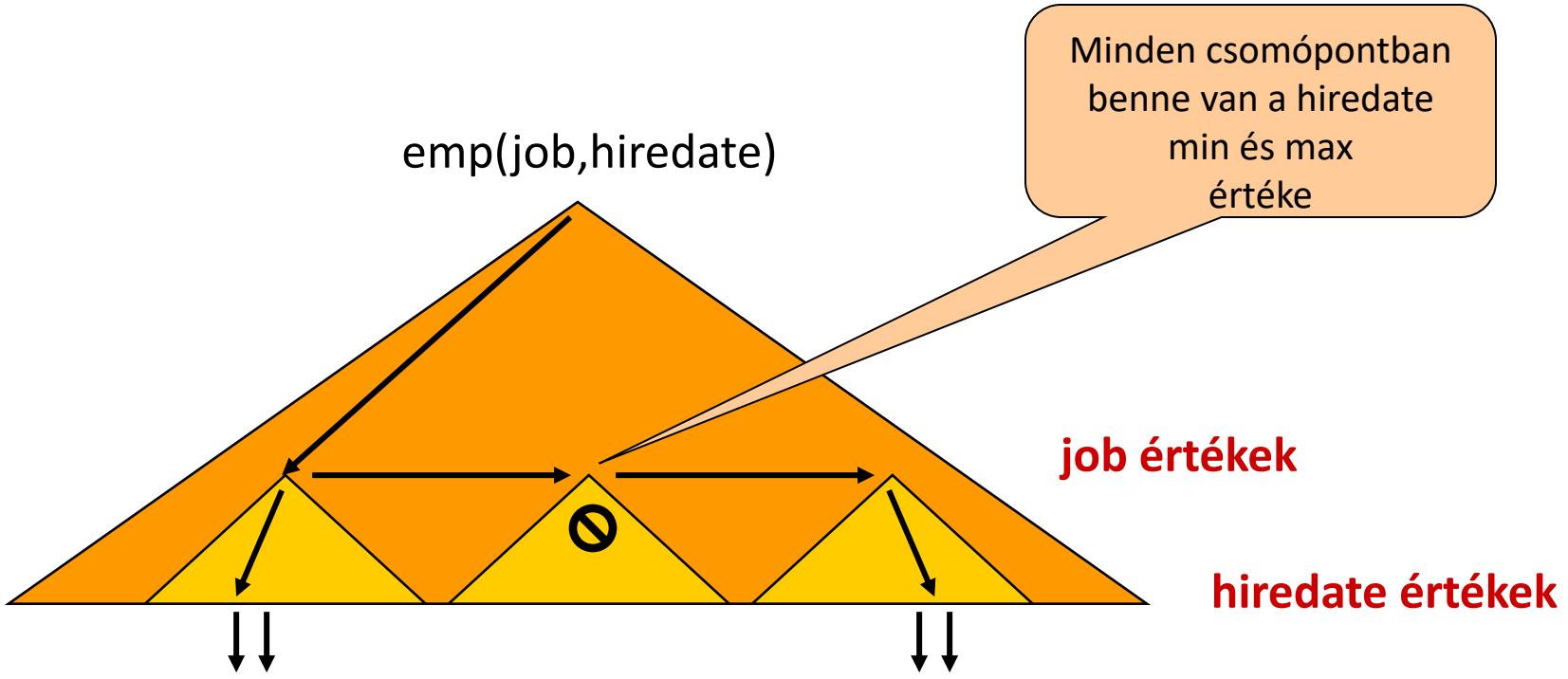
```
SELECT *  
FROM emp  
WHERE hiredate='01-01-2001';
```

Emp(job,hiredate)

```
>. SELECT STATEMENT  
>... TABLE ACCESS by rowid emp  
>.... INDEX range scan i_emp_j_h
```

- Index kihagyásos keresés (korábbi verziókban FTS)
 - „Ott használunk indexet, ahol eddig soha nem használtuk”
 - A bevezető mezőkön már nem kell predikátum
 - A B-fát sok kis al-B-fa gyűjteményének tekinti
 - Legjobban kis számosságú bevezető mezőkre működik

Index kihagyásos keresés (2.6)



Egyetlen tábla, index(2.7)

```
SELECT *  
FROM emp  
WHERE empno>100  
AND job='manager';
```

Unique Emp(empno)
Emp(job)

```
>.SELECT STATEMENT  
>...TABLE ACCESS by rowid emp  
>.....INDEX range scan i_emp_job
```

- Több index
 - Szabály: heurisztikus döntéslista alapján választ
 - Az elérhető indexeket rangsorolja
 - Költség: kiszámolja a legtöbbet kiválasztót (azaz a legkisebb költségűt)
 - Statisztikát használ

RBO heurisztikák

- Több elérhető index rangsorolása
 1. Egyenlőség egy mezős egyedi indexen
 2. Egyenlőség láncolt egyedi indexen
 3. Egyenlőség láncolt indexen
 4. Egyenlőség egy mezős indexen
 5. Korlátos intervallum keresés indexben
 - Like, Between, Leading-part, ...
 6. Nem korlátos intervallum keresés indexen
 - Kisebb, nagyobb (a bevezető részen)

Általában tippel választjuk ki, melyiket használjuk

CBO költségszámítás

- Statisztikák különböző szinteken
 - Tábla:
 - Num_rows, Blocks, Empty_blocks, Avg_space
 - Mező:
 - Num_values, Low_value, High_value, Num_nulls
 - Index:
 - Distinct_keys, Blevel, Avg_leaf_blocks_per_key, Avg_data_blocks_per_key, Leaf_blocks
- Az egyes indexek kiválasztóképességének számításához használjuk
 - Kiválasztóképesség = a sorok hány százalékát adja vissza
 - az I/O száma fontos szerepet játszik
 - FTS-t is figyelembe vesszük most!

Egyetlen tábla, index(2.1)

```
SELECT *  
FROM emp  
WHERE empno=174;
```

Unique emp(empno)

```
>.SELECT STATEMENT  
>...TABLE ACCESS by rowid emp  
>....INDEX unique scan i_emp_pk  
Or,  
>.SELECT STATEMENT  
>...TABLE ACCESS full emp
```

- CBO teljes táblabeolvasást használ, ha FTS-hez szükséges I/O < # IRS-hez szükséges I/O
 - FTS I/O a db_file_multiblock_read_count (dfmrc)-t használja
 - Typically 16
 - Egyedi keresés: $(\text{bszint} + 1) + 1$ I/O
 - FTS: $\lceil \text{táblasorok száma} / \text{dfmrc} \rceil$ I/O

CBO: csomósodási tényező

- Index szintű statisztika

- Mennyire jól rendezettek a sorok az indexelt értékekhez képest?
- Átlagos blokkszám, hogy elérjünk egyetlen értéket
 - 1 azt jelenti, hogy az intervallumkeresés olcsó
 - <táblasorok száma> azt jelenti, hogy az intervallumkeresés drága
- Arra használja, hogy több elérhető intervallumkeresést rangsoroljon

| Blck 1 | Blck 2 | Blck 3 |
|--------|--------|--------|
| ----- | ----- | ----- |
| A A A | B B B | C C C |

Clust.factor = 1

| Blck 1 | Blck 2 | Blck 3 |
|--------|--------|--------|
| ----- | ----- | ----- |
| A B C | A B C | A B C |

Clust.factor = 3

Egyetlen tábla, index(2.2)

```
SELECT *  
FROM emp  
WHERE job='manager';
```

emp(job)

```
>.SELECT STATEMENT  
>...TABLE ACCESS by rowid emp  
>....INDEX range scan i_emp_job  
Or,  
>.SELECT STATEMENT  
>...TABLE ACCESS full emp
```

- Csomósodási tényező IRS és FTS összehasonlításában
 - Ha (táblasorok / dfmrc)
<
(értékek száma * csomó.tény.) + bszint + meglátogatandó levél blokkok
akkor FTS-t használunk

Egyetlen tábla, index(2.7)

```
SELECT *
FROM emp
WHERE empno>100
AND job='manager';
```

Unique Emp(empno)
Emp(job)

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>....INDEX range scan i_emp_job
Or,
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>....INDEX range scan i_emp_empno
```

- Csomó.tényező több IRS összehasonlításában
 - Feltesszük, hogy a FTS túl sok I/O
 - Hasonlítsuk össze (értékek száma * csomó.tény.)-t, hogy válasszunk az indexek közül
 - Empno-kiválasztóképesség => értékek száma * 1 => I/O szám
 - Job-kiválasztóképesség => 1 * csomó.tény. => I/O szám

Egyetlen tábla, index(2.8)

```
SELECT *
FROM emp
WHERE job='manager'
AND depno=10
```

Emp(job)
Emp(depno)

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>....AND-EQUAL
>.....INDEX range scan i_emp_job
>.....INDEX range scan i_emp_depno
```

- Több azonos rangú, egymezős index
 - ÉS-EGYENLŐ: legfeljebb 5 egymezős intervallumkeresést von össze
 - Kombinál több index intervallumkeresést táblaelérés előtt
 - Az egyetemes intervallumkeresések rowid-halmazait összemetszi
 - CBO-nál ritkán fordul elő

Egyetlen tábla, index(2.9)

```
SELECT ename  
FROM emp  
WHERE job='manager';
```

Emp(job,ename)

```
>.SELECT STATEMENT  
>...INDEX range scan i_emp_j_e
```

- Indexek használata táblaelrés elkerülésére
 - A SELECT listán levő mezőktől és a WHERE feltétel bizonyos részein
 - Nincs táblaelrés, ha az összes mező indexben van

Egyetlen tábla, index(2.10)

```
SELECT count(*)  
FROM big_emp;
```

Big_emp(empno)

```
>. SELECT STATEMENT  
>... INDEX fast full scan i_emp_empno
```

- Gyors teljes index keresés (CBO only)
 - Ugyanazt a több blokkos I/O-t használja, mint az FTS
 - A kiválasztható indexeknek legalább egy NOT NULL mezőt kell tartalmazniuk
 - A sorok levélblokk sorrendben adódnak vissza
 - Nem indexelt mezők sorrendben

Összekapcsolás, skatulyázott ciklusok(3.1)

```
SELECT *  
FROM dept, emp;
```

```
>.SELECT STATEMENT  
>...NESTED LOOPS  
>....TABLE ACCESS full dept  
>....TABLE ACCESS full emp
```

- Teljes direkt szorzat skatulyázott ciklusos összekapcsolással (NLJ)
 - Init(RowSource1);
While not eof(RowSource1)
Loop Init(RowSource2);
 While not eof(RowSource2)
 Loop return(CurRec(RowSource1)+CurRec(RowSource2));
 NxtRec(RowSource2);
 End Loop;
 NxtRec(RowSource1);
 End Loop;

Két ciklus,
skatulyázott

Összekapcsolás, összefésüléses rendező(3.2)

```
SELECT *  
FROM emp, dept  
WHERE emp.d# = dept.d#;
```

```
>.SELECT STATEMENT  
>...MERGE JOIN  
>.....SORT join  
>.....TABLE ACCESS full emp  
>.....SORT join  
>.....TABLE ACCESS full dept
```

- Belső összekapcsolás, nincs index: összefésüléses rendező összekapcsolás (SMJ)

```
Tmp1 := Sort(RowSource1,JoinColumn);  
Tmp2 := Sort(RowSource2,JoinColumn);  
Init(Tmp1); Init(Tmp2);  
While Sync(Tmp1,Tmp2,JoinColumn)  
Loop return(CurRec(Tmp1)+CurRec(Tmp2));  
End Loop;
```

Sync továbbviszi a mutató(ka)t a következő egyezésre

Összekapcsolás (3.3)

```
SELECT *  
FROM emp, dept  
WHERE emp.d# = dept.d#;
```

Emp(d#)

```
>.SELECT STATEMENT  
>...NESTED LOOPS  
>....TABLE ACCESS full dept  
>....TABLE ACCESS by rowid emp  
>.....INDEX range scan e_emp_fk
```

- Belső összekapcsolás, csak az egyik oldal indexelt
 - NLJ a nem indexelt tábla teljes beolvasásával kezd
 - minden kinyert sornál az indexben keresünk egyező sorokat
 - A 2. ciklusban a d# (jelenlegi) értéke elérhető!
 - És felhasználható intervallumkeresésre

Összekapcsolások (3.4)

```
SELECT *
FROM emp, dept
WHERE emp.d# = dept.d#
```

Emp(d#)

Unique Dept(d#)

```
>.SELECT STATEMENT
>...NESTED LOOPS
>....TABLE ACCESS full dept
>....TABLE ACCESS by rowid emp
>.....INDEX range scan e_emp_fk
Or,
>.SELECT STATEMENT
>...NESTED LOOPS
>....TABLE ACCESS full emp
>....TABLE ACCESS by rowid dept
>.....INDEX unique scan e_dept_pk
```

- Belső összekapcsolás, minden oldal indexelt
 - RBO: NLJ, először a FROM utolsó tábláján FTS
 - CBO: NLJ, először a FROM legnagyobb tábláján FTS
 - A legnagyobb I/O nyereség FTS-nél
 - Általában kisebb tábla lesz a puffer gyorsítótárban

Összekapcsolások (3.5)

```
SELECT *  
FROM emp, dept  
WHERE emp.d# = dept.d#  
AND dept.loc = 'DALLAS'
```

Emp(d#)

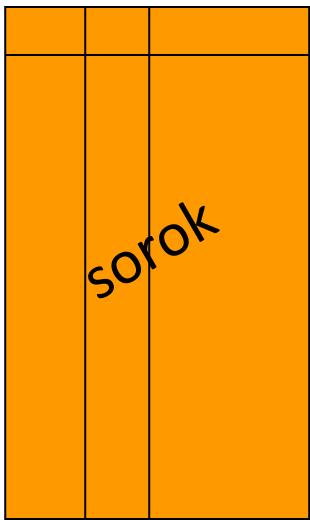
Unique Dept(d#)

```
>.SELECT STATEMENT  
>...NESTED LOOPS  
>....TABLE ACCESS full dept  
>....TABLE ACCESS by rowid emp  
>.....INDEX range scan e_emp_fk
```

- Belső összekapcsolás plusz feltételekkel
 - Skatulyázott ciklusok
 - Mindig azzal a táblával kezdjük, amelyiken plusz feltétel van

Hasítás

Tábla



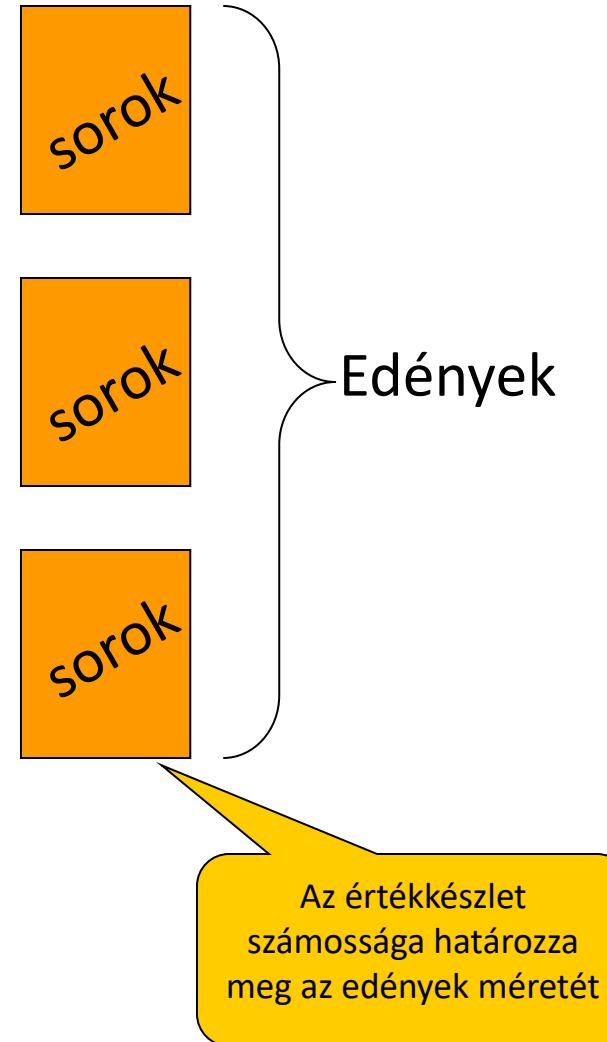
Egyenlőségi keresés
where-ben

Hasítófüggvény
pl. $\text{Mod}(cv,3)$

Tartomány = mezőértékek (cv)

Értékkészlet = hasítás értéke (eltolás)

```
SELECT *  
FROM table  
WHERE column = <érték>
```



Az értékkészlet
számosága határozza
meg az edények méretét

Összekapcsolások, Hasítás (3.6)

```
SELECT *  
FROM dept, emp  
WHERE dept.d# = emp.d#
```

Emp(d#), Unique Dept(d#)

```
>.SELECT STATEMENT  
>...HASH JOIN  
>....TABLE ACCESS full dept  
>....TABLE ACCESS full emp
```

- Tmp1 := Hash(RowSource1,JoinColumn); -- memóriában
Init(RowSource2);
While not eof(RowSource2)
Loop HashInit(Tmp1,JoinValue); -- edény megtalálása
While not eof(Tmp1)
Loop return(CurRec(RowSource2)+CurRec(Tmp1));
NxtHashRec(Tmp1,JoinValue);
End Loop; NxtRec(RowSource2);
End Loop;

Összekapcsolások, Hasítás (3.6)

- Explicit engedélyezni kell az init.ora fájlban:
 - Hash_Join_Enabled = True
 - Hash_Area_Size = <bytes>
- Ha a hasított tábla nem fér bele a memóriába
 - 1. sorforrás: átmeneti hasító cluster keletkezik
 - És kiíródik a lemezre (I/O) partícióinként
 - 2. sorforrás szintén konvertálódik ugyanazzal a hasítófüggvénytel
 - Edényenként a sorok összehasonlításra kerülnek
 - Egy edénynek bele kell férnie a memóriába, különben rossz teljesítmény

Allekérdezés (4.1)

```
SELECT dname, deptno  
FROM dept  
WHERE d# IN  
(SELECT d#  
  FROM emp);
```

```
>.SELECT STATEMENT  
>...NESTED LOOPS  
>....VIEW  
>.....SORT unique  
>.....TABLE ACCESS full emp  
>.....TABLE ACCESS by rowid dept  
>.....INDEX unique scan i_dept_pk
```

- Átalakítás összekapcsolássá
 - Átmeneti nézet keletkezik, amely hajtja a skatulyázott ciklust

Allekérdezés, korrelált(4.2)

```
SELECT *  
FROM emp e  
WHERE sal >  
  (SELECT sal  
   FROM emp m  
   WHERE m.e#=e.mgr#)
```

```
>.SELECT STATEMENT  
>...FILTER  
>....TABLE ACCESS full emp  
>....TABLE ACCESS by rowid emp  
>.....INDEX unique scan i_emp_pk
```

- Skatulyázott ciklus-szerű FILTER
 - Az 1. sorforrás minden sorára végrehajtja a 2. sorforrást és szűri az allekérdezés feltételére
 - Az allekérdezés átírható az EMP tábla ön-összekapcsolásává

Allekérdezés, korrelált (4.2)

```
SELECT *  
FROM emp e, emp m  
WHERE m.e#=e.mgr#  
AND e.sal > m.sal;
```

```
>.SELECT STATEMENT  
>...NESTED LOOPS  
>....TABLE ACCESS full emp  
>....TABLE ACCESS by rowid emp  
>.....INDEX unique scan i_emp_pk
```

- Allekérdezés átírása összekapcsolássá
 - Az allekérdezés átírható EXISTS-allekérdezéssé is

Allekérdezés, korrelált(4.2)

```
SELECT *  
FROM emp e  
WHERE exists  
(SELECT 'less salary'  
FROM emp m  
WHERE e.mgr# = m.e#  
    and m.sal < e.sal);
```

```
>.SELECT STATEMENT  
>...FILTER  
>....TABLE ACCESS full emp  
>....TABLE ACCESS by rowid emp  
>.....INDEX unique scan i_emp_pk
```

- Allekérdezés átírása EXISTS allekérdezéssé
 - Az 1. sorforrás minden sorára végrehajtja a 2. sorforrást és szűri a 2. sorforrás kinyerését

Összefűzés (4.3)

```
SELECT *
FROM emp
WHERE mgr# = 100
OR job = 'CLERK';
```

Emp(mgr#)
Emp(job)

```
>.SELECT STATEMENT
>...CONCATENATION
>....TABLE ACCESS by rowid emp
>.....INDEX range scan i_emp_m
>....TABLE ACCESS by rowid emp
>.....INDEX range scan i_emp_j
```

- Összefűzés (VAGY-feldolgoás)
 - Hasonló, mint amikor átírjuk 2 külön lekérdezésre
 - Amelyeket azután összefűzünk
 - Ha hiányzik az egyik index => teljes táblabeolvasás

Bel-lista iterátor (4.4)

```
SELECT *  
FROM dept  
WHERE d# in (10,20,30);
```

Unique Dept(d#)

```
>.SELECT STATEMENT  
>...INLIST ITERATOR  
>....TABLE ACCESS by rowid dept  
>.....INDEX unique scan i_dept_pk
```

- Iteráció felsorolt értékklistán
 - minden értékre külön végrehajtja
- Ugyanaz, mint 3 VAGY-olt érték összefűzése

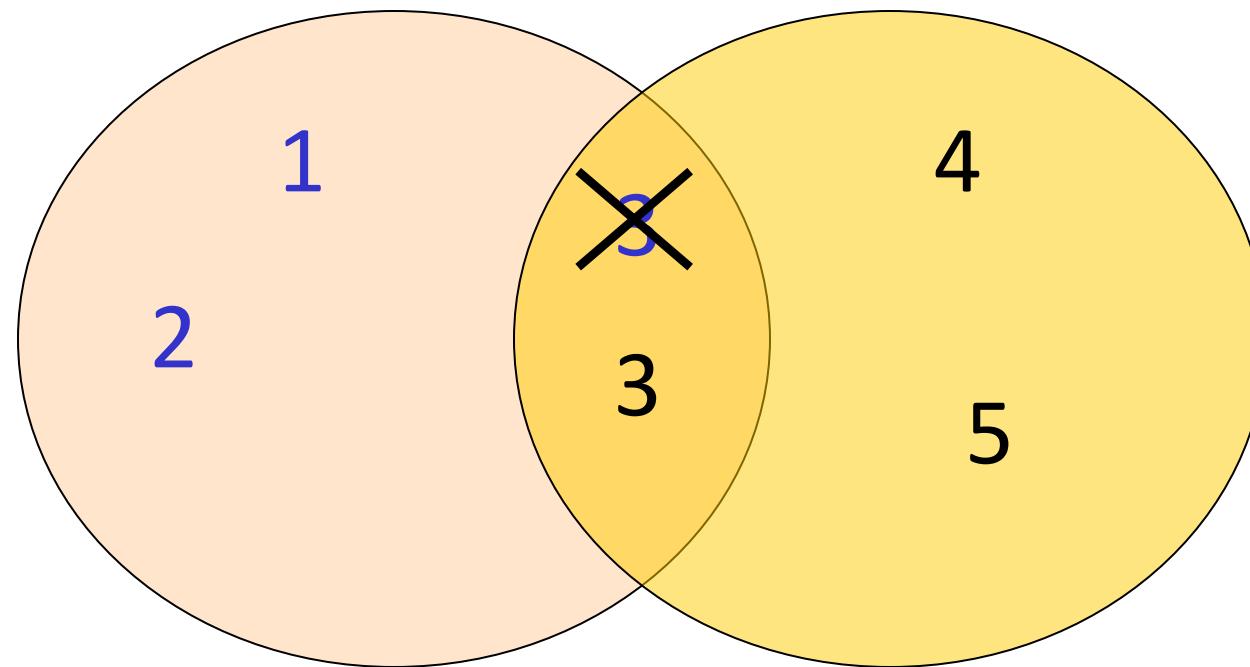
Unió (4.5)

```
SELECT empno  
FROM emp  
UNION  
SELECT deptno  
FROM dept;
```

```
>.SELECT STATEMENT  
>...SORT unique  
>....UNION  
>.....TABLE ACCESS full emp  
>.....TABLE ACCESS full dept
```

- Unió, majd egyedi rendezés
 - Az al-sorforrások külön kerülnek optimalizálásra/végrehajtásra
 - A kinyert sorokat összefűzzük
 - A halmazelmélet miatt az elemeknek egyedinek kell lenniük (rendezés)

UNION



Minden-unió (4.6)

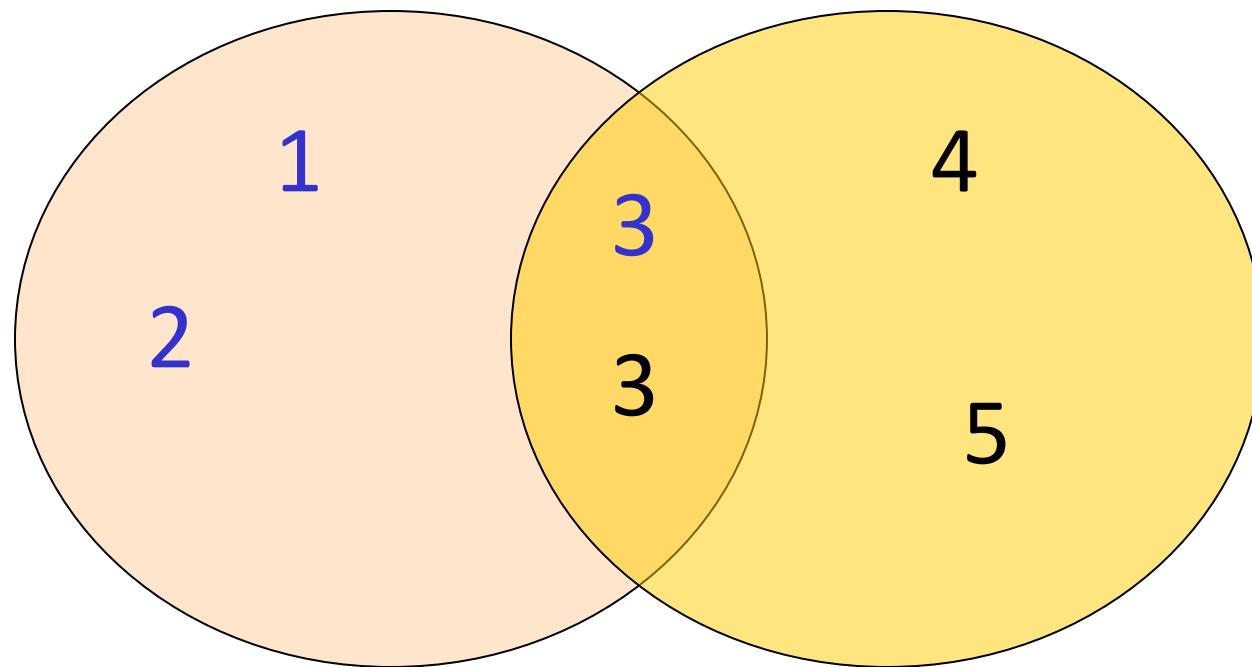
```
SELECT empno  
FROM emp  
UNION ALL  
SELECT deptno  
FROM dept;
```

```
>.SELECT STATEMENT  
>...UNION-ALL  
>....TABLE ACCESS full emp  
>....TABLE ACCESS full dept
```

- minden-unió: az eredmény zsák, nem halmaz
 - (Drága) rendezésre nincs szükség

*Használunk UNION ALL-t, ha tudjuk, hogy a zsák halmaz
(megspórolunk egy drága rendezést)*

UNION ALL



Metszet (4.7)

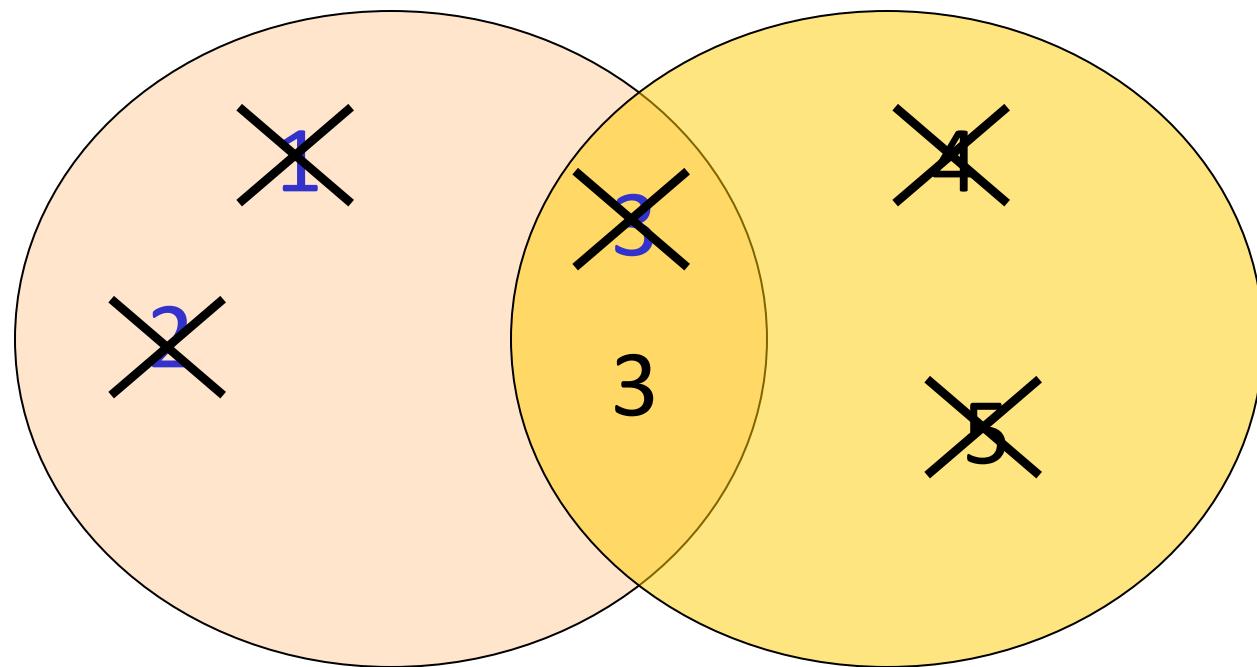
```
SELECT empno  
FROM emp  
INTERSECT  
SELECT deptno  
FROM dept;
```

```
>.SELECT STATEMENT  
>...INTERSECTION  
>....SORT unique  
>.....TABLE ACCESS full emp  
>....SORT unique  
>.....TABLE ACCESS full dept
```

- INTERSECT

- Az al-sorforrások külön kerülnek optimalizálásra/végrehajtásra
- Nagyon hasonlít az összefésüléses rendezéshez
- A teljes sorokat rendezi és összehasonlítja

INTERSECT



Különbség (4.8)

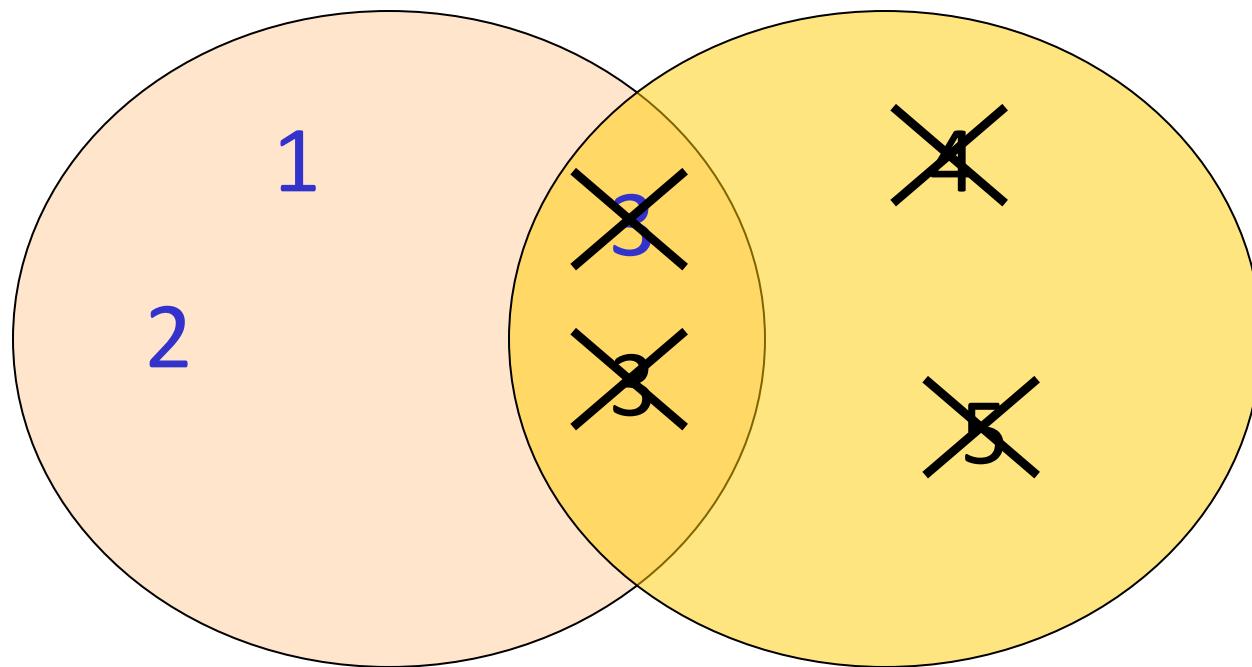
```
SELECT empno  
FROM emp  
MINUS  
SELECT deptno  
FROM dept;
```

```
>.SELECT STATEMENT  
>...MINUS  
>....SORT unique  
>.....TABLE ACCESS full emp  
>....SORT unique  
>.....TABLE ACCESS full dept
```

- MINUS

- Az al-sorforrások külön kerülnek optimalizálásra/végrehajtásra
- Hasonlít a metszet feldolgozására
 - Összehasonlítás és visszaadás helyett összehasonlítás és kizárás

MINUS



Eszközök

- Nyomkövetés
- SQL tippek
- Analizáló parancs
- Dbms_Stats csomag

Nyomkövető fájlok

- Tervmagyarázat: beletekintés végrehajtás előtt
- Nyomkövetés: beletekintés végrehajtás közben
 - Felhasznált CPU idő
 - Eltelt idő
 - Fizikai blokk I/O száma
 - Gyorsítótárazott blokk I/O száma
 - Sorforrásonként feldolgozott sorok száma
- A munkamenetet nyomkövető módba kell állítani
 - Alter session set sql_trace=true;
 - Exec dbms_system.set_sql_trace_in_session(sid,s#,T/F);

Nyomkövető fájlok

- A nyomkövető fájl az adatbázisszerveren generálódik

- TKPROF eszközzel kell formázni

```
tkprof <nyomkövető fájl> <tkp-fájl> <user>/<pw>
```

- SQL utasításonként 2 szakasz:

| call | count | cpu | elapsed | disk | query | current | rows |
|----------------|-------|------|---------|------|-------|---------|------|
| Parse | 1 | 0.06 | 0.07 | 0 | 0 | 0 | 0 |
| Execute | 1 | 0.01 | 0.01 | 0 | 0 | 0 | 0 |
| Fetch | 1 | 0.11 | 0.13 | 0 | 37 | 2 | 2 |
| total | 3 | 0.18 | 0.21 | 0 | 37 | 2 | 2 |

Nyomkövető fájlok

- 2. szakasz: bővített végrehajtási terv
 - Példa 4.2 (dolgozó fizetése nagyobb, mint a menedzseré),

```
#R Plan
2 SELECT STATEMENT
14 FILTER
14 TABLE ACCESS (FULL) OF 'EMP'
11 TABLE ACCESS (BY ROWID) OF 'EMP'
12 INDEX (UNIQUE SCAN) OF 'I_EMP_PK' (UNIQUE)
```

- Emp tartalmaz 14 rekordot
- Kettőben nincs menedzser (NULL mgr mezőérték)
- Az egyik nem létező alkalmazottra mutat
- Ketten többet keresnek, mint a menedzserük

Tippek

- Kényszerítik az optimalizálót egy konkrét lehetőség kiválasztására
 - Beágynazott megjegyzéssel valósítjuk meg

```
SELECT /*+ <tipp> */ ....
```

```
FROM ....
```

```
WHERE ....
```

```
UPDATE /*+ <tipp> */ ....
```

```
WHERE ....
```

```
DELETE /*+ <tipp> */ ....
```

```
WHERE ....
```

```
INSERT (Id. SELECT)
```

Tippek

- Gyakori tippek
 - Full(<tab>)
 - Index(<tab> <ind>)
 - Index_asc(<tab> <ind>)
 - Index_desc(<tab> <ind>)
 - Ordered
 - Use_NL(<tab> <tab>)
 - Use_Merge(<tab> <tab>)
 - Use_Hash(<tab> <tab>)
 - Leading(<tab>)
 - First_rows, All_rows, Rule

Analizáló parancs

- A statisztikát időnként generálni kell
 - Az ‘ANALYZE’ parancssal tehető meg

```
Analyze <Table | Index> <x>
<compute | estimate | delete> statistics
    <sample <x> <Rows | Percent>>
```

```
Analyze table emp estimate statistics sample 30 percent;
```

Az ANALYZE támogatása megszűnik

Dbms_Stats csomag

- Az analizáló parancs utódja
 - Dbms_stats.gather_index_stats(<owner>,<index>,<blocksample>,<est.percent>)
 - Dbms_stats.gather_table_stats(<owner>,<table>,<blocksample>,<est.percent>)
 - Dbms_stats.delete_index_stats(<owner>,<index>)
 - Dbms_stats.delete_table_stats(<owner>,<table>)

```
SQL>exec dbms_stats.gather_table_status('scott','emp',null,30);
```

Adattárház jellemzők

- Hagyományos csillag lekérdezés
- Bittérkép indexek
 - Bittérkép egyesítése, átalakítása rowid-dé
 - Egyetlen táblás lekérdezés
- Csillag lekérdezés
 - Több táblás

Hagyományos csillag lekérdezés

```
SELECT f.*  
FROM a,b,f  
WHERE a.pk = f.a_fk  
AND b.pk = f.b_fk  
AND a.t = ... AND b.s = ...
```

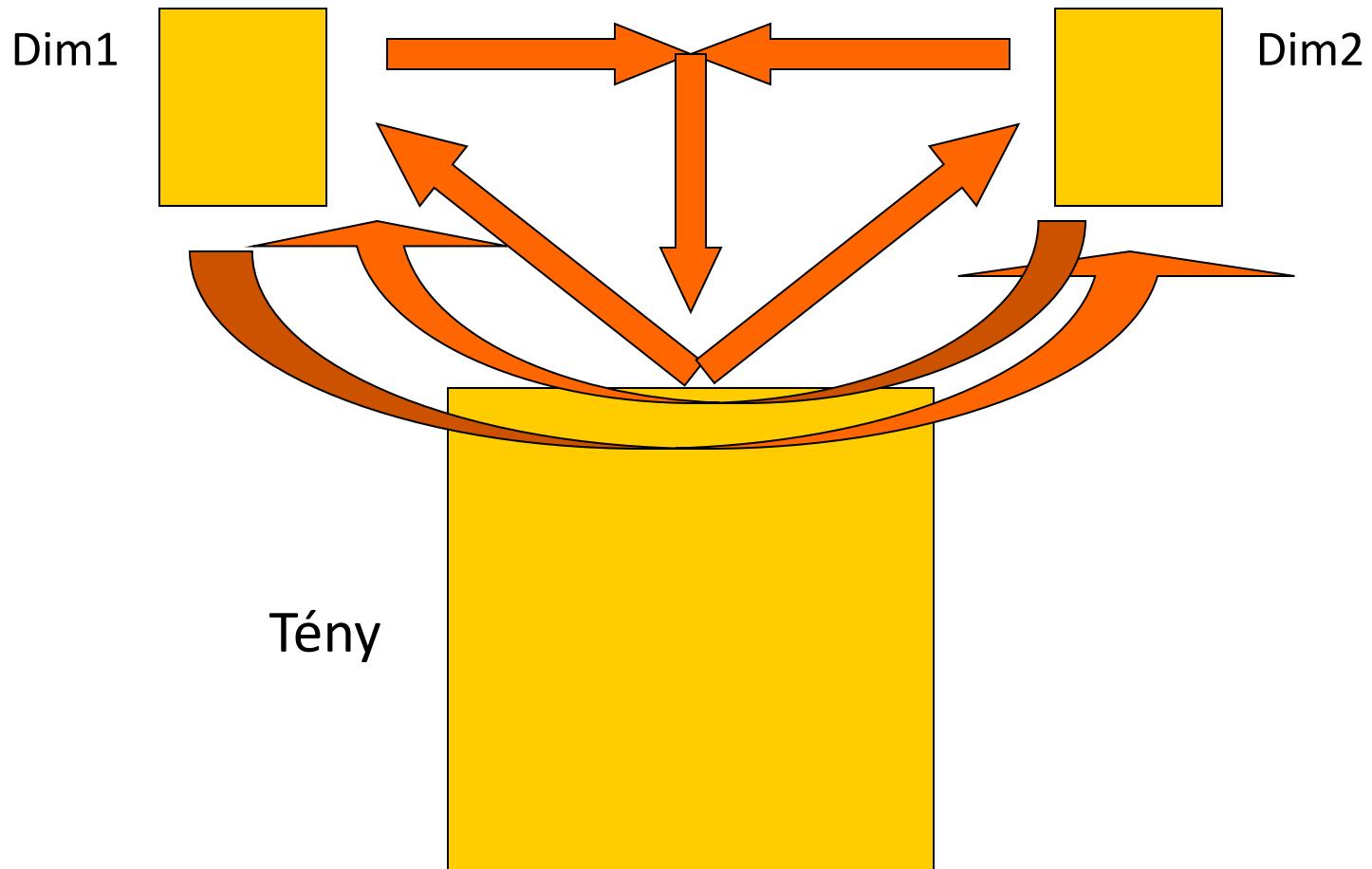
A(pk), B(pk)
F(a_fk), F(b_fk)

```
>.SELECT STATEMENT  
>...NESTED LOOPS  
>....NESTED LOOPS  
>.....TABLE ACCESS full b  
>.....TABLE ACCESS by rowid fact  
>.....INDEX range scan i_fact_b  
>.....TABLE ACCESS by rowid a  
>.....INDEX unique scan a_pk
```

- Dupla skatulyázott ciklus
 - Válasszunk kezdp táblát (A vagy B)
 - Aztán kövessük az összekapcsolási feltételeket skatulyázott ciklusokkal

Túl bonyolult az ÉS-EGYENLŐ-höz

Hagyományos csillag lekérdezés



Négy lehetséges elérési sorrend!

Hagyományos csillag lekérdezés

```
SELECT f.*  
FROM a,b,f  
WHERE a.pk = f.a_fk  
AND b.pk = f.b_fk  
AND a.t = ... AND b.s = ...
```

F(a_fk,b_fk,...)

```
>.SELECT STATEMENT  
>...NESTED LOOPS  
>....MERGE JOIN cartesian  
>.....TABLE ACCESS full a  
>.....SORT join  
>.....TABLE ACCESS full b  
>.....TABLE ACCESS by rowid fact  
>.....INDEX range scan I_f_abc
```

- Összefűzött index intervallumkeresés csillag lekérdezéshez
 - Legalább két dimenzió
 - Legalább eggyel több indexelt mező, mint dimenzió
 - Összevonás-Összekapcsolás-Direkt szorzat adja az összes lehetséges dimenziókombinációt
 - minden kombinációhoz keresünk az összefűzött indexben

Bittérkép index

| Empno | Status | Region | Gender | Info |
|--------------|---------------|---------------|---------------|-------------|
| 101 | single | east | male | bracket_1 |
| 102 | married | central | female | bracket_4 |
| 103 | married | west | female | bracket_2 |
| 104 | divorced | west | male | bracket_4 |
| 105 | single | central | female | bracket_2 |
| 106 | married | central | female | bracket_3 |

| REGION='east' | REGION='central' | REGION='west' |
|----------------------|-------------------------|----------------------|
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 0 |

Bittérkép index

```
SELECT COUNT(*)  
FROM CUSTOMER  
WHERE MARITAL_STATUS = 'married'  
AND REGION IN ('central','west');
```

| status = 'married' | region = 'central' | region = 'west' | | | |
|-----------------------|-----------------------|--------------------|---|------------|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | AND | OR | = | AND | = |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 |

Bittérkép elérés, egyetlen tábla

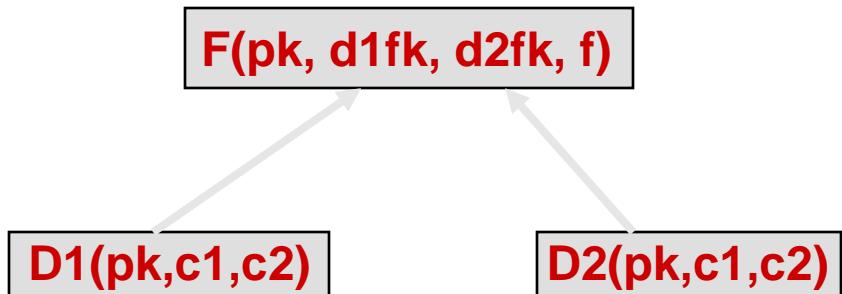
```
SELECT count(*)  
FROM customer  
WHERE status='M'  
AND region in ('C','W');
```

```
>.....TABLE ACCESS (BY INDEX ROWID) cust  
>.....BITMAP CONVERSION to rowids  
>.....BITMAP AND  
>.....BITMAP INDEX single érték cs  
>.....BITMAP MERGE  
>.....BITMAP KEY ITERATION  
>.....BITMAP INDEX range scan cr
```

- Bittérkép ÉS, VAGY és ÁTALAKÍTÁS
 - 'C' és 'W' bitsorozatok megkeresése (bittérképkulcs-iteráció)
 - Logikai VAGY végrehajtása (bittérkép összevonás)
 - Az 'M' bitsorozat megkeresése
 - Logikai ÉS a régió bitsorozattal (bittérkép és)
 - Átalakítás rowid-kké
 - Táblaelrés

Bittérkép elérés, csillag lekérdezés

Bittérkép indexek: id1, id2



```
SELECT sum(f)  
FROM F,D1,D2  
WHERE F=D1 and F=D2  
AND D1.C1=<...>  
AND D2.C2=<...>
```

```
>.....TABLE ACCESS (BY INDEX ROWID) f  
>.....BITMAP CONVERSION (TO ROWIDS)  
>.....BITMAP AND  
>.....BITMAP MERGE  
>.....BITMAP KEY ITERATION  
>.....TABLE ACCESS (FULL) d1  
>.....BITMAP INDEX (RANGE SCAN) id1  
>.....BITMAP MERGE  
>.....BITMAP KEY ITERATION  
>.....TABLE ACCESS (FULL) d2  
>.....BITMAP INDEX (RANGE SCAN) id2
```

Adattárház tippek

- Csillag lekérdezésre jellemző tippek
 - Star
 - Hagyományos: összevonásos index intervallumkeresés
 - Star_transformation
 - Egymezős bittérkép index összevonás/ÉS-ek
 - Fact(t) / No_fact(t)
 - Segíti a star_transformation-t
 - Index_combine(t i1 i2 ...)
 - Explicit megadja, mely indexeket vonja össze/ÉS-elje

ETL lehetőségek

- Új a 9i-ben
 - Külső táblák
 - Külső ASCII fájl elérése SQL-ből (csak FTS)
 - Összevonás (aka UpSert)
 - Feltételes beszúrás vagy frissítés végrehajtása
 - Többtáblás beszúrás (Multi-Table Insert, MTI)
 - Feltételesen beszúrja az allekérdezések eredményét több táblába

Elérhetőség

- Oracle7
 - Költségalapú optimalizáció
 - Hasításos összekapcsolás
- Oracle r8.0
 - Bittérkép indexek (hibamentesen)
 - Star_transformation
 - Rowid formátum (dbms_rowid)
- Oracle 8i
 - Dbms_Stats
- Oracle9i
 - Index SkipScans
 - First_rows(n)-tipp

Egy bevezetés...

- Nem fedtük le:
 - Elosztott SQL
 - Skatulyázott SQL
 - PL/SQL függvények SQL-en belül
 - Ellen-összekapcsolások
 - Nézetek feldolgozása
 - Index+hasító clusterek
 - Partícionálás / Párhuzamosítás
 - Index szervezett táblák
 - ...

SQL Tuning: Útirány

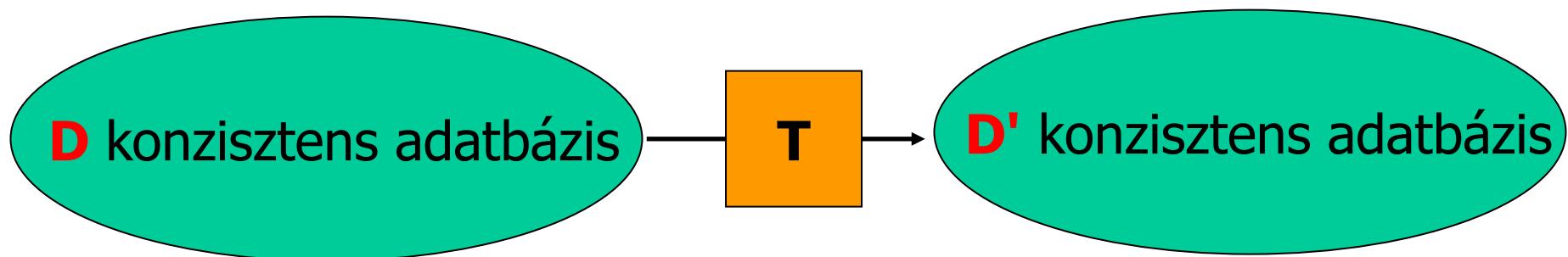
- Képes beolvasni tervet
- Képes átírni a tervet 3GL programmá
 - Ismerjük a sorforrás műveleteinket
- Képes beolvasni SQL-t
- Képes átalakítani az SQL-t üzleti lekérdezéssé
 - Ismerjük az adatmodellünket
- Képes megítélni a kimenetelt
 - Ismerjük az üzleti szabályokat / adatstatisztikákat
 - Jobban, mint a CBO
- Szakértők:
 - Optimalizáljuk az SQL-t az SQL írása közben...

Tranzakciókezelés alapfogalmai



TRANZAKCIÓ:

Konzisztenciát megtartó adatkezelő műveletek sorozata



Ezek után minden feltehetően:

Ha T tranzakció konzisztens állapotból indul

+ T tranzakció csak egyedül futna le

⇒ T konzisztens állapotban hagyja az adatbázis



Konzisztencia, megszorítások

- **Mit jelent a konzisztencia?**
- **Az adatok előre megadott feltételeket, predikátumokat elégítenek ki.**
- **Például:**
 - X az R reláció kulcsa
 - $X \rightarrow Y$ függőség teljesül R-ben
 - megengedett értékek korlátozása:
 - $\text{Domain}(X) = \{\text{piros}, \text{kék}, \text{zöld}\}$
 - α indexfájl az R reláció X attribútumának érvényes indexe
 - Semelyik dolgozó nem keres többet, mint az átlagfizetés kétszerese



Definíció:

- **Konzisztens állapot:**
kielégíti az összes feltételt (megszorítást)
- **Konzisztens adatbázis:**
konzisztens állapotú adatbázis



Általánosabb megszorítások

Tranzakciós megszorítások

- **Ha módosítjuk a fizetést, akkor az új fizetés > régi fizetés**

(Egy állapotból nem lehet ellenőrizni, mivel ez a változtatás módjára ad meg feltételt!)

- **A számla rekord törlése után legyen az egyenleg = 0.**

(Ezt sem lehet egy állapotra ellenőrizni, mivel vagy törlés miatt lett az egyenleg 0, vagy eleve 0 értéket tároltunk.)



Megjegyzés: az utóbbi szimulálható közönséges megszorítással, ha felveszünk egy törölve oszlopot.

Ha törölve=igen, akkor egyenleg=0.

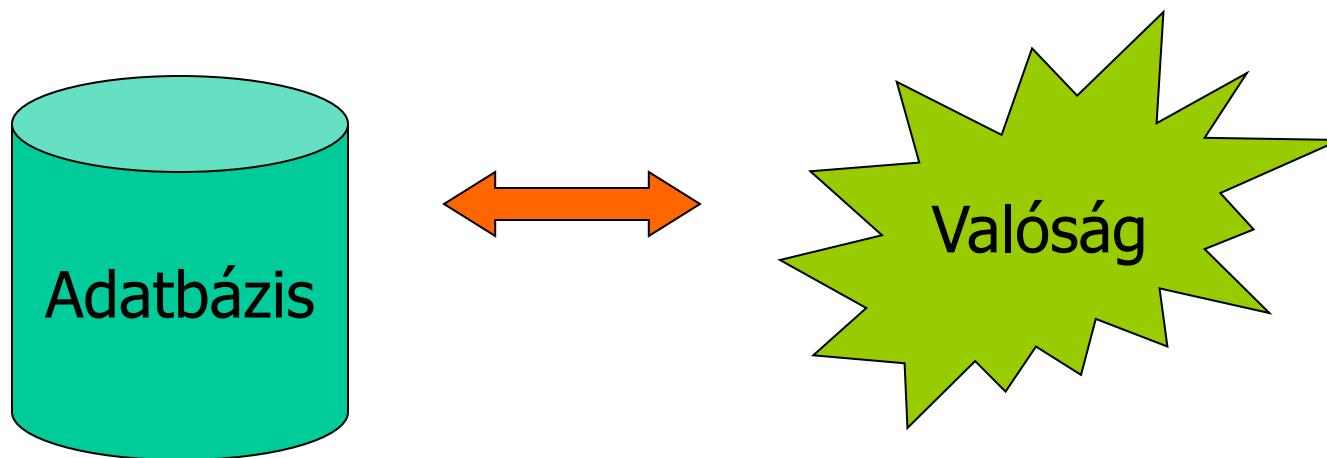
számla

| | | | |
|--------|------|----------|---------|
| AZON # | | egyenleg | törölve |
|--------|------|----------|---------|



A megszorítások hiányossága

Az adatbázis a valóságot próbálja reprezentálni, de minden összefüggést (vagyis a valóság teljes szemantikáját) lehetetlen megadni.



Vegyük észre: Az adatbázis **nem lehet állandóan** konzisztens!

Például:

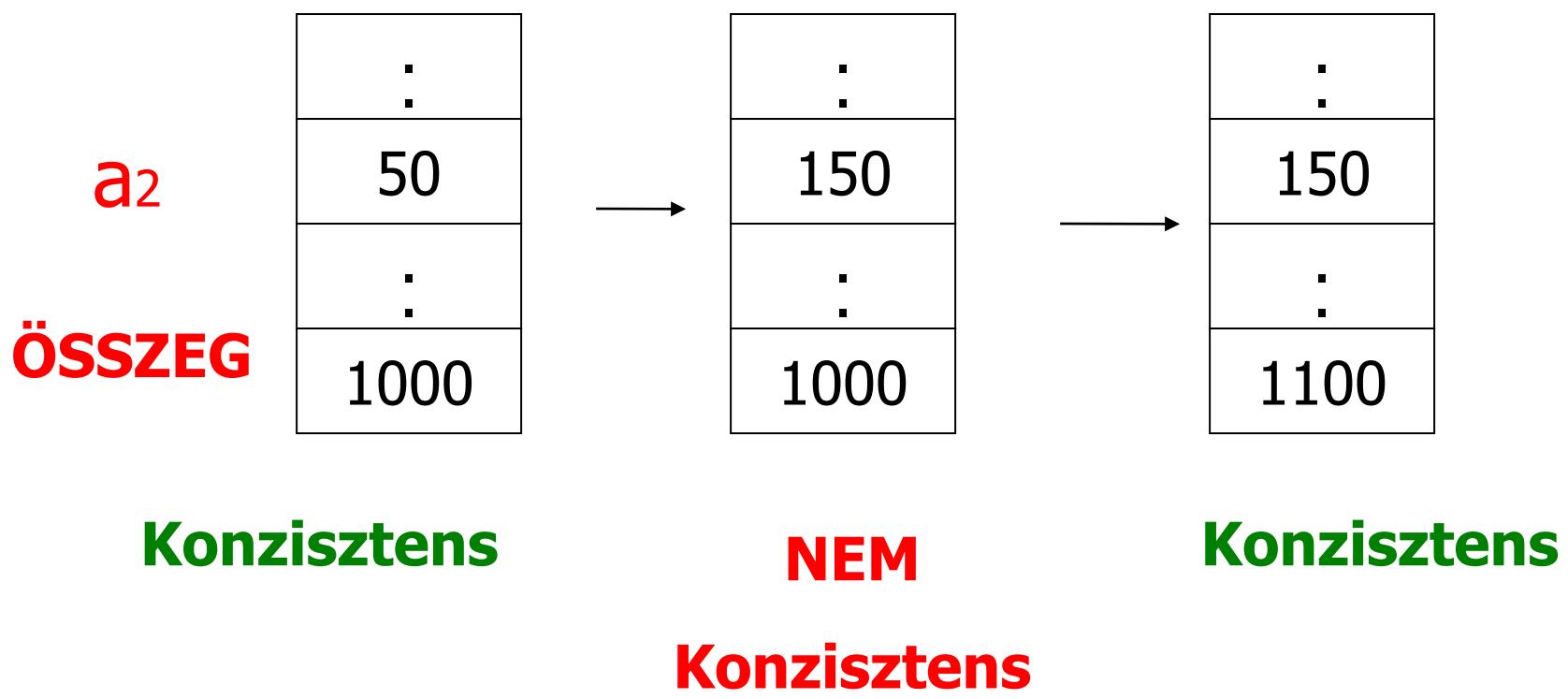
a₁ + a₂ +.... a_n = ÖSSZEG (megszorítás)

Adjunk 100-at az a₂-höz:

{ a₂ ← a₂ + 100
 összeg ← összeg + 100



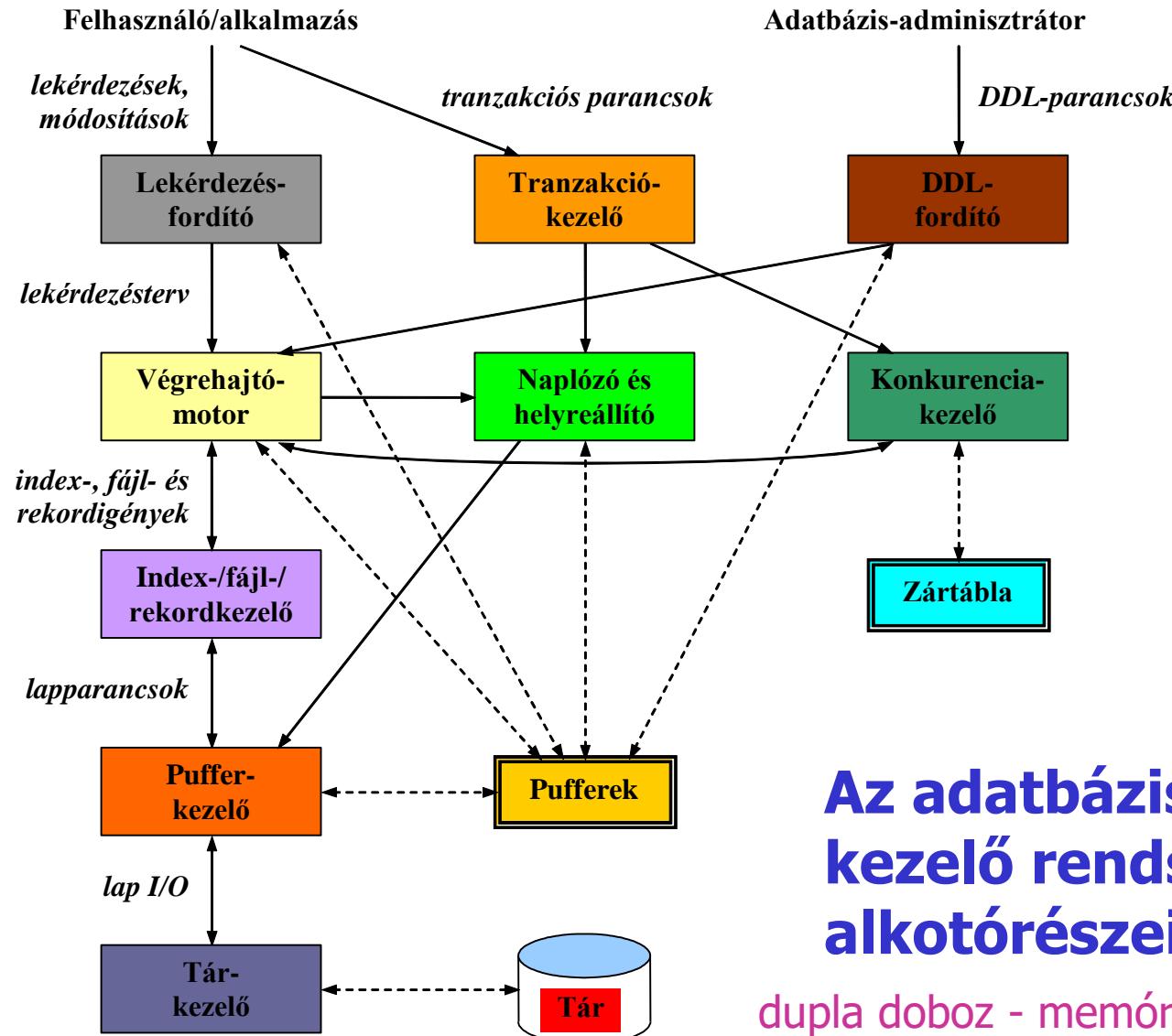
A két lépést nem tudjuk egyszerre végrehajtani, így egy pillanatra megsérül a konzisztencia.



Helyesség feltétele

1. Ha **leáll valamelyik, vagy több tranzakció (abort, vagy hiba miatt), akkor is konzisztens D adatbázist kell előállítanunk**
2. Mind egyes tranzakció induláskor **konzisztens D adatbázist lát.**





Az adatbázis-kezelő rendszer alkotórészei

dupla doboz - memória

folytonos vonal - vezérlésátadás, adatáramlással

szaggatott vonal - csak adatmozgás



A lekérdezés megválaszolása

- A lekérdezésfordító elemzi és optimalizálja a lekérdezést.
- Az eredményül kapott lekérdezés-végrehajtási tervet (a megválaszolásához szükséges tevékenységek sorozatát) továbbítja a végrehajtómotornak.
- A végrehajtómotor kisebb adatdarabokra (tipikusan rekordokra) vonatkozó kérések sorozatát adja át az erőforrás-kezelőnek.
- Az erőforrás-kezelő ismeri a relációkat tartalmazó adatfájlokat, a fájlok rekordjainak formátumát, méretét és az indexfájlokat is. Az adatkéréseket az erőforrás-kezelő lefordítja lapokra (blokkokra), és ezeket a kéréseket továbbítja a pufferkezelőnek.
- A pufferkezelő feladata, hogy a másodlagos adattárolón (általában lemezen) tárolt adatok megfelelő részét hozza be a központi memória puffereibe. A pufferek és a lemez közti adatátvitel egysége általában egy lap vagy egy lemezblokk. A pufferkezelő információt cserél a tárkezelővel, hogy megkapja az adatokat a lemezről. Megtörténhet, hogy a tárkezelő az operációs rendszer parancsait is igénybe veszi, de tipikusabb, hogy az adatbázis-kezelő a parancsait közvetlenül a lemezvezérlőhöz intézi.



A tranzakció feldolgozása.

- A lekérdezéseket és más tevékenységeket tranzakciókba csoportosíthatjuk.
- A tranzakciók olyan munkaegységek, amelyeket atomosan és más tranzakcióktól látszólag elkülönítve kell végrehajtani. (Gyakran minden egyes lekérdezés vagy módosítás önmagában is egy tranzakció.)
- A tranzakció végrehajtásának **tartósnak** kell lennie, ami azt jelenti, hogy bármelyik befejezett tranzakció hatását még akkor is meg kell tudni őrizni, ha a rendszer összeomlik a tranzakció befejezése utáni pillanatban.
- A tranzakciófeldolgozót két fő részre osztjuk:
 - **Konkurenciavezérlés-kezelő** vagy **ütemező** (scheduler): a tranzakciók elkülönítésének és atomosságának biztosításáért felelős.
 - **Naplázás- és helyreállítás-kezelő**: a tranzakciók atomosságáért és tartósságáért felelős.



A tranzakció

- A tranzakció az adatbázis-műveletek végrehajtási egysége, amely DML-beli utasításokból áll, és a következő tulajdonságokkal rendelkezik:
 - A Atomosság (atomicity):** a tranzakció „minden vagy semmit” jellegű végrehajtása (vagy teljesen végrehajtjuk, vagy egyáltalán nem hajtjuk végre).
 - C Konzisztencia (consistency):** az a feltétel, hogy a tranzakció megőrizze az adatbázis konzisztenciáját, azaz a tranzakció végrehajtása után is teljesüljenek az adatbázisban előírt konzisztenciamegszorítások (integritási megszorítások), azaz az adatalemekre és a közöttük lévő kapcsolatokra vonatkozó elvárások.
 - I Elkülönítés (isolation):** az a tény, hogy minden tranzakciónak látszólag úgy kell lefutnia, mintha ez alatt az idő alatt semmilyen másik tranzakciót sem hajtanánk végre.
 - D Tartósság (durability):** az a feltétel, hogy ha egyszer egy tranzakció befejeződött, akkor már soha többé nem veszhet el a tranzakciónak az adatbázison kifejtett hatása.
- Ezek a tranzakció **ACID-tulajdonságai**.



A tranzakció

Megjegyzések:

- **A konzisztenciát minden adottnak tekintjük.**
- **A másik három tulajdonságot viszont az adatbázis-kezelő rendszernek kell biztosítania, de ettől időnként eltekintünk.**
- **Ha egy ad hoc utasítást adunk az SQL-rendszernek, akkor minden lekérdezés vagy adatbázis-módosító utasítás egy tranzakció.**
- **Amennyiben beágyazott SQL-interfészt használva a programozó készíti el a tranzakciót, akkor egy tranzakcióban több SQL-lekérdezés és -módosítás szerepelhet. A tranzakció ilyenkor általában egy DML-utasítással kezdődik, és egy COMMIT vagy ROLLBACK parancssal végződik. Ha a tranzakció valamely utasítása egy triggerset aktivizál, akkor a trigger is a tranzakció részének tekintendő, akár csak a trigger által kiváltott további triggers. (A trigger olyan programrész, amely bizonyos események bekövetkeztekor automatikusan lefut.)**



A tranzakció feldolgozása

A tranzakciófeldolgozó a következő 3 feladatot hajtja végre:

- **naplázás**
- **konkurenciavezérlés**
- **holtpont feloldása**

1. *Naplázás:*

- Annak érdekében, hogy a tartósságot biztosítani lehessen, az adatbázis minden változását külön feljegyezzük (naplózzuk) lemezen.
- A naplókezelő (log manager) többféle eljárásmód közül választja ki azt, amelyiket követni fog.
- Ezek az eljárásmódok biztosítják azt, hogy teljesen mindegy, mikor történik a rendszerhiba vagy a rendszer összeomlása, a helyreállítás-kezelő meg fogja tudni vizsgálni a változások naplóját, és ez alapján vissza tudja állítani az adatbázist valamelyen konzisztens állapotába.
- A naplókezelő először a pufferekbe írja a naplót, és egyeztet a pufferkezelővel, hogy a pufferek alkalmas időpillanatokban garantáltan íródjanak ki lemezre, ahol már az adatok túlélhetik a rendszer összeomlását.



A tranzakció feldolgozása

2. Konkurenciavezérlés:

- A tranzakcióknak úgy kell látszódniuk, mintha egymástól függetlenül, elkülönítve végeznénk el őket.
- Az ütemező (konkurenciavezérlés-kezelő) feladata, hogy meghatározza az összetett tranzakciók résztervezékenységeinek egy olyan sorrendjét, amely biztosítja azt, hogy ha ebben a sorrendben hajtjuk végre a tranzakciók elemi tevékenységeit, akkor az összhatás megegyezik azzal, mintha a tranzakciókat tulajdonképpen egyenként és egységes egészben hajtottuk volna végre.

T1. tranzakció: $u1; u2; \dots; u10$

T2. tranzakció: $v1; v2; \dots; v103$

A két utasítássorozat nem elkülönülve jön, hanem összefésülődve:

$u1; v1; v2; u2; u3; v3; \dots; v103; u10$

A saját sorrend megmarad mindenbelül.

Ekkor olyan állapot is kialakulhat, ami nem jött volna létre, ha egymás után futnak le a tranzakciók.



A tranzakció feldolgozása

2. Konkurenciavezérlés:

T1. READ A, A++, WRITE A

T2. READ A, A++, WRITE A

Ha ezek úgy fésülődnek össze, hogy

(READ A)₁, (READ A)₂, (A++)₁, (A++)₂, (WRITE A)₁, (WRITE A)₂

akkor a végén csak egyel nő A értéke, holott kettővel kellett volna.

- A tipikus ütemező ezt a munkát azáltal látja el, hogy az adatbázis bizonyos részeire elhelyezett **zárákat** (lock) karbantartja.
- Ezek a zárák megakadályoznak két tranzakciót abban, hogy rossz kölcsönhatással használják ugyanazt az adatrészt. A zárákat rendszerint a központi memória **zárttáblájában** (lock table) tárolja a rendszer.
- Az ütemező azzal befolyásolja a lekérdezések és más adatbázis-műveletek végrehajtását, hogy megtiltja a végrehajtómotornak, hogy hozzájáruljon az adatbázis zár alá helyezett részeihez.



A tranzakció feldolgozása

3. Holpont feloldása:

- A tranzakciók az ütemező által engedélyezett zárak alapján versenyeznek az erőforrásokért. Igy előfordulhat, hogy olyan helyzetbe kerülnek, amelyben egyiküket sem lehet folytatni, mert mindegyiknek szüksége lenne valamire, amit egy másik tranzakció birtokol.
- A tranzakciókezelő feladata, hogy ilyenkor közbeavatkozzon, és töröljön (abortáljon) egy vagy több tranzakciót úgy, hogy a többit már folytatni lehessen.

Holpont (deadlock):

l1(A) jelölje, hogy T1 tranzakció zár alá helyezte az A-t, stb.

l1(A); l2(B); l3(C); l1(B); l2(C); l3(A)

sorrendben érkező zárkérések esetén egyik tranzakció se tud tovább futni.



Mitől sérülhet a konzisztencia?

- **Tranzakcióhiba** (hibásan megírt, rosszul ütemezett, félbehagyott tranzakciók.)
- **Adatbázis-kezelési hiba** (az adatbázis-kezelő valamelyik komponense nem, vagy rosszul hajtja végre a feladatát.)
- **Hardverhiba** (elvész egy adat, vagy megváltozik az értéke.)
- **Adatmegosztásból származó hiba**
például:

T1: 10% fizetésemelést ad minden programozónak
T2: minden programozót átnevez rendszerfejlesztőnek



- **Feladat. Tegyük fel, hogy az adatbázisra vonatkozó konzisztenciamegszorítás: $0 \leq A \leq B$. Állapítsuk meg, hogy a következő tranzakciók megőrzik-e az adatbázis konzisztenciáját!**

T1: A := A + B; B := A + B;

T2: B := A + B; A := A + B;

T3: A := B + 1; B := A + 1;



Hogy lehet megakadályozni vagy kijavítani a hibák okozta konziszenciasérülést?



Helyreállítás

- Első lépés: meghibásodási modell definiálása
- Események osztályozása:



Szabályos esemény: "a felhasználói kézikönyv alapján kezelhető esemény"

Előrelátható, kivételes esemény:

Rendszerösszeomlás:

- elszáll a memória
- leáll a cpu, újraindítás (reset)

Nem várt, kivételes esemény: MINDEN MÁS!



Előrelátható, kivételes események

A hibák fajtái

Az adatbázis lekérdezése vagy módosítása során számos doleg hibát okozhat a billentyűzeten történt adatbeviteli hibáktól kezdve az adatbázist tároló lemez elhelyezésére szolgáló helyiségben történő robbanásig.

- **Hibás adatbevitel**
- **Készülékhibák**
- **Katasztrófális hibák**
- **Rendszerhibák**



Előrelátható, kivételes események

• Hibás adatbevitel:

- **tartalmi hiba:** gyakran nem észrevehetők, például a felhasználó elüt egy számot egy telefonszámban.
- **formai hiba:** kihagy egy számjegyet a telefonszámból. SQL-ben típus előírások, kulcsok, megszorítások (**constraint**) definiálásával megakadályozható a hibás adatbevitel.
- **A triggerek azok a programok, amelyek bizonyos típusú módosítások (például egy relációba történő beszúrás) esetén hajtódnak végre, annak ellenőrzésére, hogy a frissen bevitt adatok megfelelnek-e az adatbázis-tervező által megszabott előírásoknak.**



Előrelátható, kivételes események

- **Készülékhibák:**

- **Kis hiba:** A lemezegységek olyan helyi hibái, melyek egy vagy több bit megváltozását okozzák, a lemez szektoraihoz rendelt paritás-ellenőrzéssel megbízhatóan felismerhetők.
- **Nagy hiba:** A lemezegységek jelentős sérülése, elsősorban az író-olvasó fejek katasztrófái, az egész lemez olvashatatlaná válását okozhatják. Az ilyen hibákat általában az alábbi megoldások segítségével kezelik:
 1. RAID
 2. Archiválás
 3. Osztott másolat



Előrelátható, kivételes események

1. A ***RAID-módszerek*** (Redundant Array of Independent Disks) valamelyikének használatával az elveszett lemez tartalma visszatölthető.
2. Az ***archiválás*** használatával az adatbázisról másolatot készítünk valamilyen eszközre (például szalagra vagy optikai lemezre). A mentést rendszeresen kell végezni vagy teljes, vagy növekményes (csak az előző mentés óta történt változásokat archiváljuk) mentést használva. A mentett anyagot az adatbázistól biztonságos távolságban kell tárolnunk.
3. Az adatbázisról fenntarthatunk ***elosztott, on-line másolatokat***. Ebben az esetben biztosítanunk kell a másolatok konziszenciáját.



Előrelátható, kivételes események

- **Katasztrófális hibák:**

- Ebbe a kategóriába soroljuk azokat a helyzeteket, amikor az adatbázist tartalmazó eszköz **teljesen tönkremegy** robbanás, tűz, vandalizmus vagy akár vírusok következtében.
- A RAID ekkor nem segít, mert az összes lemez és a paritás-ellenőrző lemezeik is egyszerre használhatatlanná válnak.
- A másik két biztonsági megoldás viszont alkalmazható katasztrófális hibák esetén is.



Előrelátható, kivételes események

• Rendszerhibák:

- minden tranzakciónak van *állapota*, mely azt képviseli, hogy mi történt eddig a tranzakcióban. Az állapot tartalmazza a tranzakció kódjában a végrehajtás pillanatnyi helyét és a tranzakció összes lokális változójának értékét.
- A rendszerhibák azok a problémák, melyek a tranzakció állapotának elvesztését okozzák.



Előrelátható, kivételes események

- Tipikus rendszerhibák az áramkimaradásból és a szoftverhibából eredők, hiszen ezek a memória tartalmának felülírásával járhatnak.
- Ha egy rendszerhiba bekövetkezik, onnantól kezdve nem tudjuk, hogy a tranzakció mely részei kerültek már végrehajtásra, beleértve az adatbázis-módosításokat is. A tranzakció ismételt futtatásával nem biztos, hogy a problémát korrigálni tudjuk (például egy mezőnek eggyel való növelése esetén).
- Az ilyen jellegű problémák legfontosabb ellenszere minden adatbázis-változtatás naplózása egy elkülönült, nem illékony naplófájlban, lehetővé téve ezzel a visszaállítást, ha az szükséges. Ehhez hibavédett naplózási mechanizmusra van szükség.



Összefoglalás

- Áttekintett fogalmak kulcsszavakban
- Tranzakció, konzisztencia, adatbáziskezelő tranzakciós moduljai, tranzakció ACID tulajdonságai, tranzakciók feldolgozása (naplózás, konkurenciakezelés, holtpontkezelés), konzisztencia sérülése, meghibásodási modellek

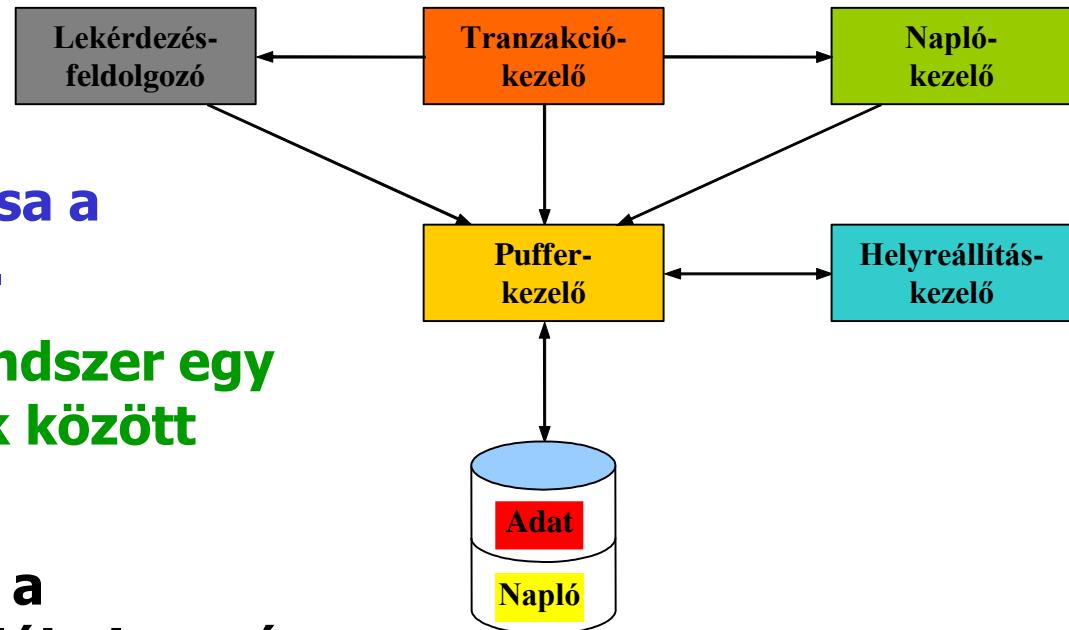


A naplókezelő és a tranzakciókezelő

A tranzakciók korrekt végrehajtásának biztosítása a tranzakciókezelő feladata.

A tranzakciókezelő részrendszer egy sor feladatot lát el, többek között

- **jelzéseket ad át a naplókezelőnek úgy, hogy a szükséges információ naplóbejegyzés formában a naplóban tárolható legyen;**
- **biztosítja, hogy a párhuzamosan végrehajtott tranzakciók ne zavarhassák egymás működését (**ütemezés**).**



A naplókezelő és a tranzakciókezelő

A tranzakciókezelő

1. a tranzakció tevékenységeiről üzeneteket küld a naplókezelőnek,
2. üzen a pufferkezelőnek arra vonatkozóan, hogy a pufferek tartalmát szabad-e vagy kell-e lemezre másolni,
3. és üzen a lekérdezésfeldolgozónak arról, hogy a tranzakcióban előírt lekérdezéseket vagy más adatbázis-műveleteket kell végrehajtania.



A naplókezelő és a tranzakciókezelő

A naplókezelő

- 1. a naplót tartja karban,**
- 2. együtt kell működnie a pufferkezelővel, hiszen a naplózandó információ elsődlegesen a memóriapufferekben jelenik meg, és bizonyos időnként a pufferek tartalmát lemezre kell másolni.**
- 3. A napló (adat lévén) a lemezen területet foglal el.**
 - **Ha baj van, akkor a helyreállítás-kezelő aktivizálódik. Megvizsgálja a naplót, és ha szükséges, a naplót használva helyreállítja az adatokat. A lemez elérése most is a pufferkezelőn át történik.**



A naplókezelő és a tranzakciókezelő

Azt minden feltehetően, hogy a háttér tár nem sérül, azaz csak a memória, illetve a puffer egy része száll el.

Az ilyen belső társérülés elleni védekezés két részből áll:

1. Felkészülés a hibára: naplázás
2. Hiba után helyreállítás: a napló segítségével egy konzisztens állapot helyreállítása

Természetesen a naplázás és a hiba utáni helyreállítás összhangban vannak, de van több különböző naplázási protokoll (és ennek megfelelő helyreállítás).



Adategység (adatbáziselem)

Feltesszük, hogy az adatbázis adategységekből, elemekből áll.

Az **adatbáziselem** (database element) a fizikai adatbázisban tártolt adatok egyfajta funkcionális egysége, amelynek értékét tranzakciókkal lehet elérni (**kiolvasni**) vagy módosítani (**kiírni**).

Az adatbáziselem lehet:

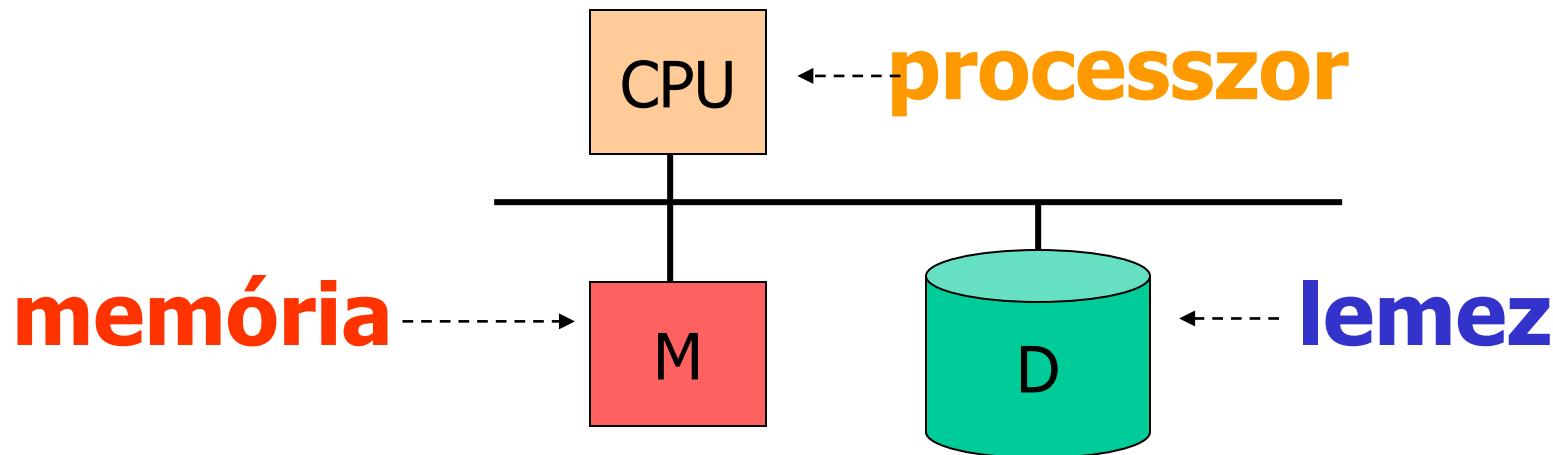
- **reláció (vagy OO megfelelője, az osztálykiterjedés),**
- **relációsor (vagy OO megfelelője, az objektum)**
- **lemezblokk**
- **lap**

Ez utóbbi a legjobb választás a naplázás szempontjából, mivel ekkor a puffer egyszerű elemekből fog állni, és ezzel elkerülhető néhány súlyos probléma, például amikor az adatbázis valamely elemének egy része van csak a nem illékony memóriában (a lemezen).



A vizsgált meghibásodási modell

A tranzakció és az adatbázis kölcsönhatásának három fontos helyszíne van:

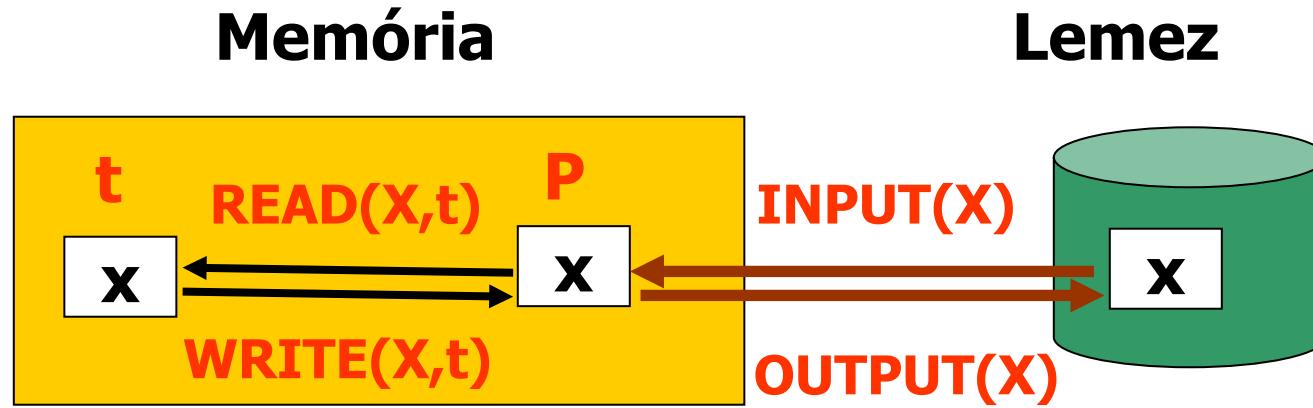


1. az adatbázis elemeit tartalmazó lemezblokkok területe; (D)
2. a pufferkezelő által használt virtuális vagy valós memóriaterület; (M)
3. a tranzakció memóriaterülete. (M)



OLVASÁS:

- Ahhoz, hogy a tranzakció egy X adatbáziselementet beolvashasson, azt előbb memóriapuffer(ek)be (P) kell behozni, ha még nincs ott.
- Ezt követően tudja a puffer(ek) tartalmát a tranzakció a saját memóriaterületére (t) beolvasni.



ÍRÁS:

- Az adatbáziselem új értékének kiírása fordított sorrendben történik: az új értéket a tranzakció alakítja ki a saját memóriaterületén, majd ez az új érték másolódik át a megfelelő puffer(ek)be.
Fontos, hogy egy tranzakció sohasem módosíthatja egy adatbáziselem értékét közvetlenül a lemezen!



Az adatmozgások alapműveletei:

1. **INPUT(X):** Az X adatbáziselementet tartalmazó lemezblokk másolása a memóriapufferbe.
2. **READ(X,t):** Az X adatbáziselem bemásolása a tranzakció t lokális változójába. Részletesebben: ha az X adatbáziselementet tartalmazó blokk nincs a memóriapufferben, akkor előbb végrehajtódik INPUT(X). Ezután kapja meg a t lokális változó X értékét.
3. **WRITE(X,t):** A t lokális változó tartalma az X adatbáziselem memóriapufferbeli tartalmába másolódik. Részletesebben: ha az X adatbáziselementet tartalmazó blokk nincs a memóriapufferben, akkor előbb végrehajtódik INPUT(X). Ezután másolódik át a t lokális változó értéke a pufferbeli X-be.
4. **OUTPUT(X):** Az X adatbáziselementet tartalmazó puffer kimásolása lemezre.



Az adatbáziselem mérete

- **FELTEVÉS:** az adatbáziselemek elférnek egy-egy lemezblokkban és így egy-egy pufferben is, azaz **feltételezhetjük, hogy az adatbáziselemek pontosan a blokkok.**
- **Ha az adatbáziselem valójában több blokkot foglalna el, akkor úgy is tekinthetjük, hogy az adatbáziselem minden blokkméretű része önmagában egy adatbáziselem.**
- A naplózási mechanizmus **atomos**, azaz vagy lemezre írja X összes blokkját, vagy semmit sem ír ki.
- A **READ** és a **WRITE** műveleteket a tranzakciók használják, az **INPUT** és **OUTPUT** műveleteket a pufferkezelő alkalmazza, illetve bizonyos feltételek mellett az **OUTPUT** műveletet a naplózási rendszer is használja.



Főprobléma: A befejezetlen tranzakciók

Például: Konzisztencia feltétel: $A=B$

T₁: $A \leftarrow A \times 2$

$B \leftarrow B \times 2$

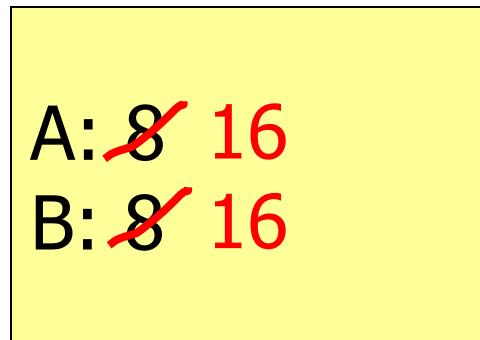
Pontosabban 8 lépésből áll:

**READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t);
OUTPUT(A); OUTPUT(B);**

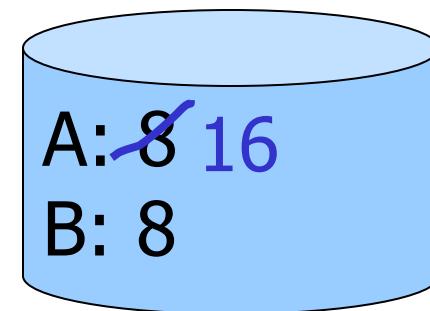


T1: Read (A,t); $t \leftarrow t \times 2$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);

Rendszerhiba!
Inkonzisztens maradna!



MEMÓRIA



LEMEZ



Az értékek változása a memóriában és a lemezen

| | <i>Tevékenység t</i> | <i>M-A</i> | <i>M-B</i> | <i>D-A</i> | <i>D-B</i> |
|-----------------|----------------------|------------|------------|------------|------------|
| 1. READ (A, t) | 8 | 8 | | 8 | 8 |
| 2. t := t*2 | 16 | 8 | | 8 | 8 |
| 3. WRITE (A, t) | 16 | 16 | | 8 | 8 |
| 4. READ (B, t) | 8 | 16 | 8 | 8 | 8 |
| 5. t := t*2 | 16 | 16 | 8 | 8 | 8 |
| 6. WRITE (B, t) | 16 | 16 | 16 | 8 | 8 |
| 7. OUTPUT (A) | 16 | 16 | 16 | 16 | 8 |
| 8. OUTPUT (B) | 16 | 16 | 16 | 16 | 16 |



- **Az atomosság miatt:**
 - nem maradhat így, vagy minden lépést végre kell hajtani, vagy egyet sem:
 - vagy **A=B=8** vagy **A=B=16** lenne jó
- **MEGOLDÁS:** naplózással
 - A **napló** (log): **naplóbejegyzések** (log records) sorozata, melyek mindegyike arról tartalmaz valami információt, hogy mit tett egy tranzakció.
 - Ha **rendszerhiba** fordul elő, akkor a napló segítségével rekonstruálható, hogy a tranzakció mit tett a hiba fellépéséig.
 - A naplót (az archívmentéssel együtt) használhatjuk akkor is, amikor **eszközhiba** keletkezik a naplót nem tároló lemezen.



Naplóbejegyzések

A tranzakciók legfontosabb történésein írjuk ide, például:

- *Ti kezdődik: (Ti, START)*
- *Ti írja A-t: (Ti, A, régi érték, új érték)*

(néha elég csak a régi vagy csak az új érték, a naplázási protokolltól függően)

- *Ti rendben befejeződött: (Ti, COMMIT)*
- *Ti a normálisnál korábban fejeződött be: (Ti, ABORT)*

A napló időrendben tartalmazza a történéseket és tipikusan a háttéráron tartjuk, amiről feltesszük, hogy nem sérült meg.

Fontos, hogy a naplóbejegyzéseket mikor írjuk át a pufferből a lemezre (például a naplókezelő kényszeríti ki, hogy COMMIT esetén a változások a lemezen is megtörténtek).



KÉTFÉLE NAPLÓZÁS:

- Egyes tranzakciók hatását visszont vissza kívánjuk vonni, azaz kérjük az adatbázis visszaállítását olyan állapotba, mintha a tekintett tranzakció nem is működött volna. (**SEMMISSÉGI – UNDO**)
- A katasztrófák hatásának kijavítását követően a tranzakciók hatását meg kell ismételni, és az általuk adatbázisba írt új értékeket ismételten ki kell írni. (**HELYREÁLLÍTÓ – REDO**)



Összefoglalás

Tranzakciókezelés és naplózás feladatai, folyamatai, adategységek választása, meghibásodás modell (részei és műveletei), tranzakciók táblázatos reprezentálása, lehetséges naplóbejegyzések, a naplózás két fajtája (Undo, Redo)

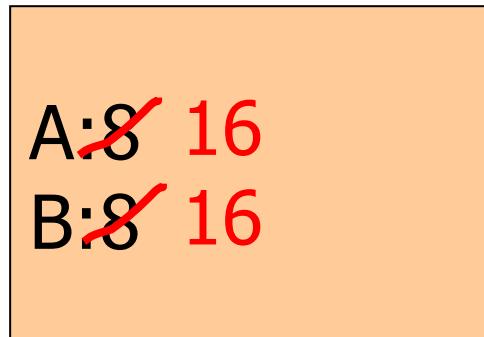


Undo log (SEMMISSÉGI NAPLÓZÁS)

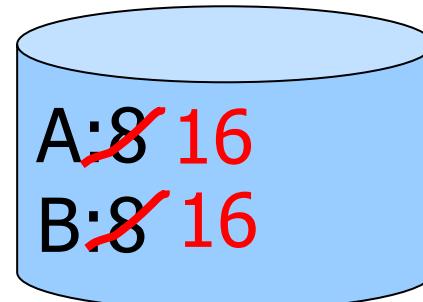
T₁: Read (A,t); $t \leftarrow t \times 2$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);

A=B

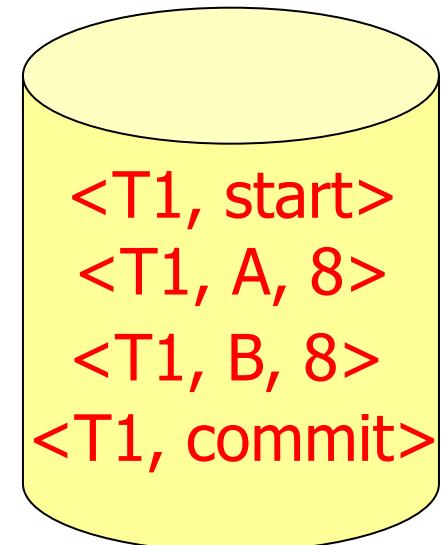
- A régi értéket naplózzuk!
- Azonnali kiírás!



MEMÓRIA



LEMEZ



NAPLÓ



Mikor írjuk ki a naplót a lemezre?

- A naplót először a memóriában frissítjük.
- Mi van, ha nem írjuk ki lemezre minden egyes változtatáskor, és elszáll a memória?

MEMÓRIA

A:~~8~~ 16

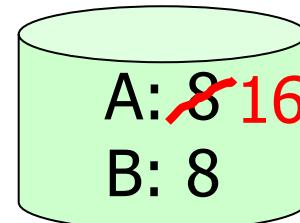
B:~~8~~ 16

Napló:

<T₁,start>

<T₁, A, 8>

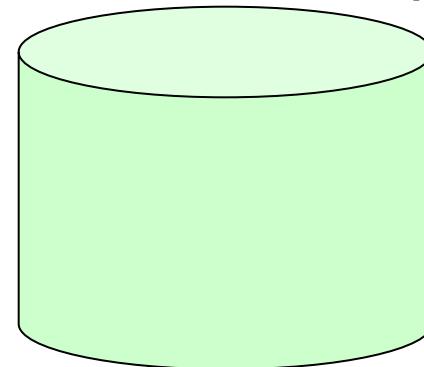
<T₁, B, 8>



Adatbázis

1. ROSSZ

Napló ÁLLAPOT

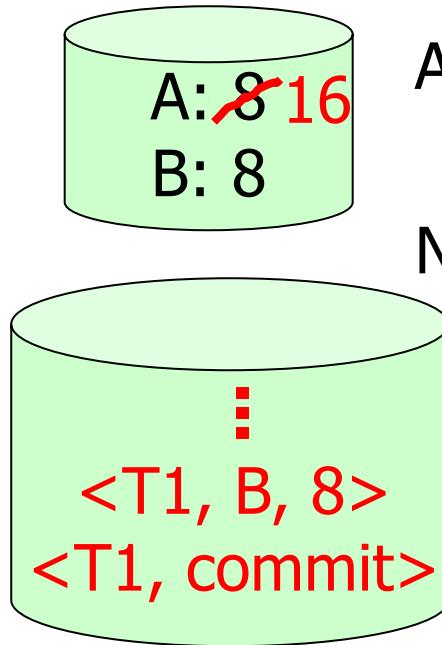


Mikor írjuk ki a naplót a lemezre?

- A naplót először a memóriában frissítjük.
- Mi van, ha előbb írjuk ki lemezre, mint hogy az adatbázist frissítettük volna, és közben elszáll a memória?

MEMÓRIA

```
A:8 16
B:8 16
Napló:
<T1,start>
<T1, A, 8>
<T1, B, 8>
<T1, commit>
```



Undo naplózás szabályai

- U1.** Ha a T tranzakció módosítja az X adatbáziselementet, akkor a (T, X , régi érték) naplóbejegyzést azelőtt kell a lemezre írni, mielőtt az X új értékét a lemezre írná a rendszer.
- U2.** Ha a tranzakció hibamentesen befejeződött, akkor a COMMIT naplóbejegyzést csak azután szabad a lemezre írni, ha a tranzakció által módosított összes adatbáziselem már a lemezre íródott, de ezután rögtön.



Undo naplózás esetén a lemezre írás sorrendje

1. **Az adatbáziselemek módosítására vonatkozó naplóbejegyzések;**
 2. **maguk a módosított adatbáziselemek;**
 3. **a COMMIT naplóbejegyzés.**
-
- **Az első két lépés minden módosított adatbáziselemre vonatkozóan önmagában, külön-külön végrehajtandó (nem lehet a tranzakció több módosítására csoportosan megtenni)!**
 - **A naplóbejegyzések lemezre írásának kikényszerítésére a naplókezelőnek szüksége van a FLUSH LOG műveletre, mely felszólítja a pufferkezelőt az összes korábban még ki nem írt naplóblokk lemezre való kiírására.**



Undo naplózás esetén a lemezre

írás sorrendje

| <i>Lépés</i> | <i>Tevékenység</i> | <i>t</i> | <i>M-A</i> | <i>M-B</i> | <i>D-A</i> | <i>D-B</i> | <i>Napló</i> |
|--------------|--------------------|----------|------------|------------|------------|------------|--------------|
| 1) | | | | | | | <T , START> |
| 2) | READ (A, t) | 8 | 8 | | 8 | 8 | |
| 3) | $t := t * 2$ | 16 | 8 | | 8 | 8 | |
| 4) | WRITE (A, t) | 16 | 16 | | 8 | 8 | <T , A , 8> |
| 5) | READ (B, t) | 8 | 16 | 8 | 8 | 8 | |
| 6) | $t := t * 2$ | 16 | 16 | 8 | 8 | 8 | |
| 7) | WRITE (B, t) | 16 | 16 | 16 | 8 | 8 | <T , B , 8> |
| 8) | FLUSH LOG | | | | | | |
| 9) | OUTPUT (A) | 16 | 16 | 16 | 16 | 8 | |
| 10) | OUTPUT (B) | 16 | 16 | 16 | 16 | 16 | |
| 11) | | | | | | | <T , COMMIT> |
| 12) | FLUSH LOG | | | | | | |



Helyreállítás UNDO napló alapján

A helyreállítás-kezelő feladata a napló használatával az adatbázist **konzisztens** állapotba visszaállítani.

- For every T_i with $\langle T_i, \text{start} \rangle$ in log:
 - If $\langle T_i, \text{commit} \rangle$ or $\langle T_i, \text{abort} \rangle$ in log, do nothing
 - Else
 - { For all $\langle T_i, X, v \rangle$ in log:
 - { write (X, v)
 - { output (X)
 - Write $\langle T_i, \text{abort} \rangle$ to log

Jó ez így?



Helyreállítás UNDO napló alapján

A helyreállítás során fontos a módosítások sorrendje!

- (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 $\begin{cases} - \text{write } (X, v) \\ - \text{output } (X) \end{cases}$
- (3) For each $T_i \in S$ do
 - write $\langle T_i, \text{abort} \rangle$ to log
- (4) Flush log

Ez miért lesz jó?



Helyreállítás UNDO napló alapján

Ha hiba történt → konzisztens állapot visszaállítása
→ nem befejezett tranzakciók hatásának törlése

Első feladat: Eldönteni melyek a sikeresen befejezett és melyek nem befejezett tranzakciók.

- Ha van (T, START) és van (T, COMMIT) → minden változás a lemezen van OK
- Ha van (T, START) , de nincs (T, COMMIT) , lehet olyan változás, ami nem került még a lemezre, de lehet olyan is ami kikerült → ezeket vissza kell állítani!



Helyreállítás UNDO napló alapján

Ha hiba történt → konzisztens állapot visszaállítása
→ nem befejezett tranzakciók hatásának törlése

Második feladat: visszaállítás

A napló végéről visszafelé (pontosabban a hibától) haladva: megjegyezzük, hogy mely Ti -re találtunk (Ti , COMMIT) vagy (Ti , ABORT) bejegyzéseket.

Ha van egy ($Ti; X; v$) bejegyzés:

- Ha láttunk már (Ti , COMMIT) bejegyzést (visszafelé haladva), akkor Ti már befejeződött, értékét kiírtuk a tárra
→ nem csinálunk semmit
- minden más esetben (vagy volt (Ti , ABORT) vagy nem)
→ X -be visszaírjuk v -t



Helyreállítás UNDO napló alapján

**Ha hiba történt → konzisztens állapot visszaállítása
→ nem befejezett tranzakciók hatásának törlése**

Harmadik feladat: Ha végeztünk, minden nem teljes *Ti*-re írunk (*Ti, ABORT*) bejegyzést a napló végére, majd kiírjuk a naplót a lemezre (**FLUSH LOG**).

Mi van ha a helyreállítás közben hiba történik? Már bizonyos értékeket visszaállítottunk, de utána elakadunk.

→ Kezdjük előlről a visszaállítást! Ha már valami vissza volt állítva, legfeljebb még egyszer „visszaállítjuk” → nem történik semmi. (A helyreállítás idempotens!)



Helyreállítás Undo naplózással

| Lépés | Tevékenység | t | M-A | M-B | D-A | D-B | Napló |
|-------|--------------|----|-----|-----|-----|-----|--------------|
| 1) | | | | | | | <T , START> |
| 2) | READ (A, t) | 8 | 8 | | 8 | 8 | |
| 3) | $t := t * 2$ | 16 | 8 | | 8 | 8 | |
| 4) | WRITE (A, t) | 16 | 16 | | 8 | 8 | <T , A , 8> |
| 5) | READ (B, t) | 8 | 16 | 8 | 8 | 8 | |
| 6) | $t := t * 2$ | 16 | 16 | 8 | 8 | 8 | |
| 7) | WRITE (B, t) | 16 | 16 | 16 | 8 | 8 | <T , B , 8> |
| 8) | FLUSH LOG | | | | | | |
| 9) | OUTPUT (A) | 16 | 16 | 16 | 16 | 8 | |
| 10) | OUTPUT (B) | 16 | 16 | 16 | 16 | 16 | |
| 11) | | | | | | | <T , COMMIT> |
| 12) | FLUSH LOG | | | | | | |

- Ha a hiba a 12) lépést követően jelentkezett:
- Tudjuk, hogy ekkor a <T, COMMIT> bejegyzést már lemezre írta a rendszer. A hiba kezelése során a T tranzakció hatásait nem kell visszaállítani, a T-re vonatkozó összes naplóbejegyzést a helyreállítás-kezelő figyelmen kívül hagyhatja.



Helyreállítás Undo naplózással

| Lépés | Tevékenység | t | M-A | M-B | D-A | D-B | Napló |
|-------|--------------|----|-----|-----|-----|-----|--------------|
| 1) | | | | | | | <T , START> |
| 2) | READ (A, t) | 8 | 8 | | 8 | 8 | |
| 3) | $t := t * 2$ | 16 | 8 | | 8 | 8 | |
| 4) | WRITE (A, t) | 16 | 16 | | 8 | 8 | <T , A , 8> |
| 5) | READ (B, t) | 8 | 16 | 8 | 8 | 8 | |
| 6) | $t := t * 2$ | 16 | 16 | 8 | 8 | 8 | |
| 7) | WRITE (B, t) | 16 | 16 | 16 | 8 | 8 | <T , B , 8> |
| 8) | FLUSH LOG | | | | | | |
| 9) | OUTPUT (A) | 16 | 16 | 16 | 16 | 8 | |
| 10) | OUTPUT (B) | 16 | 16 | 16 | 16 | 16 | |
| 11) | | | | | | | <T , COMMIT> |
| 12) | FLUSH LOG | | | | | | |

- Ha a hiba a 11) és 12) lépések között jelentkezett:
- Ha a COMMIT már lemezre íródott egy másik tranzakció miatt, akkor ez az előző eset.
- Ha a COMMIT nincs a lemezen, akkor T befejezetlen. B és A értékét visszaállítjuk, majd <T, ABORT>-t írunk a naplóba és a lemezre.



Helyreállítás Undo naplózással

| Lépés | Tevékenység | t | M-A | M-B | D-A | D-B | Napló |
|-------|--------------|----|-----|-----|-----|-----|--------------|
| 1) | | | | | | | <T , START> |
| 2) | READ (A, t) | 8 | 8 | | 8 | 8 | |
| 3) | $t := t * 2$ | 16 | 8 | | 8 | 8 | |
| 4) | WRITE (A, t) | 16 | 16 | | 8 | 8 | <T , A , 8> |
| 5) | READ (B, t) | 8 | 16 | 8 | 8 | 8 | |
| 6) | $t := t * 2$ | 16 | 16 | 8 | 8 | 8 | |
| 7) | WRITE (B, t) | 16 | 16 | 16 | 8 | 8 | <T , B , 8> |
| 8) | FLUSH LOG | | | | | | |
| 9) | OUTPUT (A) | 16 | 16 | 16 | 16 | 8 | |
| 10) | OUTPUT (B) | 16 | 16 | 16 | 16 | 16 | |
| 11) | | | | | | | <T , COMMIT> |
| 12) | FLUSH LOG | | | | | | |

- Ha a hiba a 10) és 11) lépések között lépett fel:
- Nincs COMMIT, tehát T befejezetlen, hatásainak semmissé tétele az előző esetnek megfelelően történik.



Helyreállítás Undo naplózással

| Lépés | Tevékenység | t | M-A | M-B | D-A | D-B | Napló |
|-------|--------------|----|-----|-----|-----|-----|--------------|
| 1) | | | | | | | <T , START> |
| 2) | READ (A, t) | 8 | 8 | | 8 | 8 | |
| 3) | $t := t * 2$ | 16 | 8 | | 8 | 8 | |
| 4) | WRITE (A, t) | 16 | 16 | | 8 | 8 | <T , A , 8> |
| 5) | READ (B, t) | 8 | 16 | 8 | 8 | 8 | |
| 6) | $t := t * 2$ | 16 | 16 | 8 | 8 | 8 | |
| 7) | WRITE (B, t) | 16 | 16 | 16 | 8 | 8 | <T , B , 8> |
| 8) | FLUSH LOG | | | | | | |
| 9) | OUTPUT (A) | 16 | 16 | 16 | 16 | 8 | |
| 10) | OUTPUT (B) | 16 | 16 | 16 | 16 | 16 | |
| 11) | | | | | | | <T , COMMIT> |
| 12) | FLUSH LOG | | | | | | |

- Ha a 8) és 10) lépések között következett be a hiba:
- Az előző esethez hasonlóan T hatásait semmissé kell tenni. Az egyetlen különbség, hogy az A és/vagy B módosítása még nincs a lemezén. Ettől függetlenül minden adatbáziselem korábbi értékét (8) állítjuk vissza.



Helyreállítás Undo naplózással

| Lépés | Tevékenység | <i>t</i> | M-A | M-B | D-A | D-B | Napló |
|-------|--------------|----------|-----|-----|-----|-----|--------------|
| 1) | | | | | | | <T , START> |
| 2) | READ (A, t) | 8 | 8 | | 8 | 8 | |
| 3) | $t := t * 2$ | 16 | 8 | | 8 | 8 | |
| 4) | WRITE (A, t) | 16 | 16 | | 8 | 8 | <T , A , 8> |
| 5) | READ (B, t) | 8 | 16 | 8 | 8 | 8 | |
| 6) | $t := t * 2$ | 16 | 16 | 8 | 8 | 8 | |
| 7) | WRITE (B, t) | 16 | 16 | 16 | 8 | 8 | <T , B , 8> |
| 8) | FLUSH LOG | | | | | | |
| 9) | OUTPUT (A) | 16 | 16 | 16 | 16 | 8 | |
| 10) | OUTPUT (B) | 16 | 16 | 16 | 16 | 16 | |
| 11) | | | | | | | <T , COMMIT> |
| 12) | FLUSH LOG | | | | | | |

- Ha a hiba a 8) lépésnél korábban jelentkezik:
- Az 1. szabály miatt igaz, hogy mielőtt az A és/vagy B a lemezen módosulnának, a megfelelő módosítási naplóbejegyzésnek a lemezre írt naplóban meg kell jelennie. Ez most nem történt meg, így a módosítások sem történtek meg, tehát nincs is visszaállítási feladat.



Az ellenőrzőpont-képzés

- A visszaállítás nagyon sokáig tarthat, mert el kell mennünk a napló elejéig.
- Meddig menjünk vissza a naplóban?
- Honnan tudjuk, hogy mikor vagyunk egy biztosan konzisztens állapotnál?
- Erre való a **CHECKPOINT**. Ennek képzése:
 1. Megtiltjuk az új tranzakciók indítását.
 2. Megvárjuk, amíg minden futó tranzakció **COMMIT** vagy **ABORT** módon véget ér.
 3. A naplót a pufferből a háttértárra írjuk (**FLUSH LOG**),
 4. Az adategységeket a pufferből a háttértárra írjuk.
 5. A naplóba beírjuk, hogy **CHECKPOINT**.
 6. A naplót újra a háttértárra írjuk: **FLUSH LOG**.
 7. Újra fogadjuk a tranzakciókat.
- Ezután nyilván elég az első **CHECKPOINT**-ig visszamenni, hiszen előtte minden Ti már valahogy befejeződött.



Az ellenőrzőpont-képzés

1. <T1, START>
2. <T1,A,5>
3. <T2, START>
4. <T2,B,10>
5. <T2,C,15>
6. <T1,D,20>
7. <T1, COMMIT>
8. <T2, COMMIT>
9. <CKPT>
- 10.<T3, START>
- 11.<T3,E,25>
- 12.<T3,F,30>

- Tegyük fel, hogy a 4. bejegyzés után úgy döntünk, hogy ellenőrzőpontot hozunk létre.
- A T1 és T2 aktív tranzakciók, meg kell várunk a befejeződésükét, mielőtt a <CKPT> bejegyzést a naplóba írnánk.

RENSZERHIBA

- Tegyük fel, hogy a naplórészlet végén rendszerhiba lép fel.
- HELYREÁLLÍTÁS: A naplót a végétől visszafelé elemezve T3-at fogjuk az egyetlen be nem fejezett tranzakciónak találni, így E és F korábbi értékeit kell csak visszaállítanunk.
- Amikor megtaláljuk a <CKPT> bejegyzést, tudjuk, hogy nem kell a korábbi naplóbejegyzéseket elemeznünk, végeztünk az adatbázis állapotának helyrehozásával.



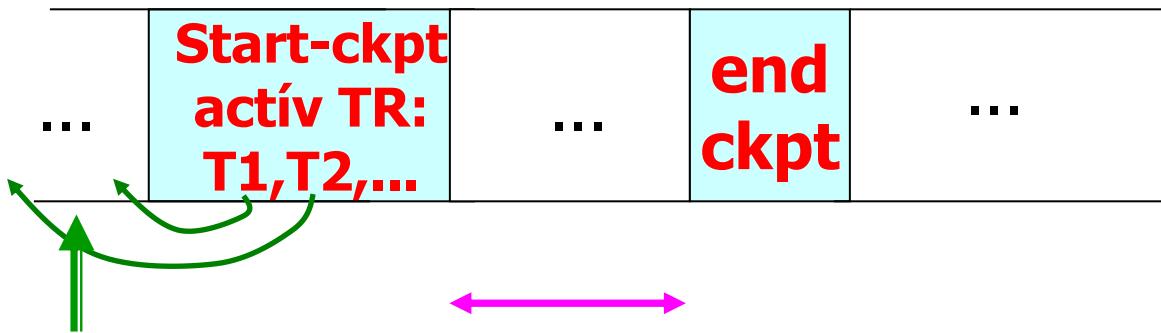
Az ellenőrzőpont-képzés

- **Probléma:** Hosszú ideig tarthat, amíg az aktív tranzakciók befejeződnek. (Új tranzakciót sokáig nem lehet kiszolgálni.)
- Megoldás: CHECKPOINT képzése működés közben.
- A módszer lépései:
 1. **<START CKPT(T_1, \dots, T_k)>** naplóbejegyzés készítése, majd lemezre írása (**FLUSH LOG**), ahol T_1, \dots, T_k az éppen aktív tranzakciók nevei.
 2. Meg kell várni a T_1, \dots, T_k tranzakciók mindegyikének normális vagy abnormális befejeződését, nem tiltva közben újabb tranzakciók indítását.
 3. Ha a T_1, \dots, T_k tranzakciók mindegyike befejeződött, akkor **<END CKPT>** naplóbejegyzés elkészítése, majd lemezre írása (**FLUSH LOG**).



Működés közbeni ellenőrzőpont (non-quiescent checkpointing)

LOG



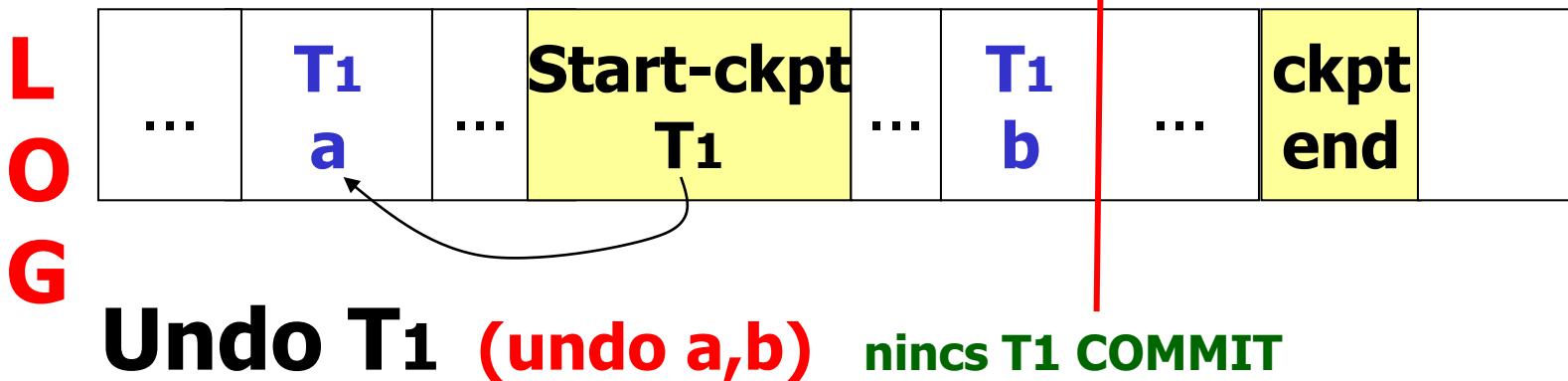
Csak az aktív
tranzakciók
előzményét
kell
magnézni!

A be nem
fejezett
tranzakciók által
írt, "PISZKOS"
adatok
kerülhetek a
lemezre!



Helyreállítás

- A naplót a végétől visszafelé elemezve megtaláljuk az összes **be nem fejezett tranzakciót**.
- A régi értékére visszaállítjuk az ezen tranzakciók által megváltoztatott adatbáziselemek tartalmát.



- Két eset fordulhat elő aszerint, hogy visszafelé olvasva a naplót az <END CKPT> vagy a <START CKPT(T₁,...,Tk)> naplóbejegyzést találjuk előbb:



Két eset fordulhat elő azértint, hogy visszafelé olvasva a naplót az <END CKPT> vagy a <START CKPT(T₁,...,T_k)> naplóbejegyzést találjuk előbb:

- 1. Ha előbb az <END CKPT> naplóbejegyzéssel találkozunk, akkor tudjuk, hogy az összes még be nem fejezett tranzakcióra vonatkozó naplóbejegyzést a legközelebbi korábbi <START CKPT(T₁,...,T_k)> naplóbejegyzésig megtaláljuk. Ott viszont megállhatunk, az annál korábbiakat akár el is dobhatjuk.**



Két eset fordulhat elő aszerint, hogy visszafelé olvasva a naplót az <END CKPT> vagy a <START CKPT(T₁,...,T_k)> naplóbejegyzést találjuk előbb:

2. Ha a <START CKPT(T₁,...,T_k)> naplóbejegyzéssel találkozunk előbb, az azt jelenti, hogy a katasztrófa az ellenőrzőpont-képzés közben fordult elő, ezért a T₁,...,T_k tranzakciók nem fejeződtek be a hiba fellépéséig.

- Ekkor a **be nem fejezett tranzakciók közül a legkorábban (t) kezdődött tranzakció indulásáig kell a naplóban visszakeresnünk, annál korábbra nem.**
- Az ezt megelőző olyan **START CKPT** bejegyzés, amelyhez tartozik **END CKPT**, biztosan megelőzi a keresett összes tranzakció indítását leíró bejegyzéseket. (S..E.t.S.S..S)
- **Ha a START CKPT előtt olyan START CKPT bejegyzést találunk, amelyhez nem tartozik END CKPT, akkor ez azt jelenti, hogy korábban is ellenőrzőpont-képzés közben történt rendszerhiba. Az ilyen „ellenőrzőpont-kezdeményeket” figyelmen kívül kell hagyni.**



Helyreállítás

- **Következmény:** ha egy **<END CKPT>** naplóbejegyzést **kiírunk lemezre**, akkor az azt megelőző **START CKPT** bejegyzésnél korábbi naplóbejegyzéseket törölhetjük.
- **Megjegyzés:** az ugyanazon tranzakcióra vonatkozó naplóbejegyzéseket összeláncoljuk, akkor nem kell a napló minden bejegyzését átnéznünk ahhoz, hogy megtaláljuk a még be nem fejezett tranzakciókra vonatkozó bejegyzéseket, elegendő csak az adott tranzakció bejegyzéseinek láncán visszafelé haladunk.



Helyreállítás

1. <T1, START >
2. <T1,A,5>
3. <T2, START >
4. <T2,B,10>
5. <**START CKPT(T1,T2)**>
6. <T2,C,15>
7. <T3, START >
8. <T1,D,20>
9. <T1, COMMIT >
10. <T3,E,25>
11. <T2, COMMIT >
12. <**END CKPT**>
13. <T3,F,30>

RENDSZERHIBA

- A naplót a végétől visszafelé vizsgálva úgy fogjuk találni, hogy **T3** egy **be nem fejezett tranzakció**, ezért hatásait semmissé kell tenni.
- Az utolsó naplóbejegyzés arról informál bennünket, hogy az **F** adatbáziselembe a **30** értéket kell visszaállítani.
- Amikor az <**END CKPT**> naplóbejegyzést találjuk, tudjuk, hogy az összes be nem fejezett tranzakció a megelőző **START CKPT** naplóbejegyzés után indulhatott csak el.
- Megtaláljuk a <T3,E,25> bejegyzést, emiatt az **E** adatbáziselem értékét **25-re** kell visszaállítani.
- Ezen bejegyzés és a **START CKPT** naplóbejegyzés között további elindult, de be nem fejeződött tranzakcióra vonatkozó bejegyzést nem találunk, így az adatbázison **mást már nem kell megváltoztatnunk**.



Helyreállítás

1. <T1, START >
2. <T1,A,5>
3. <T2, START >
4. <T2,B,10>
5. <START CKPT(T1,T2)>
6. <T2,C,15>
7. <T3, START >
8. <T1,D,20>
9. <T1, COMMIT >
10. <T3,E,25>
11. <T2, COMMIT >
12. <END CKPT>
13. <T3,F,30>

RENDSZERHIBA

- Visszafelé elemezve a naplót, megállapítjuk, hogy a T3, és a T2 nincs befejezve, tehát helyreállító módosításokat végzünk.
- Ezután megtaláljuk a <START CKPT(T1,T2)> naplóbejegyzést, emiatt az egyetlen be nem fejezett tranzakció csak a T2 lehet.
- A <T1, COMMIT> bejegyzést már láttuk, vagyis T1 befejezett tranzakció. Láttuk a <T3,START> bejegyzést is, így csak addig kell folytatnunk a napló elemzését, amíg a <T2, START> bejegyzését meg nem találjuk. Eközben még a B adatbáziselem értékét is visszaállítjuk 10-re.



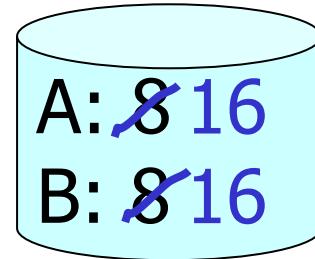
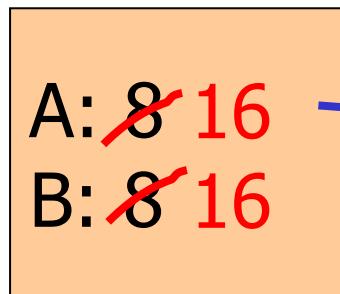
UNDO naplázás összefoglalása

- 1. UNDO naplázás U1, U2 szabály**
- 2. Helyreállítás algoritmusa**
- 3. Ellenőrzőpont leállással**
- 4. Ellenőrzőpont működés közben**



Redo logging (Helyrehozó naplózás)

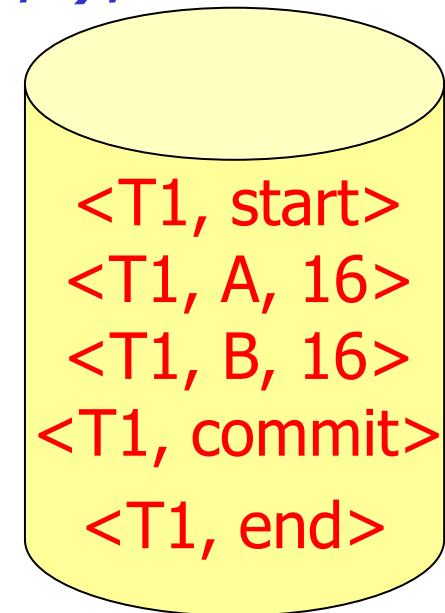
T1: Read(A,t); $t \leftarrow t \times 2$; write (A,t);
Read(B,t); $t \leftarrow t \times 2$; write (B,t);
Output(A); Output(B)



Memória

Adatbázis a lemezen

- **Az új értéket naplózzuk!**
- **Késleltetett kiírás!**



NAPLÓ



A helyrehozó naplózás szabálya

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 helyrehozó naplázás esetén a lemezre írás sorrendje

- (1) Ha egy **T** tranzakció **v**-re módosítja egy **X** adatbáziselem értékét, akkor egy **<T,X,v>** bejegyzést kell a naplóba írni.
- (2) Az adatbáziselemek módosítását leíró naplóbejegyzések lemezre írása.
- (3) A **COMMIT** naplóbejegyzés lemezre írása. (2. és 3. egy lépésben történik.)
- (4) Az adatbáziselemek értékének cseréje a lemezen.
- (5) A **<T,end>**-t bejegyezzük a naplóba, majd kiírjuk lemezre a naplót.



A helyrehozó naplózás szabályai

| <i>Lépés</i> | <i>Tevékenység</i> | <i>t</i> | <i>M-A</i> | <i>M-B</i> | <i>D-A</i> | <i>D-B</i> | <i>Napló</i> |
|--------------|--------------------|----------|------------|------------|------------|------------|--------------|
| 1) | | | | | | | <T , START> |
| 2) | READ (A, t) | 8 | 8 | | 8 | 8 | |
| 3) | $t := t * 2$ | 16 | 8 | | 8 | 8 | |
| 4) | WRITE (A, t) | 16 | 16 | | 8 | 8 | <T , A , 16> |
| 5) | READ (B, t) | 8 | 16 | 8 | 8 | 8 | |
| 6) | $t := t * 2$ | 16 | 16 | 8 | 8 | 8 | |
| 7) | WRITE (B, t) | 16 | 16 | 16 | 8 | 8 | <T , B , 16> |
| 8) | | | | | | | <T , COMMIT> |
| 9) | FLUSH LOG | | | | | | |
| 10) | OUTPUT (A) | 16 | 16 | 16 | 16 | 8 | |
| 11) | OUTPUT (B) | 16 | 16 | 16 | 16 | 16 | |
| 12) | | | | | | | <T , END> |
| 13) | FLUSH LOG | | | | | | |



Helyreállítás a REDO naplóból

- For every T_i with $\langle T_i, \text{commit} \rangle$ in log:
 - For all $\langle T_i, X, v \rangle$ in log:
 - { Write(X, v)
 - Output(X)

Jó ez így?



Helyreállítás a REDO naplóból

A helyreállítás során fontos a módosítások sorrendje!

- (1) Let $S = \text{set of transactions with } <Ti, \text{commit}> \text{ (and no } <Ti, \text{end}> \text{) in log}$
- (2) For each $<Ti, X, v>$ in log, in forward order (earliest \rightarrow latest) do:
 - if $Ti \in S$ then $\begin{cases} \text{Write}(X, v) \\ \text{Output}(X) \end{cases}$
- (3) For each $Ti \in S$, write $<Ti, \text{end}>$

Ez miért lesz jó?



Helyreállítás a módosított REDO naplóból

Nem használunk $\langle Ti, end \rangle$ bejegyzést a befejezett tranzakciókra, helyette a be nem fejezetteket jelöljük meg $\langle Ti, abort \rangle$ -tal. (Módosított REDO napló)

1. Meghatározzuk a befejezett tranzakciót (**COMMIT**).
2. Elemezzük a naplót az elejétől kezdve. minden $\langle T, X, v \rangle$ naplóbejegyzés esetén:
 - a) Ha **T be nem fejezett tranzakció**, akkor nem kell tenni semmit.
 - b) Ha **T befejezett tranzakció**, akkor v értéket kell írni az X adatbáziselembe.
3. minden **T be nem fejezett tranzakcióra** vonatkozóan $\langle T, ABORT \rangle$ naplóbejegyzést kell a naplóba írni, és a naplót ki kell írni lemezre (**FLUSH LOG**).



Helyreállítás

| <i>Lépés</i> | <i>Tevékenység</i> | <i>t</i> | <i>M-A</i> | <i>M-B</i> | <i>D-A</i> | <i>D-B</i> | <i>Napló</i> |
|--------------|--------------------|----------|------------|------------|------------|------------|--------------|
| 1) | | | | | | | <T , START> |
| 2) | READ (A , t) | 8 | 8 | | 8 | 8 | |
| 3) | $t := t * 2$ | 16 | 8 | | 8 | 8 | |
| 4) | WRITE (A , t) | 16 | 16 | | 8 | 8 | <T , A , 16> |
| 5) | READ (B , t) | 8 | 16 | 8 | 8 | 8 | |
| 6) | $t := t * 2$ | 16 | 16 | 8 | 8 | 8 | |
| 7) | WRITE (B , t) | 16 | 16 | 16 | 8 | 8 | <T , B , 16> |
| 8) | | | | | | | <T , COMMIT> |
| 9) | FLUSH LOG | | | | | | |
| 10) | OUTPUT (A) | 16 | 16 | 16 | 16 | 8 | |
| 11) | OUTPUT (B) | 16 | 16 | 16 | 16 | 16 | |

1. Ha a katasztrófa a 9) lépés után következik be:

- A <T, COMMIT> bejegyzés már lemezen van. A helyreállító rendszer **T-t** befejezett tranzakcióként azonosítja.
- Amikor a naplót az elejtől kezdve elemzi, a <T,A,16> és a <T,B,16> bejegyzések hatására a helyreállítás-kezelő az **A** és **B** adatbáziselemekbe a **16** értéket írja.



Helyreállítás

| <i>Lépés</i> | <i>Tevékenység t</i> | <i>M-A</i> | <i>M-B</i> | <i>D-A</i> | <i>D-B</i> | <i>Napló</i> |
|-----------------|----------------------|------------|------------|------------|------------|--------------|
| 1) | | | | | | <T , START> |
| 2) READ (A, t) | 8 | 8 | | 8 | 8 | |
| 3) t := t*2 | 16 | 8 | | 8 | 8 | |
| 4) WRITE (A, t) | 16 | 16 | | 8 | 8 | <T , A , 16> |
| 5) READ (B, t) | 8 | 16 | 8 | 8 | 8 | |
| 6) t := t*2 | 16 | 16 | 8 | 8 | 8 | |
| 7) WRITE (B, t) | 16 | 16 | 16 | 8 | 8 | <T , B , 16> |
| 8) | | | | | | <T , COMMIT> |
| 9) FLUSH LOG | | | | | | |
| 10) OUTPUT (A) | 16 | 16 | 16 | 16 | 8 | |
| 11) OUTPUT (B) | 16 | 16 | 16 | 16 | 16 | |

2. Ha a hiba a 8) és 9) lépések között jelentkezik:

- A <T, COMMIT> bejegyzés már a naplóba került, de nem biztos, hogy lemezre íródott.
- Ha lemezre került, akkor a helyreállítási eljárás az 1. esetnek megfelelően történik, ha nem, akkor pedig a 3. esetnek megfelelően.



Helyreállítás

| <i>Lépés</i> | <i>Tevékenység</i> | <i>t</i> | <i>M-A</i> | <i>M-B</i> | <i>D-A</i> | <i>D-B</i> | <i>Napló</i> |
|--------------|--------------------|----------|------------|------------|------------|------------|--------------|
| 1) | | | | | | | <T , START> |
| 2) | READ (A, t) | 8 | 8 | | 8 | 8 | |
| 3) | $t := t * 2$ | 16 | 8 | | 8 | 8 | |
| 4) | WRITE (A, t) | 16 | 16 | | 8 | 8 | <T , A , 16> |
| 5) | READ (B, t) | 8 | 16 | 8 | 8 | 8 | |
| 6) | $t := t * 2$ | 16 | 16 | 8 | 8 | 8 | |
| 7) | WRITE (B, t) | 16 | 16 | 16 | 8 | 8 | <T , B , 16> |
| 8) | | | | | | | <T , COMMIT> |
| 9) | FLUSH LOG | | | | | | |
| 10) | OUTPUT (A) | 16 | 16 | 16 | 16 | 8 | |
| 11) | OUTPUT (B) | 16 | 16 | 16 | 16 | 16 | |

3. Ha a katasztrófa a 8) lépést megelőzően keletkezik:

- Akkor <T, COMMIT> naplóbejegyzés még biztosan nem került lemezre, így T be nem fejezett tranzakciónak tekintendő.
- Ennek megfelelően A és B értékeit a lemezen még nem változtatta meg a T tranzakció, tehát nincs mit helyreállítani. Végül egy <ABORT T> bejegyzést írunk a naplóba.



Összehasonlítás

- Különbség a az UNDO protokollhoz képest:
- Az adat változás utáni értékét jegyezzük fel a naplóba
- Máshová rakjuk a COMMIT-ot, a kiírás elő => megtelhet a puffer
- Az UNDO protokoll esetleg túl gyakran akar írni => itt el lehet halasztani az írást



Helyrehozó naplázás ellenőrzőpont-képzés használatával

- **Új probléma:** a befejeződött tranzakciók módosításainak lemezre írása a befejeződés után sokkal később is történhet.
- **Következmény:** ugyanazon pillanatban aktív tranzakciók számát nincs értelme korlátozni, tehát nincs értelme az egyszerű ellenőrzőpont-képzésnek.
- **A kulcsfeladat** – amit meg kell tennünk az ellenőrzőpont-készítés kezdete és befejezése közötti időben – az összes olyan adatbáziselem lemezre való kiírása, melyeket **befejezett tranzakciók módosítottak**, és még nem voltak lemezre kiírva.
- Ennek megvalósításához a **pufferkezelőnek** nyilván kell tartania a piszkos puffereket (**dirty buffers**), melyekben már végrehajtott, de lemezre még ki nem írt módosításokat tárol. Azt is tudnunk kell, hogy mely tranzakciók mely puffereket módosították.



Helyrehozó naplázás ellenőrzőpont-képzés használatával

- Másrészről viszont **be tudjuk fejezni az ellenőrzőpont-képzést** az aktív tranzakciók (normális vagy abnormális) befejezésének **kivárása nélkül**, mert ők ekkor még amúgy sem engedélyezik lapjaik lemezre írását.
- A helyrehozó naplázásban a működés közbeni ellenőrzőpont-képzés a következőkből áll:
 1. **<START CKPT(T₁,...,T_k)>** naplóbejegyzés elkészítése és lemezre írása, ahol T₁,...,T_k az összes éppen aktív tranzakció.
 2. Az összes olyan adatbáziselem kiírása lemezre, melyeket olyan tranzakciók írtak pufferekbe, melyek a **START CKPT** naplóba írásakor már befejeződtek, de puffereik lemezre még nem kerültek.
 3. **<END CKPT>** bejegyzés naplóba írása, és a napló lemezre írása.



Helyreállítás ellenőrzőpont esetén

1. <T1, START>
2. <T1,A,5>
3. <T2, START>
4. <T1, COMMIT>
5. <T2,B,10>
6. <START CKPT(T2)>
7. <T2,C,15>
8. <T3, START>
9. <T3,D,20>
10. <END CKPT>
11. <T2, COMMIT>
12. <T3, COMMIT>

- Amikor az ellenőrzőpont-képzés elkezdődött, csak T2 volt aktív, de a T1 által A-ba írt érték még nem biztos, hogy lemezre került. Ha még nem, akkor A-t lemezre kell másolnunk, mielőtt az ellenőrzőpont-képzést befejezhetnénk.



Helyreállítás ellenőrzőpont 1. eset

1. <T1, START>
2. <T1,A,5>
3. <T2, START>
4. <T1, COMMIT>
5. <T2,B,10>
6. <START CKPT(T2)>
7. <T2,C,15>
8. <T3, START>
9. <T3,D,20>
10. <END CKPT>
11. <T2, COMMIT>
12. <T3, COMMIT>

RENDSZERHIBA

1. Ha a hiba előtt a naplóba feljegyzett utolsó ellenőrzőpont-bejegyzés <END CKPT>.
- Az olyan értékek, melyeket olyan tranzakciók írtak, melyek a <START CKPT(T1,...,Tk)> naplóbejegyzés megtétele előtt befejeződtek, már biztosan lemezre kerültek, így nem kell velük foglalkoznunk.



Helyreállítás ellenőrzőpont 1. eset

1. <T1, START>
2. <T1,A,5>
3. <T2, START>
4. <T1, COMMIT>
5. <T2,B,10>
6. <START CKPT(T2)>
7. <T2,C,15>
8. <T3, START>
9. <T3,D,20>
10. <END CKPT>
11. <T2, COMMIT>
12. <T3, COMMIT>

<T2,COMMIT> és <T3,COMMIT> miatt
T2 és T3 befejezett tranzakció.

Így <T2,B,10>, <T2,C,15> és
<T3,D,20> alapján a lemezre újraírjuk
a B, a C és a D tartalmát, megfelelően
10, 15 és 20 értékeket adva nekik.

RENDSZERHIBA

- Elég azokat a tranzakciókat venni, melyek az utolsó <START CKPT(T1,...,Tk)> naplóbejegyzésben a Ti-k között szerepelnek, vagy ezen naplóbejegyzést követően indultak el. (T2,T3)
- A naplóban való keresés során a legkorábbi <Ti, START> naplóbejegyzésig kell visszamennünk, annál korábbra nem.
- Ezek a START naplóbejegyzések akárhány korábbi ellenőrzőpontnál előbb is felbukkanhatnak.



Helyreállítás ellenőrzőpont 1. eset

1. <T1, START>
2. <T1,A,5>
3. <T2, START>
4. <T1, COMMIT>
5. <T2,B,10>
6. <START CKPT(T2)>
7. <T2,C,15>
8. <T3, START>
9. <T3,D,20>
10. <END CKPT>
11. <T2, COMMIT>
12. <T3, COMMIT>

<T2,COMMIT> miatt csak T2 befejezett tranzakció.

Így <T2,B,10>, <T2,C,15> alapján a lemezre újraírjuk a B, a C tartalmát, megfelelően 10, 15 értékeket adva nekik.

RENDSZERHIBA

- Elég azokat a tranzakciókat venni, melyek az utolsó <START CKPT(T1,...,Tk)> naplóbejegyzésben a Ti-k között szerepelnek, vagy ezen naplóbejegyzést követően indultak el. (T2,T3)
- T3 most be nem fejezett, így nem kell újragörgetni.
- A helyreállítást követően egy <T3, ABORT> bejegyzést írunk a naplóba.



Helyreállítás ellenőrzőpont 2. eset

1. <T1, START>
2. <T1,A,5>
3. <T2, START>
4. <T1, COMMIT>
5. <T2,B,10>
6. <START CKPT(T2)>
7. <T2,C,15>
8. <T3, START>
9. <T3,D,20>
10. <END CKPT>
11. <T2, COMMIT>
12. <T3, COMMIT>

RENSZERHIBA

S...E...S

2. Ha a naplóba feljegyzett utolsó ellenőrzőpont-bejegyzés a <START CKPT(T1,...,Tk)>.
- Az előző <END CKPT> bejegyzéshez tartozó <START CKPT(S1,...,Sm)> bejegyzésig vissza kell keresnünk, és helyre kell állítanunk az olyan befejeződött tranzakciók tevékenységének eredményeit, melyek ez utóbbi <START CKPT(S1,...,Sm)> bejegyzés után indultak, vagy az Si-k közül valók.
 - Ha nincs előző <END CKPT>, akkor a napló elejéig kell visszamenni.



Helyreállítás ellenőrzőpont 2. eset

1. <T1, START>
2. <T1,A,5>
3. <T2, START>
4. <T1, COMMIT>
5. <T2,B,10>
6. <START CKPT(T2)>
7. <T2,C,15>
8. <T3, START>
9. <T3,D,20>  RENDSZERHIBA
10. <END CKPT>
11. <T2, COMMIT>
12. <T3, COMMIT>

- Most nem találunk korábbi ellenőrzőpont-bejegyzést, így a napló elejére kell mennünk.
- Így esetünkben az egyedüli **befejezett tranzakciónak T1-et** fogjuk találni, ezért a <T1,A,5> tevékenységet helyreállítjuk.
- A helyreállítást követően <T2, ABORT> és <T3, ABORT> bejegyzést írunk a naplóba.



Redo naplózás összefoglalása

- 1. Redo naplózás R1 szabálya, írás csak a Commit után**
- 2. END nélküli naplózás**
- 3. A helyreállítás algoritmusa (a befejezett tranzakciót újra végrehajtjuk)**
- 4. Ellenőrzőpont képzése (a start chkpoint előtt befejezett tranzakcióknál kikényszerítjük a változtatások lemezreírását.)**



Semmisségi/helyrehozó (undo redo) naplázás

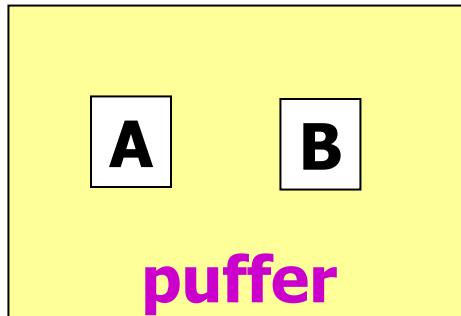
- A **semmisségi naplázás** esetén az adatokat a tranzakció befejezésekor nyomban lemezre kell írni, **nő a végrehajtandó lemezműveletek száma**.
- A **helyrehozó naplázás** minden módosított adatbázisblokk pufferben tartását igényli egészen a tranzakció rendes és teljes befejezéséig, így a napló kezelésével **nő a tranzakciók átlagos pufferigénye**.



Semmisségi/helyrehozó undo/redo) naplázás

További probléma (**blokknál kisebb adatbáziselemek esetén**): **ellentétes pufferírási igények**

(T1, COMMIT)
lemezen van



(T2, COMMIT)
nincs a lemezen

Például:

T1 az A-t módosította és befejeződött rendben

T2 a B-t módosította, de a COMMIT nincs a lemezen

Ekkor az R1 szabály miatt:

- a puffert lemezre kell írni A miatt,
- a puffert nem szabad lemezre írni B miatt.



Megoldás

- **A naplóbejegyzés négykomponensű:**
 - a $\langle T, X, v, w \rangle$ naplóbejegyzés azt jelenti, hogy a **T** tranzakció az adatbázis **X** elemének **korábbi v** értékét **w-re** módosította.
- **UR1:** Mielőtt az adatbázis bármely **X** elemének értékét – valamely **T** tranzakció által végzett módosítás miatt – a lemezen módosítanánk, **ezt megelőzően** a $\langle T, X, v, w \rangle$ naplóbejegyzésnek lemezre kell kerülnie.
- **WAL – Write After Log elv:** előbb naplózunk, utána módosítunk
- **NAGYOBB SZABADSÁG:** A $\langle T, \text{COMMIT} \rangle$ bejegyzés megelőzheti, de követheti is az adatbáziselemek lemezen történő bármilyen megváltoztatását.
- **NAGYOBB MÉRETŰ NAPLÓ:** - régi és új értéket is tároljuk



UNDO/REDO naplázás

| Lépés | Tevékenység | t | M-A | M-B | D-A | D-B | Napló |
|-------|------------------|----|-----|-----|-----|-----|------------|
| 1) | | | | | | | <T,START> |
| 2) | READ(A,t) | 8 | 8 | | 8 | 8 | |
| 3) | $t := t^2$ | 16 | 8 | | 8 | 8 | |
| 4) | WRITE(A,t) | 16 | 16 | | 8 | 8 | <T,A,8,16> |
| 5) | READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| 6) | $t := t^2$ | 16 | 16 | 8 | 8 | 8 | |
| 7) | WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <T,B,8,16> |
| 8) | FLUSH LOG | | | | | | |
| 9) | OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| 10) | | | | | | | <T,COMMIT> |
| 11) | OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |

Megjegyzés: A <T,COMMIT> naplóbejegyzés kiírása kerülhetett volna a 9) lépés elő vagy a 11) lépés mögé is.



Helyreállítás UNDO/REDO naplózás esetén

A semmisségi/helyrehozó módszer alapelvei a következők:

- 1. (REDO): A legkorábbitól kezdve állítsuk helyre minden befejezettranzakció hatását.**
- 2. (UNDO): A legutolsótól kezdve tegyük semmisse minden be nem fejezettranzakció tevékenységeit.**

Megjegyzés: A **COMMIT** kötetlen helye miatt előfordulhat, hogy egy **befejezett tranzakció néhány vagy összes változtatása még nem került lemezre**, és az is, hogy egy **be nem fejezett tranzakció néhány vagy összes változtatása már lemezen is megtörtént.**



Helyreállítás UNDO/REDO naplózás esetén

| Lépés | Tevékenység | t | M-A | M-B | D-A | D-B | Napló |
|-------|------------------|----|-----|-----|-----------|-----|------------|
| 1) | | | | | | | <T,START> |
| 2) | READ(A,t) | 8 | 8 | | 8 | 8 | |
| 3) | $t := t^2$ | 16 | 8 | | 8 | 8 | |
| 4) | WRITE(A,t) | 16 | 16 | | 8 | 8 | <T,A,8,16> |
| 5) | READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| 6) | $t := t^2$ | 16 | 16 | 8 | 8 | 8 | |
| 7) | WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <T,B,8,16> |
| 8) | FLUSH LOG | | | | | | |
| 9) | OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| 10) | | | | | | | <T,COMMIT> |
| 11) | OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |

- **Ha a katasztrófa a <T,COMMIT> naplóbejegyzés lemezre írása után történik:**
- **T-t befejezett tranzakciónak tekintjük. 16-ot írunk mind A-ba, minden B-be.**
- **A-nak már 16 a tartalma, de B-nek lehet, hogy nem, aszerint, hogy a hiba a 11) lépés előtt vagy után következett be.**



Helyreállítás UNDO/REDO naplózás esetén

| Lépés | Tevékenység | t | M-A | M-B | D-A | D-B | Napló |
|-------|------------------|----|-----|-----|-----|-----|------------|
| 1) | | | | | | | <T,START> |
| 2) | READ(A,t) | 8 | 8 | | 8 | 8 | |
| 3) | $t := t^2$ | 16 | 8 | | 8 | 8 | |
| 4) | WRITE(A,t) | 16 | 16 | | 8 | 8 | <T,A,8,16> |
| 5) | READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| 6) | $t := t^2$ | 16 | 16 | 8 | 8 | 8 | |
| 7) | WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <T,B,8,16> |
| 8) | FLUSH LOG | | | | | | |
| 9) | OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| 10) | | | | | | | <T,COMMIT> |
| 11) | OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |

- **Ha a katasztrófa a <T,COMMIT> naplóbejegyzés lemezre írását megelőzően, a 9) és 10) lépések között következett be:**
- **T befejezetlen tranzakció: Ekkor A és B korábbi értéke, 8 íródik lemezre. Az A értéke már 16 volt a lemezen, és emiatt a 8-ra való visszaállítás feltétlenül szükséges. A B értéke nem igényelne visszaállítást, de nem lehetünk biztosak benne, így végrehajtjuk.**



Helyreállítás UNDO/REDO naplózás esetén

| Lépés | Tevékenység | t | M-A | M-B | D-A | D-B | Napló |
|-------|------------------|----|-----|-----|-----|-----|------------|
| 1) | | | | | | | <T,START> |
| 2) | READ(A,t) | 8 | 8 | | 8 | 8 | |
| 3) | $t := t^2$ | 16 | 8 | | 8 | 8 | |
| 4) | WRITE(A,t) | 16 | 16 | | 8 | 8 | <T,A,8,16> |
| 5) | READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| 6) | $t := t^2$ | 16 | 16 | 8 | 8 | 8 | |
| 7) | WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <T,B,8,16> |
| 8) | FLUSH LOG | | | | | | |
| 9) | OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| 10) | | | | | | | <T,COMMIT> |
| 11) | OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |

- Ha a katasztrófa a <T,COMMIT> naplóbejegyzés lemezre írását megelőzően, a 9) lépés előtt következett be:
- T befejezetlen tranzakció: Az A és B korábbi értéke, 8 íródik lemezre. (Most A és B sem igényelné a visszaállítást, de mivel nem lehetünk biztosak abban, hogy a visszaállítás szükséges-e vagy sem, így azt (a biztonság kedvéért) minden végre kell hajtanunk.)



Helyreállítás UNDO/REDO naplózás esetén

- **Probléma (befejezett változtatást is megsemmisítünk): Az UNDO naplózáshoz hasonlóan most is előfordulhat, hogy a tranzakció a felhasználó számára korrekten befejezettnek tűnik, de még a $\langle T, COMMIT \rangle$ naplóbejegyzés lemezre kerülése előtt fellépett hiba utáni helyreállítás során a rendszer a tranzakció hatásait semmissé teszi ahelyett, hogy helyreállította volna.**
- Amennyiben ez a lehetőség problémát jelent, akkor a semmisségi/helyrehozó naplózás során egy további szabályt célszerű bevezetni:
- **UR2: A $\langle T, COMMIT \rangle$ naplóbejegyzést nyomban lemezre kell írni, amint megjelenik a naplóban.**
- Ennek teljesítéséért a fenti példában a 10) lépés ($\langle T, COMMIT \rangle$) után egy **FLUSH LOG** lépést kell beiktatnunk.



Helyreállítás UNDO/REDO naplózás esetén

- **Konkurencia problémája:**

Előfordulhat, hogy a T tranzakció rendben és teljesen befejeződött, és emiatt helyreállítása során egy X adatbáziselem T által kialakított új értékét rekonstruáljuk, melyet viszont egy be nem fejezett, és ezért visszaállítandó U tranzakció korábban módosított, ezért vissza kellene állítani az X régi értékét.

A probléma nem az, hogy először helyreállítjuk X értékét, és aztán állítjuk vissza U előttire, vagy fordítva. Egyik sorrend sem helyes, mert a végső adatbázisállapot nagy valószínűséggel így is, úgy is inkonzisztens lesz.

A konkurenciakezelésnél fogjuk megoldani, hogyan biztosítható T és U elkülönítése, amivel az ugyanazon X adatbáziselemen való kölcsönhatásuk elkerülhető.



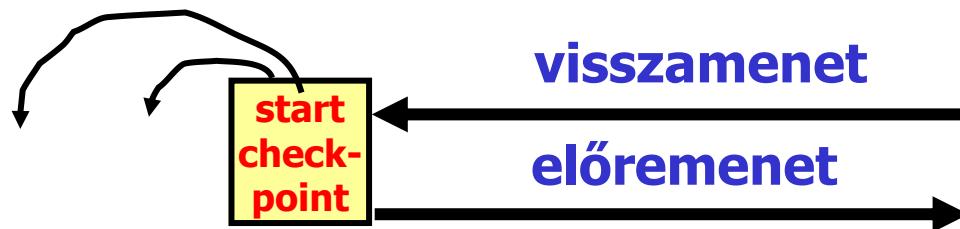
Semmisségi/helyrehozó naplázás ellenőrzőpont-képzéssel

- **Egyszerűbb, mint a másik két naplázás esetén.**
- 1. Írunk a naplóba **<START CKPT(T₁,...,T_k)>** naplóbejegyzést, ahol **T₁,...,T_k** az **aktív tranzakciók**, majd **írunk a naplót lemezre**.
- 2. **Írunk lemezre az összes piszkos puffert**, tehát azokat, melyek egy vagy több módosított adatbáziselementet tartalmaznak. A helyrehozó naplózástól eltérően itt az összes piszkos puffert lemezre írunk, nem csak a már befejezett tranzakciók által módosítottakat.
- 3. Írunk **<END CKPT>** naplóbejegyzést a naplóba, majd **írunk a naplót lemezre**.



Helyreállítás:

- **Visszamenet** (napló végetől \rightarrow az utolsó érvényes checkpoint kezdetéig)
 - meghatározzuk a befejezett tranzakciók **S** halmazát
 - megsemmisítjük azoknak a tranzakciók hatását, amelyek nincsenek **S**-ben
- **Megsemmisítjük a függő tranzakciókat**
 - visszamegyünk azokon a tranzakciókon, amelyek (checkpoint aktív tranzakciói) – **S** halmazban vannak
- **Előremenet** (az utolsó checkpoint kezdetétől \rightarrow a napló végéig)
 - helyrehozzuk az **S** tranzakcióinak hatását



Semmisségi/helyrehozó naplázás

ellenőrzőpont-képzéssel

1. <T1, START>
2. <T1,A,4,5>
3. <T2, START>
4. <T1, COMMIT>
5. <T2,B,9,10>
6. <START CKPT(T2)>
7. <T2,C,14,15>
8. <T3,START>
9. <T3,D,19,20>
10. <END CKPT>
11. <T2, COMMIT>
12. <T3, COMMIT>

- Lehetséges, hogy a T2 által B-nek adott új érték (**10**) lemezre íródik az <END CKPT> előtt, ami nem volt megengedve a helyrehozó naplázásban.
- Most lényegtelen, hogy ez a lemezre írás mikor történik meg. Az **ellenőrzőpont képzése alatt biztosan lemezre írjuk B-t** (ha még nem került oda), mivel minden piszkos (változásban érintett) puffert kiírunk lemezre.
- Hasonlóan A-t – melyet a befejezett T1 tranzakció alakított ki – **is lemezre fogjuk írni, ha még nem került oda.**



Semmisségi/helyrehozó naplázás ellenőrzőpont-képzéssel

1. <T1, START>
2. <T1,A,4,5>
3. <T2, START>
4. <T1, COMMIT>
5. <T2,B,9,10>
6. <START CKPT(T2)>
7. <T2,C,14,15>
8. <T3,START>
9. <T3,D,19,20>
10. <END CKPT>
11. <T2, COMMIT>
12. <T3, COMMIT>

Amikor olyan tranzakció hatásait állítjuk helyre, mint amilyen a T2 is, akkor a naplóban nem kell a <START CKPT(T2)> bejegyzésnél korábbra visszatekinteni, mert tudjuk, hogy a T2 által az ellenőrzőpont-képzést megelőzően elvégzett módosítások az ellenőrzőpont képzése alatt lemezre íródtak.

KATASZTRÓFA

- **T2-t és T3-at teljesen és rendesen befejezett tranzakciónak tekintjük.**
- **A T1 tranzakció az ellenőrzőpontnál korábbi. Minthogy <END CKPT> bejegyzést találunk a naplóban, így T1-ről biztosan tudjuk, hogy teljesen és rendesen befejeződött, valamint az általa elvégzett módosítások lemezre íródtak. Ezért a T2 és T3 által végrehajtott módosítások helyreállítandók, T1 pedig figyelmen kívül hagyható.**



Semmisségi/helyrehozó naplázás ellenőrzőpont-képzéssel

1. <T1, START>
2. <T1,A,4,5>
3. <T2, START>
4. <T1, COMMIT>
5. <T2,B,9,10>
6. <START CKPT(T2)>
7. <T2,C,14,15>
8. <T3,START>
9. <T3,D,19,20>
10. <END CKPT>
11. <T2, COMMIT>
12. <T3, COMMIT>

Ha T3 az ellenőrzőpont-képzés előtt már aktív tranzakció lett volna, akkor a naplóban a START CKPT bejegyzésben szereplő befejezetlen tranzakciók közül a legkorábban elindult Ti tranzakció <Ti,START> bejegyzésig kellene visszakeresnünk, hogy megtaláljuk a Ti (most T2 vagy T3) semmissé teendő tevékenységeit leíró naplóbejegyzéseket. A helyrehozó lépést viszont most is elég a START CKPT bejegyzéstől végrehajtani.

KATASZTRÓFA

- Ekkor **T2-t befejezett, T3-at pedig befejezetlen tranzakciónak kell tekintenünk.**
- **T2 tevékenységét helyreállítandó C értékét a lemezen 15-re írjuk; B-t már nem kell 10-re írnunk a lemezen, mert tudjuk, hogy ez már lemezre került az <END CKPT> előtt.**
- **A helyreállító naplázástól eltérően T3 hatásait semmissé tesszük, azaz a lemezen D tartalmát 19-re írjuk.**



Semmisségi/helyrehozó naplázás ellenőrzőpont-képzéssel

1. <T1, START>
2. <T1,A,4,5>
3. <T2, START>
4. <T1, COMMIT>
5. <T2,B,9,10>
6. <START CKPT(T2)>
7. <T2,C,14,15> → KATASZTRÓFA
8. <T3,START>
9. <T3,D,19,20>
10. <END CKPT>
11. <T2, COMMIT>
12. <T3, COMMIT>

- Ha a katasztrófa az <END CKPT> bejegyzés előtt lép fel, akkor figyelmen kívül hagyjuk az utolsó **START CKPT** bejegyzést, és az előzőek szerint járunk el.

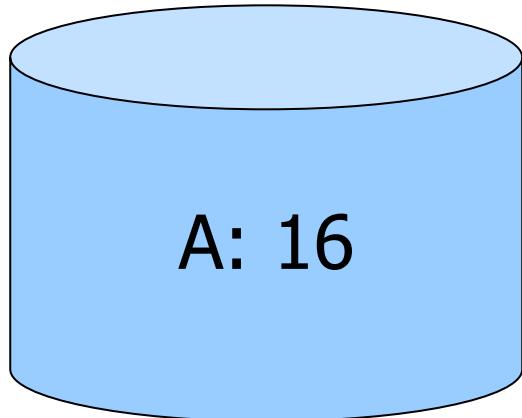


Undo/Redo naplózás összefoglalása

1. Undo/Redo naplózás, UR1 szabály, WAL, (Commit bárhol lehet)
2. Helyreállítás algoritmusa (naplóban két irányú mozgás)
3. UR2 szabály
4. Ellenőrző pont képzése (összes piszkos puffer kiírása)



Az eszközök meghibásodásának kezelése



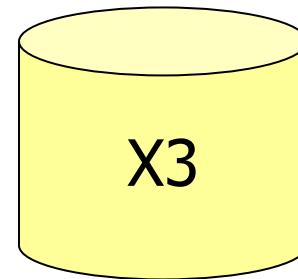
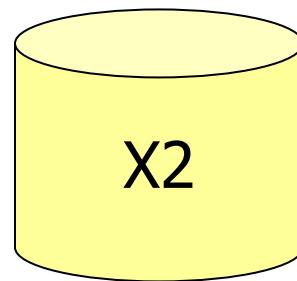
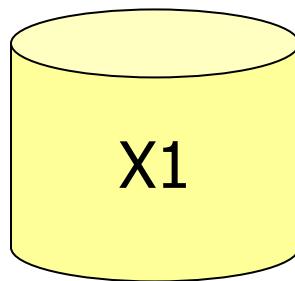
Megoldás: Készítsünk másolatokat (backup - mentés) az adatokról!



Védelmi módszerek a lemezhibák ellen:

1. Háromszoros redundancia

- 3 másolat különböző lemezeken
- Output(X) --> 3 kiírás
- Input(X) --> 3 beolvasás + szavazás



Védelmi módszerek a lemezhibák ellen:

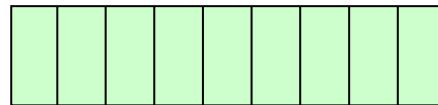
2. Többszörös írás, egyszeres olvasás

- N másolatot tartunk különböző lemezeken
 - Output(X) --> N kiírás
 - Input(X) --> 1 másolat beolvasása
 - { - ha ok, kész
 - különben egy másik másolat beolvasása
- ↔ **Feltevés:** észrevesszük, ha rossz egy adat



Védelmi módszerek a lemezhibák ellen:

3: Adatbázis mentés + napló

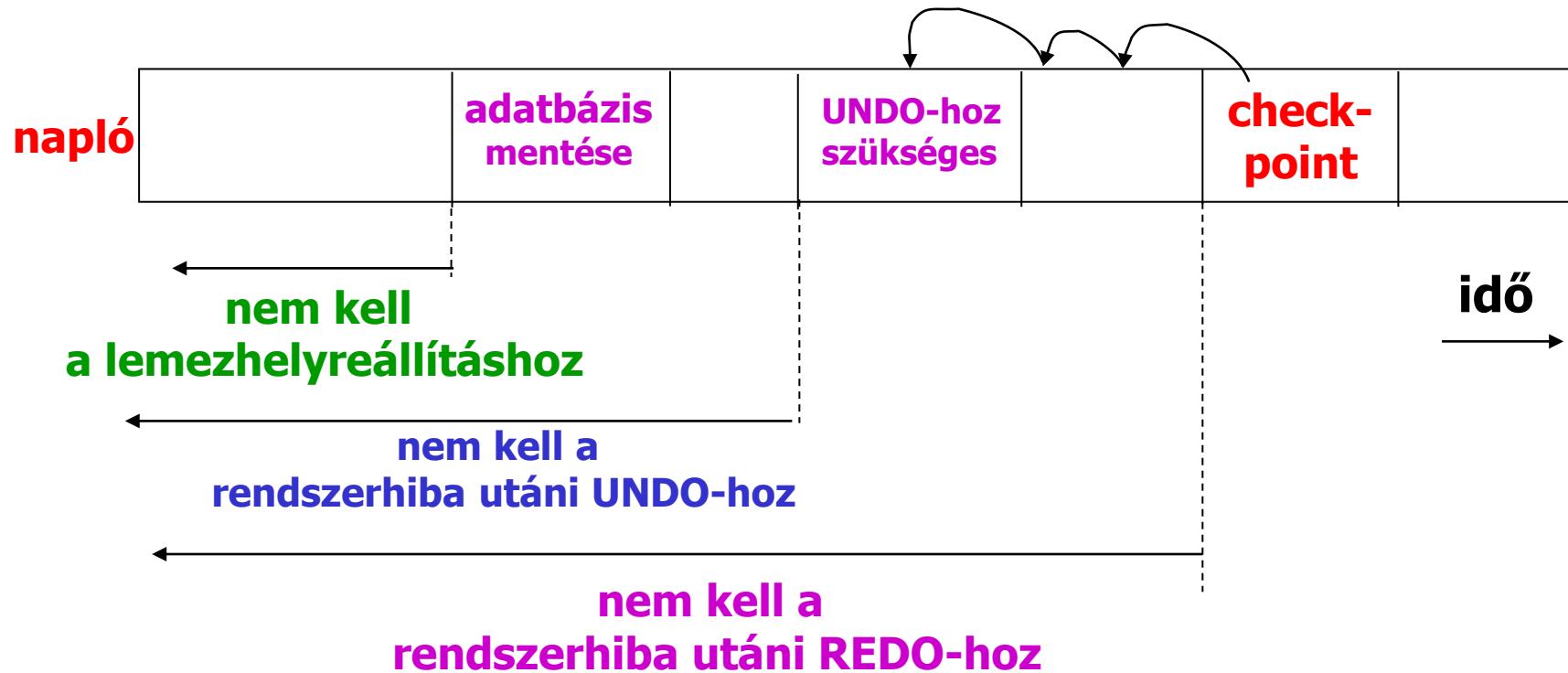


Ha az aktív adatbázis megsérül,

- 1. az adatbázis visszatöltése a mentésből**
- 2. a napló redo bejegyzéseiből naprakész állapot visszaállítása**



A napló melyik részét lehet eldobni?



Helyreállítás mentésekből és naplóból

- A napló használatával sokkal frissebb állapotot tudunk rekonstruálni.
- **Feltétel:** A biztonsági másolat készítése után történt adatbázis-változásokról keletkező **napló túlélte az eszköz meghibásodását**
- Visszaállítjuk a biztonsági másolatot, majd a napló felhasználásával a mentés óta történt adatbázis-változásokat át tudjuk vezetni az adatbázison.

Az adatbázist a naplóból akkor tudjuk rekonstruálni, ha:

1. **a naplót tároló lemez különbözik az adatbázist tartalmazó lemez(ek)től;**
2. **a naplót sosem dobjuk el az ellenőrzőpont-képzést követően;**
3. **a napló helyrehozó vagy semmisségi/helyrehozó típusú, így az új értékeket (is) tárolja.**

Probléma: A napló esetleg az adatbázisnál is gyorsabban növekedhet, így nem praktikus a naplót örökre megőrizni.



A mentések szintjei

A mentésnek két szintjét különböztetjük meg:

- ***teljes mentés*** (**full dump**), amikor az egész adatbázisról másolat készül;
- ***növekményes mentés*** (**incremental dump**), amikor az adatbázisnak csak azon elemeiről készítünk másolatot, melyek az utolsó teljes vagy növekményes mentés óta megváltoztak.

Helyreállítás: a teljes mentésből és a megfelelő növekményes mentésekből

- **A helyrehozó vagy a semmisségi/helyrehozó naplázás rendszerhiba utáni visszaállítási folyamatához hasonló módszerrel.**
- **Visszamásoljuk a teljes mentést, majd az ezt követő legkorábbi növekményes mentéstől kezdve végrehajtjuk a növekményes mentésekben tárolt változtatásokat.**



Mentés működés közben

Ha leállítjuk a rendszert, akkor nyugodtan lehet menteni.

Probléma:

- **sokáig tarthat a leállítás, újraindítás**
- **nem biztos, hogy egyáltalán le szabad állítani a rendszert**

Megoldás: működés közben mentünk

Példa:

- A, B, C és D értéke az archiválás kezdetekor rendre 1, 2, 3, 4.
- A mentés közben A értéke 5-re, C értéke 6-ra, B értéke 7-re módosul.
- Az adatbáziselemeket a mentéskor sorban másoljuk az archívumba.
- A mentés végére pedig **5, 7, 6, 4** az adatbázis állapota, a **mentett archívumba 1, 2, 6, 4** került, jóllehet ilyen adatbázis-állapot a mentés ideje alatt nem is fordult elő.

Lemez Mentés

A

A := 5

B

C := 6

C

B := 7

D



Mentés működés közben

1. A **<START DUMP>** bejegyzés naplóba írása.
2. A REDO vagy UNDO/REDO naplázási módnak megfelelő **ellenőrzőpont** kialakítása.
3. Az adatlemez(ek) teljes vagy növekményes **mentése**.
4. **A napló mentése.** A mentett naplórész tartalmazza legalább a 2. pontbeli ellenőrzőpont-képzés közben keletkezett naplóbejegyzéseket, melyeknek túl kell élniük az adatbázist hordozó eszköz meghibásodását.
5. **<END DUMP>** bejegyzés naplóba írása.

Megjegyzés: A mentés befejezésekor eldobhatjuk a naplónak azt a részét, amelyre nincs szükség a 2. pontban végrehajtott ellenőrzőpont-képzéshez tartozó helyreállítási folyamat szabályai szerint.

UNDO napló nem használható: Mivel UNDO naplázás esetén az OUTPUT műveletek a módosítási **bejegyzés naplóba írását követően bármikor lefuthatnak**, ezért előfordulhat, olyan eredményt kapunk, mintha egy tranzakció nem atomosan hajtódott volna végre.



Mentés működés közben

Tegyük fel, hogy a fenti adatbázis mentés közbeni módosításait két tranzakció, T1 (mely A-t és B-t módosította) és T2 (mely C-t módosította) végezte, melyek a mentés kezdetekor aktívak voltak. UNDO/REDO naplózási módszert alkalmazva a mentés alatti események lehetséges naplóbejegyzései a következők:

<START DUMP>

<START CKPT(T1,T2)>

<T1,A,1,5>

<T2,C,3,6>

<T2, COMMIT>

<T1,B,2,7>

<END CKPT>

a mentés befejezése

<END DUMP>

Lemez Mentés

A

A := 5

B

C := 6

C

B := 7

D



Helyreállítás mentésből és naplóból

Tegyük fel, hogy a biztonsági mentés elkészítését követően történik katasztrófa, és a napló ezt túlélte. Az érdekesség kedvéért tegyük fel, hogy a napló katasztrófát túlélt részében nincs **<T1, COMMIT>** bejegyzés, van viszont **<T2, COMMIT>**. Az adatbázist először a biztonsági mentésből visszatöltjük, így A, B, C, D értékei rendre 1, 2, 6, 4 lesznek.

<START DUMP>

<START CKPT(T1,T2)>

<T1,A,1,5>

<T2,C,3,6>

<T2, COMMIT>

<T1,B,2,7>

<END CKPT>

a mentés befejezése

<END DUMP>

- T2 befejezett tranzakció, helyreállítjuk azon lépés hatását, amely C értékét 6-ra módosította.
- T1 hatásait semmissé kell tennünk. A értékét 1-re, B értékét 2-re kell visszaállítanunk.

Lemez Mentés

A

A := 5

B

C := 6

C

B := 7

D



Az Oracle naplózási és archiválási rendszere

- Rendszerhiba esetén a **helyreállítás-kezelő automatikusan aktivizálódik**, amikor az Oracle újraindul.
- A helyreállítás a **napló (redo log)** alapján történik. A napló olyan állományok halmaza, amelyek az adatbázis változásait tartalmazzák, akár lemezre kerültek, akár nem. Két részből áll: az **online** és az **archivált naplóból**.
- Az **online napló** kettő vagy több online naplófájlból áll.
- A naplóbejegyzések ideiglenesen az **SGA (System Global Area)** memóriapuffereiben tárolódnak, amelyeket a **Log Writer (LGWR)** háttérfolyamat folyamatosan ír ki lemezre. (Az SGA tartalmazza az adatbáziselemeket tároló puffereket is, amelyeket pedig a **Database Writer** háttérfolyamat ír lemezre.)
- Ha egy felhasználói folyamat befejezte egy tranzakció végrehajtását, akkor a **LGWR** egy **COMMIT** bejegyzést is kiír a naplóba.



Az Oracle naplózási és archiválási rendszere

- Az online **naplófájlok ciklikusan töltődnek föl**. Például ha a naplót két fájl alkotja, akkor először az elsőt írja tele a LGWR, aztán a másodikat, majd újraírja az elsőt stb. Amikor egy naplófájl megtelt, kap egy sorszámot (**log sequence number**), ami azonosítja a fájlt.
- A biztonság növelése érdekében az Oracle lehetővé teszi, hogy a **naplófájlokat több példányban letároljuk**. A ***multiplexelt online naplóban*** ugyanazon naplófájlok több különböző lemezen is tárolódnak, és ezek egyszerre módosulnak. Ha az egyik lemez megsérül, akkor a napló többi másolata még mindig rendelkezésre áll a helyreállításhoz.
- Lehetőség van arra, hogy a megtelt online naplófájlokat archiváljuk, mielőtt újra felhasználnánk őket. Az ***archivált (offline) napló*** az ilyen archivált naplófájlok ból tevődik össze.



Az Oracle naplózási és archiválási rendszere

- A **naplókezelő két módban működhet**: **ARCHIVELOG** módban a rendszer minden megtelt naplófájlt archivál, mielőtt újra felhasználná, **NOARCHIVELOG** módban viszont a legrégebbi megtelt naplófájl mentés nélkül felülíródik, ha az utolsó szabad naplófájl is megtelt.
- **ARCHIVELOG** módban az adatbázis **teljesen visszaállítható rendszerhiba** és **eszközhiba után** is, valamint az adatbázist működés közben is lehet archiválni. Hátránya, hogy az archivált napló kezeléséhez külön adminisztrációs műveletek szükségesek.
- **NOARCHIVELOG** módban az adatbázis **csak rendszerhiba után** állítható vissza, eszközhiba esetén nem, és az adatbázist archiválni csak zárt állapotában lehet, működés közben nem. Előnye, hogy a DBA-nak nincs külön munkája, mivel nem jön létre archivált napló.
- A naplót a **LogMiner naplóelemző eszköz** segítségével analizálhatjuk, amelyet SQL alapú utasításokkal vezérelhetünk.



Az Oracle naplózási és archiválási rendszere

- A helyreállításhoz szükség van még egy **vezérlőfájlra** (control file) is, amely többek között az adatbázis fájlszerkezetéről és a LGWR által **éppen írt naplófájl sorszámáról** tartalmaz információkat. Az automatikus helyreállítási folyamatot a rendszer ezen vezérlőfájl alapján irányítja. Hasonlóan a naplófájlokhoz, a vezérlőfájlt is tárolhatjuk több példányban, amelyek egyszerre módosulnak. Ez a **multiplexelt vezérlőfájl**.



A rollback szegmensek és a helyreállítás folyamata

- Az Oracle az **UNDO** és a **REDO** naplózás egy **speciális keverékét** valósítja meg.
- A tranzakciók hatásainak **semmissé tételehez** szükséges információkat a **rollback szegmensek** tartalmazzák. minden adatbázisban van egy vagy több rollback szegmens, amely a **tranzakciók által módosított adatok régi értékeit tárolja** attól függetlenül, hogy ezek a módosítások lemezre íródtak vagy sem. A rollback szegmenseket használjuk az olvasási konzisztencia biztosítására, a tranzakciók visszagörgetésére és az adatbázis helyreállítására is.
- A rollback szegmens **rollback bejegyzések ből** áll. Egy rollback bejegyzés többek között a **megváltozott blokk azonosítóját** (**fájlsorszám** és a **fájlon belüli blokkazonosító**) és a **blokk régi értékét tárolja**. A rollback bejegyzés mindig előbb kerül a rollback szegmensbe, mint ahogy az adatbázisban megtörténik a módosítás. Az ugyanazon tranzakcióhoz tartozó bejegyzések össze vannak láncolva, így könnyen visszakereshetők, ha az adott tranzakciót vissza kell görgetni.



A rollback szegmensek és a helyreállítás folyamata

- A **rollback szegmenseket** sem a felhasználók, sem az adatbázis-adminisztrátorok nem olvashatják. Mindig a **SYS felhasználó a tulajdonosuk**, attól függetlenül, ki hozta őket létre.
- minden rollback szegmenshez tartozik egy **tranzakciós tábla**, amely azon tranzakciók listáját tartalmazza, amelyek által végrehajtott módosításokhoz tartozó rollback bejegyzések az adott rollback szegmensben tárolódnak. minden rollback szegmens fix számú tranzakciót tud kezelní. Ez a szám az adatblokk méretétől függ, amit viszont az operációs rendszer határoz meg. Ha explicit módon másképp nem rendelkezünk, az Oracle egyenletesen elosztja a tranzakciókat a rollback szegmensek között.



A rollback szegmensek és a helyreállítás folyamata

- Ha egy tranzakció befejeződött, akkor a rá vonatkozó **rollback bejegyzések még nem törölhetők**, mert elképzelhető, hogy még a tranzakció befejeződése előtt elindult egy olyan lekérdezés, amelyhez szükség van a módosított adatok régi értékeire. Hogy a rollback adatok minél tovább elérhetők maradjanak, a rollback szegmensbe a bejegyzések sorban egymás után kerülnek be. Amikor megtelik a szegmens, akkor **az Oracle az elejéről kezdi újra feltölteni**. Előfordulhat, hogy egy sokáig futó tranzakció miatt nem írható felül a szegmens eleje, ilyenkor a szegmenst ki kell bővíteni.
- Amikor létrehozunk egy adatbázist, **automatikusan létrejön egy SYSTEM nevű rollback szegmens** is a SYSTEM táblaterületen. Ez nem törölhető. Erre a szegmensre mindenkor szükség van, akár létrehozunk további rollback szegmenseket, akár nem. Ha több rollback szegmensünk van, akkor a SYSTEM nevűt az Oracle csak speciális rendszertranzakciókra próbálja használni, a felhasználói tranzakciókat pedig szétosztja a többi rollback szegmens között. Ha viszont túl sok felhasználói tranzakció fut egyszerre, akkor a SYSTEM szegmenst is használni fogja erre a célra.



A rollback szegmensek és a helyreállítás folyamata

- **Naplózás naplózása:** Amikor egy rollback bejegyzés a rollback szegmensbe kerül, a naplóban erről is készül egy naplóbejegyzés, hiszen a rollback szegmensek (más szegmensekhez hasonlóan) az adatbázis részét képezik.
 - A helyreállítás szempontjából nagyon fontos a módosításoknak ez a **kétszeres feljegyzése**.
1. Ha rendszerhiba történik, először a napló alapján visszaállításra kerül az **adatbázisnak a rendszerhiba bekövetkezése előtti állapota**, amely inkonzisztens is lehet. Ez a folyamat a ***rolling forward***.
 2. A helyreállítás folyamán a **rollback szegmens is visszaállítódik**, amiben az aktív tranzakciók által végrehajtott tevékenységek semmissé tételehez szükséges információk találhatók. Ezek alapján ezután minden aktív tranzakcióra végrehajtódik egy **ROLLBACK** utasítás. Ez a ***rolling back*** folyamat.



Az archiválás folyamata

- Az eszközhibák okozta problémák megoldására az Oracle is használja az **archiválást**.
- A **teljes mentés** az adatbázishoz tartozó adatfájlok, az online naplófájlok és az adatbázis vezérlőfájljának operációsrendszer-szintű mentését jelenti.
- **Részleges mentés** esetén az adatbázisnak csak egy részét mentjük, például az egy táblaterülethez tartozó adatfájlokat vagy csak a vezérlőfájlt. A részleges mentésnek csak akkor van értelme, ha a naplózás **ARCHIVELOG** módban történik. Ilyenkor a naplót felesleges újra archiválni.
- A mentés lehet **teljes** és **növekményes** is.
- Ha az adatbázist konzisztens állapotában archiváltuk, akkor **konzisztens mentésről** beszélünk. A konzisztens mentésből az adatbázist egyszerűen visszamásolhatjuk, nincs szükség a napló alapján történő helyreállításra.
- Lehetőség van az adatbázist egy régebbi mentésből visszaállítani, majd csak néhány naplóbejegyzést figyelembe véve az adatbázist egy meghatározott időpontbeli állapotába visszavinni. Ezt **nem teljes helyreállításnak** (*incomplete recovery*) nevezzük.



Összefoglalás

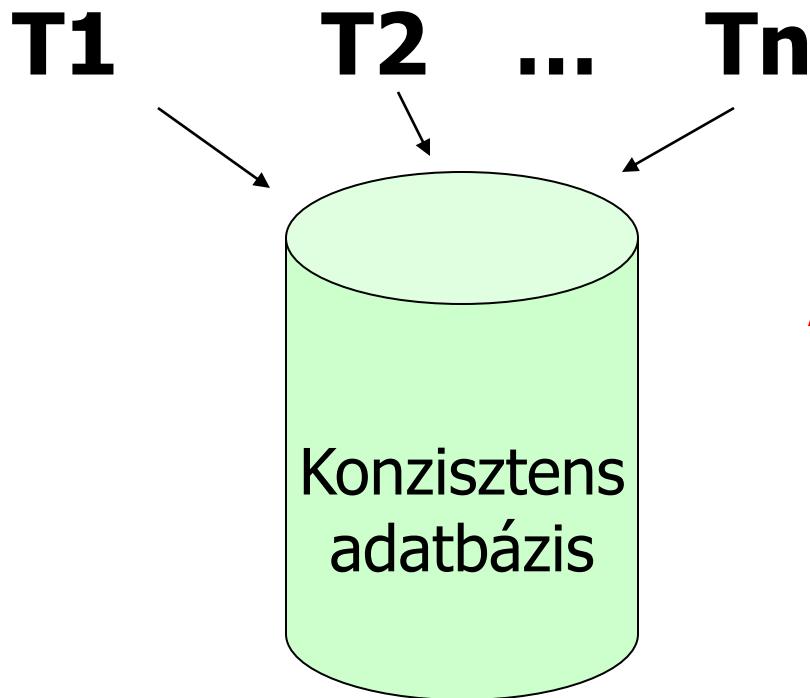
- Eszközök meghibásodása
 - Másolatok
 - Mentés leállással, működés közben
- Oracle naplázása és archiválása
 - Log fájl (REDO napló)
 - Rollback szegmens (UNDO napló)
 - Helyreállítás (rolling forward, rollback)
 - Teljes vagy részleges mentés



Konkurenciavezérlés, sorbarendezhetőség, konfliktus- sorbarendezhetőség



Egyszerre több tranzakció is ugyanazt az adatbázist használja.



Az adatbázisnak
konzisztensnek
kell maradnia!

A tranzakciók közötti egymásra hatás az adatbázis-állapot inkonziszenessé válását okozhatja, még akkor is, amikor a tranzakciók külön-külön megőrzik a konzisztenciát, és rendszerhiba sem történt.



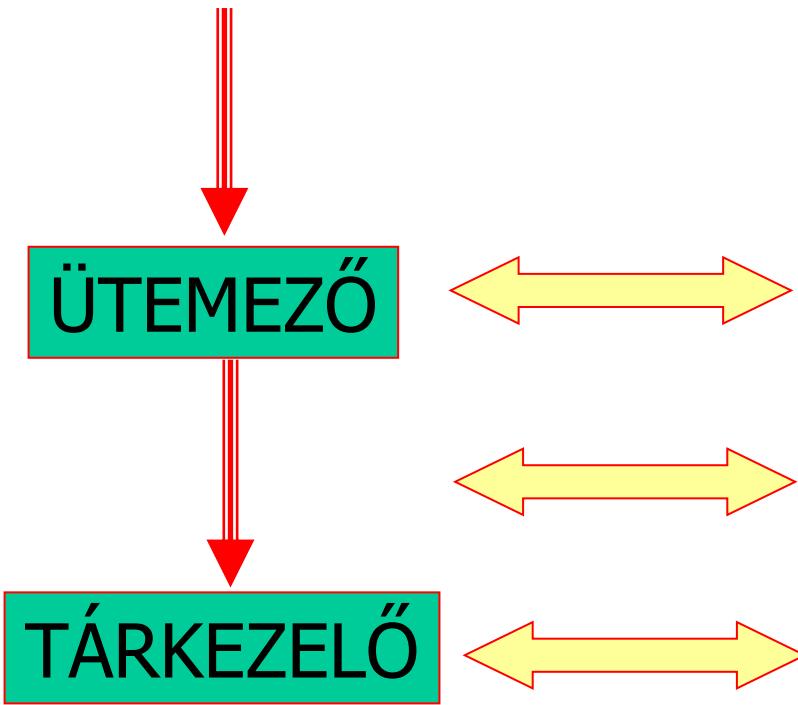
A konkurenciavezérlés

- A tranzakciós lépések szabályozásának 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) nevezük.
- Amint a tranzakciók az adatbáziselemek olvasását és írását kérik, ezek a kérések az ütemezőhöz kerülnek, amely legtöbbször közvetlenül végrehajtja azokat. Amennyiben a szükséges adatbáziselem nincs a pufferben, először a pufferkezelőt hívja meg.
- Bizonyos esetekben azonban nem biztonságos azonnal végrehajtani a kéréseket. Az ütemezőnek ekkor **késleltetnie kell** a kérést, sőt bizonyos esetben **abortálnia kell** a kérést kiadó tranzakciót.
- 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.



Az ütemező és a tárkezelő együttműködése

Kérések a
tranzakcióktól ÍRÁSRA,
OLVASÁSRA



- A konkurenciakezelés szempontjából a lényeges olvasási és írási műveletek a központi memória puffereiben történnek, nem pedig a lemezen. Tehát csak a **READ** és **WRITE** műveletek sorrendje számít, amikor a konkurenciával foglalkozunk, az **INPUT** és **OUTPUT** műveleteket figyelmen kívül hagyjuk.

| 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) |

Konzisztencia:
A=B

**Egymás után
futtatva,
megőrzik a
konzisztenciát.**



Soros ütemezések

A két tranzakciónak két soros ütemezése van, az egyikben
T1 megelőzi T2-t, a másikban T2 előzi meg T1-et

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

Mindkét soros ütemezés konzisztens (**A=B**) adatbázist eredményez, bár
a két ütemezésben a végeredmények különböznek (**250**, illetve **150**).



Sorbarendezhető ütemezések

Az első ütemezés egy **sorbarendezhető**, de nem soros. A hatása megegyezik a **(T₁, T₂) soros ütemezés hatásával**: tetszőleges konzisztens kiindulási állapotra: **A = B = c**-ből kiindulva A-nak is és B-nek is **2(c + 100)** lesz az értéke, tehát a konzisztenciát mindenkor megőrizzük.

| T ₁ | T ₂ | A | B |
|----------------|----------------|-----|---|
| READ(A,t) | | 25 | |
| t := t+100 | | | |
| WRITE(A,t) | | 125 | |
| | READ(A,s) | 125 | |
| | s := s*2 | | |
| | WRITE(A,s) | 250 | |
| READ(B,t) | | 25 | |
| t := t+100 | | | |
| WRITE(B,t) | | 125 | |
| | READ(B,s) | 125 | |
| | s := s*2 | | |
| | WRITE(B,s) | 250 | |

| T ₁ | T ₂ | A | B |
|----------------|----------------|-----|---|
| READ(A,t) | | 25 | |
| t := t+100 | | | |
| WRITE(A,t) | | 125 | |
| | READ(A,s) | 125 | |
| | s := s*2 | | |
| | WRITE(A,s) | 250 | |
| READ(B,s) | | 25 | |
| s := s*2 | | | |
| WRITE(B,s) | | 50 | |
| | READ(B,t) | 50 | |
| | t := t+100 | | |
| | WRITE(B,t) | 150 | |



Sorbarendezhető ütemezések

A második példában szereplő ütemezés **nem sorbarendezhető**. A végeredmény sosem konzisztens $A := 2(A + 100)$, $B := 2B + 100$, így **nem lehet a hatása soros ütemezéssel megegyező**. Az ilyen viselkedést a különböző konkurenciavezérlési technikákkal el kell kerülnünk.

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

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



A tranzakció szemantikájának hatása

Ez az ütemezés **elvileg sorbarendezhető**, de csak T2 speciális aritmetikai művelete (**1-gyel szorzás**) miatt. Az, hogy mivel szorzunk, lehet, hogy egy bonyolult függvény eredményeképpen dől el. Az ütemező csak az írási, olvasási műveleteket figyelve **pesszimista** alapon dönt a sorbarendezhetőségről:

- Ha T tudna A-ra olyan hatással lenni, hogy az adatbázis-állapot inkonzisztensé váljon, akkor T ezt meg is teszi.

A feltevés miatt az ütemező szerint ez az ütemezés **nem lesz sorbarendezhető**.

| T ₁ | T ₂ | A | B |
|----------------|-----------------|-----|----|
| READ(A,t) | | 25 | |
| t := t+100 | | | |
| WRITE(A,t) | | 125 | |
| | READ(A,s) | 125 | |
| | s := <u>s*1</u> | | |
| | WRITE(A,s) | 125 | |
| | READ(B,s) | | 25 |
| | s := <u>s*1</u> | | |
| | WRITE(B,s) | 25 | |
| READ(B,t) | | | 25 |
| t := t+100 | | | |
| WRITE(B,t) | | 125 | |



A tranzakciók és az ütemezések jelölése

- Csak a tranzakciók által végrehajtott **olvasások** és **írások** számítanak!

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

$T_2 : r_2(A); w_2(A); r_2(B); w_2(B);$

- Az ütemezés jelölése:

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

1. Az $r_i(x)$ vagy $w_i(x)$ azt jelenti, hogy a T_i tranzakció olvassa, illetve írja az x adatbáziselementet.
2. Egy T_i tranzakció az i indexű műveletekből álló sorozat.
3. Egy **s ütemezés** olyan műveletek sorozata, amelyben minden T_i tranzakcióra teljesül, hogy T_i műveletei ugyanabban a sorrendben fordulnak elő s-ben, mint a T_i -ben. s a tranzakciók műveleteinek **átlapolása (interleaving)**.



Konfliktus-sorbarendezhetőség

- **Elégséges feltétel:** biztosítja egy ütemezés sorbarendezhetőségét.
- A forgalomban lévő rendszerek ütemezői a tranzakciók sorbarendezhetőségére általában ezt az erősebb feltételt biztosítják, amelyet ***konfliktus-sorbarendezhetőségnak*** nevezünk.
- A ***konfliktus*** (conflict) vagy ***konfliktuspár*** olyan egymást követő műveletpár az ütemezésben, amelynek ha a **sorrendjét felcseréljük**, akkor legalább az egyik tranzakció viselkedése megváltozhat.



Nincs konfliktus

- Legyen T_i és T_j két különböző tranzakció ($i \neq j$).
 1. $r_i(X); r_j(Y)$ sohasem konfliktus, még akkor sem, ha $X = Y$,
 - mivel egyik lépés sem változtatja meg az értékeket.
 2. $r_i(X); w_j(Y)$ nincs konfliktusban, ha $X \neq Y$,
 - mivel T_j írhatja Y -t, mielőtt T_i beolvasta X -et, X értéke ettől ugyanis nem változik. Annaál súlyos hatása T_j -re, hogy T_i olvassa X -et, ugyanis ez nincs hatással arra, hogy milyen értéket ír T_j Y -ba.
 3. $w_i(X); r_j(Y)$ nincs konfliktusban, ha $X \neq Y$,
 - ugyanazért, amiért a 2. pontban.
 4. $w_i(X); w_j(Y)$ nincs konfliktusban, ha $X \neq Y$.



Konfliktus

- **Három esetben nem cserélhetjük fel a műveletek sorrendjét:**

a) $r_i(X)$; $w_i(Y)$ konfliktus,

- mivel egyetlen tranzakción belül a műveletek sorrendje rögzített, és az adatbázis-kezelő ezt a sorrendet nem rendezheti át.

b) $w_i(X)$; $w_j(X)$ konfliktus,

- mivel X értéke az marad, amit T_j számolt ki. Ha felcseréljük a sorrendjüket, akkor pedig X-nek a T_i által kiszámolt értéke marad meg. A pessimista feltevés miatt a T_i és a T_j által kiírt értékek lehetnek különbözőek, és ezért az adatbázis valamelyik kezdeti állapotára különbözni fognak.

c) $r_i(X)$; $w_j(X)$ és $w_i(X)$; $r_j(X)$ is konfliktus.

- Ha átvisszük $w_j(X)$ -et $r_i(X)$ elő, akkor a T_i által olvasott X-beli érték az lesz, amit a T_j kiírt, amiről pedig feltételeztük, hogy nem szükségképpen egyezik meg 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 T_i működését.



Konfliktus-sorbarendezhetőség

- **ELV: nem konfliktusos cserékkel az ütemezést megpróbáljuk soros ütemezéssé átalakítani.** Ha ezt meg tudjuk tenni, akkor az eredeti ütemezés sorbarendezhető 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 **elégséges feltétel** a sorbarendezhetőségre, vagyis egy konfliktus-sorbarendezhető ütemezés sorbarendezhető ütemezés is egyben.
- A konfliktus-sorbarendezhetőség **nem szükséges** ahoz, hogy egy ütemezés sorbarendezhető legyen, mégis általában ezt a feltételt ellenőrzik a forgalomban lévő rendszerek ütemezői, amikor a sorbarendezhetőséget kell biztosítaniuk.



- **Példa.** Legyen az ütemezés a következő:
 $r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B);$
- Azt állítjuk, hogy ez az ütemezés konfliktus-sorbarendezhető. A következő cserékkel 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:
 1. $r_1(A); w_1(A); r_2(A); \underline{w_2(A)}; \underline{r_1(B)}$; $w_1(B); r_2(B); w_2(B);$
 2. $r_1(A); w_1(A); \underline{r_2(A)}; \underline{r_1(B)}$; $w_2(A); w_1(B); r_2(B); w_2(B);$
 3. $r_1(A); w_1(A); r_1(B); r_2(A); \underline{w_2(A)}; \underline{w_1(B)}$; $r_2(B); w_2(B);$
 4. $r_1(A); w_1(A); r_1(B); \underline{r_2(A)}; \underline{w_1(B)}$; $w_2(A); r_2(B); w_2(B);$
 5. $r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B);$



Nem konfliktus-sorbarendezhető ütemezés

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

Ez az ütemezés **nem konfliktus-sorbarendezhető**, ugyanis A-t T1 írja előbb, B-t pedig T2. Mivel sem A írását, sem B írását nem lehet átrendezni, semmilyen módon nem kerülhet T1 összes művelete T2 összes művelete elő, sem fordítva.

| T ₁ | T ₂ | A | B |
|----------------|-----------------|-----|----|
| READ(A,t) | | 25 | |
| t := t+100 | | | |
| WRITE(A,t) | | 125 | |
| | READ(A,s) | 125 | |
| | s := <u>s*1</u> | | |
| | WRITE(A,s) | 125 | |
| | READ(B,s) | | 25 |
| | s := <u>s*1</u> | | |
| | WRITE(B,s) | 25 | |
| READ(B,t) | | 25 | |
| t := t+100 | | | |
| WRITE(B,t) | | 125 | |



Összefoglalás

- Konkurenciakezelés
 - Ütemezés, ütemező feladatai
 - Konzisztens adatbázis eredményező ütemezések
 - Sorbarendezhető (hatása megegyezik a tanzakciók valamelyien sorrendű végrehajtásával)
 - Konfliktus-sorbarendezhető (nem-konfliktusos párok cseréjével el lehet jutni sorbarendezeit ütemezésig)
 - A konfliktus-sorbarendezhetőség elégséges, de nem szükséges feltétel a sorbarendezhetőség igazolásához.



Sorbarendezhető, de nem konfliktus-sorbarendezhető ütemezés

- Tekintsük a T_1 , T_2 és T_3 tranzakciókat és egy **soros** ütemezésüket:

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 T_1 és T_2 minden értéket ír X-be, nincs hatása, ugyanis T_3 felülírja X értékét. Emiatt S_1 és S_2 X-nek is és Y-nak is **ugyanazt az értéket adja**. Mivel S_1 soros ütemezés, és S_2 -nek bármely adatbázis-állapotra ugyanaz a hatása, mint S_1 -nek, ezért **S_2 sorbarendezhető**.
- Ugyanakkor mivel nem tudjuk felcserélni $w_1(X)$ -et $w_2(X)$ -szel, így cseréken keresztül nem lehet S_2 -t valamelyik soros ütemezéssé átalakítani. Tehát S_2 sorbarendezhető, de **nem konfliktus-sorbarendezhető**.



Megelőzési gráfok és teszt a konfliktus-sorbarendezhetőségre

- **Alapötlet:** ha valahol **konfliktusban álló műveletek** szerepelnek S-ben, akkor az ezeket a műveleteket végrehajtó tranzakcióknak **ugyanabban a sorrendben kell előfordulniuk** a konfliktus-ekvivalens 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 ellent egymásnak, akkor találhatunk **konfliktusekvivalens soros ütemezést**. Ha pedig ellentmondanak egymásnak, akkor tudjuk, hogy nincs ilyen **soros ütemezés**.



Megelőzési gráfok és teszt a konfliktus-sorbarendezhetőségre

- Adott a T_1 és T_2 , esetleg további tranzakcióknak egy s ütemezése. Azt mondjuk, hogy T_1 megelőzi T_2 -t, ha van a T_1 -ben olyan A_1 művelet és a T_2 -ben olyan A_2 művelet, hogy
 1. A_1 megelőzi A_2 -t 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.
- Másképpen fogalmazva: A_1 és A_2 konfliktuspárt alkotna, ha szomszédos műveletek lennének. Jelölése: $T_1 <_s T_2$.
- Látható, hogy ezek pontosan azok a feltételek, amikor nem lehet felcserélni A_1 és A_2 sorrendjét. Tehát A_1 az A_2 előtt szerepel bármely s-sel konfliktusekvivalens ütemezésben. Ebből az következik, hogy ha ezek közül az ütemezések közül az egyik soros ütemezés, akkor abban T_1 -nek meg kell előznie T_2 -t.
- Ezeket a megelőzéseket a *megelőzési gráfban* (precedence graph) összegezhetjük. A megelőzési gráf csúcsai az s ütemezés tranzakciói. Ha a tranzakciókat T_i -vel jelöljük, akkor a T_i -nek megfelelő csúcsot az i egész jelöli. Az i csúcsból a j csúcsba akkor vezet irányított él, ha $T_i <_s T_j$.



Lemma

S_1, S_2 konfliktusekvivalens $\Rightarrow \text{gráf}(S_1) = \text{gráf}(S_2)$

Bizonyítás:

Tegyük fel, hogy $\text{gráf}(S_1) \neq \text{gráf}(S_2)$

$\Rightarrow \exists$ olyan $T_i \rightarrow T_j$ él, amely $\text{gráf}(S_1)$ -ben benne van, de $\text{gráf}(S_2)$ -ben nincs benne, (vagy fordítva.)

$\Rightarrow S_1 = \dots p_i(A) \dots q_j(A) \dots$ p_i, q_j

$S_2 = \dots q_j(A) \dots p_i(A) \dots$ **konfliktusos pár**

$\Rightarrow S_1, S_2$ nem konfliktusekvivalens. Q.E.D.



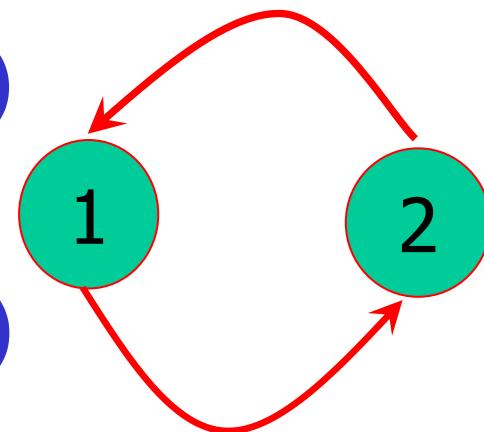
Megjegyzés:

$\text{gráf}(S_1) = \text{gráf}(S_2) \not\Rightarrow S_1, S_2$ konfliktusekvivalens

Ellenpélda:

$S_1 = w_1(A) \ r_2(A) \ w_2(B) \ r_1(B)$

$S_2 = r_2(A) \ w_1(A) \ r_1(B) \ w_2(B)$



Nem lehet semmit sem cserélni!

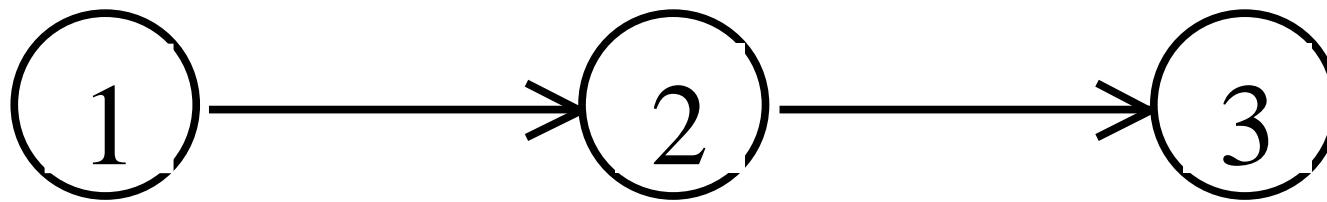


Megelőzési gráfok és teszt a konfliktus-sorbarendezhetőségre

- **Példa.** A következő s ütemezés a T_1 , T_2 és T_3 tranzakciókat tartalmazza:

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

- Az S ütemezéshez tartozó megelőzési gráf a következő:



TESZT: Ha az S megelőzési gráf tartalmaz irányított kört, akkor S nem konfliktus-sorbarendezhető, ha nem tartalmaz irányított kört, akkor S konfliktus-sorbarendezhető, és a csúcsok bármelyik topologikus sorrendje megadja a konfliktusekvivalens soros sorrendet.



Megelőzési gráfok és teszt a konfliktus-sorbarendezhetőségre

- **Egy körmentes gráf csúcsainak *topologikus sorrendje*** a csúcsok bármely olyan rendezése, amelyben minden **a → b** ére az **a** csúcs megelőzi a **b** csúcsot a **topologikus rendezésben**.

S: r₂(A); r₁(B); w₂(A); r₃(A); w₁(B); w₃(A); r₂(B); w₂(B);

- Az **S** ütemezéshez tartozó megelőzési gráf **topologikus sorrendje: (T1, T2, T3)**.



S': r₁(B); w₁(B); r₂(A); w₂(A); r₂(B); w₂(B); r₃(A); w₃(A);

- Hogy lehet S-ből S'-t megkapni szomszédos elemek cseréjével?

1. r₁(B); r₂(A); w₂(A); w₁(B); r₃(A); r₂(B); w₃(A); w₂(B);

2. r₁(B); r₂(A); w₁(B); w₂(A); r₂(B); r₃(A); w₂(B); w₃(A);

3. r₁(B); w₁(B); r₂(A); w₂(A); r₂(B); w₂(B); r₃(A); w₃(A);



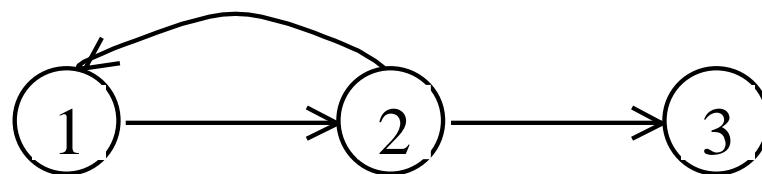
Megelőzési gráfok és teszt a konfliktus-sorrendezhetőségre

- **Példa.** Tekintsük az alábbi két ütemezést:

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

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

- Az $r_2(A) \dots w_3(A)$ miatt $T_2 <_{S_1} T_3$.
- Az $r_1(B) \dots w_2(B)$ miatt $T_1 <_{S_1} T_2$.
- Az $r_2(B) \dots w_1(B)$ miatt $T_2 <_{S_1} T_1$.
- Az S_1 ütemezéshez tartozó megelőzési gráf a következő:



- Ez a gráf nyilvánvalóan **tartalmaz kört**, ezért S_1 nem konfliktus-sorrendezhető, ugyanis láthatjuk, hogy bármely konfliktusekvivalens soros ütemezésben T_1 -nek T_2 előtt is és után is kellene állnia, tehát nem létezik ilyen ütemezés.



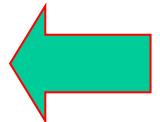
Miért működik a megelőzési gráfon alapuló tesztelés?

→ **Ha van kör a gráfban, akkor cserékkel nem lehet soros ütemezésig eljutni.**

Ha létezik a $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_1$ n darab tranzakcióból álló kör, akkor a feltételezett soros sorrendben T_1 műveleteinek meg kell előzniük a T_2 -ben szereplő műveleteket, amelyeknek meg kell előzniük a T_3 -belieket és így tovább egészen T_n -ig. De T_n műveletei miatt a T_1 -beliek mögött vannak, ugyanakkor meg is kellene előzniük a T_1 -belieket a $T_n \rightarrow T_1$ él miatt. Ebből következik, hogy ha a megelőzési gráf tartalmaz kört, akkor az ütemezés nem konfliktus-sorbarendezhető.



Miért működik a megelőzési gráfon alapuló tesztelés?



Ha nincs kör a gráfban, akkor cserékkel el lehet jutni egy soros ütemezésig.

A bizonyítás az ütemezésben részt vevő tranzakciók száma szerinti indukcióval történik:

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

Indukció: Legyen S a T_1, T_2, \dots, T_n n darab tranzakció műveleteiből álló ütemezés, és S -nek körmentes megelőzési gráfja van.

Ha egy véges gráf körmentes, akkor **van egy olyan csúcsa, amelybe nem vezet él**. Legyen a T_i egy ilyen csúcs.

Mivel az i csomópontba nem vezet él, ezért nincs S -ben olyan A művelet, amely

1. valamelyik T_j ($i \neq j$) tranzakcióra vonatkozik,
2. T_i valamely műveletét megelőzi, és
3. ezzel a művelettesel **konfliktusban van**.

Így T_i minden műveletét S legelejére mozgathatjuk át, miközben megtartjuk a sorrendjüket:

S1: (T_i műveletei) (a többi $n-1$ tranzakció műveletei)



Miért működik a megelőzési gráfon alapuló tesztelés?

Ha nincs kör a gráfban, akkor cserékkel el lehet jutni egy soros ütemezésig.
(Folytatás)

S1: (T_i műveletei) (a többi $n-1$ tranzakció műveletei)

Most tekintsük S1 második részét. Mivel ezek a műveletek egymáshoz viszonyítva ugyanabban a sorrendben vannak, mint ahogyan S-ben voltak, ennek a második résznek a megelőzési gráfját megkapjuk S megelőzési gráfjából, ha elhagyjuk belőle az i csúcsot és az ebből a csúcsból kimenő éleket.

Mivel az eredeti körmentes volt, az elhagyás után is az marad, azaz 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 S-et alakítottuk át olyan soros ütemezéssé, amelyben T_i műveletei állnak legelöl, és a többi tranzakció műveletei ezután következnek valamilyen soros sorrendben. Q.E.D.



Az ütemező eszközei a sorbareendezhetőség elérésére

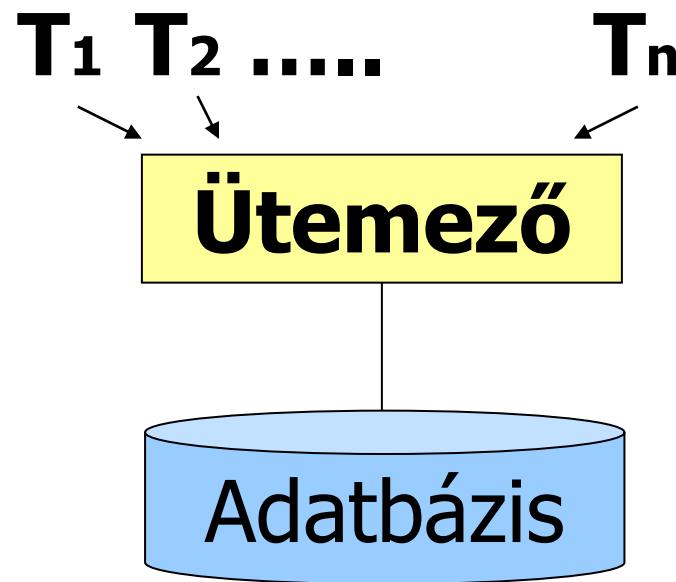
Passzív módszer:

- **hagyjuk a rendszert működni,**
- **az ütemezésnek megfelelő gráfot tároljuk,**
- **egy idő után megnézzük, hogy van-e benne kör,**
- **és ha nincs, akkor szerencsénk volt, jó volt az ütemezés.**



Az ütemező eszközei a sorbarendezetőség elérésére

Aktív módszer: az ütemező beavatkozik, és megakadályozza, hogy kör alakuljon ki.



Az ütemező eszközei a sorbareendezhetőség elérésére

- Az ütemezőnek több lehetősége is van arra, hogy kikényszerítse a sorbareendezhető ütemezéseket:
 1. zárak (ezen belül is még: protokoll elemek, pl. 2PL)
 2. időbélyegek (time stamp)
 3. érvényesítés
- Fő elv: inkább legyen szigorúbb és ne hagyjon lefutni egy olyan ütemezést, ami sorbareendezhető, mint hogy fussen egy olyan, ami nem az.

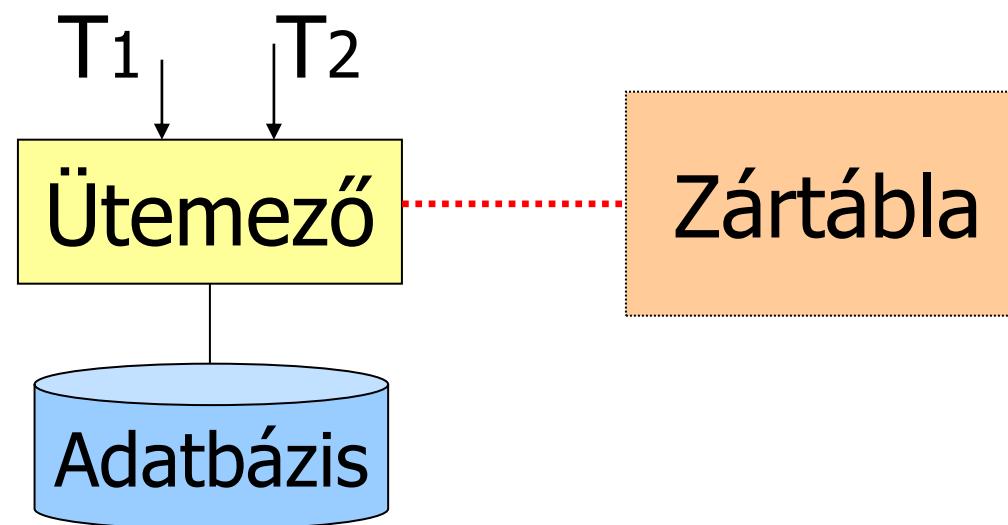


A sorbareendezhetőség biztosítása zárakkal

Két új műveletet vezetünk be:

- **li (A)** : kizárólagos zárolás (exclusive lock)
- **ui (A)**: a zár elengedése (unlock)

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 sorba-rendezhetőség kockázata.



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.



A zárak használata

- Kibővítjük a jelöléseinket a zárolás és a feloldás műveletekkel:
- $I_i(X)$: a T_i tranzakció az X adatbáziselemre *zárolást kér (lock)*.
- $u_i(X)$: a T_i tranzakció az X adatbáziselem *zárolását feloldja (unlock)*.
- Konzisztencia:
- Ha egy T_i tranzakcióban van egy $r_i(X)$ vagy egy $w_i(X)$ művelet, akkor van korábban egy $I_i(X)$ művelet, és van később egy $u_i(X)$ művelet, de a zárolás és az írás/olvasás között nincs $u_i(X)$.

$T_i: \dots I_i(X) \dots r/w_i(X) \dots u_i(X) \dots$



A zárak használata

- **Jogszerűség:**
- Ha egy ütemezésben van olyan $I_i(X)$ művelet, amelyet $I_j(X)$ követ, akkor e két művelet között lennie kell egy $u_i(X)$ műveletnek.

$S = \dots\dots I_i(X) \dots\dots u_i(X) \dots\dots$



nincs $I_j(X)$



A zárak használata

$T_1: l_1(A); r_1(A); A := A+100; w_1(A); u_1(A);$
 $l_1(B); r_1(B); B := B+100; w_1(B); u_1(B);$

$T_2: l_2(A); r_2(A); A := A*2; w_2(A); u_2(A);$
 $l_2(B); r_2(B); B := B*2; w_2(B); u_2(B);$

Mindkét tranzakció konzisztens.



A zárak használata

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

**Ez az ütemezés jogoszerű,
de nem sorba rendezhető.**



A zárolási ütemező

- A zároláson alapuló ütemező feladata, hogy akkor és csak akkor engedélyezze a kérések végrehajtását, ha azok jogoszerű ütemezéseket eredményeznek.
- Ezt a döntést segíti a zártábla, amely minden adatbáziselemhez megadja azt a tranzakciót, ha van ilyen, amelyik pillanatnyilag zárolja az adott elemet.
- A zártábla szerkezete (egyfélé zárolás esetén):
Zárolások(elem, tranzakció) relációt, ahol a T tranzakció zárolja az X adatbáziselementet.



A zárolási ütemező

$T_1: l_1(A); r_1(A); A := A+100; w_1(A); l_1(B);$

$u_1(A); r_1(B); B := B+100; w_1(B); u_1(B);$

$T_2: l_2(A); r_2(A); A := A*2; w_2(A); l_2(B);$

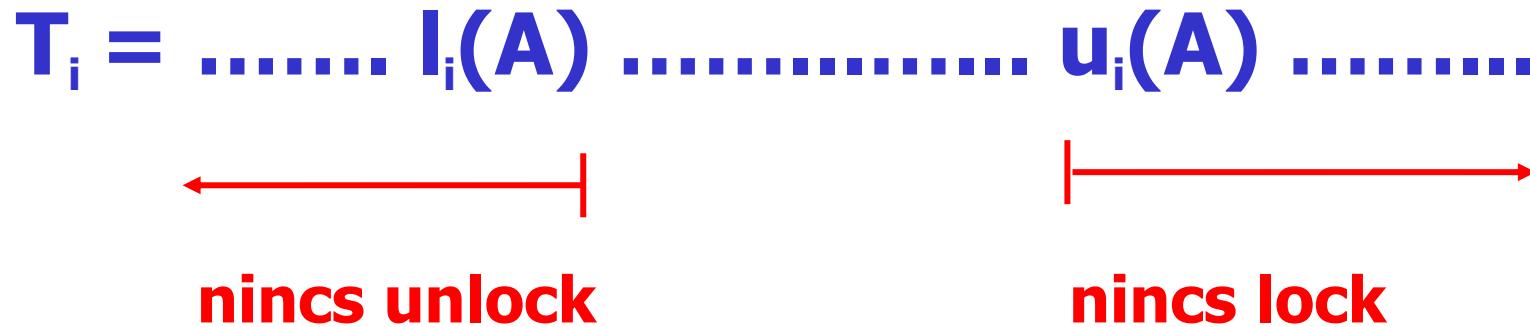
$u_2(A); r_2(B); B := B*2; w_2(B); u_2(B);$

| T_1 | T_2 | A | B |
|---------------------------|-----------------------------|-----|-----|
| $l_1(A); r_1(A);$ | | 25 | |
| $A := A+100;$ | | 125 | |
| $w_1(A); l_1(B); u_1(A);$ | $l_2(A); r_2(A);$ | 125 | |
| | $A := A*2;$ | 250 | |
| | $w_2(A);$ | | |
| | $l_2(B); \text{elutasítva}$ | | |
| $r_1(B); B := B+100;$ | | 25 | |
| $w_1(B); u_1(B);$ | $l_2(B); u_2(A); r_2(B);$ | 125 | |
| | $B := B*2;$ | 125 | |
| | $w_2(B); u_2(B);$ | 250 | |

Mivel T_2 -nek várakoznia kellett, ezért B -t akkor szorozza meg 2-vel, miután T_1 már hozzáadott 100-at, és ez **konzisztens adatbázis-állapotot** eredményez.



A kétfázisú zárolás (two-phase locking, 2PL)



Minden tranzakcióban minden zárolási művelet megelőzi az összes zárfeloldási műveletet.



A kétfázisú zárolás (two-phase locking, 2PL)

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

Ez az ütemezés **jogszerű**,

de **nem sorba rendezhető**.

A tranzakciók nem kétfázisúak!



A kétfázisú zárolás (two-phase locking, 2PL)

$T_1: l_1(A); r_1(A); A := A+100; w_1(A); l_1(B);$

$u_1(A); r_1(B); B := B+100; w_1(B); u_1(B);$

$T_2: l_2(A); r_2(A); A := A*2; w_2(A); l_2(B);$

$u_2(A); r_2(B); B := B*2; w_2(B); u_2(B);$

| T_1 | T_2 | A | B |
|---------------------------|-----------------------------|-----|-----|
| $l_1(A); r_1(A);$ | | 25 | |
| $A := A+100;$ | | 125 | |
| $w_1(A); l_1(B); u_1(A);$ | $l_2(A); r_2(A);$ | 125 | |
| | $A := A*2;$ | 250 | |
| | $w_2(A);$ | | |
| | $l_2(B); \text{elutasítva}$ | | |
| $r_1(B); B := B+100;$ | | 25 | |
| $w_1(B); u_1(B);$ | $l_2(B); u_2(A); r_2(B);$ | 125 | |
| | $B := B*2;$ | 125 | |
| | $w_2(B); u_2(B);$ | 250 | |

Elérhető egy konzisztens adatbázis-állapotot eredményező ütemezés.

A tranzakciók kétfázisúak!



A kétfázisú zárolás (two-phase locking, 2PL)

Tétel: Konzisztens, kétfázisú zárolású tranzakciók bármely S jogoszerű ütemezését át lehet alakítani konfliktusekvivalens soros ütemezéssé.

Bizonyítás: S-ben részt vevő tranzakciók száma (n) szerinti indukcióval.

Megjegyzés:

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 jogoszerű lesz.



A kétfázisú zárolás

Tétel: Konzisztens, kétfázisú zárolású tranzakciók bármely S jogszerű ütemezését át lehet alakítani konfliktusekvivalens soros ütemezéssé.

Bizonyítás:

Alapeset: Ha $n = 1$, azaz egy tranzakcióból áll az ütemezés, akkor az már önmagában soros, tehát biztosan konfliktus-sorbarendevezhető.

Indukció: Legyen S a T_1, T_2, \dots, T_n n darab konzisztens, kétfázisú zárolású tranzakció műveleteiből álló ütemezés, és legyen T_i az a tranzakció, amelyik a teljes S ütemezésben a **legelső zárfeloldási műveletet** végzi, mondjuk $u_i(x) - t$.

Azt állítjuk, hogy T_i **összes olvasási és írási műveletét** az **ütemezés legelejére tudjuk vinni** anélkül, hogy **konfliktusműveleteken** kellene áthaladnunk.



A kétfázisú zárolás

Vegyünk egy konfliktusos párt, $w_i(Y)$ -t és $w_j(Y)$ -t.
(Hasonlóan látható be más konfliktusos párra is.)

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

...; $w_j(Y)$; ...; $u_j(Y)$; ...; $l_i(Y)$; ...; $w_i(Y)$; ...

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

...; $w_j(Y)$; ...; $u_i(X)$; ...; $u_j(Y)$; ...; $l_i(Y)$; ...; $w_i(Y)$; ...

Az $u_i(X)$ művelet állhat $w_j(Y)$ előtt is. Mindkét esetben $u_i(X)$ $l_i(Y)$ előtt van, ami azt jelenti, hogy T_i nem kétfázisú zárolású, amint azt feltételeztük.



A kétfázisú zárolás

Bebizonyítottuk, hogy **S** legelejére lehet vinni **T_i** összes műveletét konfliktusmentes olvasási és írási műveletekből álló műveletpárok cseréjével.

Ezután elhelyezhetjük **T_i** zárolási és zárfeloldási műveleteit. Így **S** a következő alakba írható át:

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

Az n-1 tranzakcióból álló második rész szintén konzisztens 2PL tranzakciókból álló jogoszerű ütemezés, így alkalmazhatjuk rá az indukciós feltevést. Átalakítjuk a második részt konfliktusekvivalens soros ütemezéssé, így a teljes S konfliktus-sorbarendezhetővé vált.

Q.E.D.



A holtpont kockázata

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 : $l_2(B); r_2(B); B := B*2; w_2(B); l_2(A); u_2(B); r_2(A); A := A*2; w_2(A); u_2(A);$

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

Egyik tranzakció sem folytatódhat, hanem örökké várakozniuk kell.

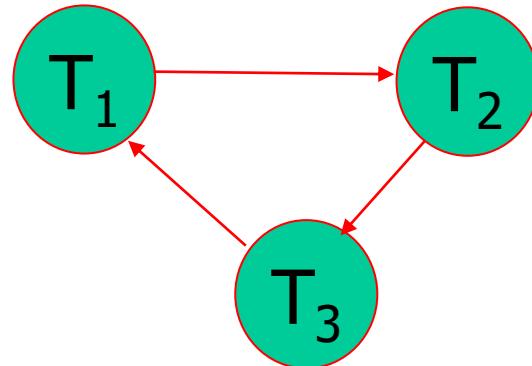
Nem tudjuk mind a két tranzakciót folytatni, ugyanis ha így lenne, akkor az adatbázis végső állapotában nem lehetne $A=B$.



Holtpont felismerése

- A felismerésben segít a zárkérések sorozatához tartozó **várakozási gráf**: csúcsai a tranzakciók és akkor van él T_i -ből 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.
- $l_1(A); l_2(B); l_3(C); l_1(B); l_2(C); l_3(A)$

Az ütemezésnek megfelelő várakozási gráf:



Holtpont felismerése

- **Tétel.** Az ütemezés során egy adott pillanatban pontosan akkor nincs holtpont, ha az adott pillanathoz tartozó várakozási gráfban nincs irányított kör.

Bizonyítás: Ha van irányított kör a várakozási gráfban, akkor a körbeli tranzakciók egyike se tud lefutni, mert vár a mellette levőre. Ez holtpont.

Ha a gráfban nincs irányított kör, akkor van topológikus rendezése a tranzakcióknak és ebben a sorrendben le tudnak futni a tranzakciók.

(Az első nem vár senkire, mert nem megy belőle ki él, így lefuthat; ezután már a másodikba se megy él, az is lefuthat, és így tovább). **Q.E.D.**



Megoldások holtpont ellen

1. Rajzoljuk folyamatosan a várakozási gráfot és ha holtpont alakul ki, akkor **ABORT**-áljuk az egyik olyan tranzakciót, aki benne van a kialakult irányított körben.

Ez egy megengedő megoldás (**optimista**), hagyja az ütemező, hogy mindenki úgy kérjen zárat, ahogy csak akar, de ha baj van, akkor erőszakosan beavatkozik. Az előző példa esetében mondjuk kilövi T_2 -t, ettől lefuthat T_1 , majd T_2 is.

2. Pesszimista hozzáállás: ha hagyjuk, hogy mindenki össze-vissza kérjen zárat, abból baj lehet. Előzzük inkább meg a holtpont kialakulását valahogyan. Lehetőségek:

- (a) minden egyes tranzakció előre elkéri az összes zárat, ami neki kellene fog. Ha nem kapja meg az összeset, akkor egyet se kér el, el se indul.

Ilyenkor biztos nem lesz holtpont, mert ha valaki megkap egy zárat, akkor le is tud futni, nem akad el. Az csak a baj ezzel, hogy előre kell minden tudni.

- (b) Felte tesszük, hogy van egy sorrend az adategységeken és minden egyes tranzakció csak eszerint a sorrend szerint növekvően kérhet újabb zárákat. Itt lehet, hogy lesz várakozás, de holtpont biztos nem lesz. Miért?



Megoldások holtpont ellen

Tegyük fel, hogy T_1, \dots, T_n irányított kört alkot, ahol T_i vár T_{i+1} -re az A_i adatelem miatt.

Ha mindegyik tranzakció betartotta, hogy egyre nagyobb indexű adatelemre kért zárat,
akkor $A1 < A2 < A3 < An < A1$ áll fenn, ami ellentmondás.

Tehát ez a protokoll is megelőzi a holtpontot, de itt is előre kell tudni, hogy milyen zárákat fog kérni egy tranzakció.

Még egy módszer, ami szintén optimista, mint az első:
Időkorlát alkalmazása: ha egy tranzakció kezdete óta túl sok idő telt el, akkor **ABORT**-áljuk.

Ehhez az kell, hogy ezt az időkorlátot jól tudjuk megválasztani.



Éhezés

Másik probléma, ami zárakkal kapcsolatban előfordulhat:

éhezés: többen várnak ugyanarra a zárra, de amikor felszabadul mindenki elviszi valaki a tranzakció orra elől.

Megoldás: adataegységenként FIFO listában tartani a várakozókat, azaz mindenki a legrégebben várakozónak adjuk oda a zárolási lehetőséget.



Különböző zármódú zárolási rendszerek

Probléma: a T tranzakciónak akkor is zárolnia kell az X adatbáziselementet, ha csak olvasni akarja X-et, írni nem.

- Ha nem zárolnánk, akkor esetleg egy másik tranzakció azalatt írna X-be új értéket, mialatt T aktív, ami nem sorba rendezhető viselkedést okoz.
- Másrészről pedig miért is ne olvashatná több tranzakció egyidejűleg X értékét mindaddig, amíg egyiknek sincs engedélyezve, hogy írja.



Osztott és kizárálagos zárak

Mivel ugyanannak az adatbáziselemnek **két olvasási művelete nem eredményez konfliktust**, így ahhoz, hogy az olvasási műveleteket egy bizonyos sorrendbe soroljuk, **nincs szükség zárolásra**.

Viszont szükséges azt az elemet is zárolni, amelyet **olvasunk**, mert **ennek az elemnek az írását nem szabad közben megengednünk**.

Az **íráshoz szükséges zár viszont „erősebb”**, mint az olvasáshoz szükséges zár, mivel ennek mind az olvasásokat, minden írásokat meg kell akadályoznia.



Osztott és kizárálagos zárak

A legelterjedtebb zárolási séma két különböző zárat alkalmaz: az *osztott zárakat* (shared locks) vagy *olvasási zárakat*, és a *kizárálagos zárakat* (exclusive locks) vagy *írási zárakat*.

Tetszőleges X adatbáziselementet vagy **egyszer lehet zárolni kizárálagosan**, vagy **akárhányszor lehet zárolni osztottan**, ha még nincs kizárálagosan zárolva.

Amikor írni akarjuk X-et, akkor X-en kizárálagos zárral kell rendelkeznünk, de ha csak olvasni akarjuk, akkor X-en akár osztott, akár kizárálagos zár megfelel.

Feltételezzük, hogy ha olvasni akarjuk X-et, de írni nem, akkor előnyben részesítjük az osztott zárolást.



Osztott és kizárálagos zárak

Az $sl_i(x)$ jelölést használjuk arra, hogy a T_i tranzakció **osztott zárat kér** az x adatbáziselemre, az $xl_i(x)$ jelölést pedig arra, hogy a T_i **kizárálagos zárat kér x -re**.

Továbbra is $u_i(x)$ -szel jelöljük, hogy T_i feloldja x zárását, vagyis felszabadítja x -et minden zár alól.



A tranzakciók konzisztenciája, a tranzakciók 2PL feltétele és az ütemezések jogoszerűsége

1. Tranzakciók konzisztenciája: Nem írhatunk kizárolagos 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 egy $xl_i(x)$ úgy, hogy közben nincs $u_i(x)$;

$sl_i(x) \dots r_i(x)$ vagy $xl_i(x) \dots r_i(x)$

b) a $w_i(x)$ írási műveletet meg kell, hogy előzze egy $xl_i(x)$ úgy, hogy közben nincs $u_i(x)$.

$xl_i(x) \dots w_i(x)$

Minden zárolást követnie kell egy ugyanannak az elemnek a zárolását feloldó műveletnek.



A tranzakciók konzisztenciája, a tranzakciók 2PL feltétele és az ütemezések jogoszerűsége

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 semmilyen Y -ra.

$sl_i(x) \dots u_i(Y)$

vagy $xl_i(x) \dots u_i(Y)$



A tranzakciók konzisztenciája, a tranzakciók 2PL feltétele és az ütemezések jogoszerűsége

3. **Az ütemezések jogoszerűsége:** 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 $u_i(x)$.

$xl_i(x) \dots u_i(x) \dots xl_j(x)$

vagy $xl_i(x) \dots u_i(x) \dots sl_j(x)$

b) Ha $sl_i(x)$ szerepel egy ütemezésben, akkor ezután nem következhet $xl_j(x)$ valamely i -től különböző j -re anélkül, hogy közben ne szerepelne $u_i(x)$.

$sl_i(x) \dots u_i(x) \dots xl_j(x)$



A tranzakciók konzisztenciája, a tranzakciók 2PL feltétele és az ütemezések jogszerűsége

Megjegyzések:

1. Az engedélyezett, hogy egy tranzakció ugyanazon elemre kérjen és tartson mind osztott, mind kizárolagos zárat, feltéve, hogy ezzel nem kerül konfliktusba más tranzakciók zárolásaival. $sl_i(x) \dots xl_i(x)$
2. Ha a tranzakciók előre tudnák, milyen zárakra lesz szükségük, akkor biztosan csak a kizárolagos zárolást kérnék, de ha nem láthatók előre a zárolási igények, lehetséges, hogy egy tranzakció osztott és kizárolagos zárakat is kér különböző időpontokban.



A tranzakciók konzisztenciája, a tranzakciók 2PL feltétele és az ütemezések jogoszerűsége

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 : $sl_2(A)$; $r_2(A)$; $sl_2(B)$; $r_2(B)$; $u_2(A)$; $u_2(B)$;

| 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 | $u_2(A)$; |
| | $u_2(B)$; |
| $xl_1(B)$; $r_1(B)$; $w_1(B)$; | |
| $u_1(A)$; $u_1(B)$; | |

Az ütemezés konfliktus-sorbarendezhető.

A konfliktusekvivalens soros sorrend a (T_2, T_1) , hiába kezdődött T_1 előbb.



A tranzakciók konzisztenciája, a tranzakciók 2PL feltétele és az ütemezések jogoszerűsége

Tétel: Konzisztens 2PL tranzakciók jogoszerű ütemezése konfliktus-sorbarendezhető.

Bizonyítás: Ugyanazok a meggondolások alkalmazhatók az osztott és kizárolagos zárakra is, mint korábban. **Q.E.D.**

Megjegyzés: Az ábrán T_2 előbb old fel zárat, mint T_1 , így azt várjuk, hogy T_2 megelőzi T_1 -et a soros sorrendben. Megvizsgálva az olvasási és írási műveleteket, észrevehető, hogy $r_1(A)$ -t T_2 összes műveletén át ugyan hátra tudjuk cserélgetni, de $w_1(B)$ -t nem tudjuk $r_2(B)$ előtt vinni, ami pedig szükséges lenne ahhoz, hogy T_1 megelőzze T_2 -t egy konfliktusekvivalens soros ütemezésben.



Kompatibilitási mátrixok

- A **kompatibilitási mátrix** minden egyes zármódhoz rendelkezik egy-egy sorral és egy-egy oszloppal.
- A sorok egy másik tranzakció által az X elemre elhelyezett záraknak, az oszlopok pedig az X-re kért zármódoknak felelnek meg.
- **A kompatibilitási mátrix használatának szabálya:**
Egy A adatbáziselemre C módú zárat akkor és csak akkor engedélyezhetünk, ha a táblázat minden olyan R sorára, amelyre más tranzakció már zárolta A-t R módban, a C oszlopban „igen” szerepel.
- **Az osztott (S) és kizárlagos (X) zárak kompatibilitási mátrixa:**

| | | S | X | Megkaphatjuk-e ezt a típusú zárat? |
|-----------------------------|---|------|-----|------------------------------------|
| Ha ilyen zár van már kiadva | S | igen | nem | Megkaphatjuk-e ezt a típusú zárat? |
| | X | nem | nem | |



Összefoglalás

- Konfliktus-sorbarendezhetőség tesztelése megelőzési gráf alapján
- Passzív módszer
- Aktív módszer
 - Zárolás
 - Időbélyezés
 - Érvényesítés
- Zárolási ütemező
- Tranzakció (konzisztens, 2PL) + Ütemezés (jogszerű)
-> sorbarendezhetőség
- Holtpont (felismerés várakozási gráffal, további megoldások),
Kiéheztetés (FIFO lista)
- Osztott és kizárólagos zárak
- Tranzakció (konzisztens, 2PL) + Ütemezés (jogszerű)
-> sorbarendezhetőség
- Kompatibilitási mátrix



Kompatibilitási mátrixok használata

1. A mátrix alapján dönti el az ütemező, hogy egy ütemezés/zárkérés legális-e, illetve ez alapján várakoztatja a tranzakciókat. Minél több az Igen a mátrixban, annál kevesebb lesz a várakoztatás.
2. A mátrix alapján keletkező várakozásokhoz elkészített várakozási gráf segítségével az ütemező kezeli a holtpontot.
3. A mátrix alapján készíti el az ütemező a megelőzési gráfot egy zárkérés-sorozathoz:
 - a megelőzési gráf csúcsai a tranzakciók és akkor van él T_i -ből T_j -be, ha van olyan A adategység, amelyre az ütemezés során Z_k zárat kért és kapott T_i , ezt elengedte, majd
 - ezután A -ra legközelebb T_j kért és kapott olyan Z_l zárat, hogy a mátrixban a Z_k sor Z_l oszlopában Nem áll.

Vagyis olyankor lesz él, ha a két zár nem kompatibilis egymással, nem mindegy a két művelet sorrendje.

..... $T_i:Z_k(A)$ $T_i:UZ_k(A)$ $T_j:Z_l(A)...$

nem kompatibilis



Kompatibilitási mátrixok használata

- A sorbarendezhetőséget a megelőzési gráf segítségével lehet eldönteni.
- Tétel. *Egy csak zárkéréseket és zárelengedéseket tartalmazó jogoszerű ütemezés sorbarendezhető akkor és csak akkor, ha a kompatibilitási mátrix alapján felrajzolt megelőzési gráf nem tartalmaz irányított kört.*

Bizonyítás:

Ha van irányított kör, akkor ekvivalens soros ütemezésben is ugyanebben a sorrendben következnének a körben szereplő tranzakciók, ami ellentmondás.

A másik irány indukcióval. A topológikus sorrendben első tranzakció lépésein előre mozgathatjuk, a hatás nem változik, és a maradék tranzakciók indukció miatt ekvivalensek egy soros ütemezéssel. **Q.E.D.**



A zárakra vonatkozó megelőzési gráf körmentességi feltétel erőssége

Tekintsük az

$l_1(A); r_1(A); u_1(A); l_2(A); r_2(A); u_2(A); l_1(A); w_1(A); u_1(A); l_2(B); r_2(B); u_2(B)$

ütemezést.

Ha megnézzük az írás/olvasás műveleteket ($r_1(A); r_2(A); w_1(A); r_2(B)$), akkor látszik, hogy az ütemezés hatása azonos a $T_2 T_1$ soros ütemezés hatásával, vagyis ez egy sorbarendezhető ütemezés zárak nélkül.

De ha felrajzoljuk a zárakra vonatkozó megelőzési gráfot (és ilyenkor persze nem nézzük, hogy milyen írások/olvasások vannak, hanem a legrosszabb esetre készülünk), akkor



lesz a gráf, és mivel ez irányított kört tartalmaz, akkor ezt elvetnénk, mert nem lesz sorbarendezhető az az ütemezés, amiben már csak a zárak vannak benne.



A sorbarendezhetőség biztosítása tetszőleges zármodellben

Az ütemező egyik lehetősége a sorbarendezhetőség elérésére, hogy folyamatosan figyeli a megelőzési gráfot és ha irányított kör keletkezne, akkor **ABORT**-ot rendel el.

Másik lehetőség a protokollal való megelőzés. Tetszőleges zármodellben értelmes a 2PL és igaz az alábbi téTEL:

TÉTEL. *Ha valamilyen zármodellben egy jogoszerű ütemezésben minden tranzakció követi a 2PL-t, akkor az ütemezéshez tartozó megelőzési gráf nem tartalmaz irányított kört, azaz az ütemezés sorbarendezhető.*

Bizonyítás: Az előzőekhez hasonlóan.

Megjegyzés: Minél gazdagabb a zármodell, minél több az IGEN a kompatibilitási mátrixban, annál valószínűbb, hogy a megelőzési gráfban nincs irányított kör, minden külön protokoll nélkül. Ez azt jelenti, ilyenkor egyre jobb lesz az **ABORT**-os módszer (azaz ritkán kell visszagörgetni egy tranzakciót).



Zárak felminősítése

- L2 erősebb L1-nél, ha a kompatibilitási mátrixban L2 sorában /oszlopában minden olyan pozícióban „NEM” áll, amelyben L1 sorában /oszlopában „NEM” áll.
- Például az SX zárolási séma esetén X erősebb S-nél (X minden zármódnál erősebb, hiszen X sorában és oszlopában is minden pozíción „NEM” szerepel).

| | | |
|---|------|-----|
| | S | X |
| S | IGEN | NEM |
| X | NEM | NEM |

- Azt mondjuk, hogy a T tranzakció felminősíti (upgrade) az L1 zárját az L1-nél erősebb L2 zárra az A adatbáziselemen, ha L2 zárat kér (és kap) A-ra, amelyen már birtokol egy L1 zárat (azaz még nem oldotta fel L1-et). Például: sl_i(A).....XI_i(A)



Zárak felminősítése

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 : $sl_2(A)$; $r_2(A)$; $sl_2(B)$; $r_2(B)$; $u_2(A)$; $u_2(B)$;

| 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 | |
| | $sl_1(B)$; $r_1(B)$; |
| | $xl_1(B)$; elutasítva |
| $xl_1(B)$; $r_1(B)$; | |
| $w_1(B)$; | |
| $u_1(A)$; $u_1(B)$; | |

| 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 | |
| | $u_2(A)$; $u_2(B)$; |
| $xl_1(B)$; $w_1(B)$; | |
| $u_1(A)$; $u_1(B)$; | |

Felminősítés nélkül

A T_1 tranzakció T_2 -vel konkurensen tudja végrehajtani az írás előtti, esetleg hosszadalmas számításait, amely nem lenne lehetséges, ha T_1 kezdetben kizárólagosan zárolta volna B-t.

Felminősítéssel



Új típusú holtpontok felminősítés esetén

| T_1 | T_2 |
|------------------------|------------------------|
| $sl_1(A)$; | $sl_2(A)$; |
| $xl_1(A)$; elutasítva | $xl_2(A)$; elutasítva |

Egyik tranzakció végrehajtása sem folytatódhat:

- vagy mindkettőnek örökösen kell várakoznia,
- vagy addig kell várakozniuk, amíg a rendszer fel nem fedezzi, hogy holtpont alakult ki, abortálja valamelyik tranzakciót, és a másiknak engedélyezi A-ra a kizárolagos zárat.
- **Megoldás: módosítási zárak**



Módosítási zárak

- Az $ul_i(x)$ módosítási zár a T_i tranzakciónak csak x olvasására ad jogot, x írására nem. Később azonban csak a módosítási zárat lehet felminősíteni írásira, az olvasási zárat nem (azt csak módosításra).
- A módosítási zár tehát nem csak a holt pontproblémát oldja meg, hanem a kiéheztetés problémáját is.

| | S | X | U | |
|---|------|-----|------|-------------------|
| S | igen | nem | igen | NEM SZIMMETRIKUS! |
| X | nem | nem | nem | |
| U | nem | nem | nem | |

Az U módosítási zár úgy néz ki, mintha osztott zár lenne, amikor kérjük, és úgy néz ki, mintha kizárolagos zár lenne, amikor már megvan.



Módosítási zárak

T_1 : $\text{ul}_1(A)$; $\text{r}_1(A)$; $\text{xl}_1(A)$; $\text{w}_1(A)$; $\text{u}_1(A)$;

T_2 : $\text{ul}_2(A)$; $\text{r}_2(A)$; $\text{xl}_2(A)$; $\text{w}_2(A)$; $\text{u}_2(A)$;

T_1

$\text{ul}_1(A)$; $\text{r}_1(A)$;

T_2

$\text{ul}_2(A)$; elutasítva

$\text{xl}_1(A)$; $\text{w}_1(A)$; $\text{u}_1(A)$;

$\text{ul}_2(A)$; $\text{r}_2(A)$;

$\text{xl}_2(A)$; $\text{w}_2(A)$; $\text{u}_2(A)$;

Nincs holtPont!



Növelési zárak

- Az **INC (A, c)** művelet:
READ (A, t) ; t := t+c; WRITE (A, t) ;
műveletek atomi végrehajtása, ahol c egy konstans.
- Tetszőleges sorrendben kiszámíthatók.
- A növelés nem cserélhető fel sem az olvasással, sem az írással.
 - Ha azelőtt vagy azután olvassuk be A-t, hogy valaki növelte, különböző értékeket kapunk,
 - és ha azelőtt vagy azután növeljük A-t, hogy más tranzakció új értéket írt be A-ba, akkor is különböző értékei lesznek A-nak az adatbázisban.
- Az **inc_i (x)** művelet:
a T_i tranzakció megnöveli az x adatbáziselementet valamely konstanssal.
(Annak, hogy pontosan mennyi ez a konstans, nincs jelentősége.)
- A műveletnek megfelelő **növelési zárat** (increment lock) **il_i (x)**-szel jelöljük.



Növelési zárak

| | S | X | I |
|---|------|-----|------|
| S | igen | nem | nem |
| X | nem | nem | nem |
| I | nem | nem | igen |

- a) **Egy konzisztens tranzakció csak akkor végezheti el x-en a növelési műveletet, ha egyidejűleg növelési zárat tart fenn rajta. A növelési zár viszont nem teszi lehetővé sem az olvasási, sem az írási műveleteket.** $il_i(X) \dots inc_i(X)$
- b) **Az $inc_i(X)$ művelet konfliktusban áll $r_j(X)$ -szel és $w_j(X)$ -szel is $j \neq i$ -re, de nem áll konfliktusban $inc_j(X)$ -szel.**
- c) **Egy jogoszerű ütemezésben bármennyi tranzakció bármikor fenntarthat x-en növelési zárat. Ha viszont egy tranzakció növelési zárat tart fenn x-en, akkor egyidejűleg semelyik más tranzakció sem tarthat fenn sem osztott, sem kizárolagos zárat x-en.**



Növelési zárak

$T_1 : sl_1(A); r_1(A); il_1(B); inc_1(B); u_1(A); u_1(B);$

$T_2 : sl_2(A); r_2(A); il_2(B); inc_2(B); u_2(A); u_2(B);$

T_1

T_2

$sl_1(A); r_1(A);$

$sl_2(A); r_2(A);$

$il_2(B); inc_2(B);$

$il_1(B); inc_1(B);$

$u_2(A); u_2(B);$

$u_1(A); u_1(B);$

Az ütemezőnek egyik kérést sem kell késleltetnie!

Zárak nélkül: **S: $r_1(A); r_2(A); inc_2(B); inc_1(B);$**

Az utolsó művelet nincs konfliktusban az előzőekkel, így előre hozható a második helyre, így a T_1, T_2 soros ütemezést kapjuk:

S1: $r_1(A); inc_1(B); r_2(A); inc_2(B);$

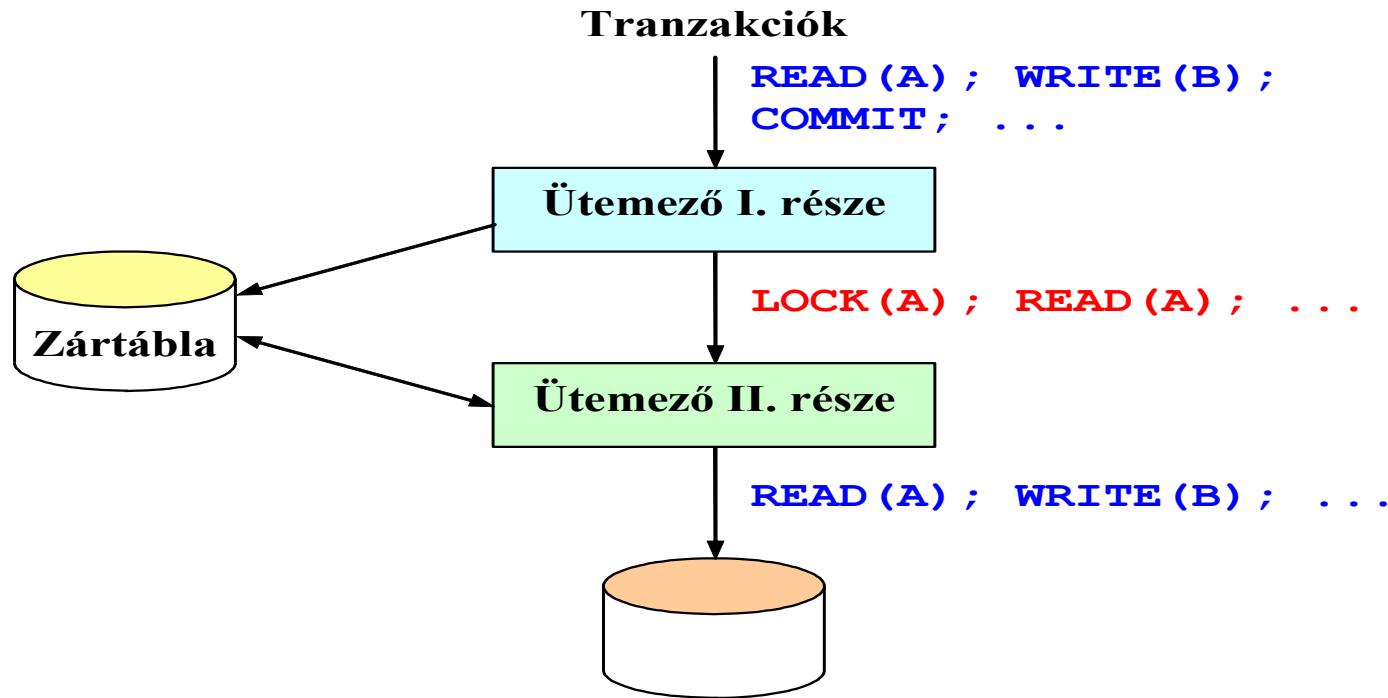


A zárolási ütemező felépítése

1. Maguk a **tranzakciók nem kérnek zárakat**, vagy figyelmen kívül hagyjuk, hogy ezt teszik. Az **ütemező szűrja be a zárolási műveleteket** 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.



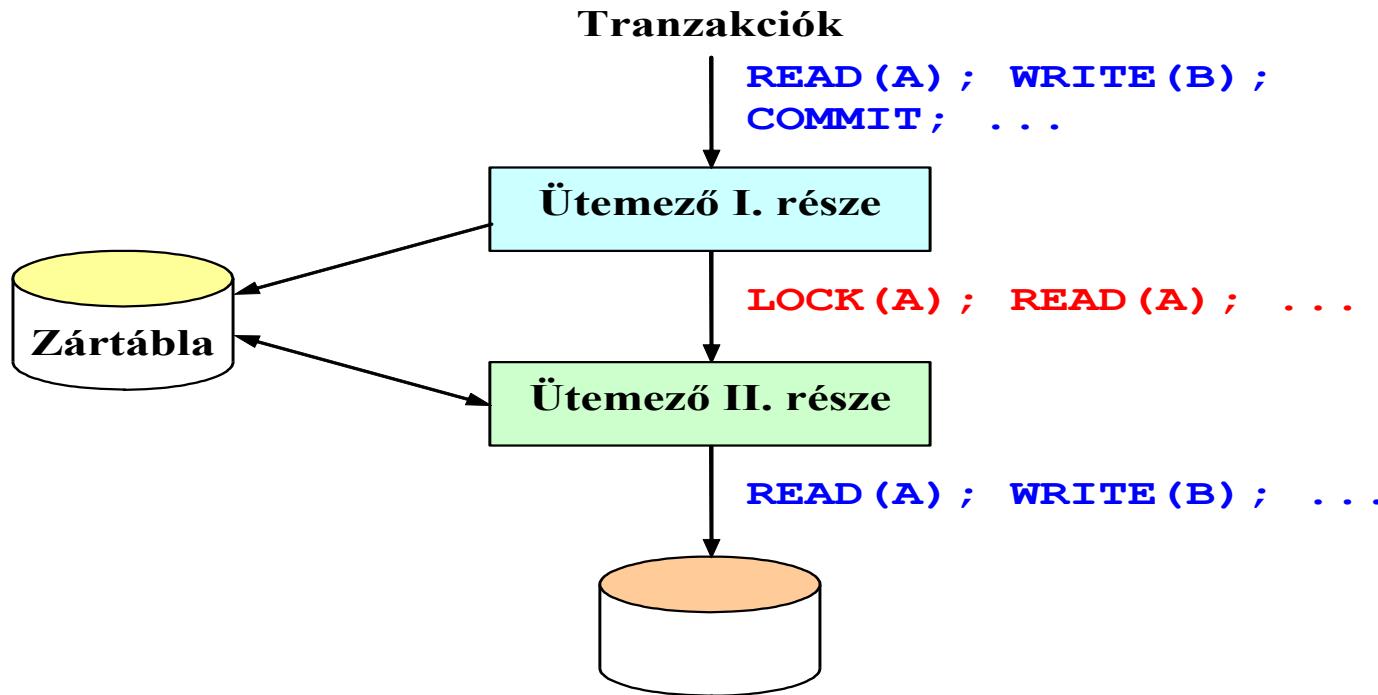
Zárolási műveleteket beszúró ütemező



1. Az I. rész fogadja a tranzakciók által generált kérések sorát, és minden adatbázis-hozzáférési művelet elő beszúrja a megfelelő zárolási műveletet. Az adatbázis-hozzáférési és zárolási műveleteket ezután átküldi a II. részhez (a **COMMIT** és **ABORT** műveleteket nem).



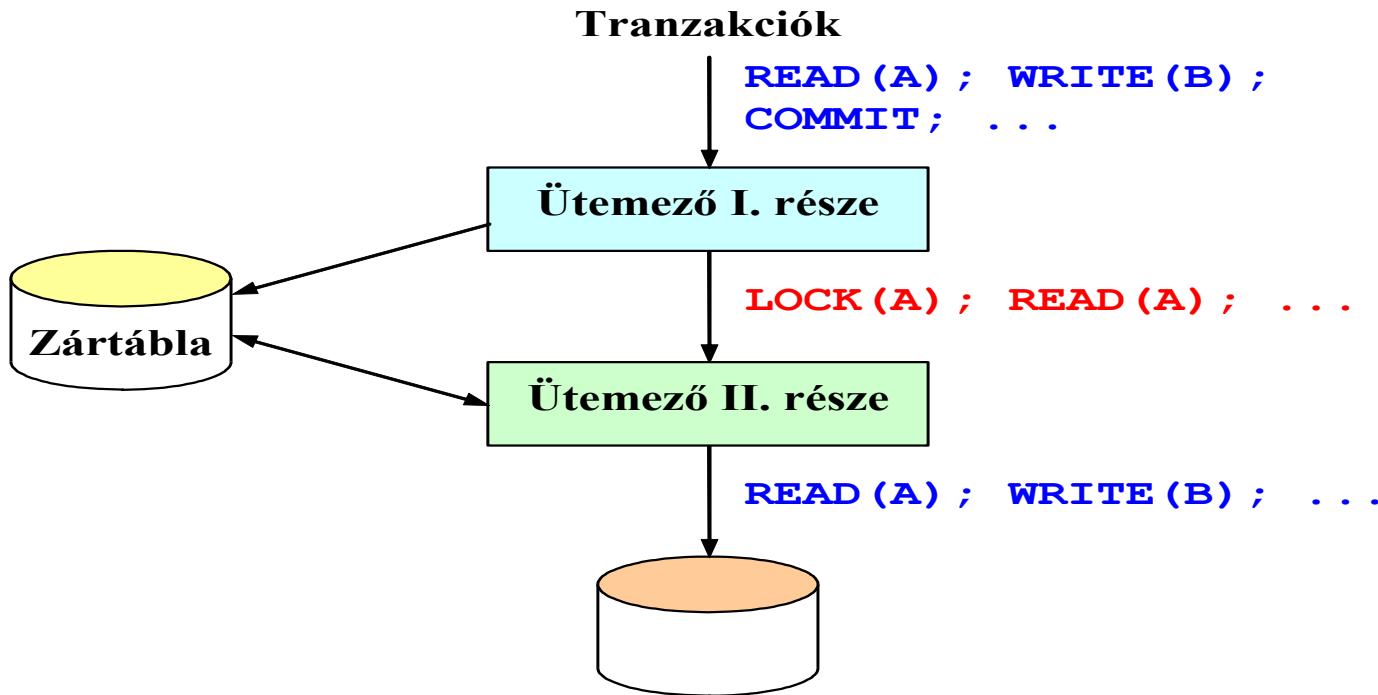
Zárolási műveleteket beszúró ütemező



2. A II. rész fogadja az I. részen keresztül érkező zárolási és adatbázis-hozzáférési műveletek sorozatát. Eldönti, hogy a T tranzakció késleltetett-e (mivel zárolásra vár). Ha igen, akkor magát a műveletet késlelteti, azaz hozzáadja azoknak a műveleteknek a listájához, amelyeket a T tranzakciónak még végre kell hajtania. Ha T nem késleltetett, vagyis az összes előzőleg kért zár már engedélyezve van, akkor megnézi, hogy milyen műveletet kell végrehajtania.



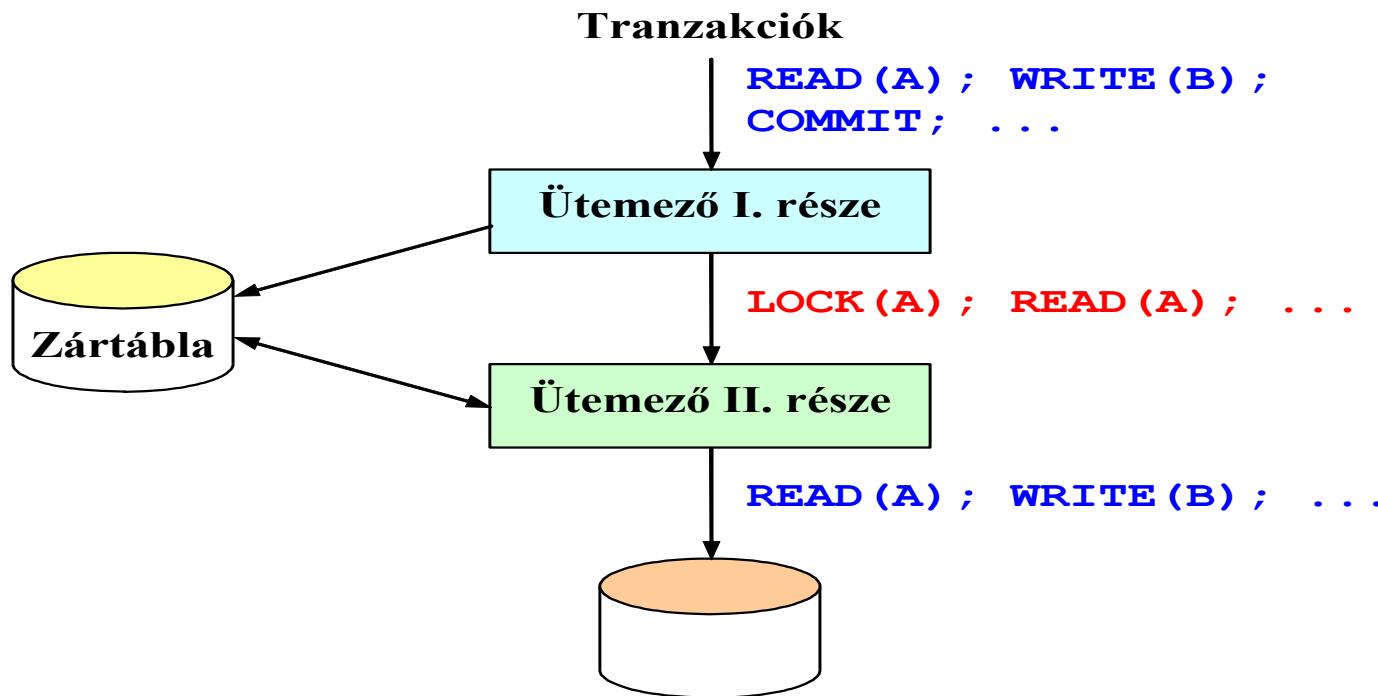
Zárolási műveleteket beszúró ütemező



- a) Ha a művelet adatbázis-hozzáférés, akkor továbbítja az adatbázishoz, és végrehajtja.
- b) Ha zárolási művelet érkezik, akkor megvizsgálja a zártáblát, hogy a zár engedélyezhető-e. Ha igen, akkor úgy módosítja a zártáblát, hogy az az éppen engedélyezett zárat is tartalmazza. Ha nem, akkor egy olyan bejegyzést készít a zártáblában, amely jelzi a zárolási kérést. Az ütemező II. része ezután késlelteti a T tranzakció további műveleteit mindaddig, amíg nem tudja engedélyezni a zárat.



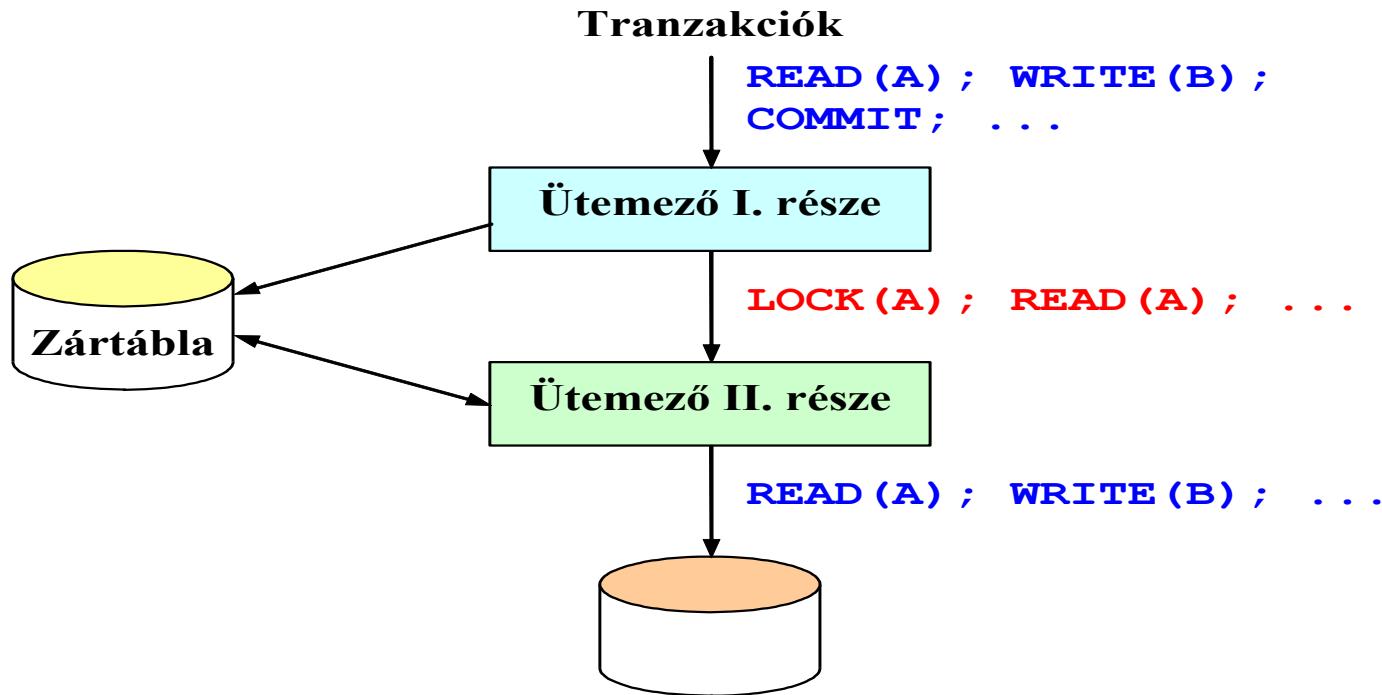
Zárolási műveleteket beszúró ütemező



3. Amikor a T tranzakciót véglegesítjük vagy abortáljuk, akkor a tranzakciókezelő **COMMIT**, illetve **ABORT** műveletek küldésével értesíti az I. részt, hogy oldja fel az összes T által fenntartott zárat. Ha bármelyik tranzakció várakozik ezen zárfeloldások valamelyikére, akkor az I. rész értesíti a II. részt.



Zárolási műveleteket beszúró ütemező



4. Amikor a II. rész értesül, hogy egy X adatbáziselemen felszabadult egy zár, akkor eldönti, hogy melyik az a tranzakció, vagy melyek azok a tranzakciók, amelyek megkapják a zárat X-re. A tranzakciók, amelyek megkapták a zárat, a késleltetett műveleteik közül annyit végrehajtanak, amennyit csak végre tudnak hajtani mindaddig, amíg vagy befejeződnek, vagy egy másik olyan zárolási kéréshez érkeznek el, amely nem engedélyezhető.



Zárolási műveleteket beszúró ütemező

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



Zárolási műveleteket beszúró ütemező

- Amikor többmódú zárak vannak, az ütemezőnek szüksége lehet arra (például felminősítésnél), hogy azonnal értesüljön, milyen későbbi műveletek fognak előfordulni ugyanazon az adatbáziselemen.

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

T_2 : $r_2(A)$; $r_2(B)$;

Felminősítés módosítási zárral:

T_1

T_2

$sl_1(A)$; $r_1(A)$;

$sl_2(A)$; $r_2(A)$;

$sl_2(B)$; $r_2(B)$;

$ul_1(B)$; $r_1(B)$;

$xl_1(B)$; elutasítva

$u_2(A)$; $u_2(B)$;

$xl_1(B)$; $w_1(B)$;

$u_1(A)$; $u_1(B)$;

Az ütemező I. része a következő műveletsorozatot adja ki:

$sl_1(A)$; $r_1(A)$;

$sl_2(A)$; $r_2(A)$; $sl_2(B)$; $r_2(B)$;

$ul_1(B)$; $r_1(B)$

$xl_1(B)$; $w_1(B)$

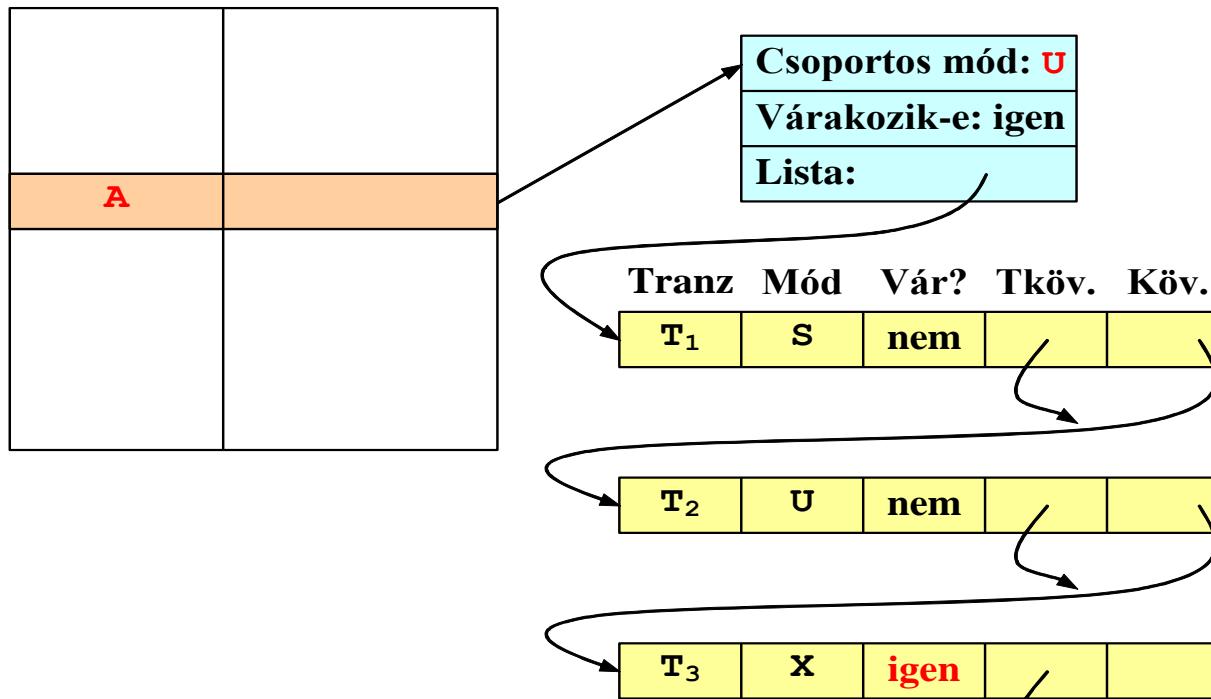
A II. rész viszont nem teljesítheti az $xl_1(B)$ -t, várakoztatja T_1 -et.

Végül T_2 végrehajtja a véglegesítést, és az I. rész feloldja a zárakat A-n és B-n. Ugyanekkor felfedezi, hogy T_1 várakozik B zárolására. Értesíti a II. részt, amely az $xl_1(B)$ zárolást most már végrehajthatónak találja. Beviszi ezt a zárat a zártáblába, és folytatja T_1 tárolt műveleteinek a végrehajtását mindaddig, ameddig tudja. Esetünkben T_1 befejeződik.



A zártábla

Adatbáziselem Zárolási információk

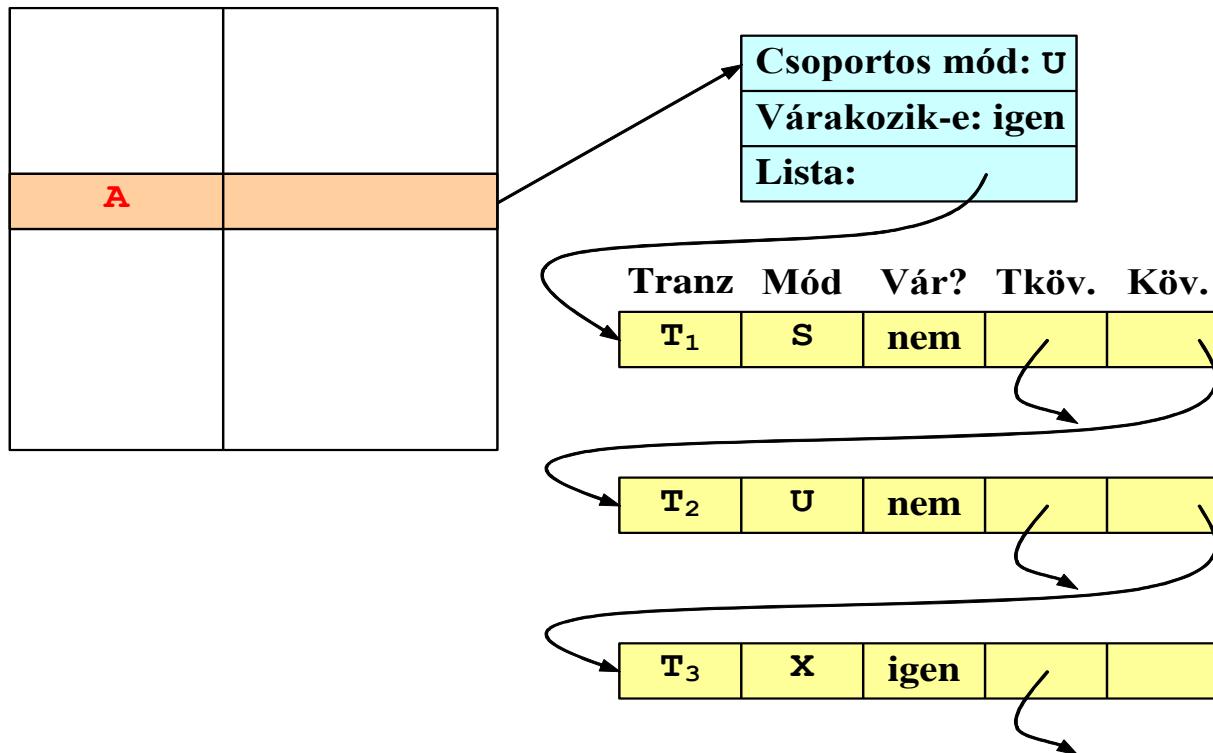


Csoportos mód az adatelemre kiadott legerősebb zár:

- a) **S** azt jelenti, hogy csak osztott zárak vannak;
- b) **U** azt jelenti, hogy egy módosítási zár van, és lehet még egy vagy több osztott zár is;
- c) **X** azt jelenti, hogy csak egy kizártlagos zár van, és semmilyen más zár nincs.

A zártábla

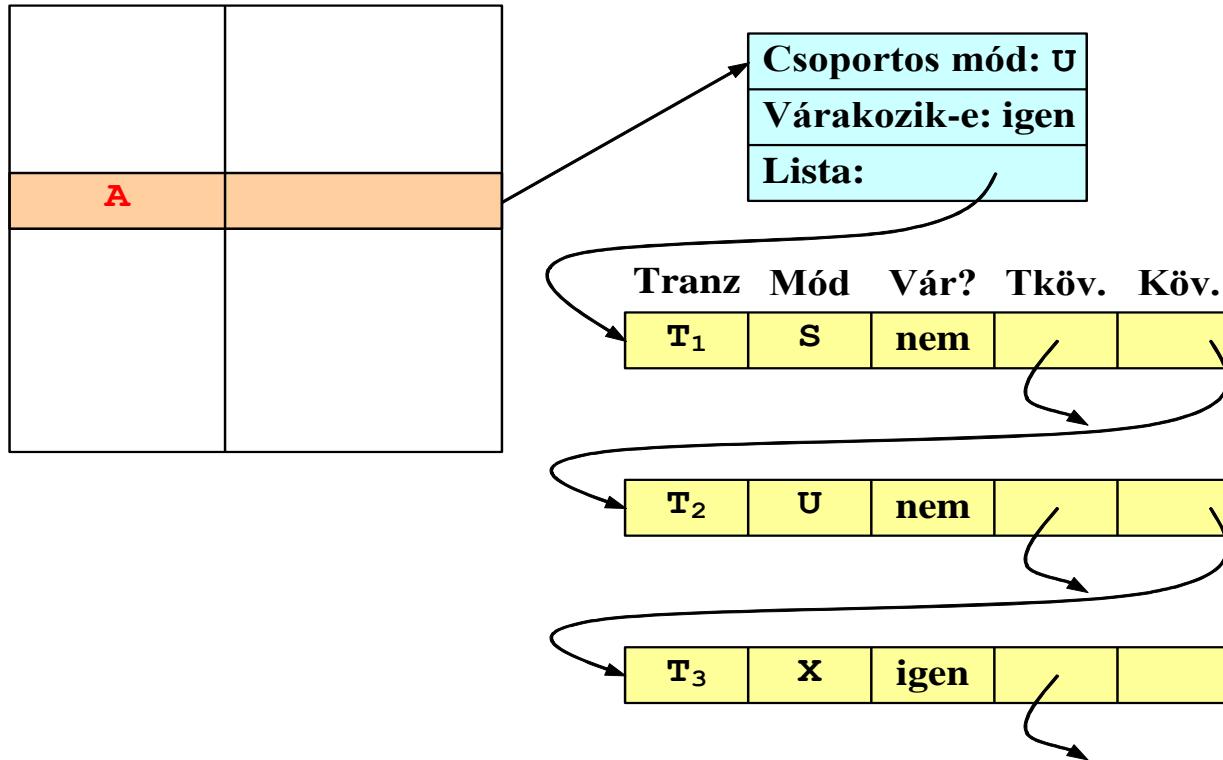
Adatbáziselem Zárolási információk



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



A zártábla



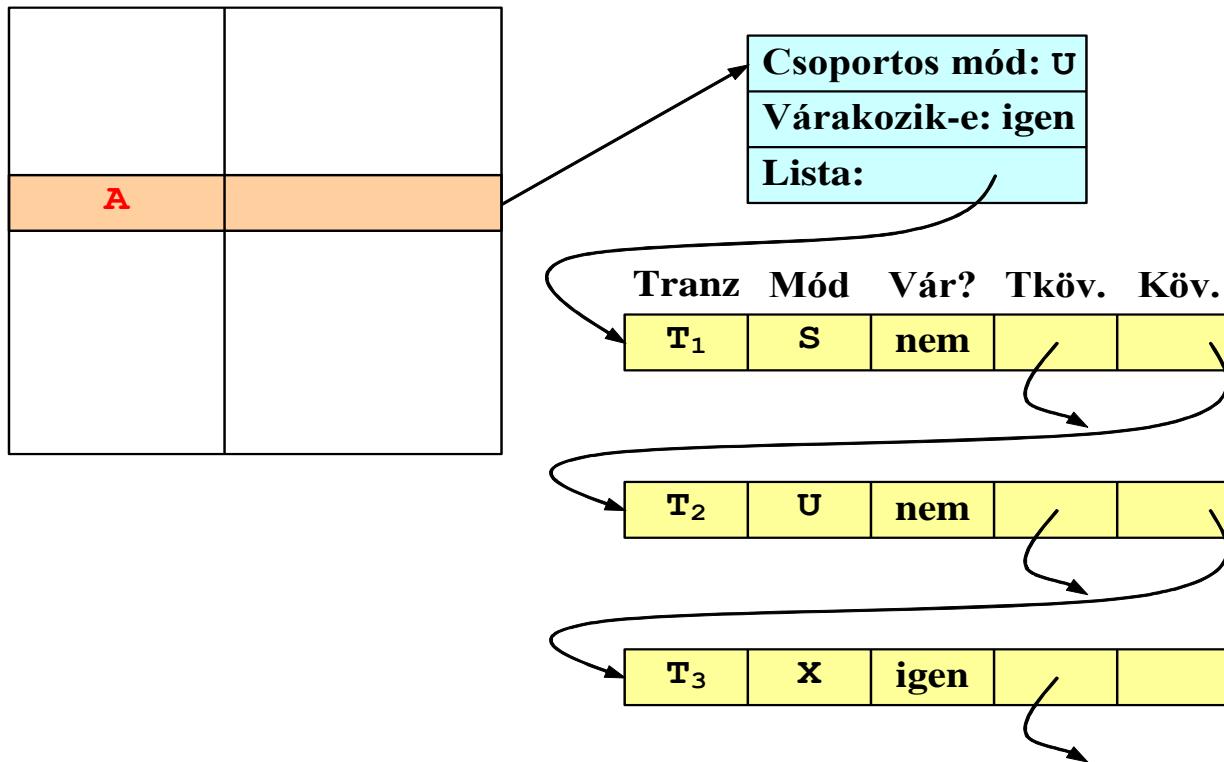
Az összes olyan tranzakciót leíró lista, amelyek vagy jelenleg zárolják **A-t, vagy **A** zárolására várakoznak.**

- a zárolást fenntartó vagy a zárolásra váró **tranzakció neve**;
- ennek a zárnak a **módja**;
- a tranzakció fenntartja-e a zárat, vagy **várakozik-e a zárra**;
- az adott tranzakció következő bejegyzése **Tköv.**



A zárolási kérések kezelése

Adatbáziselem Zárolási információk

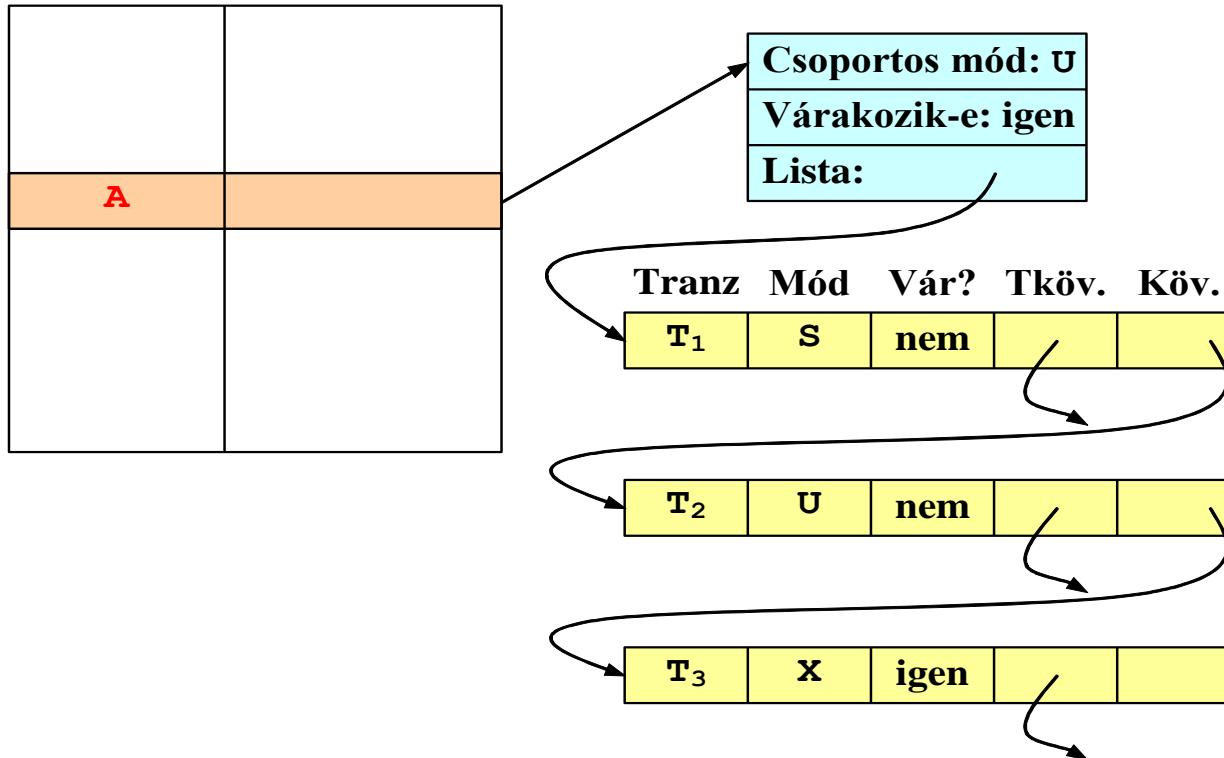


Ha a T tranzakció zárat kér A-ra. Ha nincs A-ra bejegyzés a zártáblában, akkor biztos, hogy zárok sincsenek A-n, így létrehozhatjuk a bejegyzést, és engedélyezhetjük a kérést. Ha a zártáblában létezik bejegyzés A-ra, akkor megkeressük a csoporthoz köthető módot, és ez alapján várakoztatunk, beírjuk a várakozási listába, vagy megengedjük a zárat (például ha T₂ X-et kér A-ra.)



A zárfeloldások kezelése

Adatbáziselem Zárolási információk



Ha T tranzakció feloldja az A-n lévő zárakat. Ekkor T bejegyzését A-ra a listában töröljük, és ha kell a csoportos módot is meg változtatjuk. Ha maradnak várakozó tranzakciók, akkor engedélyeznünk kell egy vagy több zárat a kért zárak listájáról.



A zárfeloldások kezelése

Több különböző megközelítés lehetséges, mindeneknek megvan a saját előnye:

- 1. Első beérkezett első kiszolgálása (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. Elsőbbségadás az osztott záraknak (priority to shared locks):** Először az összes várakozó osztott zárat engedélyezzük. Ezután egy módosítási zárolást engedélyezünk, ha várakozik ilyen. A kizártlagos zárolást csak akkor engedélyezzük, ha semmilyen más igény nem várakozik. Ez a stratégia csak akkor engedi a kiéheztést, ha a tranzakció U vagy X zárolásra vár.
- 3. Elsőbbségadás a felminősítésnek (priority to upgrading):** Ha van olyan U zárral rendelkező tranzakció, amely X zárrá való felminősítésre vár, akkor ezt engedélyezzük előbb. Máskülönben a fent említett stratégiák valamelyikét követjük.



Adatbáziselemekből álló hierarchiák kezelése

Kétféle fastruktúrával fogunk foglalkozni:

1. Az első fajta fastruktúra, amelyet figyelembe veszünk, a zárolható elemek (zárolási egységek) hierarchiája.

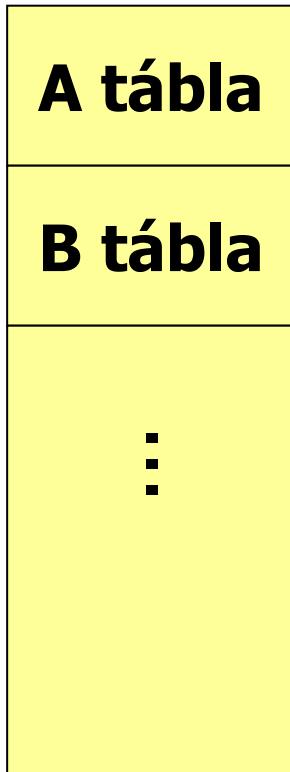
Megvizsgáljuk, hogyan engedélyezünk zárolást mind a nagy elemekre, mint például a relációkra, mind a kisebb elemekre, mint például a reláció néhány sorát tartalmazó blokkokra vagy egyedi sorokra.

2. A másik lényeges hierarchiafajtát képezik a konkurenciavezérlési rendszerekben azok az adatok, amelyek önmagukban faszervezésűek. Ilyenek például a B-fa-indexek. A B-fák csomópontjait adatbáziselemeknek tekinthetjük, így viszont az eddig tanult zárolási sémákat szegényesen használhatjuk, emiatt egy új megközelítésre van szükségünk.

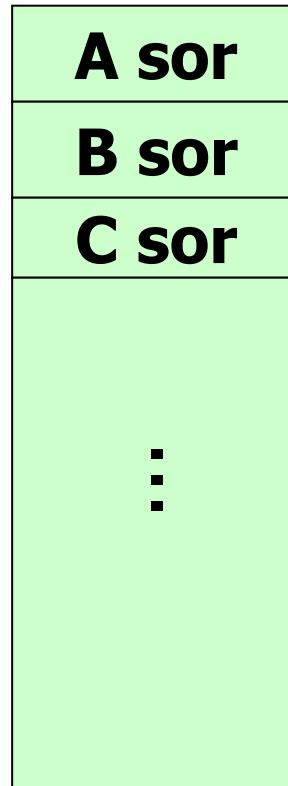


Többszörös szemcsézettségű zárak

Milyen objektumokat zároljunk?



Adatbázis



Adatbázis



Adatbázis

?



Többszörös szemcsézettségű zárák

- Az alapkérdés, hogy **kicsi vagy nagy objektumokat zároljunk?**
- Ha **nagy objektumokat (például dokumentumokat, táblákat)** zárolunk, akkor
 - kevesebb zárra lesz szükség, de
 - csökken a konkurens működés lehetősége.
- Ha **kicsi objektumokat, például számlákat, sorokat vagy mezőket** zárolunk, akkor
 - több zárra lesz szükség, de
 - nő a konkurens működés lehetősége.

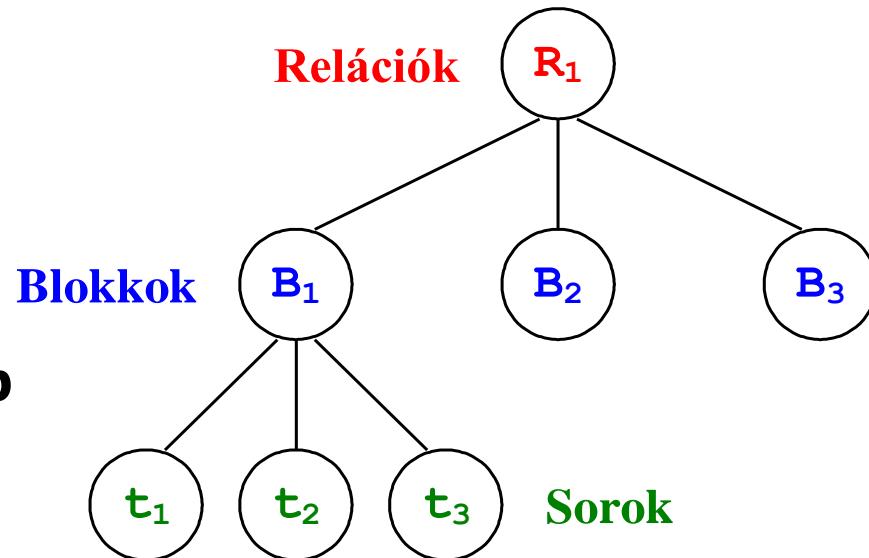
Megoldás: Kicsi ÉS nagy objektumokat is zárolhassunk!



Figyelmeztető zárak

Az adatbáziselemek több (például:három) szintjét különböztetjük meg:

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



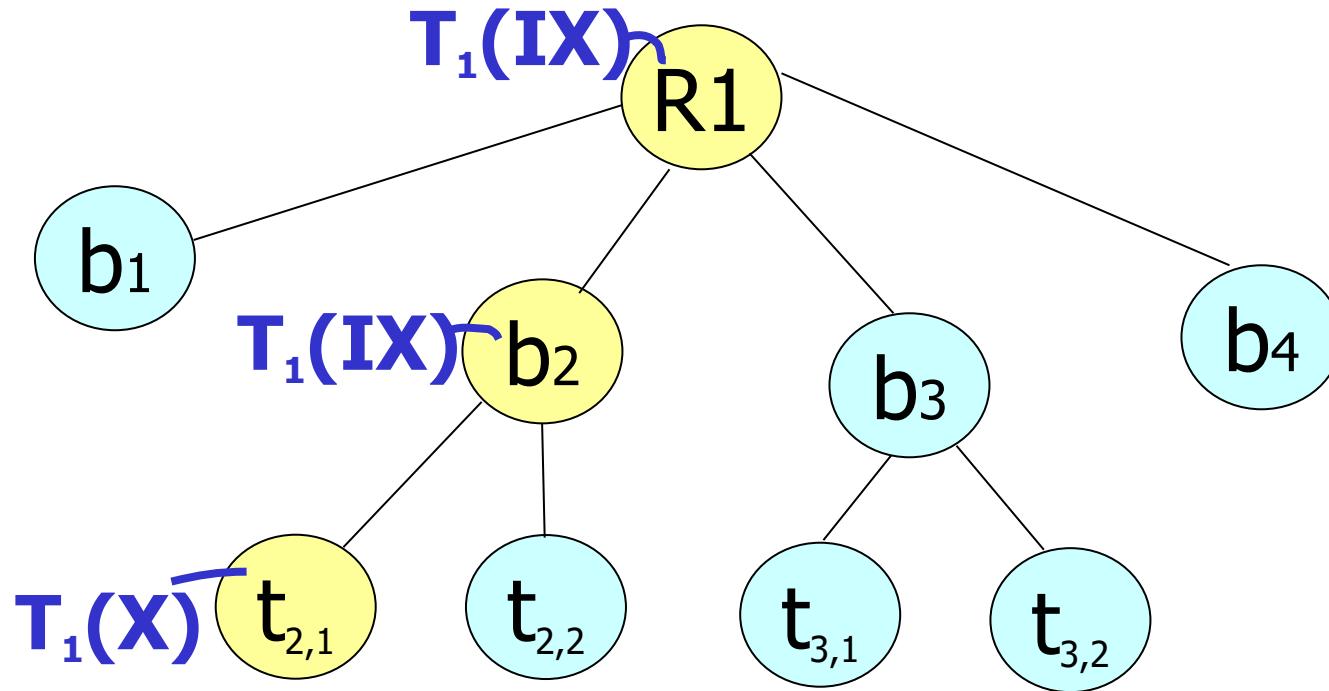
A **figyelmeztető protokoll** zárja (warning protocol):

1. a közönséges zárak: **S** és **X** (osztott és kizárólagos),
 2. figyelmeztető zárak: **IS**, **IX** (**I**=intention)
- Például **IS** azt jelenti, hogy szándékunkban áll osztott zárat kapni egy részelemen.



Figyelmeztető zárak

- A kérő zárnak megfelelő figyelmeztető zárakat kérünk az útvonal mentén a gyökérből kiindulva az adatalemig.
- Addig nem megyünk lejjebb, amíg a figyelmeztető zárat meg nem kapjuk.
- Így a konfliktusos helyzetek alsóbb szintekre kerülnek a fában.



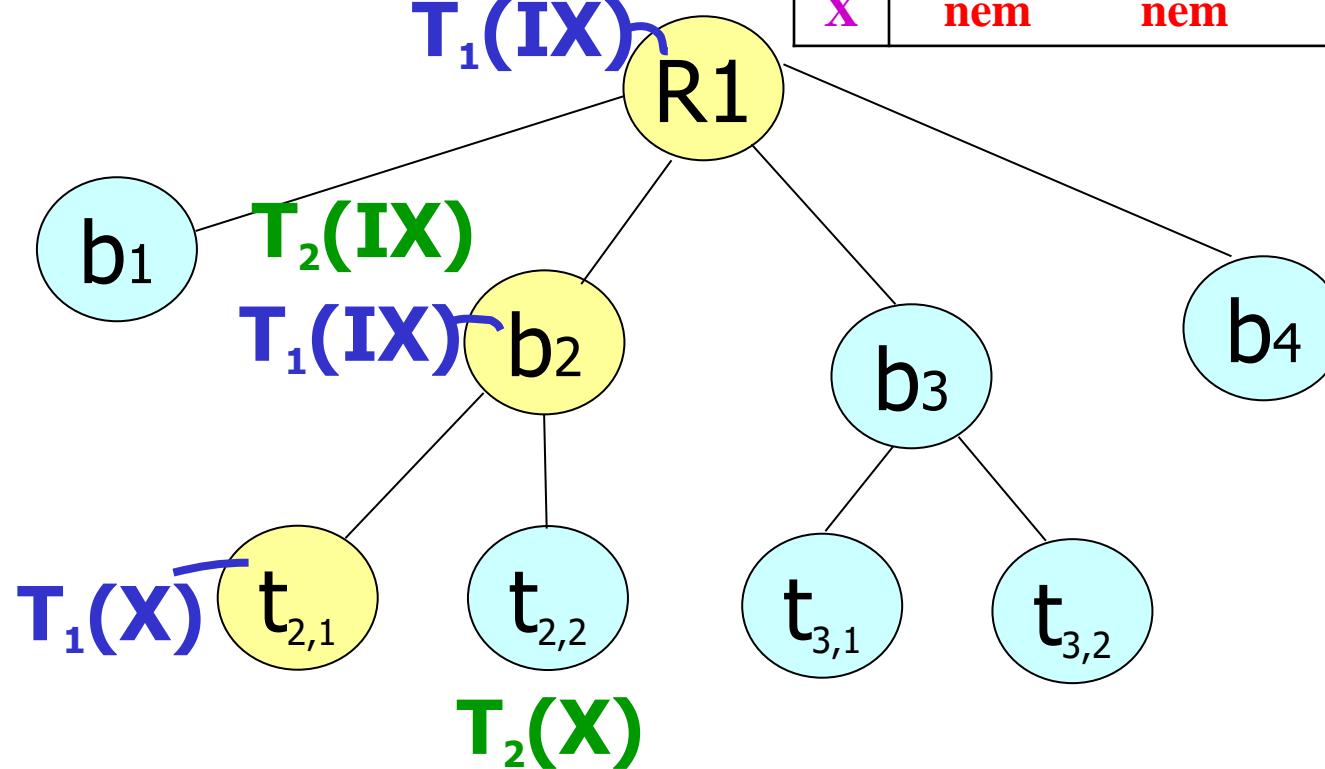
- Megkaphatja-e T_2 az X zárat a $t_{2,2}$ sorra?



Figyelmeztető zárak

Kompatibilitási mátrix:

| | IS | IX | S | X |
|----|------|------|------|-----|
| IS | igen | igen | igen | nem |
| IX | igen | igen | nem | nem |
| S | igen | nem | igen | nem |
| X | nem | nem | nem | nem |



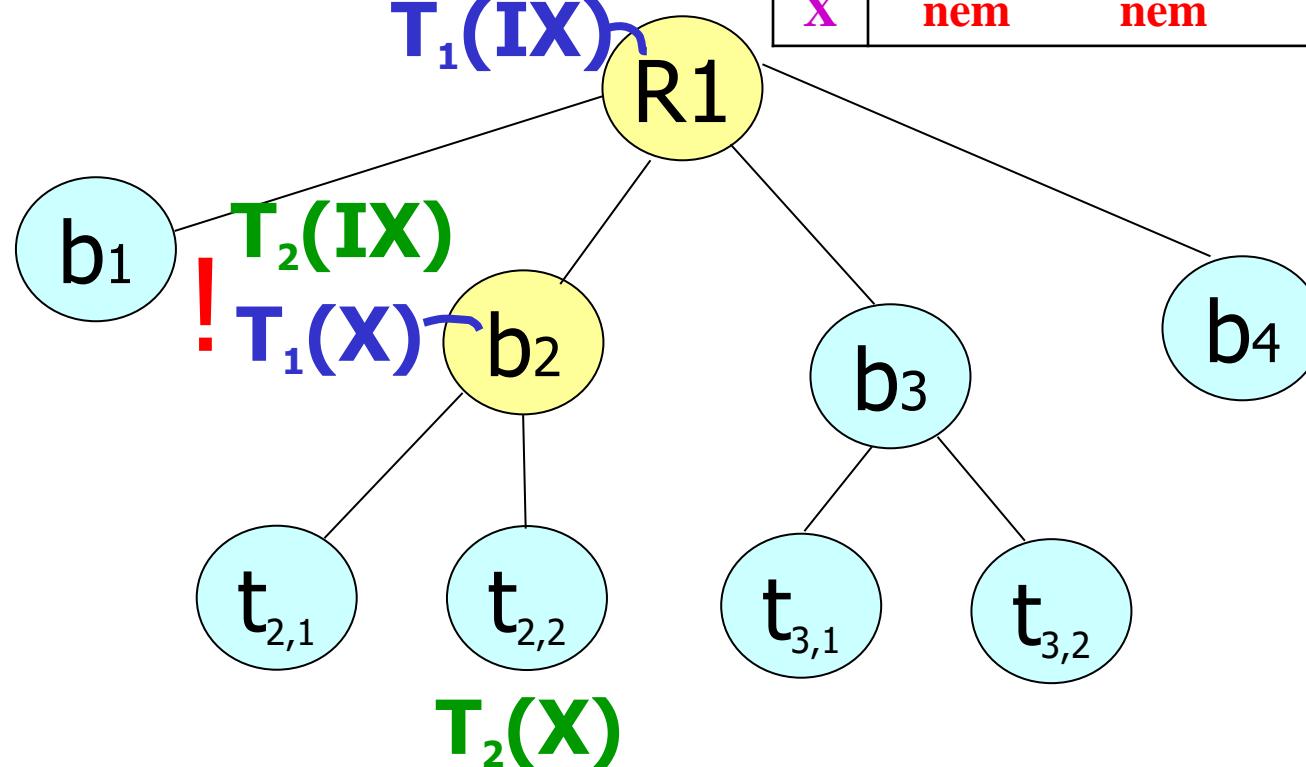
- Megkaphatja-e T₂ az X zárat a t_{2,2} sorra? IGEN



Figyelmeztető zárak

Kompatibilitási mátrix:

| | IS | IX | S | X |
|----|------|------|------|-----|
| IS | igen | igen | igen | nem |
| IX | igen | igen | nem | nem |
| S | igen | nem | igen | nem |
| X | nem | nem | nem | nem |



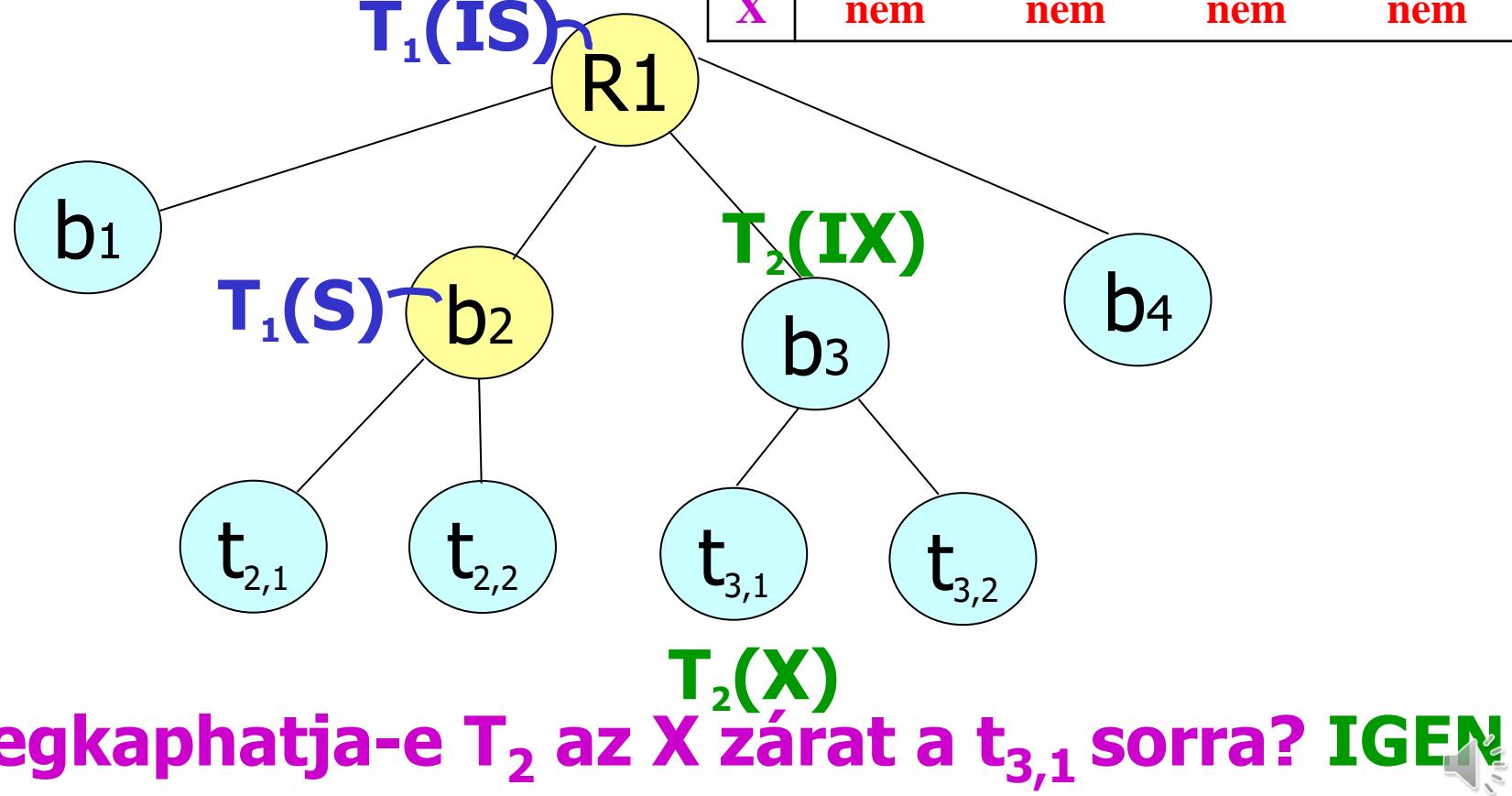
- Megkaphatja-e T_2 az X zárat a $t_{2,2}$ sorra? NEM



Figyelmeztető zárak

Kompatibilitási mátrix:

| | IS | IX | S | X |
|----|------|------|------|-----|
| IS | igen | igen | igen | nem |
| IX | igen | igen | nem | nem |
| S | igen | nem | igen | nem |
| X | nem | nem | nem | nem |



Figyelmeztető zárak

**SOR: Ha
ilyen zár van
már kiadva**

| | IS | IX | S | X |
|----|------------|------------|------------|------------|
| IS | igen | igen | igen | nem |
| IX | igen | igen | nem | nem |
| S | igen | nem | igen | nem |
| X | nem | nem | nem | nem |

**Oszlop:
Megkaphatjuk-e
ezt a típusú zárat?**

- **Ha IS zárat kérünk egy N csomópontban, az N egy leszármazottját szándékozzuk olvasni.** Ez csak abban az esetben okozhat problémát, ha egy másik tranzakció már jogosulttá vált arra, hogy az N által reprezentált teljes adatbáziselementet felülírja (**X**). Ha más tranzakció azt tervezzi, hogy N-nek csak egy részelemét írja (ezért az N csomóponton egy IX zárat helyezett el), akkor lehetőségünk van arra, hogy engedélyezzük az IS zárat N-en, és a konfliktust alsóbb szinten oldhatjuk meg, ha az írási és olvasási szándék valóban egy közös elemre vonatkozik.
- **Ha az N csomópont egy részelemét szándékozzuk írni (IX), akkor meg kell akadályoznunk az N által képviselt teljes elem olvasását vagy írását (**S vagy X**).** Azonban 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 IS-sel N-en.



Figyelmeztető zárak

SOR: Ha
ilyen zár van
már kiadva

| | IS | IX | S | X |
|----|------|------|------|-----|
| IS | igen | igen | igen | nem |
| IX | igen | igen | nem | nem |
| S | igen | nem | igen | nem |
| X | nem | nem | nem | nem |

Oszlop:
Megkaphatjuk-e
ezt a típusú zárat?

- Az **N csomópontnak megfeleltetett elem olvasása (S)** nincs konfliktusban sem egy másik olvasási zárral N-en, sem egy olvasási zárral N egy részelemén, amelyet N-en egy IS reprezentál. Azonban egy **X** vagy egy **IX** azt jelenti, hogy **más tranzakció írni fogja legalább egy részét az N által reprezentált elemnek**. Ezért nem tudjuk engedélyezni N teljes olvasását.
- Nem tudjuk megengedni az **N csomópont írását sem (X)**, ha más tranzakciónak már joga van arra, hogy olvassa vagy írja N-et (**S,X**), vagy arra, hogy megszerezze ezt a jogot N egy részelemére (**IS,IX**).



Összefoglalás

- Kompatibilitási mátrix alapján definiált zártípusok, sorbarendezhetőség feltétele
- Felminősítés, módosítási zár
- Inkrementális (növekményes) zár
- Zárolási ütemező felépítése, 2PL és konzisztencia tranzakciók generálása, jogoszerű ütemezés biztosítása, zártábla szerkezete
- Tartalmazási viszonyban álló adatelemek zárolása, figyelmeztető (szándék) zárak kompatibilitási mátrixa

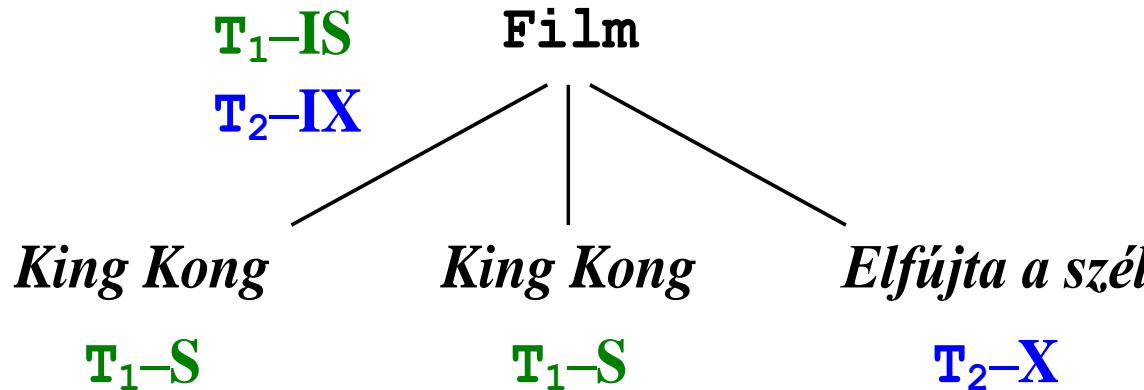


Figyelmeztető zárak

T1: SELECT * FROM Film WHERE filmCím = 'King Kong';

T2: UPDATE Film SET év = 1939 WHERE filmCím = 'Elfújta a szél';

Csak tábla és sor szintű zárolást engedjünk meg.



- Ekkor T_2 -nek szüksége van a reláció **IX** módú zárolására, ugyanis azt tervezí, hogy új értéket ír be az egyik sorba. Ez kompatibilis T_1 -nek a relációra vonatkozó **IS** zárolásával, így a zárat engedélyezzük.
- Amikor T_2 elérkezik az „Elfújta a szél” című filmhez tartozó sorhoz, ezen a soron nem talál zárat, így megkapja az **X** módú zárat, és módosítja a sort. Ha T_2 a „King Kong” című filmek valamelyikéhez próbált volna új értéket beírni, akkor várnia kellett volna, amíg T_1 felszabadítja az **S** zárákat, ugyanis az **S** és az **X** nem kompatibilisek.



Figyelmeztető zárak

SOR: Ha
ilyen zár van
már kiadva

| | IS | IX | S | X |
|----|------|------|------|-----|
| IS | igen | igen | igen | nem |
| IX | igen | igen | nem | nem |
| S | igen | nem | igen | nem |
| X | nem | nem | nem | nem |

Oszlop:
Megkaphatjuk-e
ezt a típusú zárat?

- Melyik zár erősebb a másiknál (erősebb ($<$)): ha mindenhol "nem" szerepel, ahol a gyengébben is "nem" van, de lehet ott is "nem", ahol a gyengébben "igen" van)?
- $IS < IX$ és $S < X$, de IX és S nem összehasonlítható ($<$ csak parciális rendezés).
- A csoportos mód használatához vezessünk be egy **SIX** új zárat, (ami azt jelenti, hogy ugyanaz a tranzakció S és IX zárat is tett egy adatelemre). Ekkor **SIX** minden kettőnél erősebb, de ez a legkisebb ilyen.



Csoportos mód a szándékzárolásokhoz

- **Ha mindenki van egy domináns zár (vagyis minden kiadott zárnál erősebb zár) egy elemen, akkor több zárolás hatását össze tudjuk foglalni egy csoportos móddal.**
- **A figyelmeztető zárákat is alkalmazó zárolási séma esetén az S és az IX módok közül egyik sem dominánsabb a másiknál.**
- **Ugyanaz a tranzakció egy elemet az S és IX módok mindegyikében zárolhatunk egyidejűleg.**
- **Egy tranzakció minden zárolást kérheti, ha egy teljes elemet akar beolvasni, és azután a részelemeknek egy valódi 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 új SIX zárolási módot, amelynek sora és oszlopa a „nem” bejegyzést 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.**



Csoportos mód a szándékzárásokhoz

Kérés

IS IX S SIX X

Zárolás

| | | | | | |
|-----|---|---|---|---|---|
| IS | T | T | T | T | F |
| IX | T | T | F | F | F |
| S | T | F | T | F | F |
| SIX | T | F | F | F | F |
| X | F | F | F | F | F |

T="igen"

F="nem"



Csoportos mód a szándékzárolásokhoz

P szülőn
ilyen a zár

A C gyerek
ilyen zárat kaphat

IS

IS, S

IX

IS, S, IX, X, SIX

S

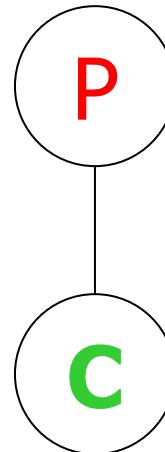
S, IS

SIX

X, IX, SIX

X

semmit



Csoportos mód a szándékzárolásokhoz

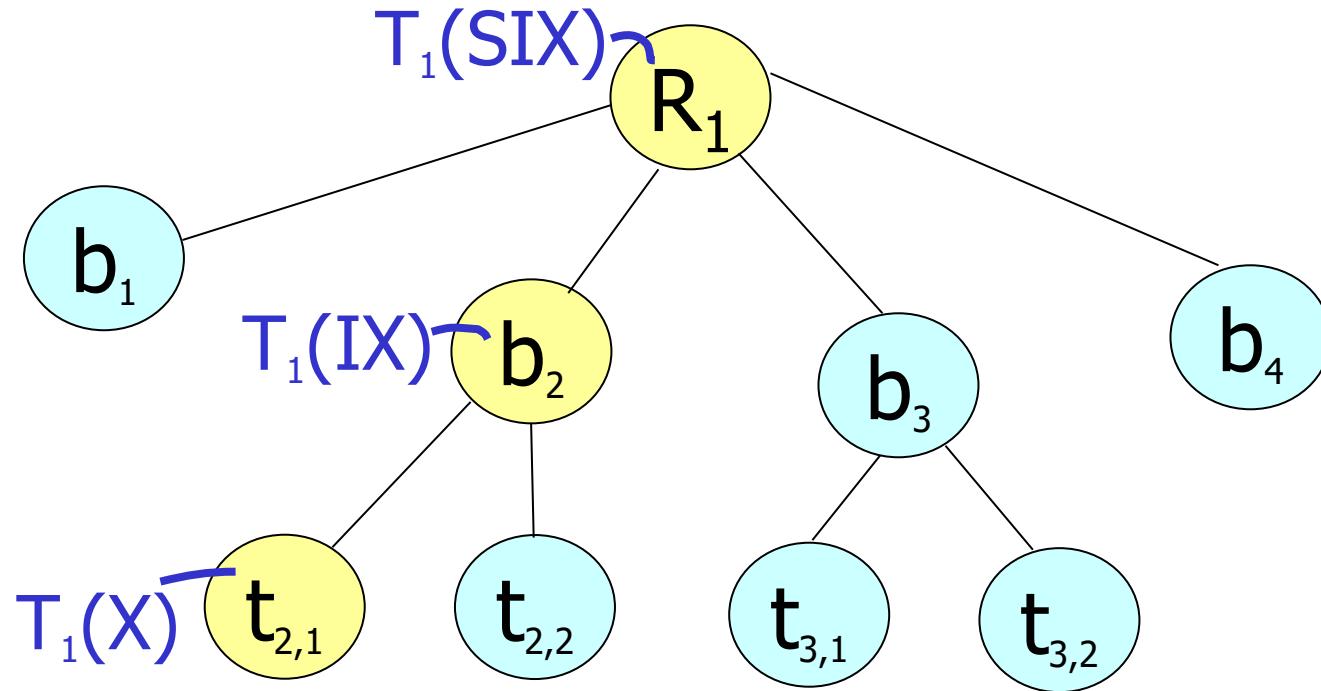
Szabályok:

- (1) A kompatibilitási mátrixnak megfelelően helyezhetjük el a zárakat.
- (2) Először a gyökeret zároljuk valamelyen módban.
- (3) Egy **Q** csúcsot egy **T_i** csak akkor zárolhat **S** vagy **IS** módban, ha a **szülő(Q)**-t **T_i** **IX** vagy **IS** módban már zárolta.
- (4) Egy **Q** csúcsot egy **T_i** csak akkor zárolhat **X,SIX,IX** módban, ha a **szülő(Q)**-t **T_i** **IX,SIX** módban már zárolta.
- (5) **T_i** a két fázisú zárolási protokollnak tesz eleget.
- (6) **T_i** csak akkor engedhet el egy zárat a **Q** csúcson, ha **Q** egyik gyerekét sem zárolja a **T_i**.



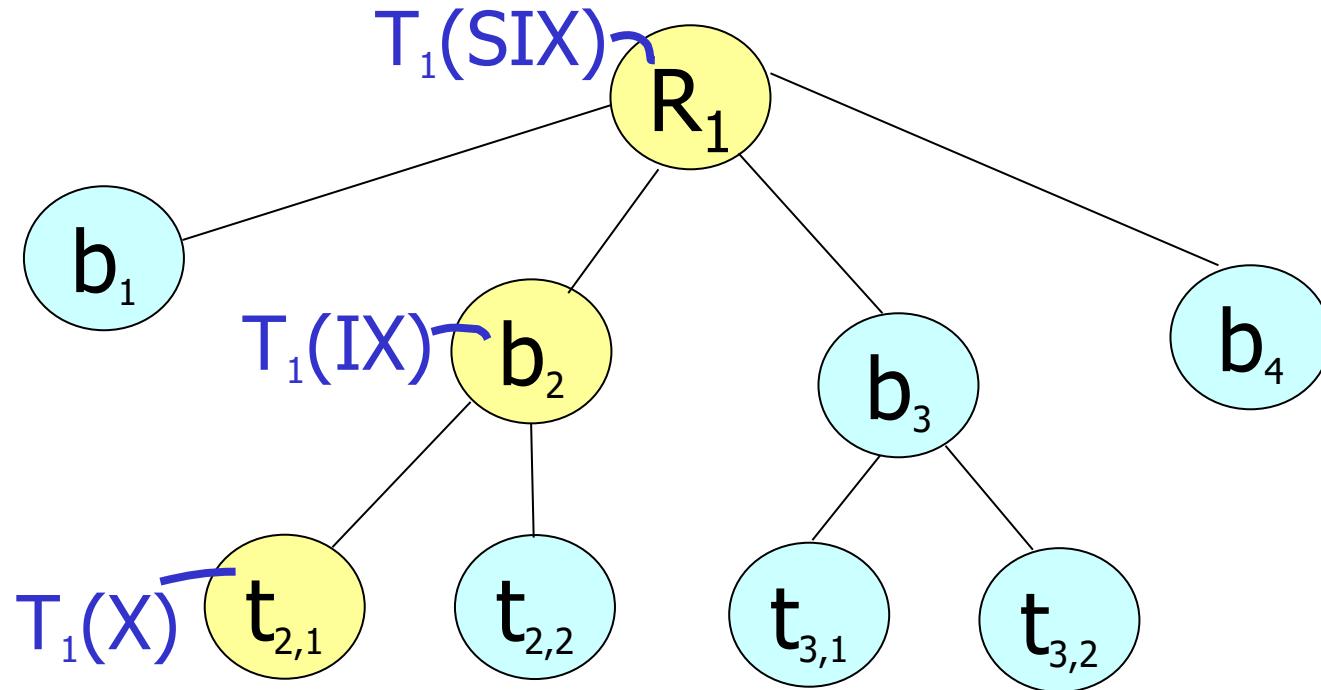
Csoportos mód a szándékzárolásokhoz

A T_2 kaphat-e $t_{2,2}$ sorra S zárat?



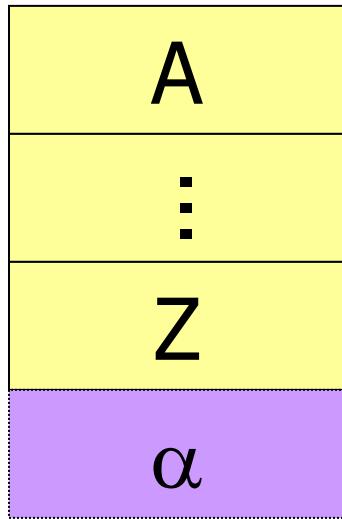
Csoportos mód a szándékzárolásokhoz

A T_2 kaphat-e $t_{2,2}$ sorra X zárat?



Nem ismételhető olvasás és a fantomok

Insert + delete + update műveletek



**Befolyásolhatja-e
egy másik
tranzakció hatását?**

← **Insert**

Mit zároljunk?

Egy nem létező sort?



Nem ismételhető olvasás és a fantomok

- Tegyük fel, hogy van egy T_1 tranzakció, amely egy adott feltételnek eleget tevő sorokat válogat ki egy relációból. Ezután hosszas számításba kezd, majd **később újra végrehajtja** a fenti lekérdezést.
- Tegyük fel továbbá, hogy a lekérdezés két végrehajtása között egy T_2 tranzakció módosít vagy töröl a táblából néhány olyan sort, amely eleget tesz a lekérdezés feltételének.
- A T_1 tranzakció lekérdezését ilyenkor **nem ismételhető (fuzzy) olvasásnak** nevezzük.
- A nem ismételhető olvasással az a probléma, hogy mást eredményez a lekérdezés másodszori végrehajtása, mint az első.
- A tranzakció viszont elvárhatja (ha akarja), hogy ha többször végrehajtja ugyanazt a lekérdezést, akkor mindenig ugyanazt az eredményt kapja.



Nem ismételhető olvasás és a fantomok

Ugyanez a helyzet akkor is, ha a T_2 tranzakció beszűr olyan sorokat, amelyek eleget tesznek a lekérdezés feltételének. A lekérdezés másodszori futtatásakor most is más eredményt kapunk, mint az első alkalommal.

Ennek az az oka, hogy most olyan sorokat is figyelembe kellett venni, amelyek az első futtatáskor még nem is léteztek.

Az ilyen sorokat nevezzük *fantomoknak* (phantom).



Nem ismételhető olvasás és a fantomok

- A fenti jelenségek általában nem okoznak problémát, ezért a legtöbb adatbázis-kezelő rendszer alapértelmezésben nem is figyel rájuk (annak ellenére, hogy minden két jelenség nem sorbarendezhető ütemezést eredményez!).
- A fejlettebb rendszerekben azonban a felhasználó kérheti, hogy a nem ismételhető olvasások és a fantomolvasások ne hajtódjanak végre.
- Ilyen esetekben rendszerint egy hibaüzenetet kapunk, amely szerint a T_1 tranzakció nem sorbarendezhető ütemezést eredményezett, és az ütemező abortálja T_1 -et.



Nem ismételhető olvasás és a fantomok

- **Figyelmeztető protokoll használata esetén viszont könnyen megelőzhetjük az ilyen szituációkat, mégpedig úgy, hogy a T_1 tranzakciónak S módban kell zárolnia a teljes relációt, annak ellenére, hogy csak néhány sorát szeretné olvasni.**
- A módosító/törlő/beszúró tranzakció ezek után IX módban szeretné zárolni a relációt. Ezt a kérést az ütemező először elutasítja és csak akkor engedélyezi, amikor a T_1 tranzakció már befejeződött, elkerülve ezáltal a nem sorbarendezhető ütemezést.



Megszorítások is sérülhetnek

Példa: R reláció (ID#,név,...)

**megszorítás: ID# kulcs
sorszintű zárolás**

| R | ID# | Név | |
|----|-----|-------|------|
| o1 | 55 | Smith | |
| o2 | 75 | Jones | |



T₁: Insert <04,Kerry,...> into R

T₂: Insert <04,Bush,...> into R

Előtte olvasási S zárak elhelyezése a többi objektumra:

T₁

S₁(o1)

S₁(o2)

Megszorítás ellenőrzése

:

Insert o3[04,Kerry,...]

T₂

S₂(o1)

S₂(o2)

Megszorítás ellenőrzés

:

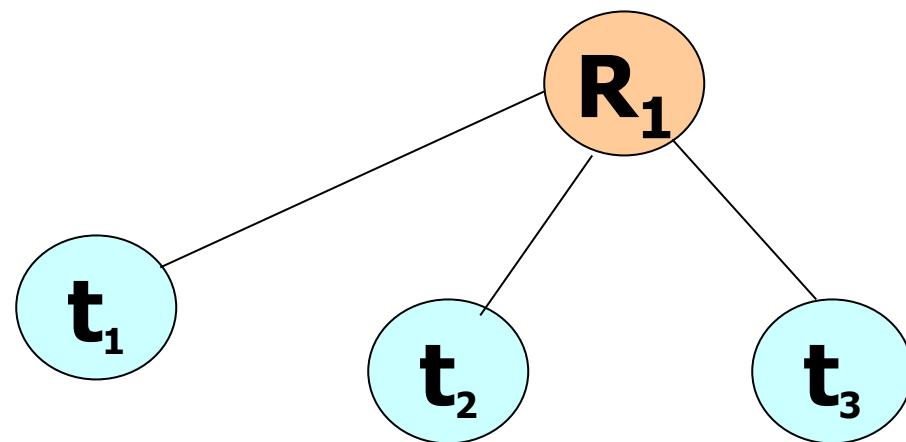
Insert o4[04,Bush,...]

Megsérül a megszorítás!



Megoldás

- Mielőtt egy Q csúcsot beszűrünk,
zároljuk a szülő(Q) csúcsot
X módban!



Példa

T1: Insert<04,Kerry>

T1

X1(R)

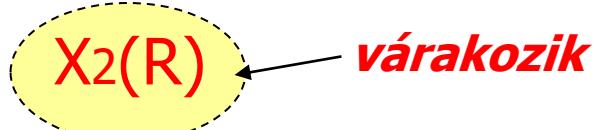
Megszorítás ellenőrzése

Insert<04,Kerry>

U(R)

T2: Insert<04,Bush>

T2



X2(R)

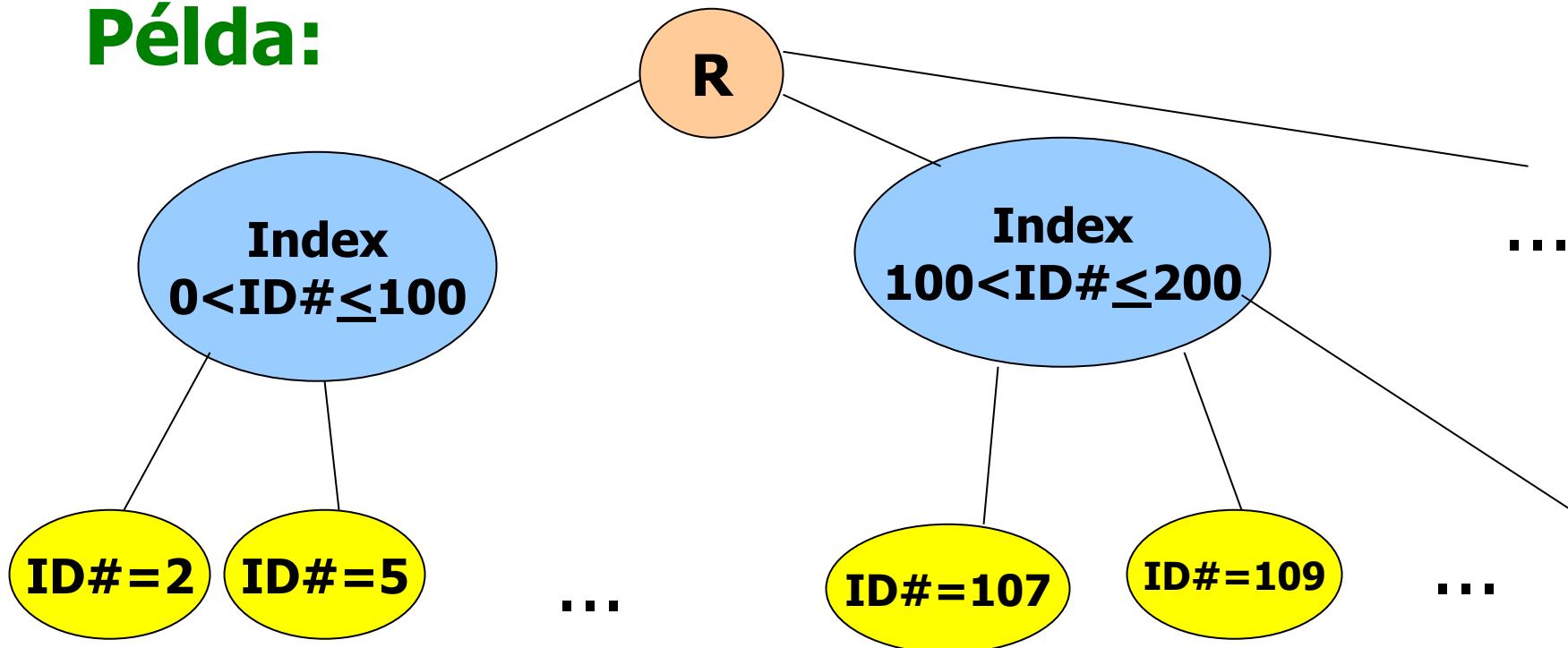
Megszorítás ellenőrzése

Hoppá! ID# = 04 már létezik!



Az R tábla sorainak elérése index alapján

Példa:



Csak egyféléképpen, a szülőkön keresztül lehet elérni egy csomópontot.



Faprotokoll

- A zárolható adategységek egy fa csúcsai.
- Például a B-fa esetén a levelekhez csak úgy juthatunk el, ha a gyökértől indulva végigjárunk egy lefelé vezető utat. Ahhoz, hogy beolvashassuk azt a levelet, ami nekünk kell, előtte be kell olvasnunk az összes felmenőjét (és ha csúcsok kettévágása vagy csúcsok összevonása úgy kívánja, írnunk is kell őket).
- Ilyenkor a szokásos technikák mennek ugyan, de nagyon előnytelenek lehetnek. Például a 2PL esetén egész addig kell tartani a zárat a gyökéren, amíg le nem értünk a levélhez, ami indokolatlanul sok várakozáshoz vezet.

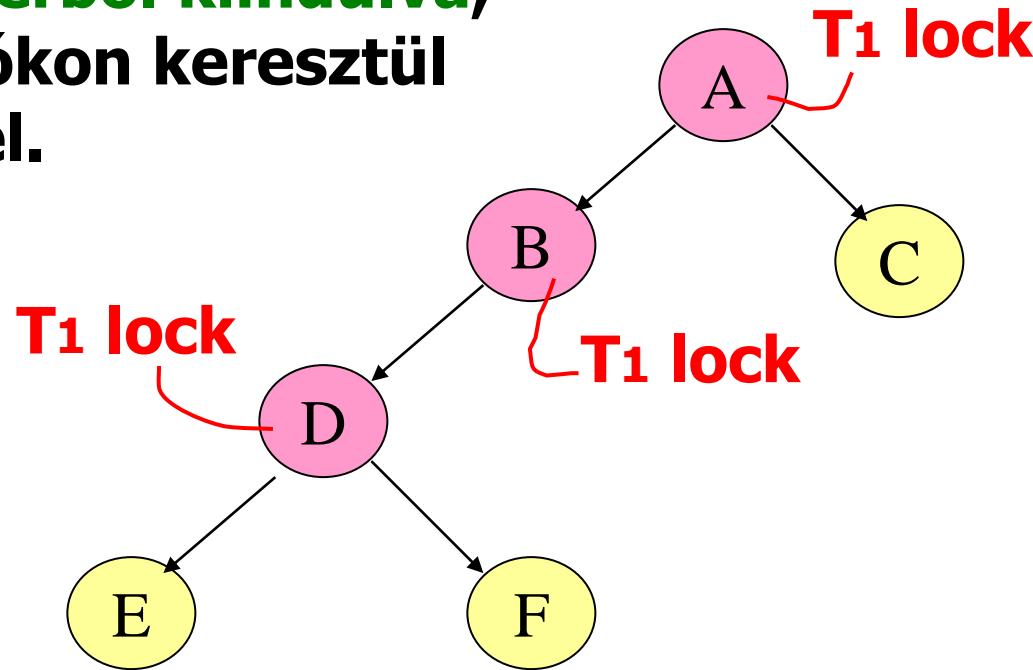


Faprotokoll

- Az esetek többségében egy B-fa gyökér csomópontját nem kell átírni, még akkor sem, ha a tranzakció beszűr vagy töröl egy sort.
- Például ha a tranzakció beszűr egy sort, de a gyökérnek az a gyereke, amelyhez hozzáférünk, nincs teljesen tele, akkor tudjuk, hogy a beszúrás nem gyűrűzik fel a gyökérig.
- Hasonlóan, ha a tranzakció egyetlen sort töröl, és a gyökérnek abban a gyerekében, amelyhez hozzáfertünk, a minimálisnál több kulcs és mutató van, akkor biztosak lehetünk abban, hogy a gyökér nem változik meg.
- Ha a tranzakció látja, hogy a gyökér biztosan nem változik meg, azonnal szeretnénk feloldani a gyökéren a zárat.
- Ugyanezt alkalmazhatjuk a B-fa bármely belső csomópontjának a zárolására is.
- A gyökéren lévő zárolás korai feloldása ellentmond a 2PL-nek, így nem lehetünk biztosak abban, hogy a B-fához hozzáférő tranzakcióknak az ütemezése sorba rendezhető lesz.
- A megoldás egy speciális protokoll a fa struktúrájú adatokhoz hozzáférő tranzakciók részére. A protokoll azt a tényt használja, hogy az elemekhez való hozzáférés lefelé halad a fán a sorbarendezhetőség biztosítása érdekében.

Példa

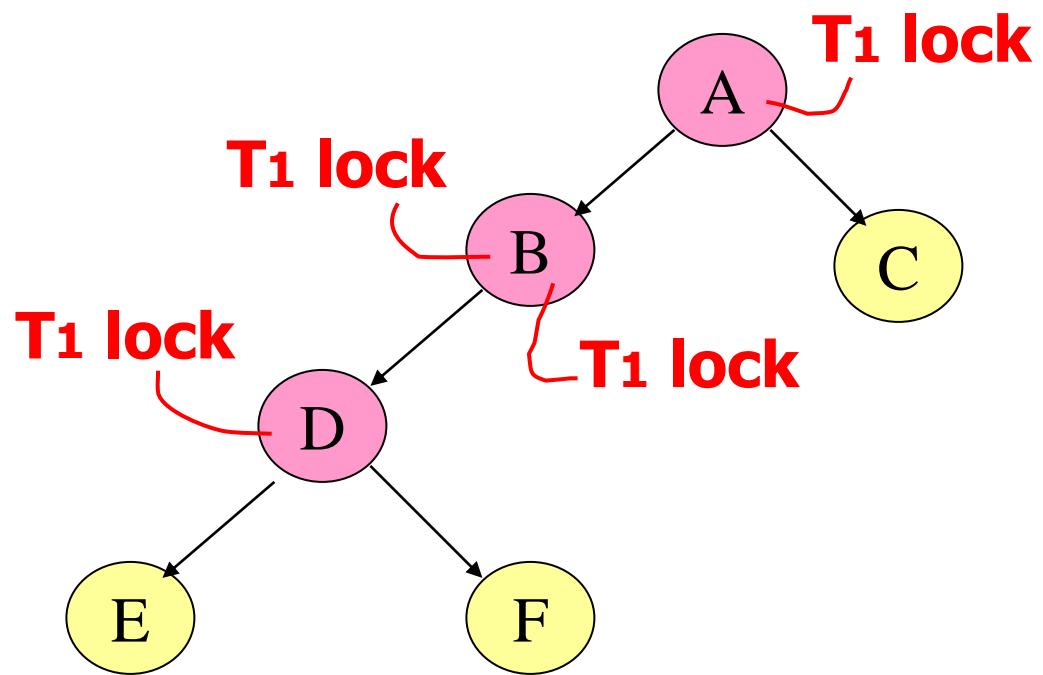
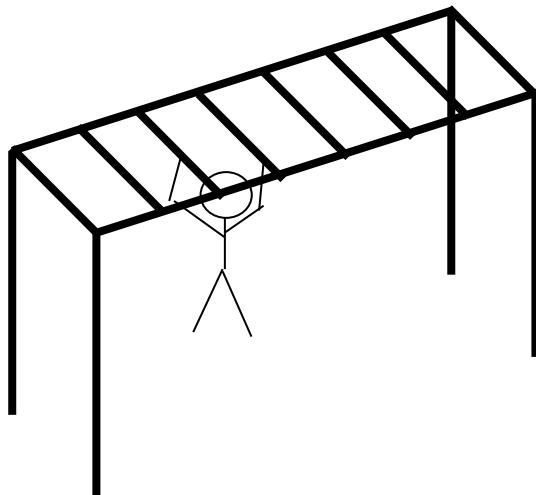
- Az összes objektumot a gyökérből kiindulva, mutatókon keresztül érjük el.



→ Elengedhetjük az A-n a zárat, ha már A-ra nincs szükségünk?



Ötlet: Mászóka



Csak egyfél zár van, de ezt az ötletet bármely zárolási módokból álló halmazra általánosíthatjuk.



Faprotokoll szabályai

Egyszerű tranzakciómodellben vagyunk (de lehetne (S/X) modellre kibővíteni), azaz

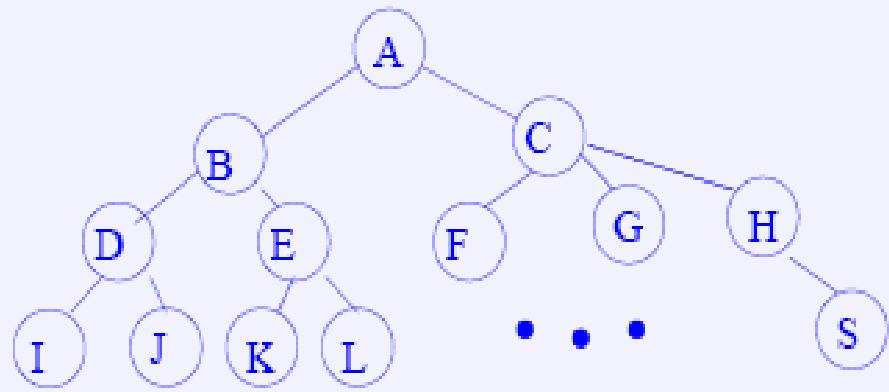
- egy zár van csak, ezt meg kell kapni íráshoz és olvasáshoz is
- zár után mindenig van UNLOCK
- nincs két különböző tranzakciónak zárja ugyanott.

A T_i tranzakció követi a faprotokollt, ha

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

Tétel. Ha minden tranzakció követi a faprotokollt egy jogoszerű ütemezésben, akkor az ütemezés sorbarendevezhető lesz, noha nem feltétlenül lesz 2PL.





A B-fa paramétere legyen 3, azaz legfeljebb 3 mutatót tartalmazhat egy csúcs. A fa belső csúcsai, A-tól H-ig, mutatókat és kulcsokat tartalmaznak. a levelekben (I-től S-ig) pedig a keresési kulcs szerint rendezetten vannak a tárolt adatok. Tegyük fel, hogy egy levélben egy tárolt elem van.

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

Ekkor a megfelelő (faprotooll szerinti, legális) ütemezés eleje

$LOCK_i(A); LOCK_i(B); UNLOCK_i(A)$ mert *B* beolvasása után látjuk, hogy neki csak két gyereke van, ha kell is csúcskettévágás, az *A*-t biztos nem érinti, *A*-t nem kell majd írni. Csak addig kellett fogni *A*-n a zárat, amíg *B*-re is megkaptuk.

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

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

Nem 2PL és ezzel nyertünk is sokat, mert amint megvolt $UNLOCK_i(A)$, akkor rögtön indulhat a következő beszúrás, ha az a jobb oldali ágán fut le. Ha 2PL lett volna, akkor $UNLOCK_i(E)$ -ig kellene várni ezzel.

Konkurenciavezérlés időbényegzőkkel

- *Eddig a zárakkal kényszerítettük ki a sorbarendezhető ütemezést.*
- *Most két másik módszert nézünk meg a tranzakciók sorbarendezhetőségének biztosítására:*

1. ***Időbényegzés*** (timestamping):

- minden tranzakcióhoz hozzárendelünk egy „időbényegzőt”.
- minden adatbáziselem utolsó olvasását és írását végző tranzakció időbényegzőjét rögzítjük, és összehasonlítjuk ezeket az értékeket, hogy biztosítsuk, hogy a tranzakciók időbényegzőinek megfelelő soros ütemezés ekvivalens legyen a tranzakciók aktuális ütemezésével.

2. ***Érvényesítés*** (validation):

- Megvizsgáljuk a tranzakciók időbényegzőit és az adatbáziselemeiket, 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.



Konkurenciavezérlés időbényezőkkel

- Mindkét megközelítés **optimista** abban az értelemben, hogy **feltételezik, nem fordul elő nem sorba rendezhető viselkedés**, és csak akkor tisztázza a helyzetet, amikor ez nyilvánvalóan nem teljesül.
- Ezzel ellentétben minden **zárolási módszer azt feltételezi, hogy „a dolgok rosszra fordulnak”**, hacsak a tranzakciót azonnal meg nem akadályozzák abban, hogy nem sorba rendezhető viselkedésük alakuljon ki.
- Az **optimista** megközelítések abban különböznek a zárolásuktól, hogy az egyetlen ellenszerük, amikor valami rosszra fordul, hogy **azt a tranzakciót, amely nem sorba rendezhető viselkedést okozna, abortálják, majd újraindítják**.
- A **zárolási ütemezők** ezzel ellentétben **késleltetik a tranzakciót, de nem abortálják** őket, hacsak nem alakul ki holtpont. (Késleltetés az optimista megközelítések esetén is előfordul, annak érdekében, hogy elkerüljük a nem sorba rendezhető viselkedést.)
- Általában az **optimista ütemezők akkor jobbak** a zárolásinál, amikor **sok tranzakció csak olvasási műveleteket** hajt végre, ugyanis az ilyen tranzakciók önmagukban soha nem okozhatnak nem sorba rendezhető viselkedést.



Időbényezők

- **Minden egyes T tranzakcióhoz hozzá kell rendelni egy egyedi számot, a **TS(T)** *időbényezőt* (time stamp).**
- **Az időbényezőket növekvő sorrendben kell kiadni abban az időpontban, amikor a tranzakció az elindításáról először értesíti az ütemezőt.**
- **Két lehetséges megközelítés az időbényezők generálásához:**
 1. **Az időbényezőket a rendszerőre felhasználásával hozzuk létre.**
 2. **Az ütemező karbantart egy számlálót. minden alkalommal, amikor egy tranzakció elindul, a számláló növekszik eggyel, és ez az új érték lesz a tranzakció időbényezője. Igy egy később elindított tranzakció nagyobb időbényezőt kap, mint egy korábban elindított tranzakció.**



Adatelemek időbényezői és véglegesítési bitjei

- **Minden egyes X adatbáziselemhez hozzá kell rendelnünk két időbényezőt és esetlegesen egy további bitet:**
- 1. **RT(X): X olvasási ideje** (read time), amely a legmagasabb időbényező, ami egy olyan tranzakcióhoz tartozik, amely már olvasta X-et.
- 2. **WT(X): X írási ideje** (write time), amely a legmagasabb időbényező, ami egy olyan tranzakcióhoz tartozik, amely már írta X-et.
- 3. **C(X): X véglegesítési bitje** (commit bit), amely akkor és csak akkor igaz, ha a legújabb tranzakció, amely X-et írta, már véglegesítve van.
- A **C(X)** bit célja, hogy elkerüljük azt a helyzetet, amelyben egy T tranzakció egy másik U tranzakció által írt adatokat olvas be, és utána U-t abortáljuk. Ez a probléma, **amikor T nem véglegesített adatok „piszkos olvasását” hajtja végre**, az adatbázis-állapot inkonzisztenssé válását is okozhatja.



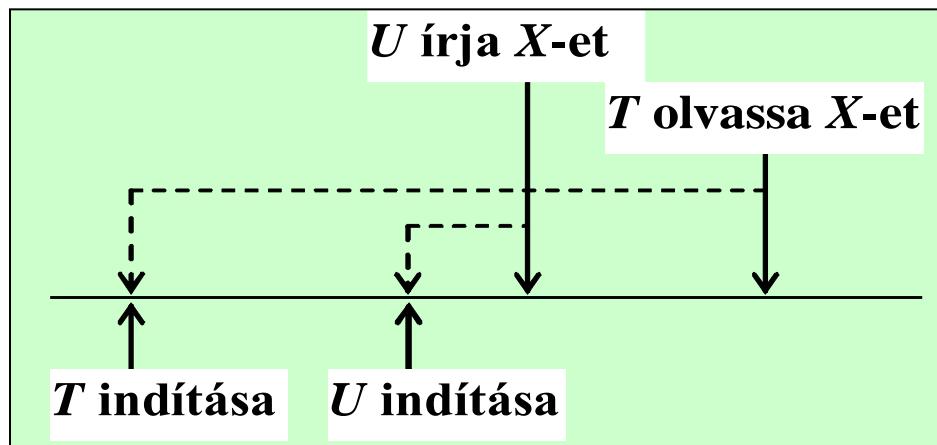
Fizikailag nem megvalósítható viselkedések

- Az ütemező olvasáskor és íráskor ellenőrzi, hogy ez abban a sorrendben történik-e, mintha a tranzakciót az időbélyegzőjük szerinti növekvő, soros ütemezésben hajtottunk volna végre.
- Ha nem, akkor azt mondjuk, hogy a viselkedés fizikailag nem megvalósítható és ilyenkor beavatkozik az ütemező.
- Kétféle probléma merülhet fel:
 1. Túl késői olvasás
 2. Túl késői írás



1. Túl késői olvasás

- A **T tranzakció megróbálja olvasni az X adatbáziselementet**, de X írási ideje azt jelzi, hogy X jelenlegi értékét azután írtuk, miután T-t már elméletileg végrehajtottuk, vagyis **TS(T) < WT(X)**.



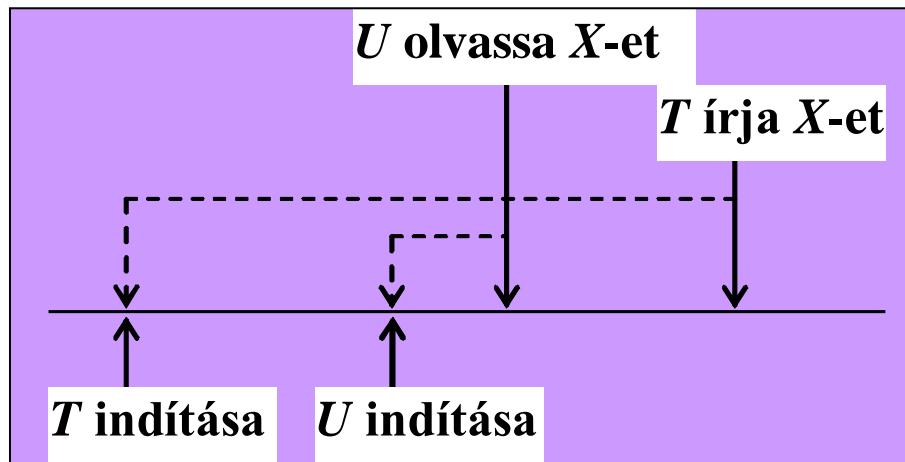
A **megoldás**, hogy **T-t abortáljuk**, amikor ez a probléma felmerül.



2. Túl késői írás

- A **T tranzakció megpróbálja írni az X adatbáziselementet**, de 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
 $WT(X) \leq TS(T) < RT(X)$ vagy
 $TS(T) < RT(X) < WT(X)$.

Semelyik más tranzakció sem írta X-et, amellyel felülírta volna a T által írt értéket, és ezzel érvénytelenítette volna T hatását.

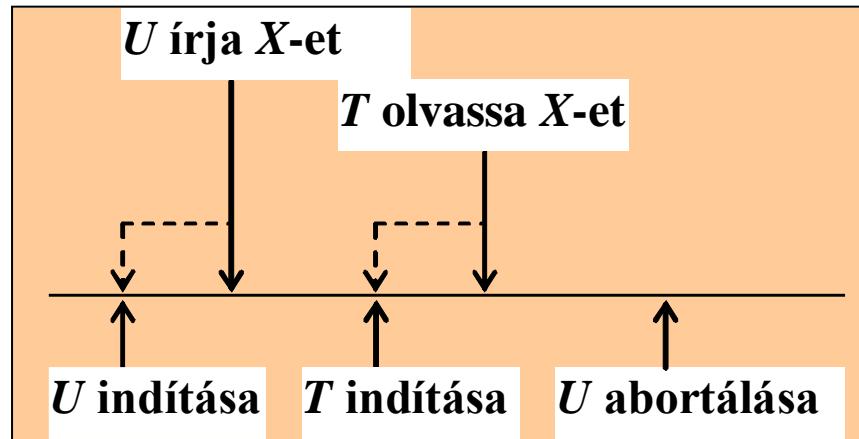


A megoldás, hogy T-t abortáljuk, amikor ez a probléma felmerül.



A piszkos adatok problémái

- Bár nincs fizikailag nem megvalósítható abban, hogy T olvassa X-et, mégis jobb a T általi olvasást azutánra elhalasztani, hogy U véglegesítését vagy abortálását már elvégeztük, különben az ütemezésünk nem lesz konfliktus-sorbarendezhető. Azt, hogy U még nincs véglegesítve, onnan tudjuk, hogy a C(X) véglegesítési bit hamis.

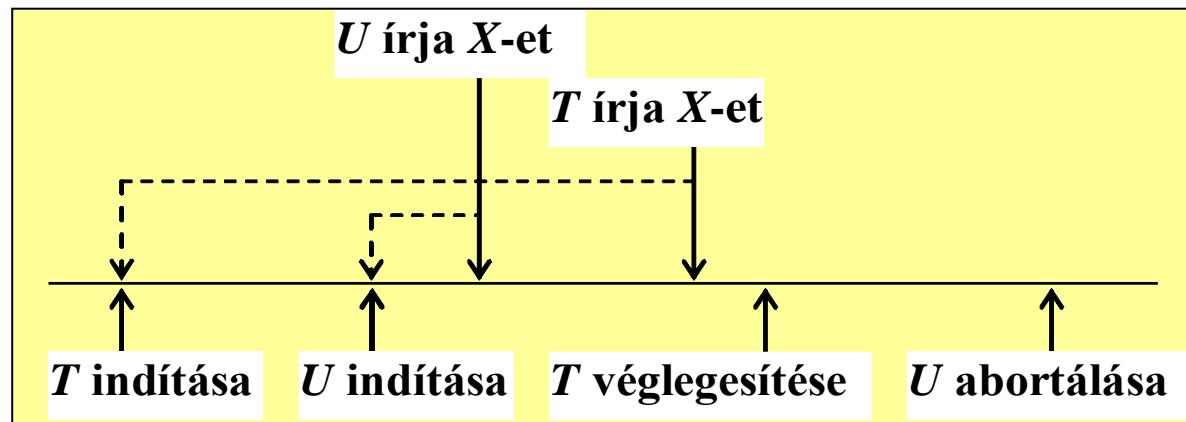


- A piszkos olvasás problémája véglegesítési bit nélkül is megoldható: Amikor abortálunk egy U tranzakciót, meg kell néznünk, hogy vannak-e olyan tranzakciók, amelyek olvastak egy vagy több U által írt adatbáziselemet. Ha igen, akkor azokat is abortálnunk kell. Ebből aztán további abortálások következhetnek, és így tovább. Ezt továbbgyűrűző visszagörgetésnek nevezzük. Ez a megoldás azonban alacsonyabb fokú konkurenciát engedélyez, mint a véglegesítési bit bevezetése és a késleltetés, ráadásul előfordulhat, hogy nem helyreállítható ütemezést kapunk. Ez abban az esetben következik be, ha az egyik „abortalandó” tranzakciót már véglegesítettük.



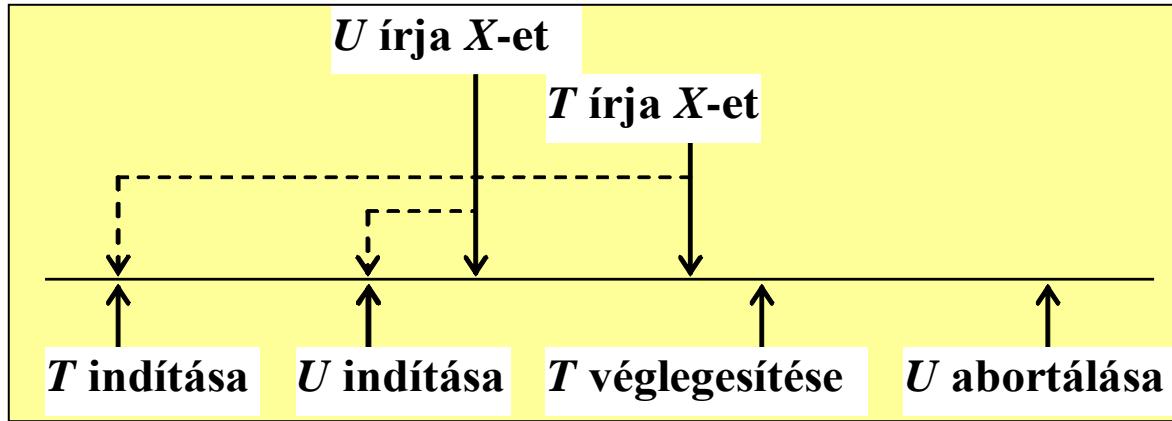
Egy másik probléma (Thomas-féle írás)

- Itt U írja először X-et. Amikor T írni próbál, a megfelelő művelet semmit sem végez, tehát elhagyható. Nyilvánvalóan nincs más V tranzakció, amelynek X-ből a T által beírt értéket kellene beolvasnia, és ehelyett az U által írt értéket olvasná, ugyanis ha V megpróbálná olvasni X-et, abortálnia kellene a túl késői olvasás miatt. X későbbi olvasásainál az U által írt értéket kell olvasni, vagy X még későbbi, de nem T által írt értékét. Ezt az ötletet, miszerint azokat az írásokat kihagyhatjuk, amelyeknél későbbi írási idejű írást már elvégeztünk, **Thomas-féle írási szabálynak** nevezzük.



PROBLÉMA: Ha **U-t később abortáljuk**, akkor X-nek az U által írt értékét ki kell törölünk, továbbá az előző értéket és írási időt vissza kell állítanunk. Minthogy **T-t véglegesítettük**, úgy látszik, hogy **X T által írt értékét kell a későbbi olvasásokhoz használnunk**. Mi viszont **kihagytuk a T általi írást**, és már túl késő, hogy helyrehozhassuk ezt a hibát.

Egy másik probléma (Thomas-féle írás)



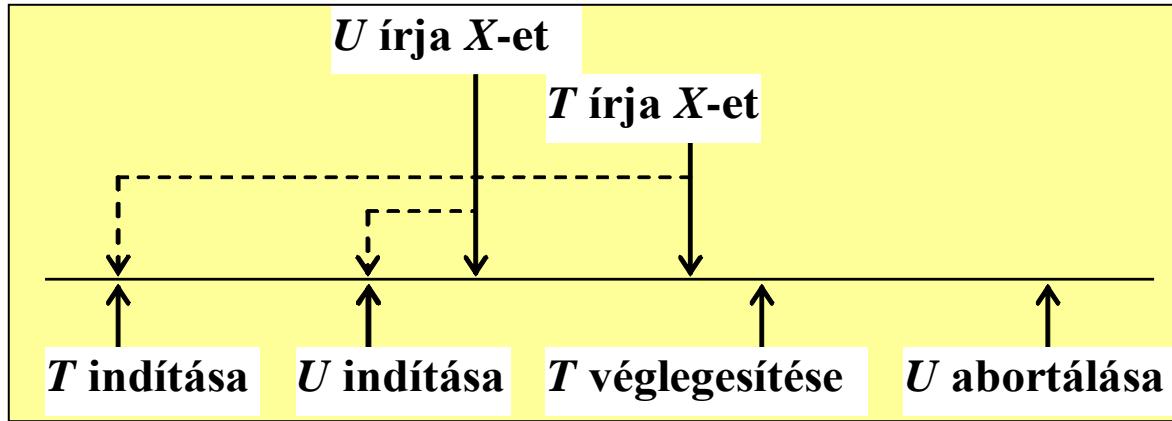
1. MEGOLDÁS: Amikor a T tranzakció írja az X adatbáziselementet, és azt látjuk, hogy X írási ideje nagyobb T időbényezőjénél (azaz $TS(T) < WT(X)$), valamint hogy az X-et író tranzakció még nincs véglegesítve (azaz $C(X)$ hamis), akkor **T-t késleltetjük mindaddig, amíg $C(X)$ igazzá nem válik.**

2. MEGOLDÁS: Amikor a T tranzakció írja az X adatbáziselementet, és azt látjuk, hogy X írási ideje nagyobb T időbényezőjénél (azaz $TS(T) < WT(X)$), **T-t visszagörgetjük.**

- Nyilván ez a megoldás **alacsonyabb fokú konkurenciát engedélyez**, mint a véglegesítési bit bevezetése és a késleltetés, és ha el akarjuk kerülni a piszkos olvasásokat, akkor az abortálás miatt most is **továbbgördülő visszagörgetéshez** és **nem helyreállítható ütemezéshez** juthatunk.



Egy másik probléma (Thomas-féle írás)



3. MEGOLDÁS: X minden írásánál hozzuk létre X és WT(X) egy új változatát, és csak akkor írjuk felül ezek „eredeti” változatait, ha $TS(T) \geq WT(X)$. Ekkor sem késleltetjük a tranzakciókat, és ha abortál az a tranzakció, melynek időbélyegzője a legnagyobb $WT(X)$ érték, akkor megkeressük a többi letárolt $WT(X)$ értékek közül a legnagyobbat, és ezután ezt, illetve az ehhez tartozó X értéket tekintjük „eredetinek”. Ezen az ötleten alapul a többváltozatú időbélyegzés, ami szintén megoldást nyújt a Thomas-féle írási szabály problémájára.

Látható, hogy az időbélyegzési technika alapváltozatában (amikor nem használunk vélegesítési bitet és nincs késleltetés) nem léphet fel holtponti helyzet, előfordulhat viszont továbbgyűrűző visszagörgetés és nem helyreállítható ütemezés.



Az időbényezőn alapuló ütemezések szabályai

- Az ütemezőnek egy T tranzakciótól érkező olvasási vagy írási kérésre adott válaszában az alábbi választásai lehetnek:
 1. Engedélyezi a kérést.
 2. Abortálja T-t (ha T „megsérти a fizikai valóságot”), és egy új időbényezővel újraindítja. Azt az abortálást, amelyet újraindítás követ, gyakran visszagörgetésnek (rollback) nevezük.
 3. Késlelteti T-t, és később dönti el, hogy abortálja T-t, vagy engedélyezi a kérést (ha a kérés olvasás, és az olvasás piszkos is lehet, illetve ha a kérés írás, és alkalmazzuk a Thomas-féle írási szabályt).
- Összegezhetjük azokat a szabályokat (4 szabályt), amelyeket az időbényezőket használó ütemezőnek követnie kell ahhoz, hogy biztosan konfliktus-sorbarendezerő ütemezést kapjunk.



Konkurenciavezérlés érvényesítéssel

- Az **érvényesítés** (validation) az **optimista konkurenciavezérlés** másik típusa, amelyben a tranzakcióknak megengedjük, hogy zárolások nélkül hozzáférjenek az adatokhoz, és a megfelelő időben ellenőrzük a tranzakció sorba rendezhető viselkedését.
- Az érvényesítés alapvetően abban különbözik az időbélyegzéstől, hogy **itt az ütemező 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 adatbáziselekbe, egy „**érvényesítési fázison**” megy keresztül, amikor a **beolvasott és kiírandó elemek halmazait összehasonlítjuk** más **aktív tranzakciók írásainak halmazaival**. Ha fellép a fizikailag nem megvalósítható viselkedés kockázata, a tranzakciót **visszagörgetjük**.



Az Oracle konkurenciavezérlési technikája

- Az Oracle alapvetően a **zárolás módszerét** használja a konkurenciavezérléshez.
- Felhasználói szinten a **zárolási egység** lehet a **tábla** vagy annak egy **sora**.
- A **zárakat az ütemező helyezi el és oldja fel**, de lehetőség van arra is, hogy a **felhasználó (alkalmazás)** kérjen zárat.
- Az Oracle alkalmazza a **kétfázisú zárolást, a figyelmeztető protokollt** és a **többváltozatú időbényezőket** is némi módosítással.



Többszintű konkurenciavezérlés Oracle-ben

- Az Oracle minden **leérdezés** számára biztosítja az **olvasási konzisztenciát**, azaz a leérdezés által olvasott adatok egy időpillanatból (a leérdezés kezdetének pillanatából) származnak.
 - Emiatt a leérdezés **sohasem olvas piszkos adatot**,
 - és **nem látja azokat a változtatásokat sem**, amelyeket a leérdezés végrehajtása alatt véglegesített tranzakciók eszközöltek.

Ezt **utasítás szintű olvasási konzisztenciának** nevezük.

- Kérhetjük egy **tranzakció összes leérdezése** számára is a konzisztencia biztosítását, ez a **tranzakció szintű olvasási konzisztencia**.
 - Ezt úgy érhetjük el, hogy a tranzakciót **sorba rendezhető**
 - **vagy csak olvasás módban futtatjuk**.
 - Ekkor a tranzakció által tartalmazott összes leérdezés a tranzakció **indításakor fennálló adatbázis-állapotot látja**, kivéve a tranzakció által korábban végrehajtott módosításokat.



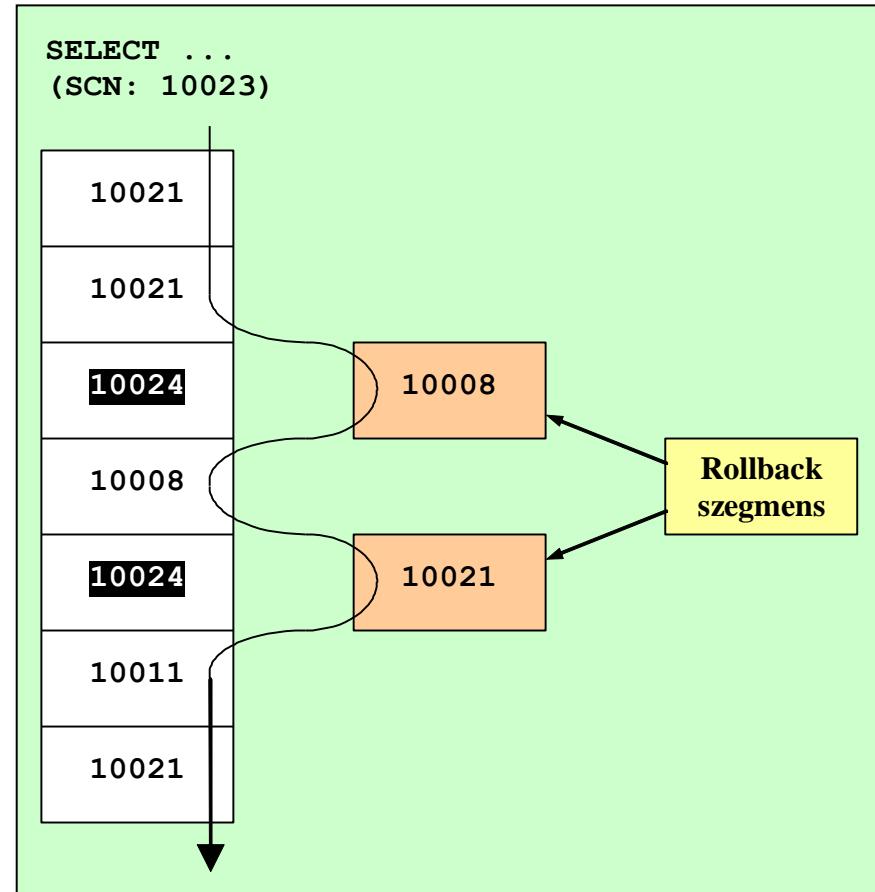
Többszintű konkurenciavezérlés Oracle-ben

- A kétféle olvasási konzisztencia eléréséhez az Oracle a **rollback szegmensekben** található információkat használja fel.
- A **rollback szegmensek** tárolják azon **adatok régi értékeit, amelyeket még nem véglegesített vagy nemrég véglegesített tranzakciók változtattak meg.**
- Amint egy lekérdezés vagy tranzakció megkezdi működését, meghatározódik a **system change number (SCN)** aktuális értéke. Az SCN a **blokkokhoz** mint adatbáziselemekhez tartozó **időbélyegzőnek** tekinthető.



Többszintű konkurenciavezérlés Oracle-ben

- Ahogy a lekérdezés olvassa az adatblokkokat, összehasonlítja azok SCN-jét az aktuális SCN értékkel, és csak az aktuálisnál kisebb SCN-nel rendelkező blokkokat olvassa be a tábla területéről.
- A nagyobb SCN-nel rendelkező blokkok esetén a rollback szegmensből megkeresi az adott blokk azon verzióját, amelyhez a legnagyobb olyan SCN érték tartozik, amely kisebb, mint az aktuális, és már véglegesített tranzakció hozta létre.



A 10023 előtt indult tranzakciók módosításait már elvileg láthatja.

A 10024-es blokkok esetén a régi példányokat a rollback szegmensből olvassuk ki.



A tranzakcióelkülönítési szintek

- Az SQL92 ANSI/ISO szabvány a **tranzakcióelkülönítés négy szintjét definiálja**, amelyek abban különböznek egymástól, hogy az alábbi **három jelenség** közül melyeket engedélyezik:
- ***piszkes olvasás***: a tranzakció olyan adatot olvas, amelyet egy másik, még nem véglegesített tranzakció írt;
- ***nem ismételhető (fuzzy) olvasás***: a tranzakció újraolvas olyan adatokat, amelyeket már korábban beolvasott, és azt találja, hogy egy másik, már véglegesített tranzakció módosította vagy törölte őket;
- ***fantomok olvasása***: a tranzakció újra végrehajt egy lekérdezést, amely egy adott keresési feltételnek eleget tevő sorokkal tér vissza, és azt találja, hogy egy másik, már véglegesített tranzakció további sorokat szúrt be, amelyek szintén eleget tesznek a feltételnek.



A négy tranzakcióelkülönítési szint a következő:

| | piszkos olvasás | nem ismételhető olvasás | fantomok olvasása |
|--|-----------------|-------------------------|-------------------|
| <i>nem olvasásbiztos</i> (read uncommitted) | lehetséges | lehetséges | lehetséges |
| <i>olvasásbiztos</i> (read committed) | nem lehetséges | lehetséges | lehetséges |
| <i>megismételhető olvasás</i> (repeatable read) | nem lehetséges | nem lehetséges | lehetséges |
| <i>sorbarendezhető</i> (Serializable) | nem lehetséges | nem lehetséges | nem lehetséges |

Az Oracle ezek közül az

- 1. olvasásbiztos** és a
- 2. sorbarendezhető** elkülönítési szinteket ismeri, valamint egy
- 3. csak olvasás (read-only) módot**, amely nem része a szabványnak.



Az Oracle tranzakcióelkülönítési szintjei

1. **Olvasásbiztos:**

- **SET TRANSACTION ISOLATION LEVEL READ COMMITTED;**
- **Ez az alapértelmezett tranzakcióelkülönítési szint.**
- **Egy tranzakció minden lekérdezése csak a lekérdezés (és nem a tranzakció) elindítása előtt véglegesített adatokat látja.**
- **Piszkos olvasás sohasem történik.**
- A lekérdezés két végrehajtása között a lekérdezés által olvasott adatokat más tranzakciók megváltoztathatják, ezért **előfordulhat nem ismételhető olvasás és fantomok olvasása** is.
- **Olyan környezetekben célszerű ezt a szintet választani, amelyekben várhatóan kevés tranzakció kerül egymással konfliktusba.**



Az Oracle tranzakcióelkülönítési szintjei

2. Sorbarendevezhető:

- SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
- A sorba rendezhető tranzakciók csak a tranzakció elindítása előtt véglegesített változásokat látják, valamint azokat, amelyeket maga a tranzakció hajtott végre INSERT, UPDATE és DELETE utasítások segítségével.
- A sorba rendezhető tranzakciók nem hajtanak végre nem ismételhető olvasásokat, és nem olvasnak fantomokat.
- Ezt a szintet olyan környezetekben célszerű használni, amelyekben nagy adatbázisok vannak, és rövidek a tranzakciók, amelyek csak kevés sort módosítanak, valamint ha kicsi az esélye annak, hogy két konkurens tranzakció ugyanazokat a sorokat módosítja, illetve ahol a hosszú (sokáig futó) tranzakciók elsősorban csak olvasási tranzakciók.
- Az Oracle csak akkor engedi egy sor módosítását egy sorbarendevezhető tranzakciónak, ha el tudja dönteni, hogy az adott sor korábbi változásait olyan tranzakciók hajtották végre, amelyek még a sorbarendevezhető tranzakció elindítása előtt véglegesítődtek. Ennek eldöntésére az Oracle a blokkokban tárolt vezérlőinformációkat használja, amelyek megmondják, hogy az adott blokkban az egyes sorokat mely tranzakciók módosították, és hogy ezek a módosítások véglegesítettek-e. (SCN – írási időbényező és commit bit)
- Amennyiben egy sorbarendevezhető tranzakció megpróbál módosítani vagy törölni egy sort, amelyet egy olyan tranzakció változtatott meg, amely a sorba rendezhető tranzakció indításakor még nem véglegesítődött, az Oracle hibaüzenetet ad („Cannot serialize access for this transaction”).



Az Oracle tranzakcióelkülönítési szintjei

3. Csak olvasás:

- **SET TRANSACTION READ ONLY;**
- A csak olvasást végző tranzakciók csak a tranzakció elindítása előtt végezhetnek változásokat látják, és nem engednek meg **INSERT, UPDATE** és **DELETE** utasításokat.



A zárolási rendszer

- Bármelyik elkülönítési szintű tranzakció használja a **sor szintű zárolást**, ezáltal egy T tranzakciónak **várnia kell**, ha olyan sort próbál írni, amelyet egy még nem véglegesített konkurens tranzakció módosított.
- T megvárja, míg a másik tranzakció véglegesítődik vagy abortál, és felszabadítja a zárat.
 - Ha **abortál**, akkor T végrehajthatja a sor módosítását, függetlenül az elkülönítési szintjétől.
 - Ha a másik tranzakció **véglegesítődik**,
 - akkor T csak akkor hajthatja végre a módosítást, ha az elkülönítési szintje az **olvasásbiztos**.
 - Egy **sorbarendezhető** tranzakció ilyenkor abortál, és „**Cannot serialize access**” hibaüzenetet ad.
- A zárakat az Oracle **automatikusan kezeli**, amikor SQL-utasításokat hajt végre. Mindig a **legkevésbé szigorú zármódot alkalmazza**, így biztosítja a legmagasabb fokú konkurenciát. Lehetőség van arra is, hogy a felhasználó kérjen zárat.



A zárolási rendszer

- Egy tranzakcióban szereplő SQL-utasításnak adott zár a tranzakció befejeződéséig fennmarad (**kétfázisú zárolás**). Ezáltal a tranzakció egy utasítása által végrehajtott változtatások csak azon tranzakciók számára láthatók, amelyek azután indultak el, miután az első tranzakció véglegesítődött.
- Az Oracle akkor **szabadítja fel a zárakat**,
 - amikor a tranzakció **véglegesítődik**
 - vagy **abortál**,
 - illetve ha **visszagörgetjük a tranzakciót egy mentési pontig** (akkor a mentési pont után kapott zárak szabadulnak fel).



Zármódok

- Az Oracle a zárákat a következő általános kategóriákba sorolja:
 1. **DML-zárok (adatzárok):** az adatok védelmére szolgálnak;
 2. **DDL-zárok (szótárzárok):** a sémaobjektumok (pl. táblák) szerkezetének a védelmére valók;
 3. **belső zárok:** a belső adatszerkezetek, adatfájlok védelmére szolgálnak, kezelésük teljesen automatikus.
- **DML-zárákat** két szinten kaphatnak a tranzakciók:
 - **sorok szintjén**
 - és **teljes táblák szintjén.**
- Egy tranzakció **tetszőleges számú sor szintű zárat** fenntarthat.
- **Sorok szintjén** csak egyfélle zármód létezik,
 - a **kizárólagos (írási – X).**



Zármódok

- A többváltozatú időbélyegzés és a sor szintű zárolás kombinációja azt eredményezi, hogy a tranzakciók csak akkor versengenek az adatokért, ha ugyanazokat a sorokat próbálják meg írni. Részletesebben:
 - Adott sorok olvasója nem vár ugyanazon sorok írójára.
 - Adott sorok írója nem vár ugyanazon sorok olvasójára, hacsak az olvasó nem a SELECT ... FOR UPDATE utasítást használja, amely zárolja is a beolvasott sorokat.
 - Adott sorok írója csak akkor vár egy másik tranzakcióra, ha az is ugyanazon sorokat próbálja meg írni ugyanabban az időben.
- Egy tranzakció kizárlagos DML-zárat kap minden egyes sorra, amelyet az alábbi utasítások módosítanak:
 - INSERT,
 - UPDATE,
 - DELETE
 - és SELECT ... FOR UPDATE.



Zármódok

- **Ha egy tranzakció**
 - **egy tábla egy sorára zárat kap,**
 - **akkor a teljes táblára is zárat kap,**

hogy elkerüljük az olyan DDL-utasításokat, amelyek felülírnák a tranzakció változtatásait, illetve hogy fenntartsuk a tranzakciónak a táblához való hozzáférés lehetőségét.
- **Egy tranzakció tábla szintű zárat kap, ha a táblát az alábbi utasítások módosítják:**
 - **INSERT,**
 - **UPDATE,**
 - **DELETE,**
 - **SELECT ... FOR UPDATE**
 - **és LOCK TABLE.**
- **Táblák szintjén ötféle zármódot különböztetünk meg:**
 - 1. row share (RS) vagy subshare (SS),**
 - 2. row exclusive (RX) vagy subexclusive (SX),**
 - 3. share (S),**
 - 4. share row exclusive (SRX) vagy share-subexclusive (SSX)**
 - 5. és exclusive (X).**
- **Ezek a módok a felsorolás sorrendjében egyre erősebbek.**



Zármódok

- A következő táblázat összefoglalja, hogy az egyes utasítások milyen zármódot vonnak maguk után, és hogy milyen zármódokkal kompatibilisek:

| SQL-utasítás | Zármód | RS | RX | S | SRX | X |
|---|--------|----|----|----|-----|---|
| SELECT ... FROM tábla | - | I | I | I | I | I |
| INSERT INTO tábla | RX | I | I | N | N | N |
| UPDATE tábla | RX | I* | I* | N | N | N |
| DELETE FROM tábla | RX | I* | I* | N | N | N |
| SELECT ... FROM tábla ... FOR UPDATE | RS | I* | I* | I* | I* | N |
| LOCK TABLE tábla IN ROW SHARE MODE | RS | I | I | I | I | N |
| LOCK TABLE tábla IN ROW EXCLUSIVE MODE | RX | I | I | N | N | N |
| LOCK TABLE tábla IN SHARE MODE | S | I | N | I | N | N |
| LOCK TABLE tábla IN SHARE ROW EXCLUSIVE MODE | SRX | I | N | N | N | N |
| LOCK TABLE tábla IN EXCLUSIVE MODE | X | N | N | N | N | N |

* Igen, ha egy másik tranzakció nem tart fenn konfliktusos sor szintű zárat, különben várakozik.

- A lekérdezések tehát **sohasem járnak zárolásokkal**, így más tranzakciók is lekérdezhetik vagy akár módosíthatják a lekérdezett táblát, akár a kérdéses sorokat is.
- Az Oracle ezért gyakran hívja a lekérdezéseket **nemblokkoló lekérdezéseknek**. Másrészt a lekérdezések sohasem várnak zárfeloldásra, mindenki végrehajtódhatnak.



Zárak felminősítése és kiterjesztése

- A módosító utasítás a sor szintű záron kívül a módosított sorokat tartalmazó táblákra is elhelyez egy-egy RX zárat.

Ha a tartalmazó tranzakció már fenntart egy S, SRX vagy X zárat a kérdéses táblán,

akkor nem kap külön RX zárat is,

Ha pedig RS zárat tartott fenn,

akkor az felminősül RX zárrá.
- Mivel sorok szintjén csak egyfajta zármód létezik (kizárolagos), nincs szükség felminősítésre.
- Táblák szintjén az Oracle automatikusan felminősít egy zárat erősebb módúvá, amikor szükséges. Például egy SELECT ... FOR UPDATE utasítás RS módban zárolja a táblát. Ha a tranzakció később módosít a zárolt sorok közül néhányat, az RS mód automatikusan felminősül RX módra.



Zárak felminősítése és kiterjesztése

- Zárak kiterjesztésének (escalation) nevezük azt a folyamatot, amikor a szemcsézettség egy szintjén (pl. sorok szintjén) lévő zárakat az adatbázis-kezelő rendszer a szemcsézettség egy magasabb szintjére (pl. a tábla szintjére) emeli.
- Például ha a felhasználó sok sort zárol egy táblában, egyes rendszerek ezeket automatikusan kiterjesztik a teljes táblára.
- Ezáltal csökken a zárak száma, viszont nő a zárolt elemek zármódjának erőssége.
- Az Oracle nem alkalmazza a zárkiterjesztést, mivel az megnöveli a holtpontok kialakulásának kockázatát.



Összefoglalás

- Figyelmeztető zárak csoportos módja
- Nem ismételhető olvasás, fantomok, megszorítások
- Indexek zárolása, mászóka-elv, Fa protokoll
- Zárakat nem használó ütemezők (időbényegzéses, érvényesítéses)
- Az időbényegzéses ütemező optimista működése, túl késői olvasás, túl késői írás, piszkos adatok olvasása, Thomas-féle írás
- Oracle konkurenciakezelése (két szintű olvasási konzisztencia, két szintű zárolása, figyelmeztető csoportos módú zárak a táblákra, felminősítés, többváltozatú időbényegzés)
- A tranzakciók 4 féle ISO szabvány szerinti elkülönítési szintje

