Információs rendszerek

Tematika

- **Helyreállíthatóság (Failure Recovery) (8. fejezet)**
- **Konkurenciavezérlés (Concurrency control) (9. fejezet)**
- **Tranzakciókezelés (Transaction processing) (10. fejezet)**
- **Oracle megoldások**

Az adatok helyessége

- **Azt szeretnénk, hogy az adatok mindig pontosak, helyesek legyenek!**

EMP

Name	Age
White	52
Green	**3421**
Gray	1

Konzisztencia, megszorítások

- **Mit jelent a konzisztencia?**
- **Az adatok előre megadott feltételeket, predikátumokat elégítenek ki.**
- **Például:**
  - **X az R reláció kulcsa**
  - **X → Y függőség teljesül R-ben**
  - **megengedett értékek korlátozása:**
    - **Domain(X) = {piros, kék, 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

A feltételrendszert ontológiának hívjuk.

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.

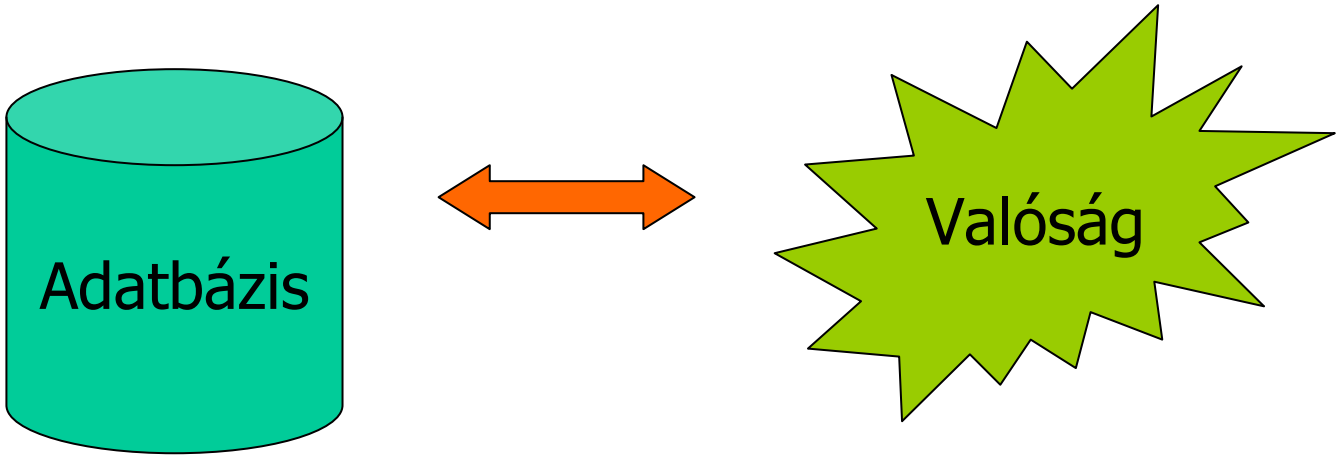
Ha törölve=igen, akkor egyenleg=0.

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!

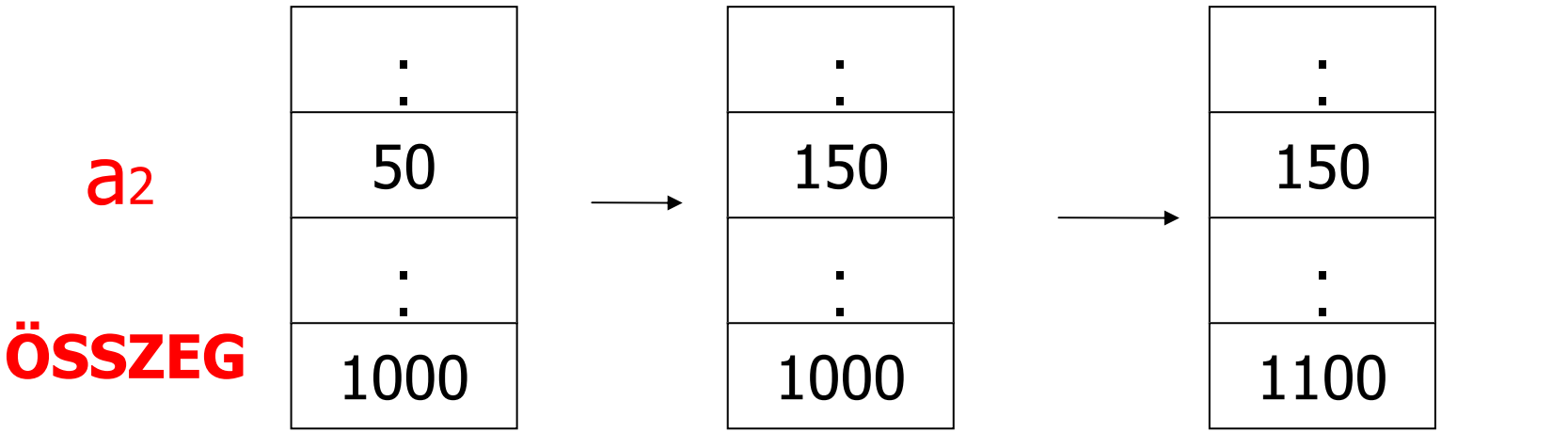
Például:

$a_1 + a_2 + \ldots a_n$ = **ÖSSZEG** (**megszorítás**)

Adjunk 100-at az a_2-höz:

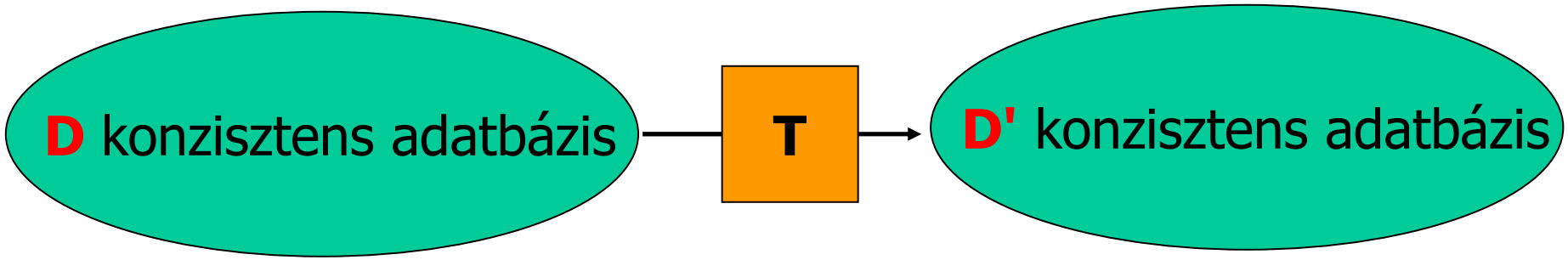
$$\begin{cases} a_2 \leftarrow a_2 + 100 \\ \text{ÖSSZEG} \leftarrow \text{ÖSSZEG} + 100 \end{cases}$$

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



Konzisztens **NEM Konzisztens** **Konzisztens**

TRANZAKCIÓ:
Konzisztenciát megtartó adatkezelő műveletek sorozata



Ezek után mindig feltesszük:

Ha T tranzakció konzisztens állapotból indul

+ T tranzakció csak egyedül futna le

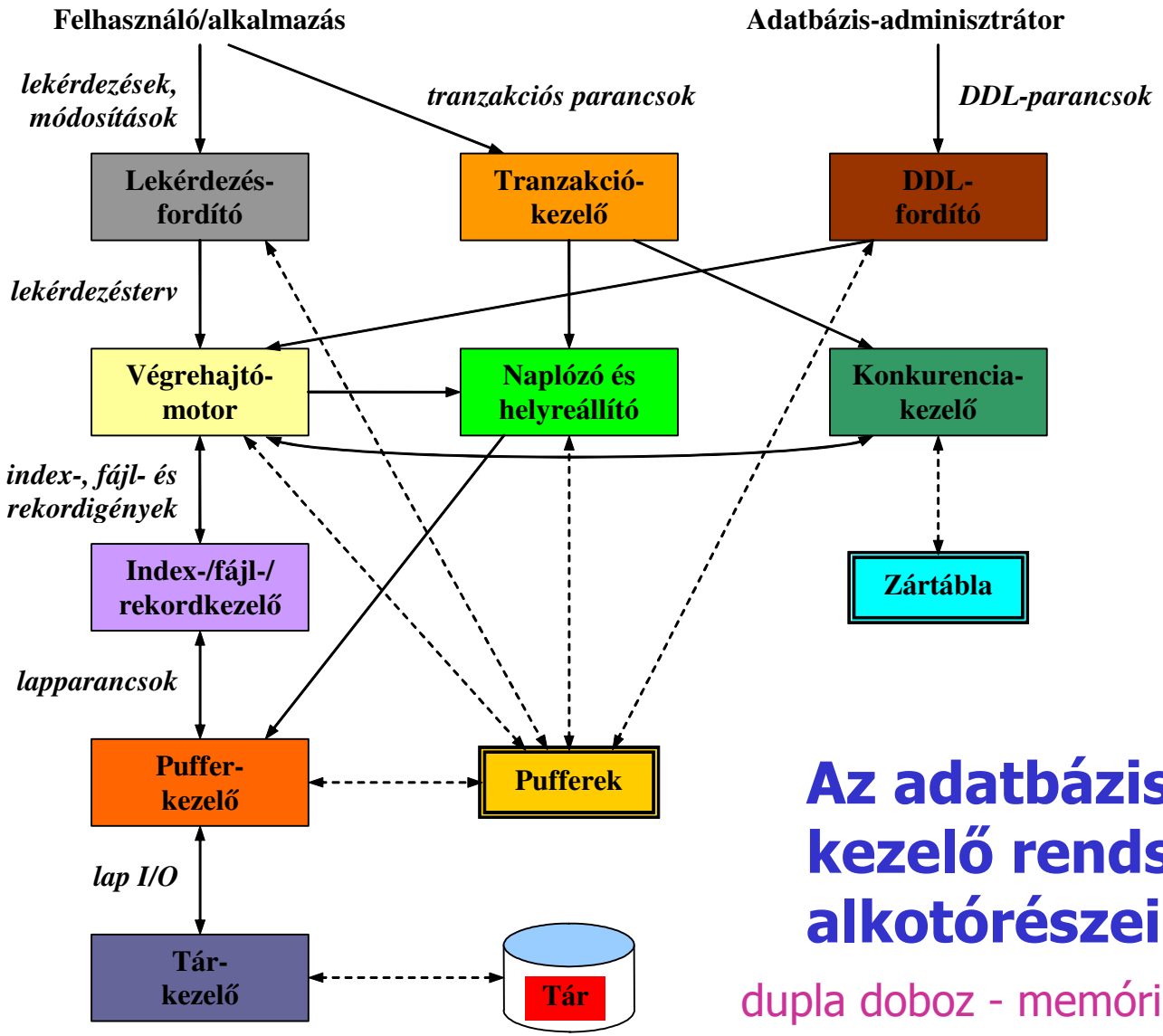
⇒ T konzisztens állapotban hagyja az adatbázis

Helyesség feltétele

1. **Ha leáll valamelyik, vagy több tranzakció (abort, vagy hiba miatt), akkor is**

   konzisztens D adatbázist kapunk

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 **tartós**nak 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.
  - ***Naplózás- 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 „mindent 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 adatelemekre 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 mindig 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 paranccsal végződik. Ha a tranzakció valamely utasítása egy triggert aktivizál, akkor a trigger is a tranzakció részének tekintendő, akárcsak a trigger által kiváltott további triggerek. (A trigger olyan programrész, amely bizonyos események bekövetkeztekor automatikusan lefut.)**

A tranzakció feldolgozása

A tranzakciófeldolgozó a következő 3 feladatot hajtja végre:

- **naplózás**
- **konkurenciavezérlés**
- **holtpont feloldása**

1. ***Naplózás*:**
   - **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 valamilyen 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 résztevékenységeinek 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 egészként hajtottuk volna végre.**

T1. tranzakció: $u1; u2; : : : ; u10$

T2. tranzakció: $v1; v2; : : : ; v103$

A két utasítássorozat nem elkülönülve jön, hanem összefésülődve:

$$u1; v1; v2; u2; u3; v3; : : : ; v103; u10$$

A saját sorrend megmarad mindkettőn belül.

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 eggyel nő A értéke, holott kettővel kellett volna.

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

A tranzakció feldolgozása

***3. Holtpont feloldása*:**

- **A tranzakciók az ütemező által engedélyezett zárak alapján versenyeznek az erőforrásokért. Így 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.**

Holtpont (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 komponens 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 konzisztenciasérülést?

- 8. fejezet: **hibák utáni helyreállítás**
- 9. fejezet: **párhuzamos végrehajtás miatti hibák**
- 10. fejezet: **adatmegosztás, párhuzamos végrehajtás és rendszerhibák**

Helyreállítás

- **Első lépés: meghibásodási modell definiálása**
- **Események osztályozása:**

Események → **szabályos**

→ **kivételes** → **előrelátható**

→ **nem várt**

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 dolog 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 olvashatatlanná 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 konzisztenciáját.

Előreláthatható, kivételes események

- **Katasztrofális 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 katasztrofális 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.**

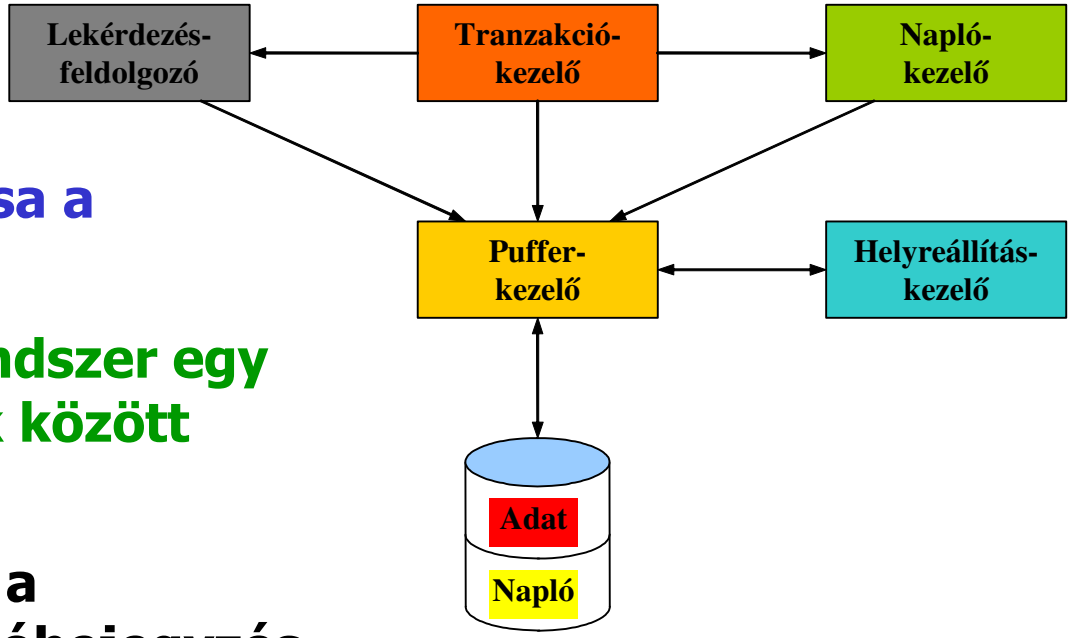
Előrelátható, kivételes események

- Tipikus rendszerhibák az áramkimaradásból és a szoftverhibákból 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.

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 mindig feltesszük, hogy a háttértár 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: naplózás

2. Hiba után helyreállítás: a napló segítségével egy konzisztens állapot helyreállítása

Természetesen a naplózás és a hiba utáni helyreállítás összhangban vannak, de van több különböző naplózási 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).

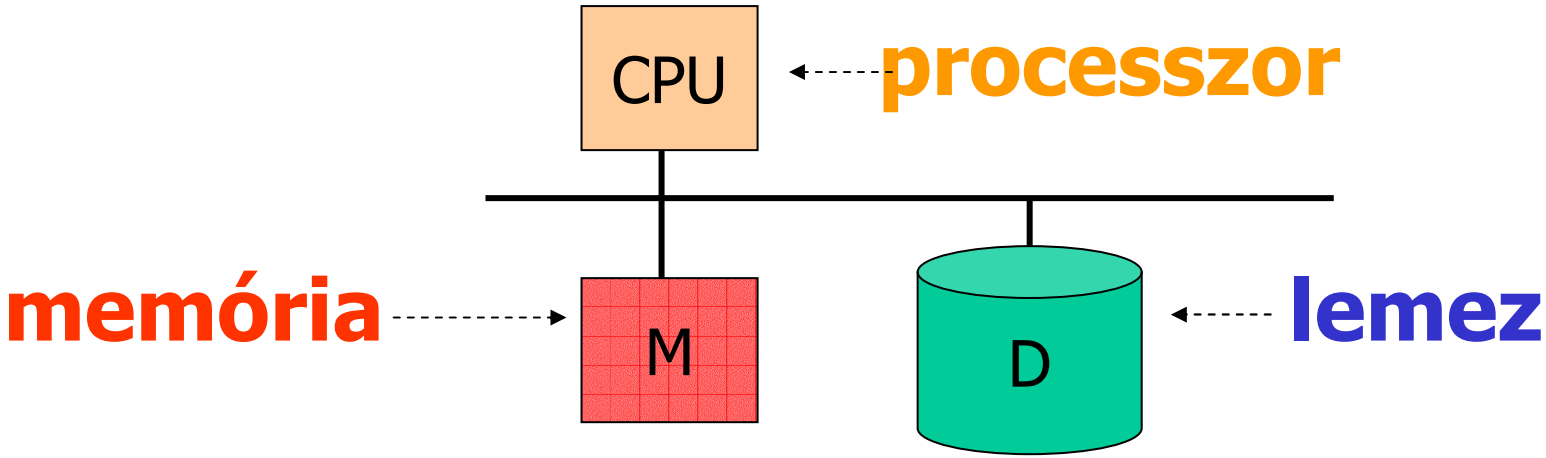
Az adatbáziselem lehet:

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

Ez utóbbi a legjobb választás a naplózás 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

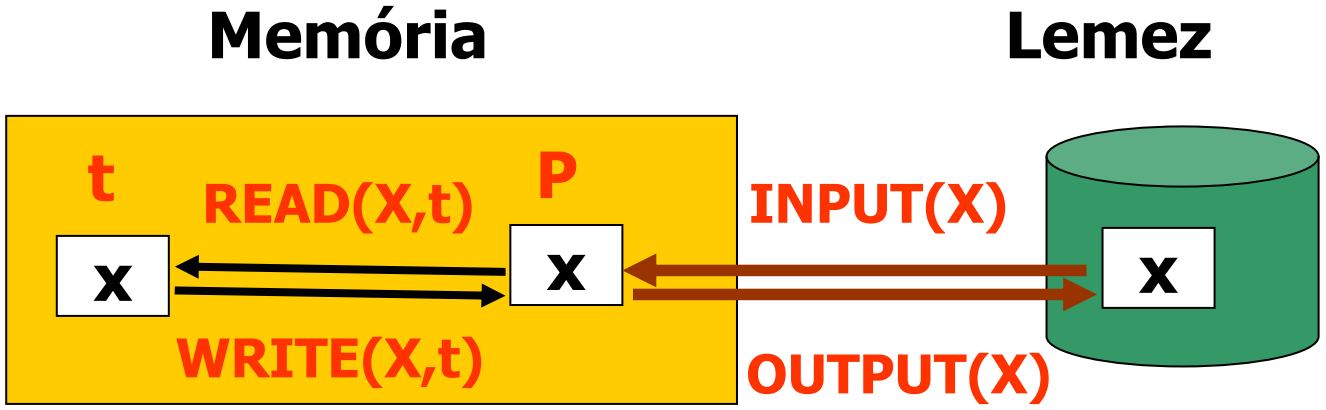
A tranzakció és az adatbázis kölcsönhatásának három fontos helyszíne van:



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 adatbáziselemet 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 adatbáziselemet tartalmazó lemezblokk másolása a memóriapufferbe.**
2. **READ(X,t): Az X adatbáziselem bemásolása a tranzakció t lokális változójába. Részletesebben: ha az X adatbáziselemet 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 adatbáziselemet 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 adatbáziselemet 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

T_1: $A \leftarrow A \times 2$

$B \leftarrow B \times 2$

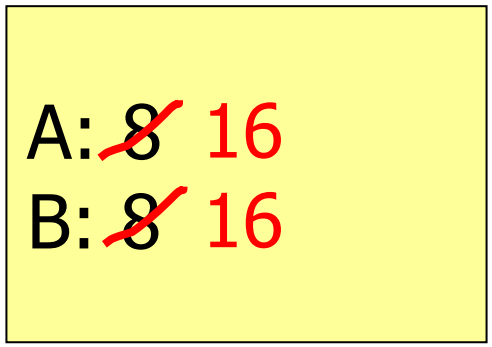
Pontosabban 8 lépésből áll:

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

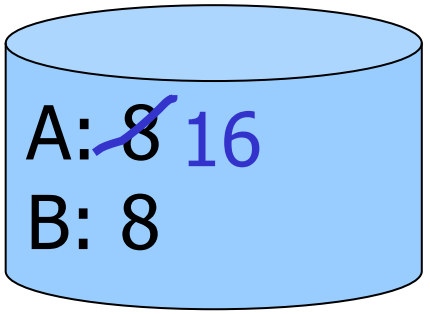
T_1: Read (A,t); $t \leftarrow t \times 2$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);

Rendszerhiba!

Inkonzisztens maradna!



MEMÓRIA



LEMEZ

Az értékek változása a memóriában és a lemezen

	Tevékenység	*t*	*M*-A	*M*-B	*D*-A	*D*-B
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 történéseit 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 naplózási protokolltól függően)

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

A napló időrendben tartalmazza a történéseket és tipikusan a háttértáron 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.

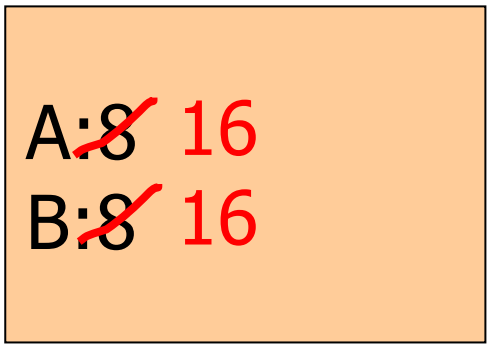
KÉTFÉLE NAPLÓZÁS:

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

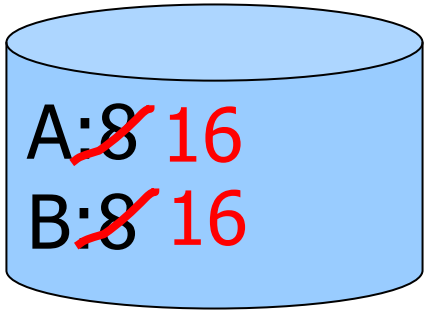
Undo log (SEMMISSÉGI NAPLÓZÁS)

T_1: Read (A,t); $t \leftarrow t \times 2$ A=B
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);

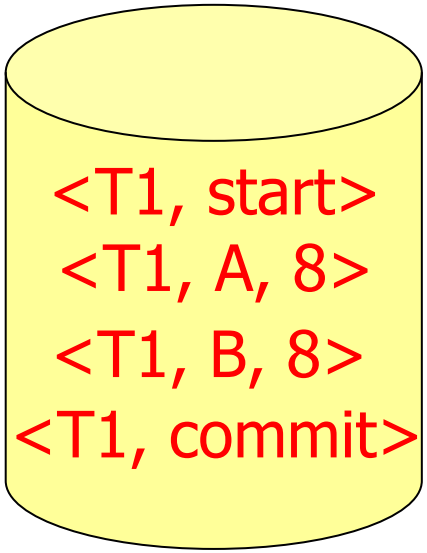
- **A régi értéket naplózzuk!**
- **Azonnali kiírás!**



MEMÓRIA



LEMEZ



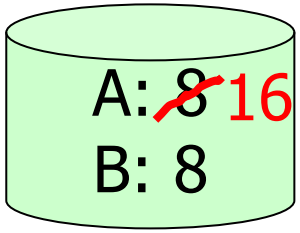
NAPLÓ

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

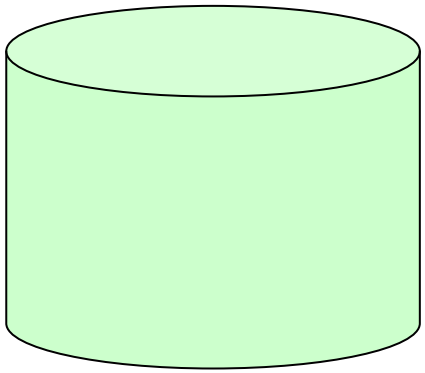
A: ~~8~~ 16
B: ~~8~~ 16
Napló:
<T_1,start>
<T_1, A, 8>
<T_1, B, 8>



Adatbázis

1. ROSSZ állapot

Napló

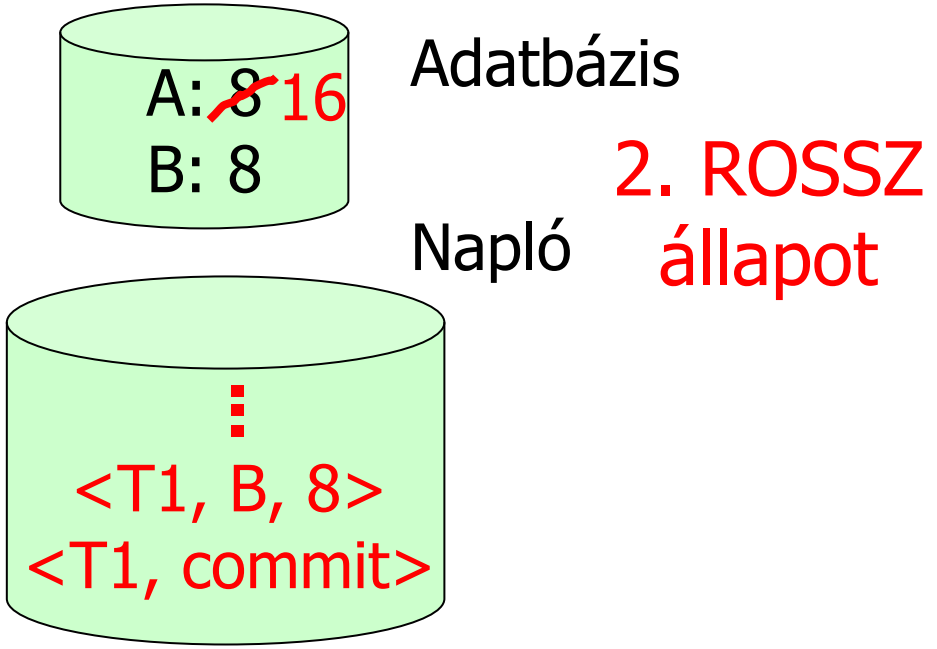


Mikor írjuk ki a naplót a lemezre?

- **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:
<T_1,start>
<T_1, A, 8>
<T_1, B, 8>
<T1, commit>



Undo naplózás szabályai

U1. Ha a T tranzakció módosítja az X adatbáziselemet, 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

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						

Helyreállítás UNDO napló alapján

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

- **For every Ti with <Ti, start> in log:**
  - **If <Ti,commit> or <Ti,abort> in log, do nothing**
  - **Else**
    - **For all <Ti, *X*, *v*> in log:**
      - **write (*X*, *v*)**
      - **output (*X*)**
    - **Write <Ti, abort> to log**

Jó ez így?

Helyreállítás UNDO napló alapján

A helyreállítás során fontos a módosítások sorrendje!

(1) Let S = set of transactions with
    <Ti, start> in log, but no
    <Ti, commit> (or <Ti, abort>) record in log

(2) For each <Ti, X, v> in log,
    in reverse order (latest → earliest) do:
    - if Ti ∈ S then
        - write (X, v)
        - output (X)

(3) For each Ti ∈ S do
    - write <Ti, abort> 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 írjunk (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 előről 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 lemezen. Ettől függetlenül mindkét adatbáziselem korábbi értékét (8) állítjuk vissza.**

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 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>

RENDSZERHIBA

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

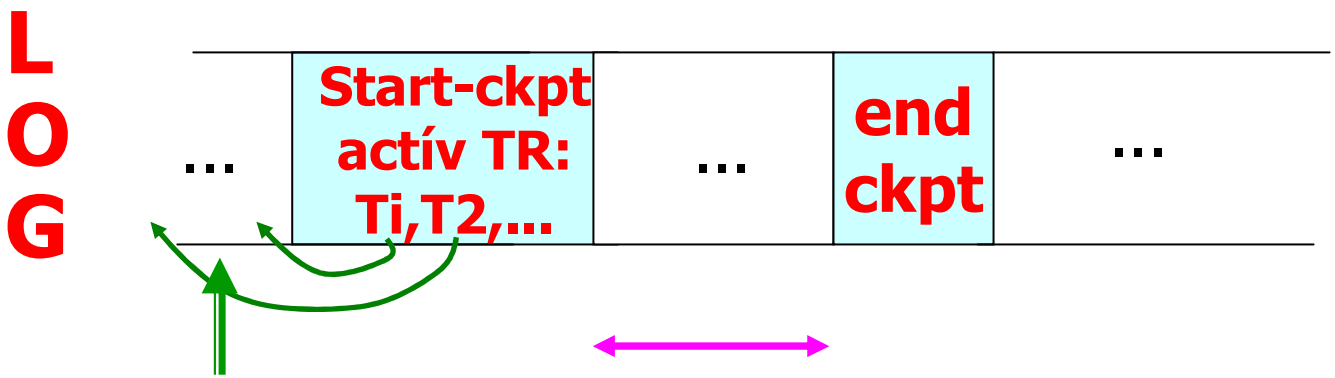
- Tegyük fel, hogy a naplórészlet végén rendszerhiba lép fel.
- HELYREÁLLÍTÁS: A naplót a végétől visszafelé elemezve T3-at fogjuk az egyetlen be nem fejezett tranzakciónak találni, így E és F korábbi értékeit kell csak visszaállítanunk.
- Amikor megtaláljuk a <CKPT> bejegyzést, tudjuk, hogy nem kell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

Az ellenőrzőpont-képzés

- Probléma: Hosszú ideig tarthat, amíg az aktív tranzakciók befejeződnek. (Új tranzakciókat sokáig nem lehet kiszolgálni.)
- Megoldás: CHECKPOINT képzése működés közben.
- A módszer lépései:

1. <START CKPT(T1,...,Tk)> naplóbejegyzés készítése, majd lemezre írása (FLUSH LOG), ahol T1,...,Tk az éppen aktív tranzakciók nevei.
2. Meg kell várni a T1,...,Tk tranzakciók mindegyikének normális vagy abnormális befejeződését, nem tiltva közben újabb tranzakciók indítását.
3. Ha a T1,...,Tk tranzakciók mindegyike befejeződött, akkor <END CKPT> naplóbejegyzés elkészítése, majd lemezre írása (FLUSH LOG).

Működés közbeni ellenőrzőpont (non-quiescent checkpointing)

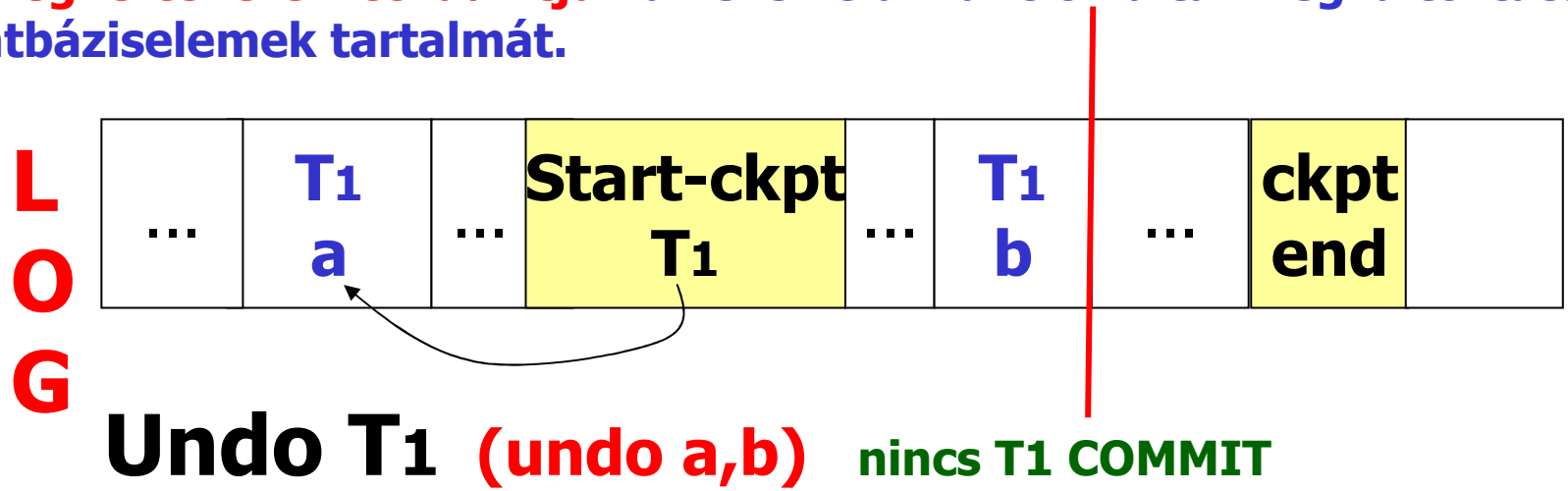


Csak az aktív tranzakciók előzményét kell megnézni!

A be nem fejezett tranzakciók által írt, "PISZKOS" adatok kerülhettek 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 aszerint, hogy visszafelé olvasva a naplót az <END CKPT> vagy a <START CKPT(T1,…,Tk)> 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,…,Tk)> 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,…,Tk)> naplóbejegyzést találjuk előbb:

2. **Ha a <START CKPT(T1,…,Tk)> naplóbejegyzéssel találkozunk előbb, az azt jelenti, hogy a katasztrófa az ellenőrzőpont-képzés közben fordult elő, ezért a T1,…,Tk 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 haladnunk.**

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

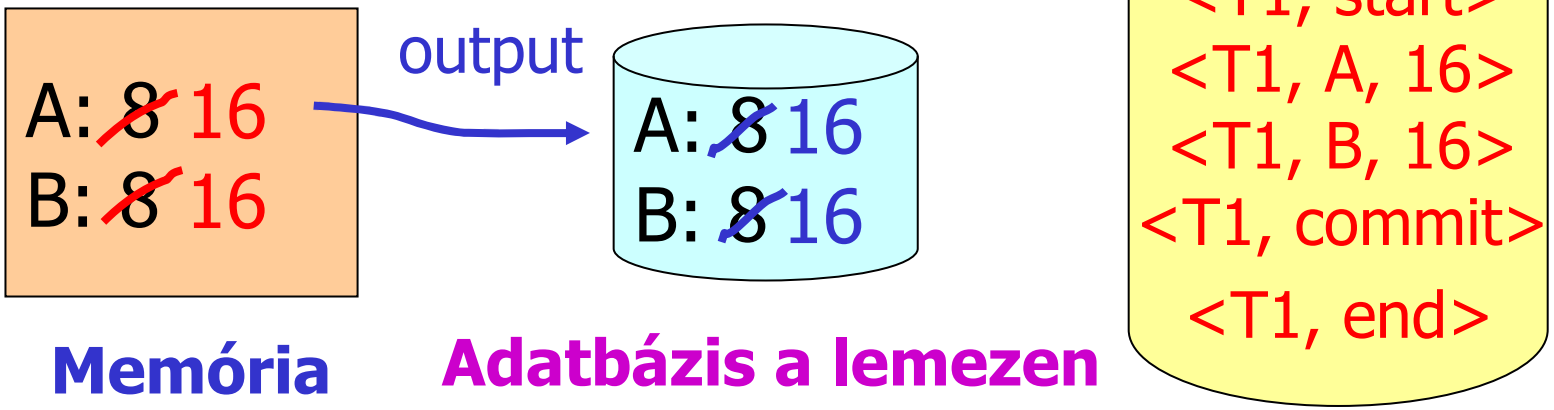
1. <T1, START >
2. <T1,A,5>
3. <T2, START >
4. <T2,B,10>
5. <START CKPT(T1,T2)>
6. <T2,C,15>
7. <T3, START >
8. <T1,D,20>
9. <T1, COMMIT >
10. <T3,E,25>
11. <T2, COMMIT >
12. <END CKPT>
13. <T3,F,30>

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.

Redo logging (Helyrehozó naplózás)

T1: Read(A,t); t← t×2; write (A,t);
Read(B,t); t←t×2; write (B,t);
Output(A); Output(B)



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

A helyrehozó naplózás szabálya

R1. Mielőtt az adatbázis bármely **X** elemét a lemezen módosítanánk, az **X** módosítására vonatkozó összes naplóbejegyzésnek, azaz **<T,X,v>**-nek és **<T, COMMIT>**-nak a lemezre kell kerülnie.

A helyrehozó naplózás 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

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,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 Ti with <Ti, commit> in log:
  - For all <Ti, X, v> in log:
    - Write(X, v)
    - Output(X)

Jó ez így?

Helyreállítás a REDO naplóból

A helyreállítás során fontos a módosítások sorrendje!

(1) Let S = set of transactions with <Ti, commit> (and no <Ti, end>) in log

(2) For each <Ti, X, v> in log, **in forward order** (**earliest** $\rightarrow$ **latest**) do:

- if Ti $\in$ S then { Write(X, v); Output(X) }

(3) For each Ti $\in$ S, write <Ti, end>

Ez miért lesz jó?

Helyreállítás a módosított REDO naplóból

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

1. Meghatározzuk a befejezett tranzakciókat (COMMIT).
2. Elemezzük a naplót az elejétől kezdve. Minden <T,X,v> 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 <T, ABORT> naplóbejegyzést kell a naplóba írni, és a naplót ki kell írni lemezre (FLUSH LOG).

Helyreállítás

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,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 elejétől 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

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,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

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,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 elé => megtelhet a puffer**
- **Az UNDO protokoll esetleg túl gyakran akar írni => itt el lehet halasztani az írást**

Helyrehozó naplózás 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 naplózás 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 naplózásban a működés közbeni ellenőrzőpont-képzés a következőkből áll:

1. <START CKPT(T1,…,Tk)> naplóbejegyzés elkészítése és lemezre írása, ahol T1,…,Tk 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>

RENDSZERHIBA

<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.

- 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>

RENDSZERHIBA

<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.

- 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>

RENDSZERHIBA

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>
10. <END CKPT>
11. <T2, COMMIT>
12. <T3, COMMIT>

RENDSZERHIBA

- 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.

Semmisségi/helyrehozó (undo/redo) naplózás

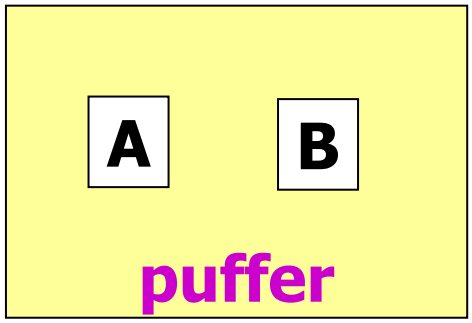
- **A semmisségi naplózás esetén az adatokat a tranzakció befejezésekor nyomban lemezre kell írni, nő a végrehajtandó lemezműveletek száma.**

- **A helyrehozó naplózás 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) naplózás

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 <T,X,v,w> 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 <T,X,v,w> naplóbejegyzésnek lemezre kell kerülnie.

- WAL – Write After Log elv: előbb naplózunk, utána módosítunk

- NAGYOBB SZABADSÁG: A <T, COMMIT> bejegyzés megelőzheti, de követheti is az adatbáziselemek lemezen történő bármilyen megváltoztatását.

- NAGYOBB MÉRETŰ NAPLÓ: - régi és új értéket is tároljuk

UNDO/REDO naplózás

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 elé 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 befejezett tranzakció hatását.**
2. **(UNDO): A legutolsótól kezdve tegyük semmissé minden be nem fejezett tranzakció 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, mind 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) mindig 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 <T,COMMIT> 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 <T,COMMIT> naplóbejegyzést nyomban lemezre kell írni, amint megjelenik a naplóban.**
- **Ennek teljesítéséért a fenti példában a 10) lépés (<T,COMMIT>) 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.

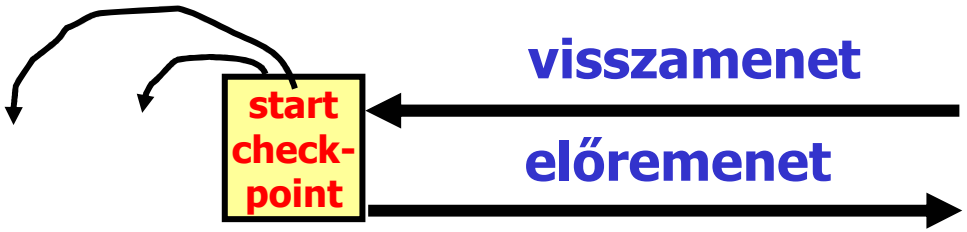
Semmisségi/helyrehozó naplózás ellenőrzőpont-képzéssel

- Egyszerűbb, mint a másik két naplózás esetén.

1. Írjunk a naplóba <START CKPT(T1,…,Tk)> naplóbejegyzést, ahol T1,…,Tk az aktív tranzakciók, majd írjuk a naplót lemezre.
2. Írjuk lemezre az összes piszkos puffert, tehát azokat, melyek egy vagy több módosított adatbáziselemet tartalmaznak. A helyrehozó naplózástól eltérően itt az összes piszkos puffert lemezre írjuk, nem csak a már befejezett tranzakciók által módosítottakat.
3. Írjunk <END CKPT> naplóbejegyzést a naplóba, majd írjuk a naplót lemezre.

Helyreállítás:

- **Visszamenet** (napló végétől ➜ 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 ➜ a napló végéig)
  - helyrehozzuk az S tranzakcióinak hatását



Semmisségi/helyrehozó naplózás 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 naplózásban.
- 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 naplózás 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 naplózás 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> bejegyzéséig 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

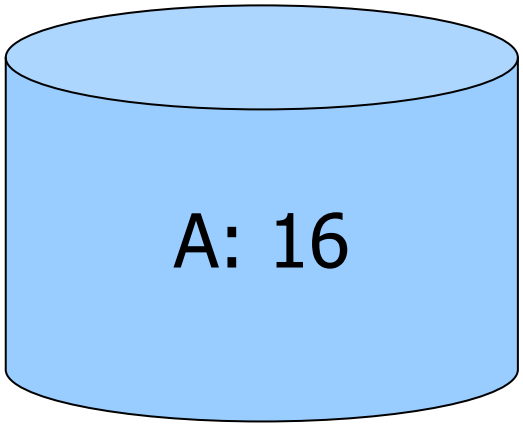
- Ekkor T2-t befejezett, T3-at pedig befejezetlen tranzakciónak kell tekintenünk.
- T2 tevékenységét helyreállítandó C értékét a lemezen 15-re írjuk; B-t már nem kell 10-re írnunk a lemezen, mert tudjuk, hogy ez már lemezre került az <END CKPT> előtt.
- A helyreállító naplózástól eltérően T3 hatásait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

Semmisségi/helyrehozó naplózás 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 a katasztrófa az <END CKPT> bejegyzés előtt lép fel, akkor figyelmen kívül hagyjuk az utolsó START CKPT bejegyzést, és az előzőek szerint járunk el.

Az eszközök meghibásodásának kezelése

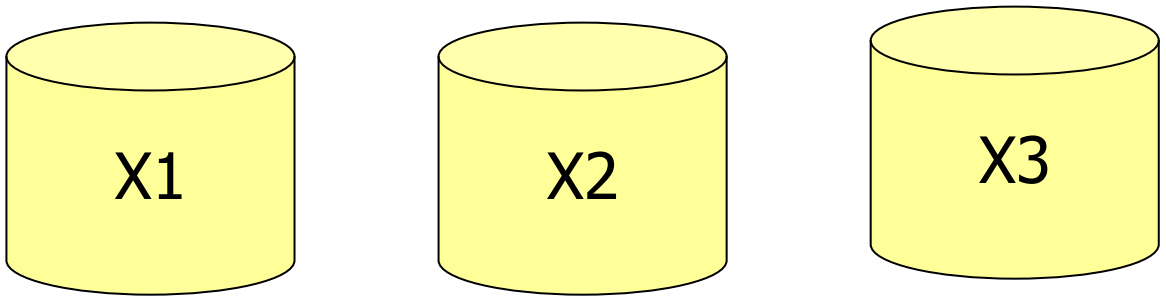


Megoldás: **Készítsünk másolatokat (backup - mentés) az adatokról!**

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:

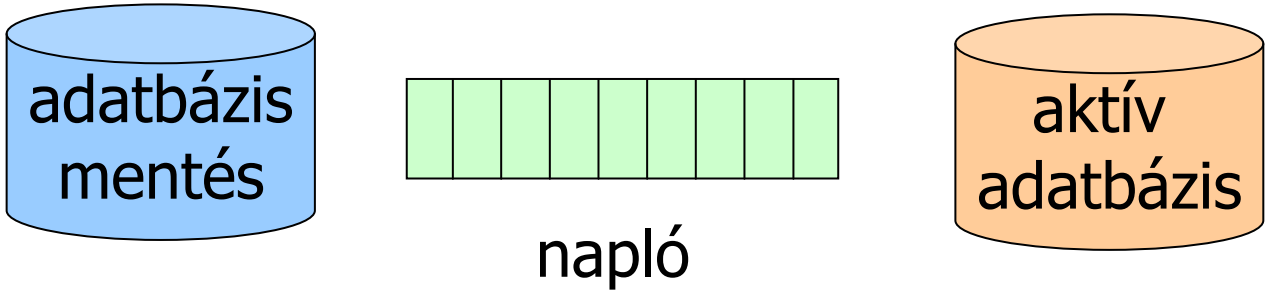
2. Többszörös írás, egyszeres olvasás

- 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:

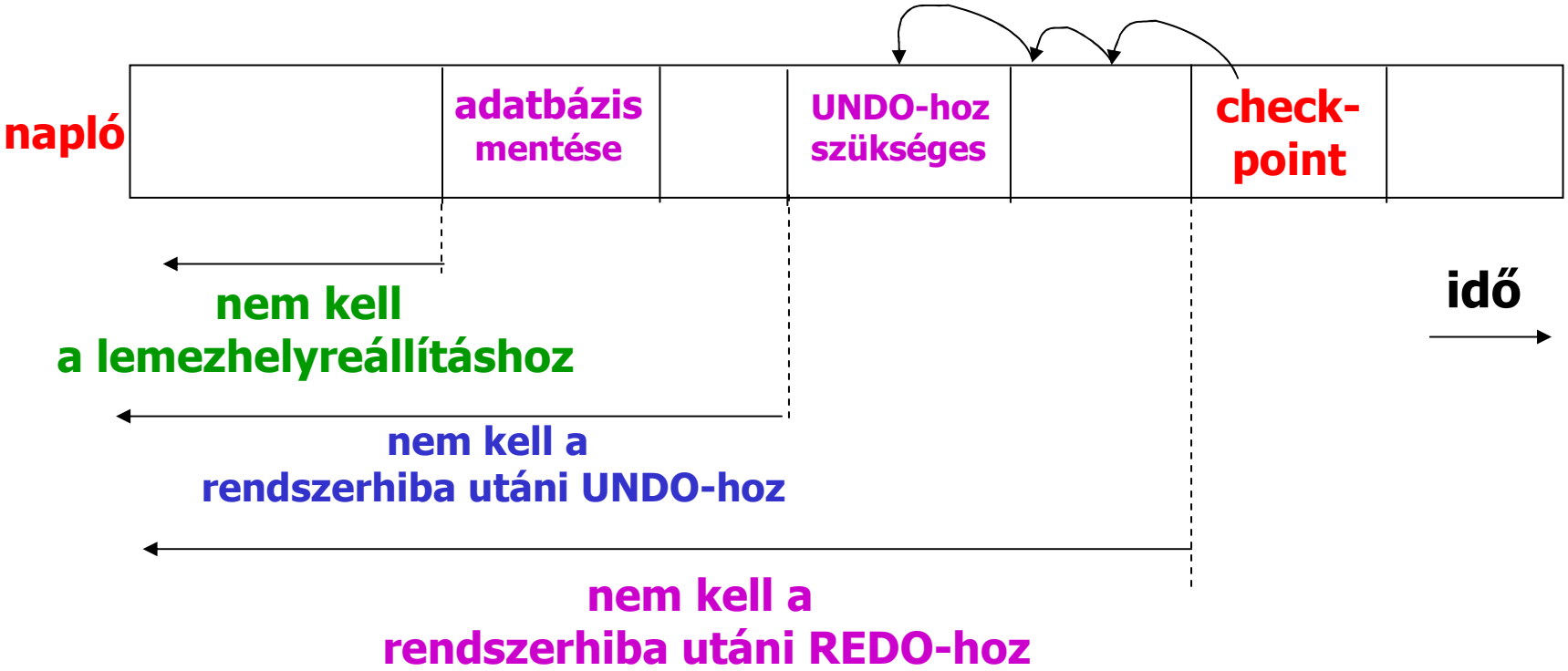
3: Adatbázis mentés + napló



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 naplózás 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 vag UNDO/REDO naplózási 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 naplózás 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>

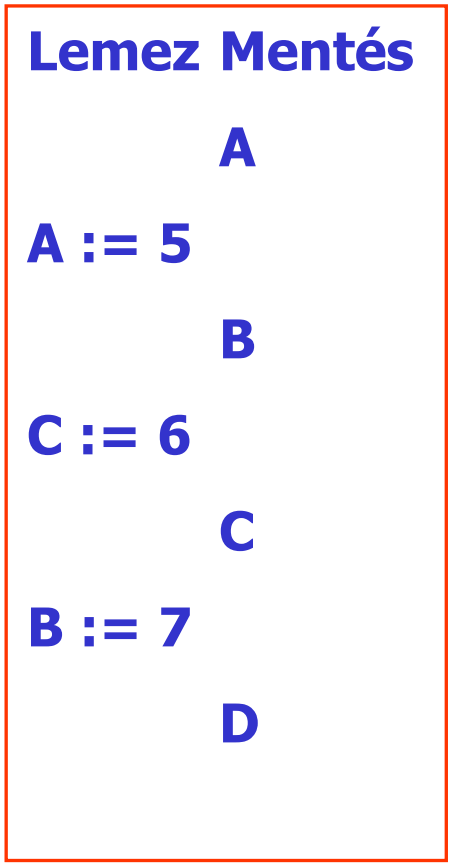
Lemez	Mentés
	A
A := 5	
	B
C := 6	
	C
B := 7	
	D

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.



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 naplófájlokból 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 tételéhez 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 bejegyzésekből* 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 kezelni. 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 mindig 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 tételéhez 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.