november 9.

Hálós adatmodell

CODASYL – DBTG ~1970-81

mai unortodox (NoSQL, Big Data) rendszereknél kiderült, hogy ezek a régi dolgok túl jók, pedig évtizedekig nem nyúltak hozzá.

elemi adat → n-esek / tuple / rekord → relációs adatbázis

elemi adat → rekord → set → hálós adatbázis

rekord: egyedpéldányok; set: kapcsolatpéldányok

Def.: Rekord típus:

$$T_R \quad (A_1, A_2, \dots, A_n) \rightarrow \text{attribútumnevek}$$

$$\uparrow \quad \uparrow \quad \uparrow$$

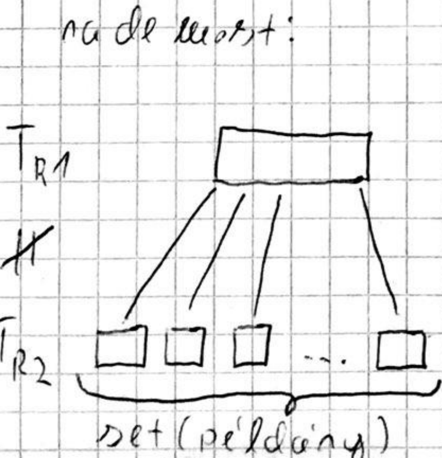
$$D_1 \quad D_2 \quad D_n \rightarrow \text{domain-ek}$$

Rekord példány:

$$(d_1, d_2, \dots, d_n) \in D_1 \times D_2 \times \dots \times D_n \quad (d_i \in D_i \ \forall i)$$

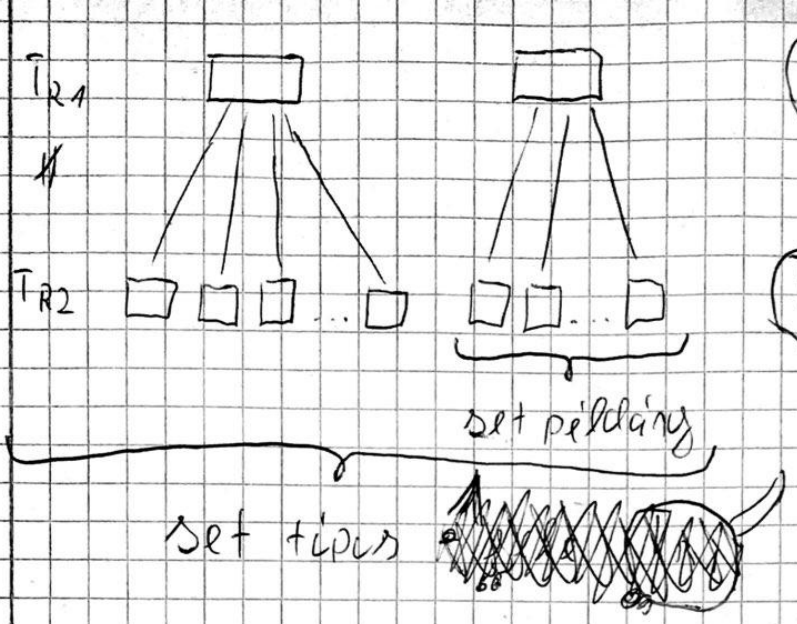
ezen a szinten nincsen semmilyen nagyon érdemi eltérés a relációs világtól.

na de most:



az egyikből pontosan 1 példány

a másikból 0..∞ akármennyi



owner rekord típus

nyíl: set típus
member rekord típus

itt most lehet látni, hogy itt a rekordtípusok közötti valamilyen függőségről dolog van

formálisabban

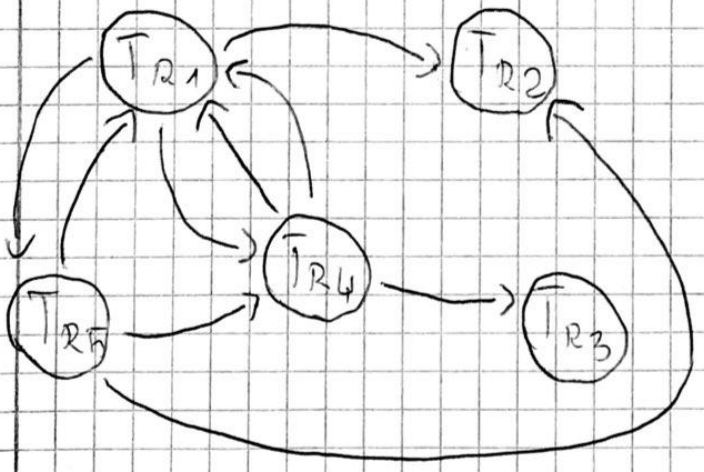
$$S: \underbrace{\tilde{f}}_{\text{összes példánya } T_{R2}\text{-nek}}(T_{R2}) \longrightarrow \tilde{f}(T_{R1})$$

member rek. owner rek.

adott owner adott member rekord

így adott rekord mint (példány) azonosító + egy adott set példányt, különben nem lenne egyértelmű a leképezés

pl.: hálós séma

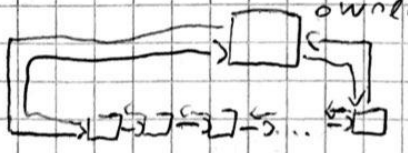


keresésnél egy rekord példányig akarunk eljutni.

valahogy keresésnél ebben a hálóban kell navigálni

keresés gyakorlata

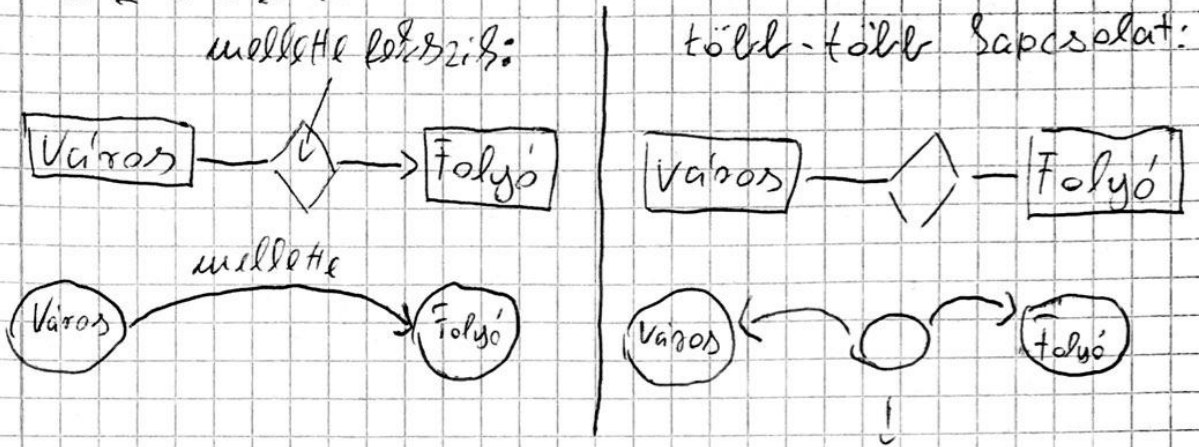
implementáció: pl. láncolt lista



ezek a pointerek tipikusan a rekordban vannak

Mire jó ez?

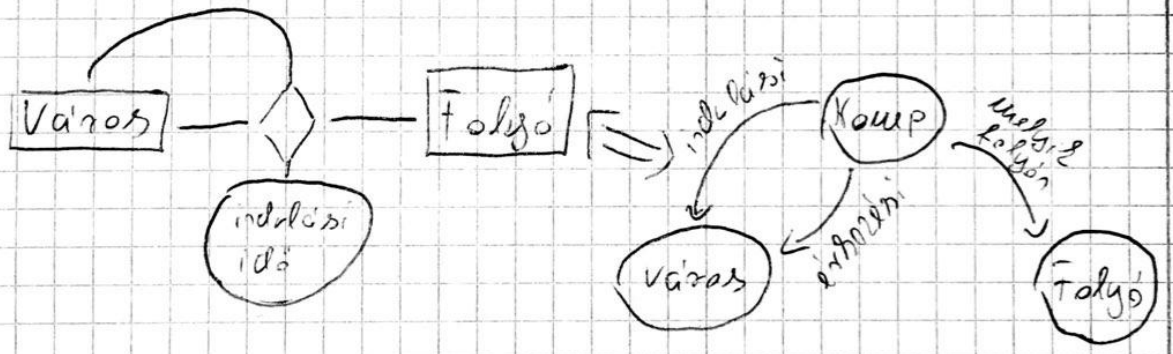
ER leképezésre hálós sémára



a többi féle leképezés megtalálható a jegyzetben, javallott megnézni, mert nem is annyira triviális.

!

mese: kompjáratok folyókon át városokat kötnek össze



azt várnád az emberi, hogy a sok mutató miatt a hálós adatmodell lassabb, mint a relációs,

de ez nem igaz!: ;H a dolgok össze vannak kötve, nem kell keresgélni, csak navigálni!

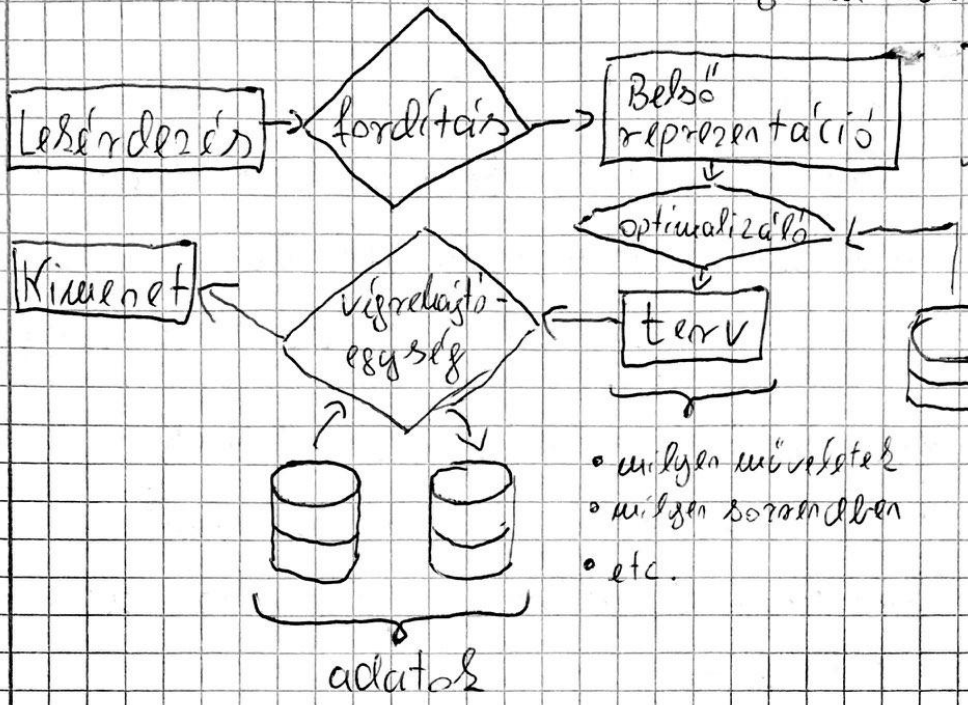
általában egy nagyságrenddel gyorsabb.

régen a gyenge gépek korában mindent szét kellett optimalizálni.

(előadás 11.)

Lekérdezésoptimalizálás

deklaratív lekérdezéseket valahogy át kell alakítani sima imperatív programkódlá.



} tipikusan valami relációalgebra szerű

} tipikusan nem a konkrét adatokat, csak valamilyen statisztikát

ugyanarra a lekérdezésre számtalan lehetséges relációalgebrai kifejezés adható.

∃ formális átalakításos relációalgebrai kifejezésre

ezek minőségére vonatkozóan,
nélkülözhetetlen, nélkülözhető
ezeket felismerő, kiszűrő eljárások.

Heurisztikus (szabály-alapú) optimalizálás:

az adatok statisztikáit nem veszi figyelembe

A heurisztikus optimalizáció egész egyszerű:

- sima kanonikus alak
- szelekciók süllyesztése: ezeket jobb minél hamarabb elvégezni, hogy kevesebb adattal dolgozhassunk
- levelek átrendezése
- θ-illesztés bevezetése: keresztszorzat helyett
- projekció süllyesztése: ugyanabból a motivációból, mint a szelekciók süllyesztése

ez adatok ∀-ra működik, teljesen általános optimalizáció. Ha erősebb dolgokat akarunk csinálni, akkor már figyelembe kell venni az adatok statisztikáit.

Költség alapú optimalizálás

valamilyen költségfüggvény alapján próbál optimalizálni.

↳ de mi alapján?

- a fizikai adatszervezésnél megszokhattuk, ez a blokkműveletek száma alapján történik

itt használunk mindenféle mérőszámokat, melyeknek egy része a fizikai adatszervezésből megszokott, néhány új.
ezeket most ide feleslegesen felsorolnám.

november 16.

- Objektum-orientált rendszerek:

  Saját feldolgozási környezet

  de most:

  ## Többfelhasználós működés

  többáltalános probléma.

  Legyen a tranzakció fogalmán át kívánt megközelíteni

  atom: műveletek sorozata,
  vagy mind sikeres, vagy
  egy sem

- Probléma: többszörös használatnál: → elemi anomáliák

  a.) - elveszett módosítás
  (lost update)

T_1	T_2
READ A	
A = A+1	READ A
	A = A+1
	WRITE A
WRITE A	

→ ütemezés

- nem megismételhető olvasás
  (non-repeatable read)

T_1	T_2
READ A	
READ B	
B = B+A	
WRITE B	
	READ A
	A = A+1
	WRITE A
READ A	
READ C	
C = A+C	
WRITE C	

- fantom olvasás
  (phantom-read)

T_1	T_2
READ $\tilde{r}$	
	INSERT y TO $\tilde{r}$
READ $\tilde{r}$	

- piszkos olvasás
  (dirty read)

T_1	T_2
READ A	
A = A+1	
WRITE A	
	READ A
	A = A+1
	WRITE A
	COMMIT
ABORT	

$\tilde{r}$: rekordok halmaza

alapvető igazság: ha nincs közös adat, akkor nem léphetnek fel ilyen anomáliák

a közösen használt adatok hozzáférését kell szabályozni

mit várunk el a tranzakcióktól az adatbázis kezelés kontextusában?

Atomicity
Consistency → rajta csak sikeresen befejezett tranzakciók hatása jelenik meg
Isolation → olyan eredményeket várunk el, mintha a tranzakciók egyedül futnának
Durability → védve legyenek az adatok mindenféle problémától, veszélytől (áramszünet, atomcsapás)

Zárak: adathozzáférési privilégium, adható, visszavonható

→ erre lesznek jók.

közös erőforrás: adatok

szintaxis: LOCK A
UNLOCK A

LOCK A →

UNLOCK A

T_1 { READ A   A=A+1   WRITE A }

T_2 {   READ A   A=A+1 WRITE A   }

→ ténylegesen itt fog lefutni

a zárak tök jók, de azért van velük egy rakás probléma.

Problémái, mint:

- Holtpont (deadlock)
- Éhezés (livelock)

Megoldásai:
szerializálás zárásra

(FIFO) ütemezés

Def.: várakoztatási gráf
csúcspontok: tranzakciók
élek: $T_i \to T_j$ ∃, ha T_i várakoztatja T_j-t

Tétel:
∄ deadlock, pontosan akkor, ha a várakoztatási gráf DAG (körmentes)

Biz.: triviális

Def.: Soros ütemezés: nem lappolódnak át tranzakciók (izolációs tulajdonság)
nem soros ütemezés: átlappolódnak
→ sorosítható: ∃ soros ekvivalense
nem sorosítható: ∄ — " —

az a soros ütemezés, mely a adatelemekhez tranzakcióinak ugyanazt az eredményt állítja elő.

jegyzet 166. oldal ábra

November 23.)

- az előző részek tartalmából:
  a problémák a közös erőforrásokkal vannak.
  izolációt akarunk

  a tranzakciók eredménye olyan legyen,
  mintha egyedül birtokolná az összes erőforrást.

  - ehhez zárakat használnak, de persze semmi nem tökéletes, vannak ezzel problémáik
    - deadlock
    - livelock

  vagy soros ütemezést,
  melynél definíció szerint
  nem merülnek fel a közös hozzáférés problémái,
  alapból izoláltak lesznek a tranzakciók.

---

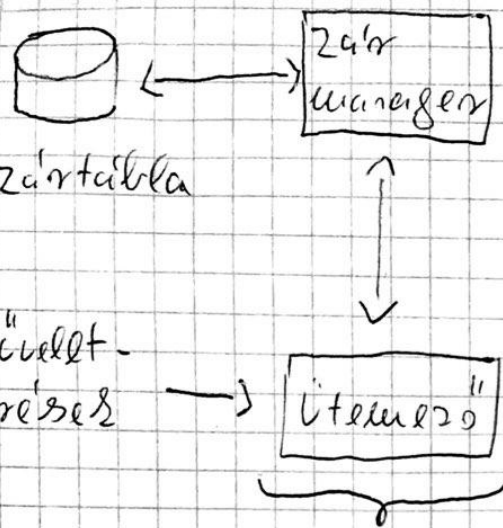
további követelmények:

- ha zárunk egy zárat engedjük is el
- ha zárt zárat pedig zárni ne hasaljon eli várjon

} amennyiben ezek teljesülnek, legális

számítástechnikailag eldönteni valamiről, hogy sorosítható-e, az nagyon nehéz.

még amúgy is, nem ismerjük előre a tranzakciókat, így
- ez a dolog, hogy majd jól kitalálom előre, hogy mit kéne csinálni az teljesen elvetendő



Sorosítási protokollok

valamilyen szabályrendszer, amivel követelményként a tranzakciók sorosíthatóak lesznek.

tranzakció modellek:

szabályos tranzakciósat jellemző tulajdonságok ügyintézésre.

- egyszerű modell
- R/W modell

egyszerű:

egyféle zár van, egy egységen egy időben csak egy zár lehet. ha enyém a zár, írhatok & olvashatok. egy tranzakcióé minden, többieké semmi.

hogy lehet ilyenkor eldönteni, hogy valami sorosítható-e?

Sorosítási gráf (precedencia gráf):

- csúcspontok: tranzakciók
- élek: $T_i \to T_j$ él $\exists$, ha T_i: LOCK A_{in} ... UNLOCK A_{in}
  T_j: ..... LOCK A_{in}

miért akarunk ezzel? sorosíthatóságra kíváncsiak.

és közben más nem történt, T_j egyből akarja a zárat T_i után, T_i után egyből futni akar

gondolgunk kell: ha azt akarjuk, hogy a soros ütemezésünk ekvivalens legyen T_j csak T_i után futhat le

Tétel:

egy S ütemezés sorosítható $\iff$ precedenciagráf DAG

T_1: LOCK A UNLOCK A LOCK B, UNLOCK B

T_2: LOCK A UNLOCK A

T_3: LOCK B UNLOCK B

vegyük észre, hogy a kutyát nem érdekli, hogy a tranzakciók mit csinálnak. csak az érdekel sorosíthatóság szempontjából, hogy mikor zárnak/nyitnak.

$T_1 \to T_2$
T_3
} ez DAG, így ez sorosítható $\to$ $T_3 T_1 T_2$, itt most ez 1 egy $\exists$. vegyük észre, hogy ez topológiai sorrend

hogyan használható ez a gyakorlatban?

↳ a precedenciagráfot dinamikusan, on-the-fly fenntartom és akkor tudok belőle okos lenni.

amikor zárás van, akkor kört keresek a gráfban:

- ha nincs benne, akkor minden kapja, melyet a tranzakció
- ha a zárkérés irányított kört eredményezne, az adott tranzakciót kilövöm.

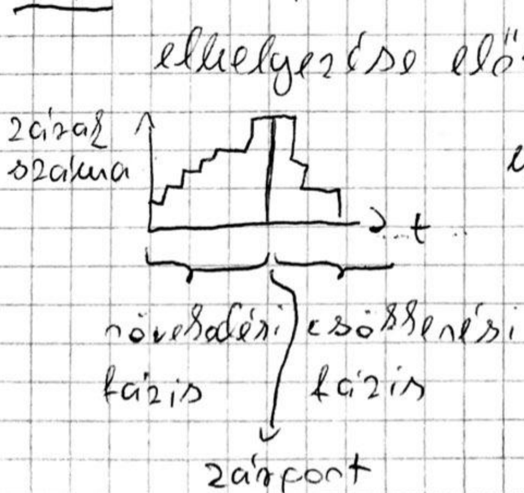
lehet, hogy ez valami fontos volt. kilövés után újra kell ütemezni, aztán egyszer majd lefut.

akár valamit, és meg azt mondom, hogy takarodj. ez nem egészséges. ilyenkor fordulhat elő piszkos írás.

Kétfázisú zárolás (2PL) - protokoll

szabályok, amiket ha rákényszerítünk a tranzakciókra, varázslatos módon sorosíthatóak lesznek

def.: UNLOCK nem lehetséges az összes szükséges zár elhelyezése előtt.



monoton nő, majd monoton csökken

tétel: legális 2PL ütemezések sorosíthatóak.

biz: indirekt tfh. nem sorosítható

⇕

∃ irányított kör a gráfban

$T_{i_1} \to T_{i_2} \to \dots \to T_{i_n} \to T_{i_1}$

ez ellentmond 2PL-nek, hiszen T_{i_1} nem unlockolhatna mielőtt lockolna

↯

✓

T_{i_1}	UX_1	
T_{i_2}	LX_1	
	UX_2	
T_{i_3}	LX_2	
T_{i_n}	UX_n	
T_{i_1}	LX_1	

R/W modell

két féle zárművelet:

- RLOCK A: olvashatók, ill. más is olvashat, de nem írhatók.
- WLOCK A: írhatók/olvashatók, más semmit.

az egyszerűhöz hasonlóan menjünk végig és nézzük meg mi sül ki belőle.

T_i: RLOCK A ... UNLOCK A

T_j: WLOCK A

ha T_j T_i előtt futna, módosíthatná az adatot, mielőtt T_i olvassa.

így $T_i \to T_j$

T_i: WLOCK A WLOCK A
T_j: WLOCK A } $T_i \to T_j$, hiszen T_j már a T_i által exkluzívan módosított adatot akarja olvasni/írni

T_i: W LOCK A CLOCK A
T_j: RLOCK A } $T_i \to T_j$ az előzőhöz hasonlóan

T_i: RLOCK A CLOCK A
T_j: RLOCK A } $T_i \not\to T_j$ ilyenkor mindegy a sorrend

az egyszerű modellben triviális, hogy A a 4 esetben szép él.
így ebben az esetben kicsit az esély irányított gráf kialakulására a gráfban.

tétel:

az ilyen módon definiált gráf aciklikus

aciklikus sorosíthatóság ($\Leftarrow$) gráf DAG

tétel: tranzakció nem engedett el zárat mielőtt meg nem szerezte az összes zárat

tétel: 2PL logikailag ekvivalens sorosíthatósággal

Zártábla:

	RLOCK	WLOCK
RLOCK	I	N
WLOCK	N	N

} megfelelő

Részes

tranzakciós műveletes:

	INC	RL	WL
INC	I	N	N
RL	N	I	N
WL	N	N	N

de sima lekérdező zárműveletet is definiálni, de ith most ez elég lesz.

Hierarchikus adategységek esete:

milyen nyereséget érhetünk el, ha az adatokat nem függetlennek tekintjük?

ebben az esetben egy csomópont alá egy részgráf tartozik

- a.) egyedi zárás
- b.) lockolja a teljes részgráfot

Fa protokoll: egyszerű tranzakciós modell & egyedi zárak.

szabályok:
- első lock akárhova mehet
- további LOCKok csak akkor, ha a tranzakció a szülőre már tett LOCK-ot
- kétszer ugyanazt nem LOCKolhatom

<u>tétel:</u>

a fa protokoll szabályait követő legális ütemezések sorosíthatóak

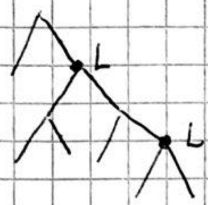
november 30.,

Hierarchikus dolgok folytatása:

Fa protokoll: tudtuk úgy biztosítani a sorosíthatóságot, hogy a protokoll nem lett két fázisú.

Figyelmeztető protokoll:

LOCK A implicit zárat tesz az adott részgráfra



potenciális zárkonfliktus

LOCK A

WARN A : WARN-ot más is tehet, lockot nem

UNLOCK A

szabályok:

- A tranzakció első művelete vagy LOCK gyökér vagy WARN gyökér
- akkor csinálhatunk LOCK-ot vagy WARN-t, hogyha a szülő elemen van WARN a mostani tranzakcióból.
- egy LOCK vagy WARN eltávolítható, ha a gyerekein már nincs semmi a jelenlegi tranzakcióból
- 2 fázisú: az összes zárat el kell helyezni, hogy el tudjuk kezdeni leszedni őket

tétel:

ha egy tranzakció legális a figyelmeztető protokoll kontextusában $\Rightarrow$ a tranzakció sorosítható és zárkonfliktus-mentes

új fejezet:

Tranzakcióhibák kezelése

Lehetséges hibakategóriák:

tranzakció-hibák:
1.) felhasználó kilövi vagy 0-val oszt, egyéb runtime hibák
2.) deadlock miatt az ütemező kilövi
3.) sorosítási feltétel nem teljesítése miatt lövi ki az ütemező

erről külön lesz szó ← 4.) rendszerhiba (operatív tár tartalma sérült)

erről nem lesz szó ← 5.) médiahiba (háttértár tartalma sérült)

Hogyan védekezzünk a tranzakcióhibák ellen?

def.: commit (kész) pont

a tranzakció ∀ zárat megkapott és ∀ műveletet elvégzett, amik tranzakcióhibákat okozhattak volna.

Lavina: egy piszkos adat olvasása miatt számtalan tranzakció eredményét kell kiszedni az adatbázisból

↓

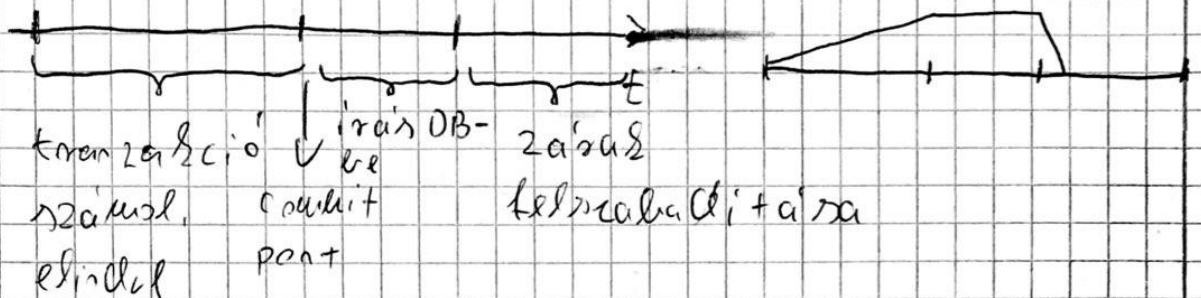
költséges kiküszöbölni

lehetséges megoldás: commit pont előtt valami ideiglenes munkaterületen dolgozik, utána pedig a tényleges adatbázisba is bevezetjük a változtatásokat.

def: szigorú protokoll

nem ismer a adatbázisba a commit pont előtt.

gyakorlati megvalósítás:
szigorú kétfázisú protokoll ⟶ locks száma:



tétel:

a szigorú kétfázisú protokoll szabályait követő logikai tranzakciók sorosítható ütemtervek.

ha ezt csinálják, akkor hosszú tranzakciók sokára kerülhetnek sorra.
→ Hatékonysági kérdés

tranzakciós teljesítmény:

sikeres tranzakciók száma / idő

de mennyit csinálhat?

dolgok, amik nem növelik a tranzakciós teljesítményt:

a.) zárakkal kapcsolatos várakozás
b.) abortált tranzakciók ismétlése
c.) nem kétfázisú protokollokból a lavinaszerű kaszkád költségei.

reák alapján megkülönböztetünk:

agresszív és konzervatív protokollokat.

Sok/kevés energiát fektet az a-val kapcsolatos dolgokra

↓

reménykedik, hogy nem lesz abort, mert akkor spórolt

a-val kapcsolatos dolgokat sokat csinálja

↓

körbebástyázza magát, ami drága, de utána nagy valószínű-séggel sikeres lesz

összefoglaló: ha kevés a közös adat, akkor az agresszív gyorsabb.

példák:

ultrakonzervatív: szigorú közlekedési, 4 zárat előre elkészít egyszerre, ill. csak akkor kapcsol meg egy zárat, ha az előtte szekvenciában áll a várakozó sorban.

ez így sorosítható, lavinamentes, patthelyzetes, ill. elakadásmentes.

Rendszerhibák kezelése

(visszaállítás, recovery)

gyakran nem tárgyalják, akiknek csak valami középszintű fogalmuk van az adatbázis-kezelésről ↑

operatív tár sérült, háttértár ok.

! a rendszerhibák ellen naplózással védekezünk !

tranzakciónapló: adatbázisban végrehajtott műveletek history-ja

miből áll ez?

< tranzakció ID {begin, commit, abort} >

< tranzakció ID, adategység, új érték, régi érték >

Ökölszabály: először ∀ a naplót írjuk, csak utána magát az adatbázist.

ezt tipikusan karbantartjuk az operatív tárban, aztán néha kiírjuk a háttértárra.

Na de most ezzel gyakorlatilag 2x csinálunk mindent. Ez nem hangzik valami gyorsnak.

ha azt csináljuk, hogy az operatív tárba naplózunk, aztán a naplók alapján írunk a háttértárra esetleg jó lehet

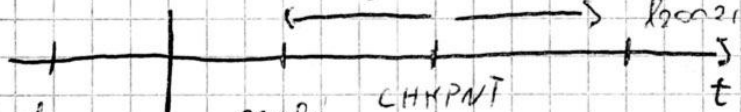
Valójában a naplózás overheadje nem vészes.

Undo: ez a dolog nagyon lassú, mert lehet visszaállítás.

Redo protokoll:

- nem kell undo, így nem kell a régi értéket tárolni a naplóban.
- 2PL + naplózás
  - tranzakció indul: <TRID, begin>
    <TRID, új érték>
    <TRID, commit>
    innentől bármikor tönkremehet a tranzakció rekonstruálható lesz { napló stabil tárra írása
    adatok írása
    zárak felszabadítása

De ez alapján hogyan állítjuk helyre?



→ innentől megint előrefelé megyünk a naplóban, és ha az előbbiek közül megtalálja azokat a tranzakciókat, amikhez tartozik commit. addig megyünk, míg checkpointot találunk!

→ újrajátsszuk azokat a tranzakciókat, amikhez commit tartozik. addig megyünk, míg checkpointot találunk!

december 7.,

legutóbb:

tranzakciók során fellépő hibák kezelése:

- tranzakcióhibák → commit point segít
- rendszerhibák → naplózás segít
- (médiahibák)

legtöbb probléma: piszkos adat & lavinák

protokollok: aggresszív & konzervatív

↓ kevesebb safety-measure

↓ több overhead, de hibatűrőbb

megoldás: szigorú protokollok (pl. szigorú kétfázisú)

naplózás esetén gyakorlatilag mindent duplán tárolunk, persze a napló próbál tömör lenni, az adatbázis maga pedig a teljesítményre törekszik

redo protokoll: szigorú kétfázisú + naplózás

↳ kérdés volt a végén, hogy hogyan lehet megtalálni az utolsó konzisztens állapotot

↓

checkpointok:

- ideiglenesen megtiltjuk az új tranzakciókat
- ∀ tranzakció ∀ műveletet végezze el
- a memóriából mindent írjunk ki a diszkre

végül írjuk ki, hogy az adatbázis elérte egy checkpoint-ba. (először naplózzuk a tényét, majd a naplót írjuk a háttértárra)

(bankipari adatbázis egy hetes visszaállítás - egy hét semmit tevés)

Időbélyeges tranzakciókezelés

cél, még mindig: sorosítható legyenek a tranzakciók

időbélyeg: A tranzakcióhoz hozzárendelt számérték, mely arányos a tranzakció kezdőidejével

ezzel szeretnénk, hogy az ütemező – lehetőleg valami egyszerű módon – soros ekvivalenst csináltassa

ez nem lesz más, mint egy sima növekvő sorrend: $t(T_1) < t(T_2) < \ldots < t(T_n)$

időbélyeg (felé) tranzakció

(ez annyira nem is meglepő, az előbbi commit pontossal csinált hasonlót)

R/W modell:

R(A): olvasási időbélyeg: annak a tranzakciónak időbélyege, mely A-t legutóbb olvasta

W(A): írási időbélyeg: ugyanaz, mint az előző, csak írásra

ebből próbál logikázni az ütemező (varázstáblázat):

	t olvasni akar	t írni akar
$t(T) < R(A)$ $t(T) < w(A)$	abort T	abort T
$t(T) < R(A)$ $t(T) > w(A)$	T olvashat $R(A)$ változatlan	abort T
$t(T) > R(A)$ $t(T) < w(A)$	abort T	abort T
$t(T) > R(A)$ $t(T) > w(A)$	T olvashat $R(A) := t(T)$	T írhat $w(A) := t(T)$

Időbélyegzés szigorú protokollok:

a fenti táblázat alapján lesz csomó piszkos adat & lavina.

ennek már látszik, hogy a szigorú protokollok szoktak megoldást jelenteni.

jobb ötletet nem tudtak kitalálni, mint hogy zárakat vezetünk be.

Időbélyegzés verziók mellett (MVCC):

írásnál nem felülírjuk az adatokat, hanem új verziót hozunk létre.

ezzel a fenti táblázat olvasásos oszlopát abortmentessé lehet tenni.

Elosztott adatbázis kezelés

autonóm adatbázis-szerverekből álló rendszerek, ahol a csomópontok kommunikálni képesek

SPOF: single point of failure

↓

nincs mit tenni, duplázunk, többszörözünk mindent!
adatosztási kérdések.

egy elosztott adatbázis a külvilág felé egyetlen egységes adatbázisként viselkedik.
(logikailag egyetlen adatbázis)

→ magát belül eldönti, hogy mit hogyan és hol csinál.

logikai adatbázis: amit a felszínen látnak
fizikai adatbázis: ami valójában a háttérben van

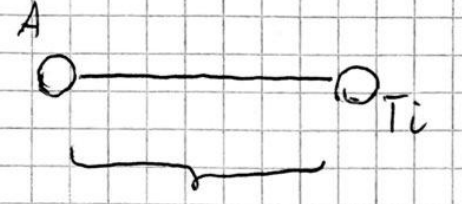
logikai adategység: azok, amikkel a felhasználók dolgoznak
fizikai adategység: replikátumok összessége a csomópontokon

nincs egy teljesítménygörbéje
határa: jó, ha közelről szedem az adatot

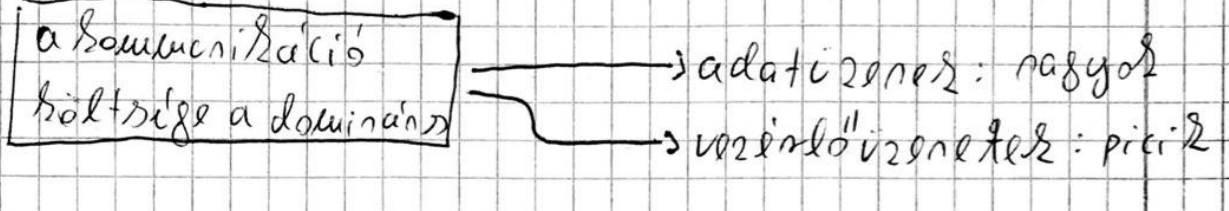
logikai szint: logikai adategységekkel – így működnek, mint eddig
fizikai szint: ami a háttérben megy, abból

logikai (globális) tranzakciók: globális adategységekre
fizikai (lokális) tranzakciók: erre fordítódnak le a globálisak,
ő pl ilyenre.

elosztott zárolás:

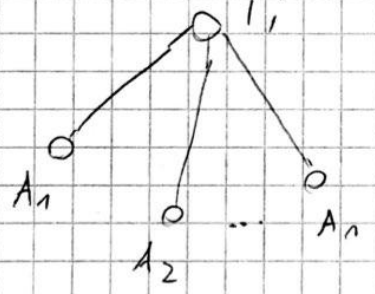


T_i tranzakció szeretné elérni
A adategységet, ami egy másik
csomóponton van.



így erre próbálunk
optimalizálni

előfordulhat, hogy A sok helyen
van:



WALL protokoll: „write locks all"

WLOCK A: globális adategységen globális zár
↓ ebből lehet csinálni

WLOCK Ai: lokális adategységen lokális zár

→ érvényes, ha ∀ WLOCK Ai példányon sikerül

RLOCK A —> RLOCK Ai
↳ sikeres, ha legalább egy példányon sikeres.

globális kompatibilitás:

mátrix:

	R	W
R	I	N
W	N	N

} ez ugyanaz, mint lokális esetben.

de ez mennyire kielégítő?

- WLOCK
  - vezérlőüzenet oda
  - vezérlőüzenet vissza
  - adatüzenet

} ez így úgy de 2air ellenőrzésével

december 9.,

Kulcsgondolat: kellő% adatot több node-on is tárolunk

mindenből van globális / logikai, ill. lokális / fizikai

talán a legérdekesebb a zárszolgáltatás:

WALL: R/W módszer

WLOCK A → ∀ WLOCK Ai

RLOCK A → ∃ RLOCK Ai

↓ globális adat

↓ lokális adat

vannak más megoldások is:

Többségi zárolás:

Zárolás sikeres, ha a fizikai zárak többségét sikerült zárni (R/W)

Kompatibilitási mátrix a WALL-ival megegyezik

WLOCK A	$\frac{n+1}{2}$	$\frac{n+1}{2}$	n adatizzel
RLOCK A	$\frac{n+1}{2}$	$\frac{n+1}{2}$	1 adat

ez egy végső, a másik a WALL, a köti össze:

% az n-ből protokoll:

$\frac{n}{2} < \% < n$ WLOCK %

RLOCK n+1-%

és még tovább is megy a logat:

Elsődleges példányok:

egy kitüntetett csomópont dönti el, hogy egy adott adategységgel a zárkérések validak-e.

centrális csúcs módszere: legprimitívebb típus, ∀ adategységre egy kitüntetett node van (mindnek ugyanaz)

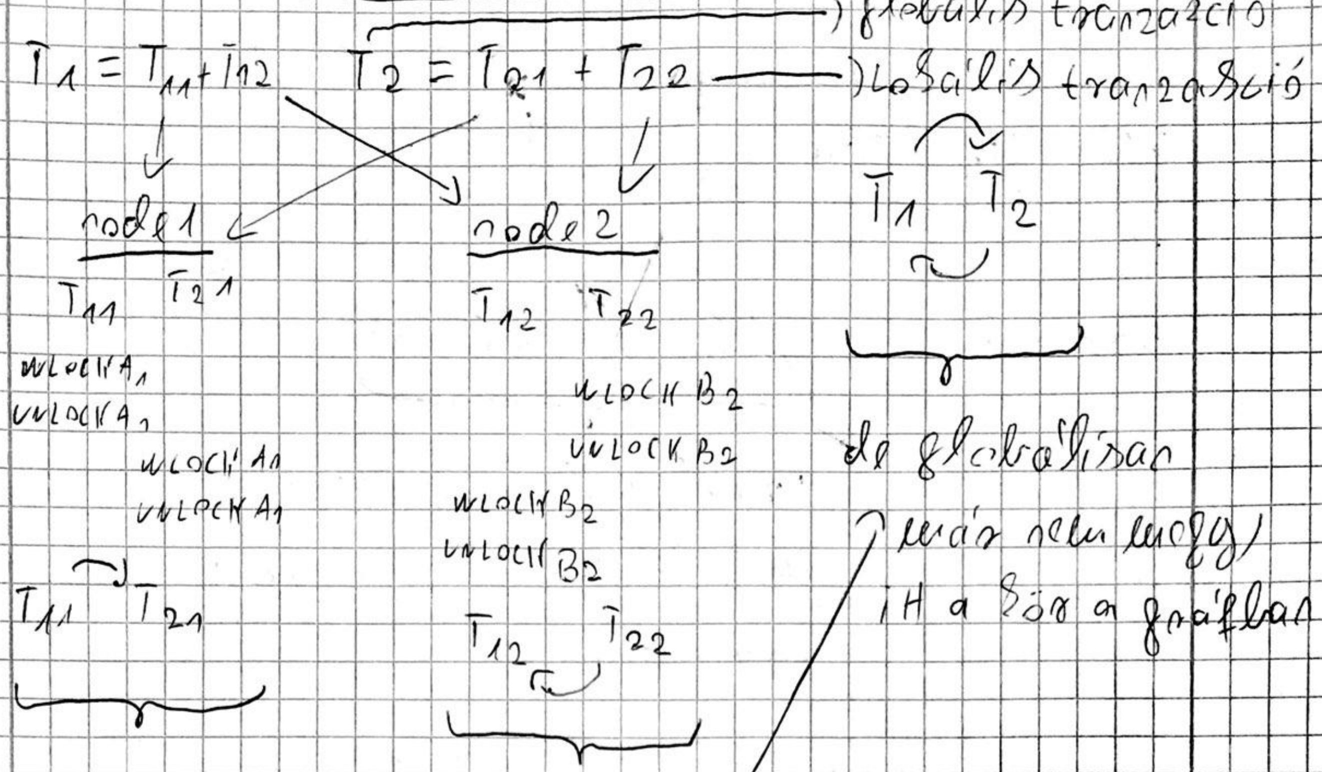
Tokenes protokoll:
elsődleges példányok adaptív verziója:
az elsődleges példány nem fix, hanem változik.
a változtatást tokenekkel oldjuk meg.

WT(A): csak akkor lehet 1 WT, ha ∄ RT
RT(A): ha ∄ WT, akkor akárhány RT lehet

ha egy node akarja a tokent, akkor egy broadcastban megkérdezi, hogy kinél van és hogy le tud-e mondani róla, ha kell.

ha le tud mondani róla, akinél van, akkor a következő broadcast-ban bejelenti, hogy akkor most az övé a token,

Elosztott tranzakciókezelés



lokálisan a tranzakciók tök szépen sorosíthatóak lennének

hát most kiderült, hogy ami alapból tök jó volt elosztott rendszereknél már nem működnek.

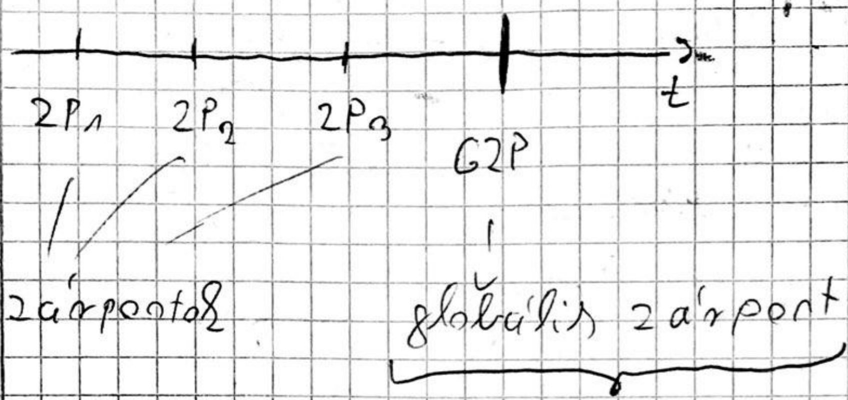
mit lehet tenni?

alapgondolat: globálisan erőltessük a kétfázisúságot, a csomópontok üzenetekkel megbeszélik, hogy [illegible].

globális zárpont – globális kétfázisúság

globális kétfázisúságot követő legális globális tranzakciók sorosíthatóak.

így néz ez ki valahogy:



erről valahogy üzenetekkel megállapodnak a lokális tranzakciók.

na de mi a helyzet a lavinákkal?

- közönök így vannak, ráadásul elosztott rendszerekben még sokkal rosszabb a helyzet (nem tudjuk milyen node-okon volt piszkos írás)

megint a szigorú protokollok jelentik a megoldást: a globális zárponthoz hasonlóan lesz globális commit point, amit hasonlóan a lokális tranzakciók bonyolnak le egymás között

distributed agreement protocol-nak nevezik ezt a dolgot, és rendkívül macerás megcsinálni

ezért

→ gyakorlatban úgy megy, hogy valakit kitüntetnek főnöknek és mindenki vele beszélget.

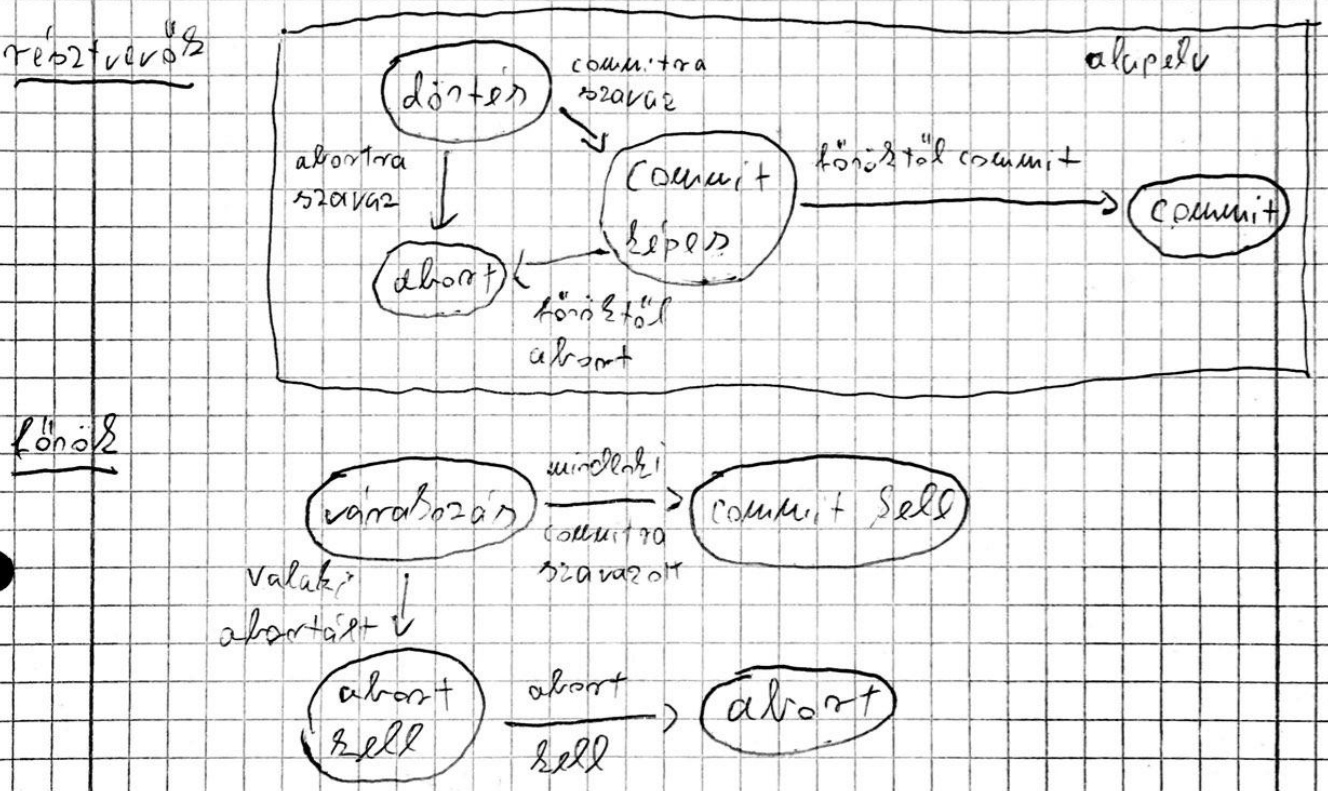
másik dolog, ami csökkenti a komplexitást, hogy azt mondjuk, hogy a commit point lesz a záspont is (mivel a co. p. úgyis mindig később van), így csak egyszer kell megegyezni ez a csomópontokat

hogy kell ezt implementálni?

↓

2PC és 3PC protokollok

cél: résztranzakciók a commit point-ban meg tudjanak egyezni



megjegyzés: főnök bárki lehet, sőt lehet résztvevő is

alkérdés: Batman

mi ezzel a baj? a valóságban ha egy résztvevőnél valami hiba van (elmegy a net, etc.); befagy (commit közben állapotban) és borul az egész.

ezért a valóságban ezt ki szokták egészíteni, 2PC-vel mondjuk (timeout-ok, egyebek).

ezeket nem rajzolom le, mert a könyvben úgyis sokkal szebben benne vannak.

→ alapvetően arról van szó, hogy van egy kezdőállapot a döntés előtt, ahol várnak a főnök utasításaira, hogy kezdjenek szavazni.

2PC

ha valaki commit képes állapotban van túl sokáig, akkor helyreállítási állapotba lép és segítséget kér a többi node-tól. ha ezt valaki hallgatja, aki már továbbjutott commit képes állapotból, akkor ő segít, hogy végül commit, vagy abort lett a végleges (és a segítségre szoruló node egy elakadt állapotban vár a segítő üzenetre)

persze ez nem tökéletes, meg semmi sem ...

mi van, ha egy node-nak senki sem segít, ha kéri?

3PC: csak akkor commitoljunk, hogyha mindenki tudja, hogy mindenki commitra szavazott (atomiság ne sérüljön) (nem csak azt, hogy mindenki commitra szavazott)

ezzel praktikusan bejön +1 állapot, ezért hívjuk 3 fázisúnak.