- *Tervek felsorolására szolgáló stratégiák:* A fizikai tervek tartományában a legjobb megtalálására irányuló keresés szokásos megközelítései között szerepelnek: a dinamikus programozás (táblázatban összegyűjti az adott logikai terv egyes részkifejezéseihez tartozó legjobb terveket), a Selinger-féle dinamikus programozás (a nem rendezett eredmény mellett rendezett eredményt adó terveket is megtart a táblázatban), mohó megközelítések (lokális értelemben optimális döntéseket hoz a fizikai tervvel kapcsolatos addigi választások alapján), valamint az elágazás-és-korlát (csak azokat a terveket sorolja fel, amelyekről nem lehet azonnal tudni, hogy az addig talált legjobb tervnél rosszabb).
- *Bal-mély összekapcsolási fák:* Amikor több reláció összekapcsolásakor kiválasztunk egy csoportosítást és sorrendet, bevett gyakorlat, hogy a keresést a bal-mély fákra korlátozzuk. Ezek bináris fák egyetlen gerinccel, ami a bal oldalon fut lefelé, és csak olyan levelekkel, amelyek jobb gyerekek. Az összekapcsolási kifejezéseknek ez az alakja várhatóan hatékony tervet eredményez, és a megvizsgálandó fizikai tervek számát is korlátozza.
- *Fizikai terv a kiválasztáshoz:* Ha lehetséges, akkor egy kiválasztást ketté kell vágni a kiválasztás alapjául szolgáló reláció egy alkalmas indexének olvasására (ahol általában egy olyan feltételt használunk, amelyben az indexelt attribútumot egy konstanssal tesszük egyenlővé), valamint egy azt követő szűrési műveletre. A szűrés megvizsgálja az indexolvasás által visszaadott sorokat, és csak azokat engedi tovább, amelyek a kiválasztási feltétel többi részét (ami nem az indexolvasás alapja) is kielégítik.
- *Futószalag-technika és materializáció:* Ideális esetben minden egyes fizikai operátor eredményét felhasználja egy másik operátor, és az eredmény átadása a központi memóriában történik („futószalag-technika"), esetleg egy iterátor használatával a kettő közötti adatáramlás vezérlésére. Néha azonban annak van előnye, hogy egy operátor eredményét eltároljuk („materializáljuk"), helyet takarítva meg ezáltal a központi memóriában más operátorok számára. A fizikai lekérdezéstervet generálónak tehát a közbülső eredmények futószalagosításával és materializációjával is számolni kell.

7.9. Irodalomjegyzék

A 6. fejezet irodalomjegyzékében említett áttekintő tanulmányok a lekérdezésfordítás szempontjából lényeges anyagokat is tartalmaznak. Javasoljuk még az [1] áttekintést, amely kereskedelmi forgalomban lévő rendszerek lekérdezésoptimalizálóit vizsgálja.

Korai tanulmányok a lekérdezésoptimalizálásról a [4], [5] és [3]. A [7] – szintén korai tanulmány – egyesíti a kiválasztások – a fában lefelé történő – tologatásának elvét az összekapcsolási sorrend megválasztására szolgáló mohó algoritmussal. A [2] cikk a „Selinger-féle optimalizálás" forrása, valamint leírja a System R-optimalizálót is, amely a maga idejében a legigényesebb kísérlet volt lekérdezésoptimalizálásra.

Az SQL2 teljes nyelvtanát a [6]-ban találhatja meg az olvasó.

1. G. Graefe (ed.), *Data Engineering* **16**:4 (1993), special issue on query processing in commercial database management systems, IEEE.
2. P. Griffiths-Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price, „Access path selection in a relational database system", *Proc. ACM SIGMOD Intl.Conf. on Management of Data* (1979), pp. 23–34.
3. P. A. V. Hall, „Optimization of a single relational expression in a relational database system", *IBM J. Research and Development* **20**:3 (1976), pp. 244–257.
4. F. P. Palermo, „A database search problem", in: J. T. Tou (ed.) *Information Systems COINS IV*, Plenum, New York, 1974.
5. J. M. Smith and P. Y. Chang, „Optimizing the performance of a relational algebra database interface", *Comm. ACM* **18**:10 (1975), pp. 568–579.
6. `ftp://jerry.ece.umassd.edu/isowg3/dbl/BASEdocs/public/sql-92.bnf`
7. E. Wong and K. Youssefi, „Decomposition – a strategy for query processing", *ACM Trans. on Database Systems* **1**:3 (1976), pp. 223–241.

8. fejezet

A rendszerhibák kezelése

E fejezetben figyelmünket az adatbázis-kezelő rendszereknek az adatok meghibásodásával foglalkozó részére fordítjuk. Két fő témakör, melyeket tanulmányozni kell:

1. Az adatokat meg kell védeni a rendszerhibáktól. E fejezet azon technikákkal foglalkozik, melyek célja a *helyreállíthatóság*, azaz az adatok integritásának (épségének és összefüggéseinek) megőrzése a rendszerhibák előfordulásakor.
2. Az adatoknak nem szabad sérülniük több hibamentes lekérdezés vagy adatbázis-módosítások egyszerre való végrehajtásakor sem. Ezzel a 9. és 10. fejezet foglalkozik.

A helyreállíthatóság biztosítására az elsődleges technika a *naplózás* (log), mely valamilyen biztonságos módszerrel rögzíti az adatbázisban végrehajtott módosítások történetét. Három különböző – a „semmisségi" (undo), a „helyrehozó" (redo) és a „semmisségi/helyrehozó" (undo/redo) – naplózási módszert tanulmányozunk. Foglalkozunk továbbá a *helyreállítással*, azzal az eljárással, amikor a naplót felhasználva rekonstruáljuk az adatbázis hiba bekövetkezése előtti állapotát. A naplózás és a helyreállítás egy nagyon fontos vonatkozása az olyan helyzetek elkerülése, amikor a naplót a távoli múltra vonatkozóan kellene tanulmányozni. Így meg fogunk ismerni egy fontos technikát, az „ellenőrzőpont" használatát, mellyel korlátozzuk a helyreállítás során elemzendő napló(rész) hosszát.

Végül az „archiválással" foglalkozunk, mellyel biztosíthatjuk, hogy az adatbázis nemcsak az ideiglenes rendszerhibákat, de a teljes adatbázis elveszítését is „túlélje". Ezzel a módszerrel az adatbázis legfrissebb másolatára (az archivált adatbázisra) és a naplózott információkra támaszkodva rekonstruáljuk az adatbázis valamely korábbi állapotát.

8.1. A helyreállítható beavatkozások példái és modelljei

A vizsgálódásunkat azzal kezdjük, hogy áttekintjük a hibafajtákat, és azt, hogy az adatbázis-kezelő rendszerek mit tud(hat)nak tenni velük. Először a „rendszerhibákat" vagy „katasztrófákat" vesszük szemügyre. Ezen hibafajták elhárítására tervezték a

naplózási és helyreállítási módszereket. A 8.1.4. részben bemutatjuk a pufferkezelés modelljét is, mely a rendszerhibákból való helyreállítás minden meggondolásának alapja. Ugyanez a modell szükséges a következő fejezetben is, amikor az adatbázisok több tranzakcióval történő egyidejű (konkurens) elérését vizsgáljuk.

8.1.1. A hibák fajtái

Az adatbázis lekérdezése vagy módosítása során számos dolog hibát okozhat. A problémák felsorolása a billentyűzeten történt adatbeviteli hibáktól kezdve az adatbázist tároló lemez elhelyezésére szolgáló helyiségben történő robbanásig folytatható. A következő pontokban a legfontosabb hibaokokat csoportosítjuk, valamint összefoglaljuk, hogy az adatbázis-kezelő rendszerek mit tehetnek ezek előfordulásakor.

Hibás adatbevitel

Az adatok tartalmi hibáit sokszor nem tudjuk észrevenni. Ha például a hivatalnok elüt egy számot az ön telefonszámán, akkor az adat még úgy néz ki, mint egy telefonszám, mely az öné is *lehetne*. Másrészről, ha a hivatalnok kifelejt egy számot az ön telefonszámából, akkor az adat nyilvánvalóan hibás, mert nem felel meg a telefonszám formai követelményeinek[1].

A modern adatbázis-kezelő rendszerek számos szoftverelemet biztosítanak a fentihez hasonló adatbeviteli hibák felismerésére. Például az SQL2- és SQL3-szabványokban, s az SQL összes közismert megvalósításaiban az adatbázis tervezője megadhat előírásokat, mint például kulcsra, idegen kulcsra vagy értékekre vonatkozó megszorításokat (hogy például a telefonszámnak 10 számjegyből kell állnia). A triggerek azok a programok, melyek bizonyos típusú módosítások (például az R relációba való sor beszúrása) esetén hajtódnak végre, annak ellenőrzésére, hogy a frissen bevitt adatok megfelelnek-e az adatbázis-tervező által megszabott előírásoknak.

Készülékhibák

A lemezegységek olyan helyi hibái, melyek egy vagy néhány bit megváltozását okozzák, a lemez szektoraihoz rendelt paritás-ellenőrzéssel megbízhatóan felismerhetők, amint erről a 2.2.5. részben már szó volt. A lemezegységek jelentős sérülése, elsősorban a fejek (író-olvasó fejek) katasztrófái, az egész lemez olvashatatlanná válását okozhatják. A katasztrofális hibákat általában az alábbi megoldások segítségével kezelik:

1. A 2.6. részben már megismert RAID-módszerek valamelyikének használatával az elveszett lemez tartalma visszatölthető.

[1] Az egyszerűség kedvéért tegyük most fel, hogy a telefonszámok egyetlen formai előírásnak kell hogy megfeleljenek.

2. *Archiválás* (mentés) használatával az adatbázisról másolatot készítünk valamely eszközre pl. szalagra vagy optikai lemezre. A mentést rendszeresen kell végezni vagy teljes vagy növekményes (csak a változások archiválása) mentést használva. A mentett anyagot az adatbázistól biztonságos távolságban kell tárolnunk. Az archiválást a 8.5. részben tárgyaljuk.
3. Az archiválás helyett az adatbázisról fenntarthatunk elosztott, on-line másolatokat is. Ezen másolatok konzisztenciáját (összhangját az eredetivel) biztosító mechanizmusokat a 10.6. részben tanulmányozzuk.

Katasztrofális hibák

Ebbe a kategóriába soroljuk azokat a helyzeteket, amikor az adatbázist tartalmazó eszköz teljesen tönkremegy. A példák közé tartoznak a robbanás, a tűz, a vandalizmus és a vírusok is. A RAID ekkor nem segít, mert az összes adatlemezek és a paritás-ellenőrző lemezeik is egyszerre használhatatlanná válnak. Ugyanakkor más biztonsági megoldások – mint az archiválás és a redundáns osztott másolatok – használata az ilyen típusú katasztrófák elleni védekezésre is alkalmas.

Rendszerhibák

A lekérdező- és az adatbázis-módosító eljárásokat *tranzakcióknak* nevezzük. A tranzakció, hasonlóan más programokhoz, lépések sorozatát hajtja végre, gyakori esetben ezen lépések közül néhány az adatbázist fogja módosítani. 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, melyekre később még szükség lehet.

A *rendszerhibák* azok a problémák, melyek a tranzakció állapotának elvesztését okozzák. Tipikus rendszerhibák az áram kimaradásból és a szoftverhibákból eredők. Azért, hogy átlássuk, miért okozza az állapot elvesztését az olyan probléma, mint az áramkimaradás, vegyük figyelembe, hogy – más programokhoz hasonlóan – a tranzakció lépései is elsődlegesen a memóriában fordulnak elő. Eltérően a lemeztől, a memória tartalma „illékony", amint erről a 2.1.6. részben szó volt. Ez azt jelenti, hogy az áramkimaradás a memória tartalmának elvesztését okozza, amíg a lemezeken tárolt adatok sértetlenek maradnak. Hasonlóan, egy szoftverhiba a memória egy részének felülírását okozhatja, előfordulhat, hogy éppen a programunk állapotában szereplő értékeket is felülírja.

Ha a memória tartalma elveszett, a tranzakció állapota is elveszett, innentől 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, ha a tranzakció az adatbázisban valamely értéket 1-gyel kell hogy növeljen, nem tudhatjuk, hogy az ismétléskor szükséges-e megismételni az 1-gyel való növelést, vagy sem. A rendszerhibákból származó problémák legfontosabb ellenszere: minden adatbázis-változtatás naplózása egy elkülönült, nem illékony naplófájlban, lehetővé téve ezzel a visszaállítást, ha az szükséges. Az a mechanizmus, ahogy a naplózás módszerét hibavédetté tesszük, meglepően bonyolult, amint azt a 8.2. rész elején látni fogjuk.

Tranzakciók és triggerek

A tranzakció kibővíthető triggerek használatával vagy más, az adatbázissémában előforduló aktív elemekkel. Ha a tranzakció módosítási tevékenységet is tartalmaz, és ez egy vagy több triggert is aktivizál, akkor a triggerakciók is a tranzakció részei lesznek. Egyes rendszerekben a triggerek kiválthatják újabb triggerek működését. Ha így van, akkor az összes kiváltott akciók a triggersorozatot aktivizáló tranzakció részének számítanak.

8.1.2. Részletesebben a tranzakciókról

Mielőtt folytatnánk az adatbázisok hibából adódó helyreállítási lehetőségeinek tanulmányozását, részletesebben tisztáznunk kell a tranzakciókra vonatkozó alapelgondolásokat. A tranzakció az adatbázis-műveletek végrehajtási egysége. Például, ha egy ad hoc utasítást adunk az SQL-rendszernek, akkor minden lekérdezés vagy adatbázis-módosító utasítás egy tranzakció. Amennyiben beágyazott SQL-interface-t használva a programozó készíti el a tranzakciót, akkor *egy* tranzakcióban – a használt programozási nyelv utasításaiként – több SQL-lekérdezés és -módosítás is szerepelhet. Tipikus beágyazott SQL- rendszerben a tranzakció adatbázis-akciók végrehajtásával kezdődik, és egy `COMMIT` vagy `ROLLBACK` („abort") paranccsal fejeződik be.

Amint a 8.1.3. részben látni fogjuk, a tranzakciót atomosan kell végrehajtani, ami azt jelenti, hogy mindent-vagy-semmit módon és időben egy egységként kell működnie. A tranzakciók korrekt végrehajtásának biztosítása a *tranzakciókezelő* feladata. A tranzakciókezelő részrendszer egy sor feladatot lát el, közöttük:

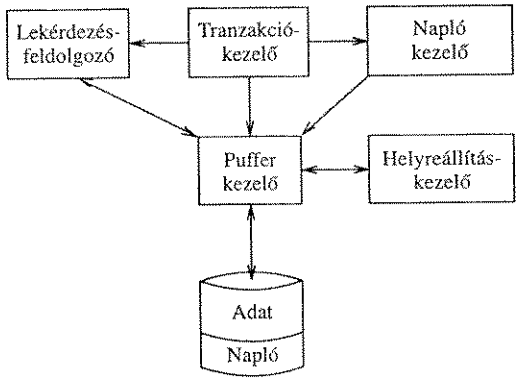
1. Jelzéseket ad át a naplókezelőnek (alább részletezzük) úgy, hogy a szükséges információ „naplóbejegyzés" formában a naplóban tárolható legyen.
2. Biztosítja, hogy a párhuzamosan végrehajtott tranzakciók ne zavarhassák egymás működését („ütemezés"; lásd a 9.1. részben).

A tranzakciókezelőt és kapcsolatait a 8.1. ábra mutatja. A tranzakciókezelő a tranzakció tevékenységeiről üzeneteket küld a naplókezelőnek, üzen a pufferkezelőnek arra vonatkozóan, hogy a pufferek tartalmát szabad-e vagy kell-e lemezre másolni, és üzen a lekérdezésfeldolgozónak arról, hogy a tranzakcióban előírt lekérdezéseket vagy más adatbázis-műveleteket kell végrehajtania.

A naplókezelő a naplót tartja karban. Együtt kell működnie a pufferkezelővel, hiszen a naplózandó információ elsődlegesen a memóriapufferekben jelenik meg, és bi-

zonyos időnként a pufferek tartalmát lemezre kell másolni. A napló, adat lévén, a lemezen területet foglal el, ahogy ezt a 8.1. ábrán is jelezzük.

Végezetül a 8.1. ábrán látható helyreállítás-kezelő szerepéről: ha baj van, akkor aktivizálódik. Megvizsgálja a naplót, és ha szükséges, a naplót használva helyreállítja az adatokat. Mint mindig, a lemez elérése a pufferkezelőn át történik.



8.1. ábra. *A naplókezelő és a tranzakciókezelő*

8.1.3. A tranzakciók korrekt végrehajtása

Mielőtt a rendszerhibák javításával foglalkoznánk, meg kell értenünk, hogy mit is jelent a tranzakciók „korrekt" végrehajtása. Először is tegyük fel, hogy az adatbázis „elemekből" áll. Azt, hogy mi az „elem", nem fogjuk precízen meghatározni, de úgy tekintjük, hogy az adatbázis elemeinek van valamilyen értékük, és ezt az értéket tranzakciókkal lehet elérni (kiolvasni) vagy módosítani. Más-más adatbázisrendszerek más-más megnevezést használnak az elemekre, de többnyire az alábbiak közül választanak:

1. Relációk vagy az objektumorientált megfelelője: az osztály kiterjedése.
2. Lemezblokkok vagy -lapok.
3. A relációk sorai vagy az objektumorientált megfelelői: objektumok.

A példáinkban az adatbázis elemeit tekinthetjük soroknak vagy sok példában egyszerűen egész számoknak. Ugyanakkor a gyakorlatban számos jó ok van arra is, hogy a 2. választást – lemezblokkokat vagy -lapokat – tekintsünk az adatbázis elemeinek. Ekkor a puffer tartalma egyszerű elemekké válik, s ezzel elkerülhető a naplózás és a tranzakciók néhány súlyosabb problémája, amelyeket majd periodikusan kifejtünk a különféle technikák tanulása során. Nem használva a lemezblokkméreténél nagyobb adatbáziselemeket, megelőzzük az olyan helyzeteket is, amikor a hiba fellépésekor az adatbázis valamely elemének egy része, de nem az egész van csak a nem illékony memóriában.

Hihető-e a korrektség alapelve?

Legyen adott egy olyan adatbázis-tranzakció, mely lehetővé teszi ad hoc módosító parancs kiadását a terminálról (a felhasználó készülékéről) esetleg olyan valaki számára, aki nem ismeri az adatbázis tervezője által elgondolt összefüggéseket. Nyilvánvaló-e ekkor, hogy az adatbázist konzisztens állapotából az összes lehetséges tranzakciók ismét konzisztens állapotba viszik? Az explicit megszorítások betartását az adatbázisrendszer kényszeríteni tudja azzal, hogy az olyan tranzakciókat, melyek megsértik az előírt összefüggéseket, a rendszer visszautasítja, s így az adatbázisban semmilyen változtatás nem történik. Az implicit megszorítások azok, melyeket nem tudunk egzakt módon jellemezni. Az egyetlen lehetőségünk a korrektség alapelvének érvényesítésére annak feltételezése, hogy ha valaki jogot kap az adatbázisban módosítani, akkor neki legyen joga annak eldöntésére is, hogy melyek az elvárt implicit megszorítások.

Az adatbázis összes elemeinek pillanatnyi értékét az adatbázis-*állapotának* nevezzük[2]. Bizonyos adatbázis-állapotokat konzisztensnek tekintünk, míg a többi adatbázis-állapotot inkonzisztensnek minősítjük. A konzisztens állapotok kielégítik az adatbázissémára vonatkozó összes megszorításokat, mint például a kulcsokra és az elemek értékeire vonatkozó előírásokat. Túl ezen, a konzisztens állapotnak ki kell elégítenie az implicit megszorításokat is, melyek az adatbázis tervezőjének elgondolásaiban szerepelnek. Az implicit megszorításokat részben az adatbázisséma részének tekintett triggerek alkalmazásával lehet biztosítani, de kikényszeríthetjük az adatbázisra vonatkozó rendtartási előírásokkal is. Használhatunk a felhasználónak szóló figyelmeztető üzeneteket is, amikor módosítja az adatbázist.

8.1. példa: Tegyük fel, hogy adatbázisunk a következő relációkból áll

```
Szerepel(filmCím, év, színészNév)
FilmSzínész(név, cím, nem, születési_idő)
```

Előírhatjuk a következő idegen kulcsra vonatkozó megszorítást: minden `színészNév` értéknek meg kell jelennie a `FilmSzínész név` értékeként; vagy a következő értékelőírást tehetjük: a `nem` értéke csak `'F'` vagy `'N'` lehet. Az adatbázis állapota akkor és csakis akkor konzisztens, ha az összes előírásokat kielégítik a két reláció pillanatnyi értékei. □

A tranzakciókra vonatkozó alapvető feltételezésünk:

- *A korrektség alapelve:* Ha a tranzakciót minden más tranzakciótól függetlenül („egyedül") és rendszerhiba nélkül végrehajtjuk, és ha indulásakor az adatbázis

[2] Ne keverjük össze az adatbázis-állapotot a tranzakció állapotával; utóbbiak a tranzakció lokális változóinak értékei, s ezek nem adatbáziselemek.

konzisztens állapotban volt, akkor a tranzakció befejezése után ismét konzisztens állapotban lesz.

A korrektség alapelvéhez kapcsolódik a naplózás technikája, melyet e fejezetben tárgyalunk, és a konkurencia vezérlési mechanizmus, melyet a 9. fejezetben tárgyalunk. Ebből két dolog következik:

1. A tranzakció *atomi*, azaz teljes egészében vagy végrehajtandó, vagy egyáltalán nem. Ha a tranzakciónak csak egy részét sikerült végrehajtani, akkor nagy esélyünk van arra, hogy az általa előállított adatbázis-állapot nem lesz konzisztens állapot.
2. A párhuzamosan végrehajtott tranzakciók nagy eséllyel inkonzisztens állapothoz vezethetnek, hacsak meg nem tesszük a 9. fejezetben tárgyalt megelőző lépéseket.

8.1.4. A tranzakciók alaptevékenységei

Vizsgáljuk meg részletesen a tranzakció és adatbázis kölcsönhatását. A kölcsönhatásoknak három fontos színhelye van:

1. Az adatbázis elemeit tartalmazó lemezblokkok területe.
2. A pufferkezelő által használt virtuális vagy valós memóriaterület.
3. A tranzakció memóriaterülete.

Ahhoz, hogy a tranzakció egy adatbáziselemet beolvashasson, azt előbb memória-puffer(ek)be kell behozni, ha még nincs ott. Ezt követően tudja a puffer(ek) tartalmát a tranzakció saját memóriaterületére beolvasni. Az adatbáziselem új értékének kiírása fordított sorrendben történik. Az új értéket a tranzakció alakítja ki saját memóriaterületén, majd ez az új érték másolódik át a megfelelő puffer(ek)be.

A pufferek tartalmát vagy azonnal lemezre lehet írni, vagy nem; az erre vonatkozó döntés általában a pufferkezelő joga. Amint már korábban láthattuk, a naplózó rendszer használatának egyik legfőbb lépése a rendszerhibákból való helyreállíthatóság biztosítása érdekében a pufferkezelő ösztönzése a pufferbeli blokkok megfelelő időpontban történő lemezre írására. Ugyanakkor a lemez I/O-műveletek számának csökkentésére az adatbázisrendszerek megengedik/megengedhetik a módosításoknak csak az illékony memóriában történő végrehajtását, legalábbis bizonyos ideig és arra megfelelő feltételek teljesülése esetén.

A naplózási algoritmusoknak és más tranzakciókezelő algoritmusoknak részletes tanulmányozása során megfelelő jelölésekre lesz szükségünk, melyekkel a különböző területek közötti adatmozgást tudjuk leírni. A következő alapműveleteket fogjuk használni:

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 az 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.

Pufferek szerepe a lekérdezések feldolgozásában és a tranzakciókban

Ha visszagondolunk a lekérdezésfeldolgozással foglalkozó fejezet pufferhasználati elemzésére, akkor a jelenlegi nézőpontunkban változás tapasztalható. A 6. és 7. fejezetekben a puffereket elsősorban a lekérdezés kiértékelése közben szükséges ideiglenes táblák elhelyezésére használtuk. Ez a pufferek egyik fontos alkalmazása ekkor, mivel senki nem igényli az ideiglenes értékek megőrzését, így ezen pufferek tartalmát általában nem kell naplózni. Másrészről, azon pufferek tartalmát, melyek az adatbázisból beolvasott elemeket tartalmazzák, meg *kell* őrizni, különösen ha a tranzakció módosítja őket.

A fenti műveleteknek addig van értelmük, amíg az adatbáziselemek elférnek egy-egy lemezblokkban és így egy-egy pufferben is. Ezt az esetet úgy is tekinthetjük, hogy az adatbáziselemek *pontosan* a blokkok. Adatbáziselem lehet az adatbázis egy-egy sora is. Mindaddig így tekinthetjük, amíg a relációséma nem engedi meg nagyobb („hosszabb") sorok előfordulását, mint amennyi hely egy blokkban rendelkezésre áll. Ha az adatbáziselem több blokkot foglal el, akkor úgy is tekinthetjük, hogy az adatbáziselem minden blokkméretű része önmagában egy adatbáziselem. A naplózási mechanizmus, melyet arra használunk, hogy a tranzakció ne fejeződhessen be az X kiírása nélkül, atomos; azaz X összes blokkját vagy lemezre írja, vagy semmit sem ír ki. A továbbiakban a naplózási meggondolásokban úgy tekintjük, hogy:

- Az adatbáziselem nem nagyobb egy blokknál.

Fontos figyelembe venni, hogy ezen parancsokat kiadó komponensek különbözőek. A `READ` és `WRITE` utasításokat a tranzakciók használják, az `INPUT` és `OUTPUT` utasításokat a pufferkezelő alkalmazza, ezen túl, ahogy már láttuk, bizonyos feltételek esetén az `OUTPUT` utasítást a naplózási rendszer is használja.

8.2. példa: Annak bemutatására, hogy a tranzakció mikor és hogyan használja a fenti alapműveleteket, tegyük fel, hogy az adatbázis két, A és B eleme tartalmának, az adatbázis minden konzisztens állapotában meg kell egyeznie[3].

[3] Természetesen feltehető a kérdés, hogy miért használnánk két különböző elemet, melyek tartalma mindig megegyezik ahelyett, hogy egyetlen elemet alkalmaznánk. Mindazonáltal, ennek az egyszerű numerikus megszorításnak a teljesítése jól jellemez nagyon sok valóságos megszorítást, mint például amikor előírják, hogy a repülőn az eladott helyek száma 10%-nál

A T tranzakció tartalmazza a következő két lépést:

```
A := A*2;
B := B*2;
```

Vegyük figyelembe, hogyha a tranzakcióra az egyetlen konzisztenciaelvárás az $A = B$, továbbá ha T korrekt adatbázis-állapotban indul, és tevékenységét rendszerhiba, valamint a párhuzamosan működő tranzakciókkal való kölcsönhatás nélkül be tudja fejezni, akkor az adatbázis befejezéskori állapotának is konzisztensnek kell lennie. Ekkor T megduplázva két azonos tartalmú elem értékét, kap két új, azonos értékű elemet.

T végrehajtása maga után vonja A és B lemezről való beolvasását, az aritmetikai műveletek a T lokális memória változóiban kerülnek végrehajtásra, végül A és B új értékei visszaírásra kerülnek a puffereikbe. T-t hat lényeges lépésből állónak tekinthetjük:

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

Ehhez még hozzáadódik az, hogy a pufferkezelő önállóan végrehajt OUTPUT lépéseket a pufferek tartalmának lemezre történő visszaírása végett. A 8.2. ábra a T elemi lépéseit és az őket követő, a pufferkezelő által végrehajtott OUTPUT utasításokat szemlélteti. Tegyük fel, hogy kezdetben $A = B = 8$. Az A és B pufferbeli és lemezen tárolt értékei és a T tranzakció t lokális változójának értékei lépésenként a következők:

Tevékenység	t	*Mem* A	*Mem* B	*Lemez* A	*Lemez* B
READ(A,t)	8	8		8	8
t := t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
READ(B,t)	8	16	8	8	8
t := t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

8.2. ábra. *A tranzakció lépései, és hatásuk a memóriában és a lemezen*

T első lépésében beolvassa A-t, mely igény a pufferkezelőben kiváltja az INPUT(A) utasítást, ha A még nincs a pufferben. A értéke a READ utasítás hatására a T tranzakció memóriaterületére a t változóba is bemásolódik. A következő lépés megduplázza t tartalmát, ennek nincs hatása sem A pufferbeli, sem A lemezen tárolt értékére. A harmadik lépés írja t-t A pufferébe, s ennek nincs hatása A lemezen tárolt értékére. A következő három lépés ugyanez, csak B-re vonatkozóan. Végül az utolsó két lépésben másolódik A és B lemezre.

Figyeljük meg, hogy ezen lépések összességének végrehajtása alatt az adatbázis konzisztenciája megőrződik. Ha OUTPUT(A) végrehajtása előtt rendszerhiba fordul elő, akkor ennek nincs hatása a lemezen tárolt adatbázisra, az még olyan, mintha T egyáltalán nem is működött volna, s így a konzisztencia megőrződött. Ha rendszerhiba áll elő OUTPUT(A) végrehajtása után, de még OUTPUT(B) végrehajtása előtt, akkor az adatbázis inkonzisztens állapotban marad. Azt nem tudjuk megelőzni, hogy ilyen szituáció soha elő ne forduljon, de lépéseket tehetünk azért, hogy amikor mégis bekövetkezik, akkor a problémát elháríthassuk – vagy A és B értékének 8-ra való visszaállításával vagy mindkettő 16-ra növelésével. □

---

többel nem haladhatja meg a fedélzeten lévő ülések számát, vagy amikor előírják, hogy a bank kölcsönei összegének meg kell egyeznie a bank követelései összegével.

8.1.5. Feladatok

8.1.1. feladat: Tegyük fel, hogy az adatbázisra vonatkozó konzisztenciamegszorítás: $0 \leq A \leq B$. Állapítsuk meg, hogy következő tranzakciók megőrzik-e az adatbázis konzisztenciáját?

* a) `A := A + B; B := A + B;`
b) `B := A + B; A: = A + B;`
c) `A := B + 1; B := A + 1;`

8.1.2. feladat: A 8.1.1. feladat mindegyik tranzakciójához a számításokon kívül tegyük hozzá a beolvasó-kiíró tevékenységeket is, és mutassuk be a tranzakciók lépésenkénti hatását a memóriában és a lemezen tárolt adatokra. Tegyük fel, hogy kezdetben $A = 5$ és $B = 10$. Mondjunk valamit arról is, hogy lehetséges-e megfelelő OUTPUT műveletekkel biztosítani az adatbázis konzisztenciáját, a tranzakciók végrehajtása közben fellépő hibák esetében is.

8.2. Semmisségi (undo) naplózás

A naplózás tanulmányozását annak elemzésével kezdjük, hogy milyen úton biztosítható a tranzakciók atomossága – ami az adatbázisra nézve abban mutatkozik meg, hogy a tranzakciót vagy teljes egészében végrehajtjuk, vagy egyáltalán nem. A *napló* nem más, mint a *naplóbejegyzések* (log records) sorozata, melyek mindegyike arról tartalmaz valami információt, hogy mit tett egy tranzakció. A tranzakciók tevékenysége nyomon követhető azáltal, hogy a tranzakció működésének hatása lépésenként naplózódik, ugyanez történik az összes tranzakcióval. A tranzakciók nyomkövetése bonyolultabbá teszi a naplózást; nem elegendő egyszerűen a tranzakció végén a tranzakció történetének naplózása.

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. Általános-

Miért oly erős a tranzakciók abortálási hajlama?

Elgondolkodhatunk azon, hogy miért abortálnak (fejeződnek be a normálisnál korábban) a tranzakciók ahelyett, hogy teljesen rendesen befejeződnének. Ennek számos oka van. A legegyszerűbb ok, amikor magában a tranzakció kódjában hiba van, például egy zérussal való osztás fordul elő, melyet a tranzakció „kilövésével" (cancel) kezel a rendszer. Az adatbázis-kezelő rendszer is számos okkal abortáltathatja a tranzakciót. Példaként a tranzakció holtponti helyzetbe (deadlock) kerülhet, ha egy vagy több másik tranzakció lekötve tart olyan erőforrásokat (például ugyanazon adatbáziselembe új érték beírásának joga), melyeket mások is használni kívánnak. A 10.3. részben látni fogjuk, hogy ehhez hasonló szituációkban a rendszer kénytelen egy vagy több tranzakciót abortálni.

ságban a katasztrófák hatásának kijavítását követően a tranzakciókat meg kell ismételni, és az általuk adatbázisba írt új értékeket ismételten ki kell írni. Egyes tranzakciók a munkájukat vissza kívánják vonni, azaz kérik az adatbázis visszaállítását olyan állapotba, mintha a tekintett tranzakció nem is működött volna.

Az általunk vizsgált első naplózási stílus, melyet *semmisségi* (undo) *naplózásnak*[4] neveznek, csak az utóbbi típusú helyreállításra alkalmas. Ha nem teljesen biztos, hogy a tranzakció hatásai teljesen befejeződtek és lemezen tárolódtak, akkor minden olyan változtatást, melyet a tranzakció tehetett az adatbázisban, semmissé kell tenni, azaz az adatbázist vissza kell állítani a tranzakció működése előtti állapotába.

Ebben a részben a naplóbejegyzések alapelgondolását kívánjuk bemutatni, beleértve a tranzakció teljes és hibátlan befejezését, a *véglegesítési* (commit) tevékenységet, és ennek hatását az adatbázis állapotára és a naplózásra. Áttekintjük azt is, hogy maga a napló hogyan keletkezik a memóriában és hogyan íródik ki a lemezre a „flush-log" (naplókiírás) művelettel. Végül megvizsgáljuk konkrétan a semmisségi naplózást, és megtanuljuk, hogyan használhatjuk a katasztrófákból való helyreállításra. Elkerülendő azt, hogy helyreállítás során a teljes naplót át kelljen vizsgálni, bemutatjuk az „ellenőrzőpont-képzés" (checkpointing) ötletét, mely lehetővé teszi, hogy a napló régi részét eldobjuk[5]. Az ellenőrzőpont-képzés módszerét a semmisségi naplózáshoz kapcsolódóan ebben a fejezetben tanulmányozzuk.

[4] Az undo naplózást semmisségi naplózásnak, vagy visszavonási naplózásként is fordítják. *A fordító megjegyzése.*

[5] Ha „csak" a helyreállításra használjuk a naplót, akkor az utolsó ellenőrzőpont-képzésnél korábban keletkezett naplórészt valóban eldobhatjuk, de ha a naplót a korábban történt akciók utólagos elemzésére is fel kívánjuk használni, akkor meg kell tartanunk. Itt figyelembe kell vennünk, hogy a napló állandóan és jelentősen növekedik, de szerencsére a háttértárolók kapacitása is nő, a fajlagos tárolási költség pedig csökken. *A fordító megjegyzése.*

8.2.1. Naplóbejegyzések

Úgy kell tekintenünk, hogy a napló mint fájl, kizárólag bővítésre van megnyitva. Tranzakció végrehajtásakor a naplókezelőé a feladat, hogy minden fontos eseményt a naplóban rögzítsen. A napló blokkjai mindenkor naplóbejegyzésekkel vannak feltöltve, mindegyik bejegyzés egy-egy naplózandó eseményre vonatkozik. A naplóblokkokat elsődlegesen a memóriában hozza létre a rendszer, és a pufferkezelő az adatbázisrendszer többi blokkjaihoz hasonlóan kezeli őket. A naplóblokkokat, amint csak lehetséges, a nem illékony tárolóra írja a rendszer, erről bővebben a 8.2.2. részben szólunk.

A naplózás minden típusa a naplóbejegyzésnek számos formáját használja. E részben a következőkkel foglalkozunk:

1. <START *T*>: Ez a bejegyzés jelzi a *T* tranzakció (végrehajtásának) elkezdődését.
2. <COMMIT *T*>: A *T* tranzakció rendben befejeződött, az adatbázis elemein már semmi további módosítást nem kíván végrehajtani. A *T* által végrehajtott összes adatbázis-módosítás már megtörtént a lemezen. Minthogy azt nem tudjuk felügyelni, hogy a pufferkezelő mikor dönt a memóriablokkok lemezre másolásáról, így általában nem lehetünk biztosak abban, hogyha meglátjuk a <COMMIT *T*> naplóbejegyzést, akkor a változtatások a lemezen már megtörténtek. Ha ragaszkodunk ahhoz, hogy a módosítások már a lemezen is megtörténjenek, ezt az igényt a naplókezelőnek kell kikényszerítenie (mint például a semmisségi naplózás esetében).
3. <ABORT *T*>: A *T* tranzakció nem tudott sikeresen befejeződni. Ha a *T* tranzakció abortált, az általa tett változtatásokat nem kell a lemezre másolni. A tranzakciókezelő feladata annak biztosítása, hogy az ilyen változtatások ne jelenjenek meg a lemezen, vagy ha volt valami hatásuk a lemezen, akkor az törlődjék. Az abortált tranzakció hatásainak helyreállításával a 10.1.1. részben foglalkozunk.

A semmisségi (undo) naplózáshoz csak egyetlen további naplóbejegyzésre van

Milyen nagy a módosítást leíró naplóbejegyzés?

Ha az adatbáziselemek lemezblokkok, és a módosítást leíró naplóbejegyzés tartalmazza az adatbáziselem régi (módosítás előtti) értékét (vagy mind a régi, mind az új értékét, amint a 8.4. részben a semmisségi/helyrehozó naplózásnál látni fogjuk), akkor előfordulhat, hogy a naplóbejegyzés a blokknál nagyobb méretű lesz. Ez nem feltétlen probléma, mert minden hagyományos fájlhoz hasonlóan, a naplót lemezblokkok sorozatának tekinthetjük, mely bájtsorozatot tartalmaz, a (technikai) blokkhatároktól függetlenül. Ezáltal mód nyílik a napló tömörítésére is. Például bizonyos körülmények között csak a módosításokat kell naplóznunk, azaz csak a tranzakció által módosított sor érintett attribútumainak neveit és azok régi értékeit. A változtatások „logikai naplózása" témájával a 10.1.5. részben foglalkozunk.

szükségünk, a *módosítási bejegyzésre* (update record), mely a <*T*,*X*,*v*> hármas. Ezen bejegyzés jelentése: a *T* tranzakció módosította az *X* adatbáziselemet, melynek módosítás előtti értéke *v* volt. A módosítási bejegyzés által leírt változtatás rendesen csak a memóriában történt meg, a lemezen nem; azaz a naplóbejegyzés a `WRITE` tevékenységre vonatkozik, nem pedig az `OUTPUT`ra! (Emlékeztetünk a két művelet közötti különbségre, amit a 8.1.4. részben már láttunk.) Megjegyezzük még, hogy a semmisségi naplózás nem rögzíti az adatbáziselem új értékét, csak a módosítás előtti értéket. Amint látni fogjuk, a semmisségi naplózást alkalmazó rendszerekben a helyreállításkezelő feladata a tranzakció lehetséges hatásainak semmissé tétele, amelyhez elegendő csak a régi értékek tárolása.

8.2.2. A semmisségi naplózás szabályai

Ahhoz, hogy a rendszerhibák utáni helyreállításra a semmisségi naplózást használhassuk, a tranzakcióknak két előírást kell kielégíteniük. Ezek a szabályok arra vonatkoznak, hogy a pufferkezelőnek hogyan kell működnie, valamint előírnak bizonyos, a tranzakció szabályos befejezésekor elvégzendő tevékenységeket. Ezeket itt összefoglalva:

U_1: Ha a *T* tranzakció módosítja az *X* adatbáziselemet, akkor a <*T*,*X*,*v*> típusú naplóbejegyzést *azt megelőzően* kell lemezre kiírni, mielőtt *X* új értékét a lemezre írná a rendszer.

U_2: Ha a tranzakció hibamentesen teljesen befejeződött, akkor a `COMMIT` naplóbejegyzést csak *azt követően* szabad lemezre írni, hogy a tranzakció által módosított összes adatbáziselem már lemezre íródott, de ezután viszont a lehető leggyorsabban.

Összefoglalva az U_1 és U_2 szabályokat, az egy tranzakcióhoz tartozó lemezre írásokat a következő sorrendben kell megtenni:

a) Az adatbáziselemek módosítására vonatkozó naplóbejegyzések kiírása.
b) Maguknak a módosított adatbáziselemeknek a kiírása.
c) A `COMMIT` naplóbejegyzés kiírása.

Az a) és b) lépések minden módosított adatbáziselemre vonatkozóan önmagukban, külön-külön végrehajtandók (nem lehet a tranzakció több módosítására csoportosan megtenni)!

A naplóbejegyzések lemezre írásának kikényszerítésére a naplókezelőnek szüksége van a *flush-log* parancsra, mely felszólítja a pufferkezelőt az összes korábban még ki nem írt naplóblokkoknak a lemezre való kiírására, valamint azon pufferek kiírására, amelyek tartalma utolsó kiírásuk óta megváltozott. A `FLUSH LOG` parancsot a tevékenységek közé fogjuk iktatni. A tranzakciókezelőnek szüksége van arra is, hogy a pufferkezelőt az adatbáziselemekre vonatkozó `OUTPUT` akció végrehajtására felszólíthassa. A folytatásban be fogjuk mutatni a tranzakció lépései közé illesztett `OUTPUT` tevékenységet is.

8.3. példa: A semmisségi naplózás fényében vizsgáljuk meg újra a 8.2. példában már megismert tranzakciót. A 8.3. ábra a 8.2. ábra kibővítése, bemutatván a naplóbejegyzéseket is, és a naplókiírási tevékenységet is a *T* tranzakció végrehajtása során. Megjegyezzük, hogy a fejlécben rövidítéseket kellett használnunk; M-*A* rövidítést használjuk „*A*-nak memóriapufferbe másolása", D-*B*-t pedig „*B*-nek lemezre másolása" jelentéssel, és hasonlóan a többi rövidítésben is.

A 8.3. ábra 1) sorában a *T* tranzakció elkezdődik. Az első, ami történik, az a <`START` *T*> bejegyzés naplóba írása. A 2) sor *A*-nak *T* általi beolvasását jelenti. A 3) sor *t* módosítása, melynek nincs semmilyen hatása sem a lemezen tárolt adatbázisra, sem annak memóriapufferben található egyetlen részére sem. Sem a 2), sem a 3) sor nem igényel naplóbejegyzést, mert nincs hatásuk az adatbázisra.

A 4) sor *A* új értékének pufferbe írása. *A* ezen módosítására vonatkozik a <*T*,*A*,8> naplóbejegyzés, mely azt rögzíti, hogy *A* korábbi értékét, 8-at *T* megváltoztatta. Megjegyezzük, hogy az új érték, 16, nincs megemlítve a semmisségi naplózás naplójában.

Lépés	*Tevékenység*	*t*	*M*-A	*M*-B	*D*-A	*D*-B	*Napló*
1)							<`START` *T*>
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)							<`COMMIT` *T*>
12)	`FLUSH LOG`						

8.3. ábra. *Tevékenységek és naplóbejegyzéseik*

Más naplózási módszerek áttekintése

A „helyrehozó" (redo) naplózás (8.3. részben tárgyaljuk), a katasztrófát követő helyreállítás során helyrehozza az összes olyan tranzakció hatását, melyek már elindultak, de még nem fejeződtek be. A helyrehozó naplózás szabályai biztosítják, hogy ne legyen szükséges az olyan tranzakciók helyrehozása, melyekre vonatkozó `COMMIT` bejegyzést a naplóban megtaláljuk. A „semmisségi/helyrehozó" (undo/redo) naplózás (8.4. részben tárgyaljuk) a katasztrófát követő helyreállítás során semmissé teszi az összes olyan tranzakció hatását, amely még nem fejeződött be, és helyre kívánja hozni azokat a tranzakciókat, melyek már befejeződtek. Ezenfelül a napló és a pufferek kezelési szabályai biztosítani kívánják, hogy ezek a lépések az adatbázis minden sérülését helyreállítsák.

Az 5)-től 7)-ig sorokban a B-re vonatkozóan ugyanazon lépések hajtódnak végre, mint korábban A-ra. E ponton a T rendben befejeződött, tevékenységét véglegesíteni kell. A megváltozott A és B értékét lemezre kell másolni, betartva a semmisségi (undo) naplózás két szabályát, a következő lépéseknek kötött sorrendben kell megtörténnie.

Első, hogy A és B addig nem másolható lemezre, amíg a módosítást leíró naplóbejegyzések lemezre nem kerülnek. Ezt a 8) lépéssel biztosítjuk, a `FLUSH LOG` hatására az eddigi összes naplóbejegyzés lemezre íródik. E kiírást követően a 9) és 10) lépések A-t és B-t lemezre másolják. Ezeket a lépéseket a T teljes befejeződése, a véglegesítés érdekében a tranzakciókezelő igénye szerint a pufferkezelő valósítja meg.

S ekkor lehetséges a T teljes és sikeres befejezése, ezt jelzendő a 11) lépésben <COMMIT T> bejegyzés a naplóban íródik. Végül a 12) lépésben ismét ki kell adni a `FLUSH LOG` utasítást azért, hogy biztosítsuk a <COMMIT T> naplóbejegyzés lemezre való kiírását. Ezen naplóbejegyzés lemezre való kiírása nélkül, bár olyan helyzetben vagyunk, hogy a tranzakció teljesen és hibamentesen befejeződött, ennek a napló későbbi elemzésekor nem fogjuk nyomát találni. Az ilyen szituációk olyan furcsa viselkedést eredményezhetnek, hogy hiba esetén, amint a 8.2.3. részben látni fogjuk, a felhasználó azt tapasztalja, hogy a tranzakció hibamentesen rendesen befejeződött, a lemezre kiírt módosítások mégis semmissé váltak, a tranzakció ténylegesen abortált[6]. □

8.2.3. Helyreállítás a semmisségi naplózás használatával

Tételezzük fel, hogy rendszerhiba fordult elő. Előfordulhat, hogy valamely adott tranzakció által végzett adatbázis-módosítások közül bizonyosak lehet, hogy már lemezre íródtak, míg más módosítások – melyeket ugyanezen tranzakció hajtott végre – nem jutottak el a lemezre. Ha így történt, a tranzakció nem atomosan hajtódott végre, ennek következtében az adatbázis inkonzisztens állapotba kerülhetett. A *helyreállítás-kezelő* (recovery manager) feladata – a napló használatával – az adatbázist konzisztens állapotba visszaállítani.

Ebben a részben csak a legegyszerűbb helyreállítás-kezelő módszerrel foglalkozunk, mely a teljes naplót látja, függetlenül annak méretétől, és a napló vizsgálatával hajtja végre az adatbázis módosításait. A 8.2.4. részben egy sokkal finomabb megközelítést mutatunk be, amikor ellenőrzőpont periodikus készítésével a rendszer korlátozza azt a távolságot, ameddig a helyreállítás-kezelőnek a korábbi történéseket (a naplóban) vizsgálnia kell.

A helyreállítás-kezelő első feladata a tranzakciók felosztása sikeresen befejezett és nem befejezett tranzakciókra. Ha található <COMMIT T> naplóbejegyzés, akkor az U_2 szabálynak megfelelően a T tranzakció által végrehajtott összes adatbázis-módosítások már korábban lemezre íródtak. Így a T tranzakció önmagában, a hiba fellépésekor, nem hagyhatta az adatbázist inkonzisztens állapotban.

[6] A tranzakció valójában nem abortált, de egy később fellépett hiba elemzésekor, mivel a rendszer nem talál <COMMIT T> naplóbejegyzést, úgy tekinti, hogy a tranzakció nem tudott teljesen és rendesen befejeződni, ezért a helyreállító rendszer a tranzakció hatásait semmissé teszi, végeredményben az történik, mintha T abortált volna. *A fordító megjegyzése.*

Háttértevékenységek, naplózás, pufferkezelés

A 8.3. ábrán látottaknak megfelelően a tranzakció tevékenységei és a naplóbejegyzések sorozata azt az elképzelést sugallják, mintha ezek a tevékenységek elkülönülten következnének be. Ugyanakkor az adatbázis-kezelő rendszer számos tranzakció szimultán kezelését kell hogy megoldja. Így egy T tranzakció négy naplóbejegyzése a naplóban más tranzakciók naplóbejegyzéseivel keveredhet. Ezenfelül, ha a másik tranzakciók valamelyike is a napló lemezre írását kezdeményezi (`FLUSH LOG` kiadásával), akkor a T-re vonatkozó naplóbejegyzések esetleg már korábban lemezre kerülnek, mint ahogy azt a 8.3. ábrán látható `FLUSH LOG` utasítások okoznák. Abból nem származik probléma, ha az adatbázis módosítására vonatkozó naplóbejegyzések a szükségesnél korábban jelennek a naplóban. A <COMMIT T> naplóbejegyzést úgysem fogjuk a T `OUTPUT` utasításai végrehajtásának befejezésénél korábban kiírni, ezzel biztosítani tudjuk, hogy a módosított adatbázisértékek korábban megjelenjenek a lemezen, mint a `COMMIT` naplóbejegyzés.

Kényes helyzet áll elő, ha az A és B, két adatbáziselem, közös blokkban található. Akkor egyikük lemezre írása maga után vonja a másikuk kiírását is. Legrosszabb esetben az egyik adatbáziselem túl korai kiírásával megsértjük az U_1 szabályt. Ez szükségessé tehet további előírásokat a tranzakcióra nézve azért, hogy a semmisségi naplózási módszer használható legyen. Például a 9.3. részben ismertetett zárolási módszert kell használnunk annak megelőzésére, nehogy két tranzakció, egyszerre, ugyanazon blokkot használja (e példában lemezblokkokat tekintünk adatbáziselemeknek). Ilyen és hasonló problémák akkor jelentkeznek, amikor az adatbáziselemek blokkok részei, ez motiválja azt a javaslatunkat, hogy a blokkokat adatbáziselemeknek tekintsük.

Amennyiben feltételezzük, hogy a naplóban találunk <START T> bejegyzést, de nem találunk <COMMIT T> bejegyzést, akkor a T végrehajthatott az adatbázisban olyan módosításokat, melyek még a hiba fellépése előtt lemezre íródtak, amíg más változtatások a memóriapufferekben sem történtek meg, vagy a memóriapufferben megtörténtek ugyan, de a lemezre már nem íródtak ki. Ilyen esetben a *T nem komplett tranzakció*, és hatását semmissé kell tenni, azaz a T által módosított adatbáziselemek értékét vissza kell állítani korábbi értékeikre. Szerencsére az U_1 szabály betartása biztosítja, hogy ha T, a hiba jelentkezése előtt, az X értékét módosította, akkor a hiba jelentkezése előtt már a lemezen lévő naplóba kellett kiíródni egy <T,X,v> bejegyzésnek. S így a helyreállítás során módunkban áll a v értéket az X adatbáziselembe visszaírni. Megjegyezzük, hogy ez a szabály bizonyítottnak tekinti, hogy X korábbi értéke v volt, de ennek tényleges ellenőrzésére alkalmatlan. (A <T,X,v> naplóbejegyzésnek hinnünk kell, annak helyességét önmagában nem tudjuk ellenőrizni.)

Minthogy a naplóban számos, rendesen befejezett és teljesen be nem fejezett tranzakció nyomát találhatjuk, s ezek közül több tranzakció módosíthatta az X adatbázis-

elemet is, így nagyon ügyelnünk kell arra, hogy milyen sorrendben állítjuk vissza X korábbi tartalmát. Ezért a helyreállítás-kezelő a naplót a végéről kezdi átvizsgálni (tehát az utoljára felírt bejegyzéstől a korábban felírottak irányába). Amint halad a napló átvizsgálásával, megjegyzi mindazon T tranzakciókat, melyekre vonatkozóan a naplóban <COMMIT T> vagy <ABORT T> bejegyzést talált. Amint halad visszafelé, amikor $<T,X,v>$ bejegyzést lát, akkor:

1. Ha ugyanerre a T tranzakcióra vonatkozó COMMIT bejegyzéssel már találkozott, akkor nincs teendője, T rendesen és teljesen befejeződött, hatásait nem kell tehát semmissé tenni.
2. Minden más esetben T nem teljes vagy abortált tranzakció. A helyreállítás-kezelő X értékét v-re kell hogy cserélje.[7]

Miután a helyreállítás-kezelő végrehajtotta a fenti változtatásokat, minden, korábban abortált, nem teljes T tranzakcióra vonatkozóan <ABORT T> naplóbejegyzést ír a naplóba, és kiváltja annak naplófájlba való kiírását is (FLUSH LOG). Ekkor az adatbázis normál használata folytatódhat, új tranzakciók végrehajtása kezdődhet.

8.4. példa: Tekintsük a 8.3. ábrán és a 8.3. példában látott tevékenységeket. Rendszerhiba számos különböző időpontban felléphet; tekintsük át az összes lényeges, különböző esetet:

1. A hiba a 12) lépést követően jelentkezett. Tudjuk, hogy ekkor a <COMMIT T> bejegyzést már lemezre írta a rendszer. A hiba kezelése során a T tranzakció hatásait már nem kell visszaállítani, s a T-re vonatkozó összes naplóbejegyzést a helyreállítás-kezelő figyelmen kívül hagyhatja.
2. A hiba a 11) és 12) lépések között keletkezett. Ekkor előfordulhat, hogy a COMMIT bejegyzést tartalmazó naplóbejegyzés már lemezre íródott, például, ha a naplóbejegyzés kiírását másik tranzakció már kérte a pufferkezelőtől. Ha így történt, akkor T-re vonatkozólag a hiba kezelése az 1) esethez hasonló. Ha azonban a COMMIT bejegyzés a lemezen nem található, akkor a helyreállítás-kezelő a T tranzakciót befejezetlennek tekinti. Ahogy olvassa a naplót visszafelé, először a $<T,B,8>$ bejegyzést fogja megtalálni (a T tranzakcióra vonatkozólag). Ennek megfelelően a lemezen a B tartalmába a 8-at állítja vissza. Majd a $<T,A,8>$ naplóbejegyzés miatt A tartalmába kerül 8. Végezetül <ABORT T> bejegyzést ír a naplóba és a lemezre.
3. Ha a hiba a 10) és 11) lépések között lépett fel, akkor a COMMIT bejegyzés még biztosan nem történt meg, tehát T befejezetlen, hatásainak semmissé tétele a 2) esetnek megfelelően történik.
4. A 8) és 10) lépések között bekövetkező hiba fellépésekor az előző 3) esethez hasonlóan T hatásait semmissé kell tenni. Az egyetlen különbség, hogy az A és/vagy B módosítása még nem jelent meg a lemezen. Ettől függetlenül mindkét adatbázis-elem korábbi értékét, 8-at, állítja vissza a rendszer.

[7] Ha T abortált, akkor az összes hatását az adatbázisban mindenképpen vissza kell állítani.

A helyreállítás közben bekövetkező (újabb) katasztrófákról

Tegyük fel, hogy egy korábbi hiba utáni helyreállítás közben ismét rendszerhiba lép fel. A semmisségi (undo) naplózás oly módon van megtervezve, hogy a korábbi érték változtatás előtti tárolása következtében a helyreállító lépések idempotensek; ami azt jelenti, hogy a helyreállító tevékenység többszöri végrehajtása pontosan ugyanolyan hatású, mint egyszeri végrehajtása. Arra koncentrálunk csak, hogyha találunk $<T,X,v>$ naplóbejegyzést, akkor nem számít, hogy X értéke már v, X értékét (esetleg ismételten) v-re állíthatjuk. Hasonlóan semmi problémát nem okoz, ha a helyreállítási folyamat egészét (vagy félbemaradt részét) többször megismételjük, az esetleg már visszaállított értékeket ismételten visszaállítjuk. Ugyanezt fogjuk látni az e fejezetben tárgyalt többi naplózási módszerek esetében is. Minthogy a helyreállító tevékenység idempotens, másodszor (többször is) probléma nélkül megismételhetjük, függetlenül a korábbi helyreállító akció által végrehajtott módosításoktól.

5. Amennyiben a hiba a 8) lépésnél korábban jelentkezik, akkor még az sem biztos, hogy a T tranzakcióra vonatkozó naplóbejegyzések közül bármi is lemezre (a naplóba) került-e, de nem is fontos, mivel az U_1 szabály miatt tudjuk, hogy mielőtt az A és/vagy B adatbáziselemek a lemezen módosulnának, a megfelelő módosítási naplóbejegyzésnek a naplóban meg kell jelennie. Esetünkben ez nem történt meg, következésképp a módosítások sem történtek meg, tehát nincs is visszaállítási feladat. □

8.2.4. Az ellenőrzőpont-képzés

Mint láttuk, a helyreállítás elvben a teljes napló átvizsgálását igényli. Ha a naplózásban (az eddig bemutatott) semmisségi (undo) naplózás módszerét követjük, akkor, ha egy tranzakció a COMMIT naplóbejegyzést már kiírta a naplóba, akkor az ezen tranzakcióra vonatkozó naplóbejegyzésekre a helyreállítás során nincs már szükség[8]. Gondolhatnánk arra, hogy a tranzakcióra vonatkozó, a COMMITot megelőző naplóbejegyzéseket törölhetnénk a naplóból, de ezt nem mindig tehetjük meg. Ennek oka az, hogy gyakran sok tranzakció működik egyszerre, ha a naplót egy tranzakció befejezése után csonkítanánk, esetleg elveszítenénk más, még aktív tranzakciókra vonatkozó bejegyzéseket, s így – ha szükség lenne rá –, nem tudnánk a naplót a helyreállításra használni.

E lehetséges probléma megoldására a legegyszerűbb mód, a naplóra vonatkozóan, ismétlődően *ellenőrzőpontot* képezni. Az egyszerű ellenőrzőpont képzése:

[8] A tranzakció teljesen és helyesen befejeződött, nem kell tehát semmissé tenni hatásait, a napló bejegyzéseire ez okból már valóban nincs szükség, a tevékenységek utólagos elemzése miatt azonban még e naplóbejegyzések is fontosak lehetnek. *A fordító megjegyzése.*

<START T_1>
<T_1,A,5>
<START T_2>
<T_2,B,10>
<T_2,C,15>
<T_1,D,20>
<COMMIT T_1>
<COMMIT T_2>
<CKPT>
<START T_3>
<T_3,E,25>
<T_3,F,30>

8.4. ábra. *Napló egyszerű ellenőrzőpont-képzéssel*

1. Új tranzakcióindítási kérések kiszolgálásának leállítása.
2. A még aktív tranzakciók helyes és teljes befejezésének vagy abortálásának és a COMMIT vagy az ABORT bejegyzés naplóba írásának kivárása.
3. A napló lemezre kiírása (FLUSH).
4. <CKPT>[9] naplóbejegyzés képzése és kiírása a naplóba, és ismételt FLUSH.
5. Tranzakcióindítási kérések kiszolgálása.

Az ellenőrzőpont kiírását megelőzően végrehajtott tranzakciók mind befejeződtek, s az U_2 szabálynak megfelelően módosításaik már lemezre kerültek. Ennek megfelelően – ezen tranzakciók tevékenységére nézve – egy esetleges későbbi hiba elhárításakor már nem igényel a rendszer helyreállítást. A helyreállítás során a naplót a végétől visszafelé csak a <CKPT> bejegyzésig kell elemezni azért, hogy a nem befejezett tranzakciókat azonosítsuk. Amikor a <CKPT> bejegyzést megtaláljuk, ebből tudjuk, hogy már láttuk az összes befejezetlen tranzakciót. Mivel az ellenőrzőpont-képzés alatt újabb tranzakció nem indulhatott, látnunk kellett a befejezetlen tranzakciókhoz tartozó összes naplóbejegyzést. Ezért nem szükséges a <CKPT> bejegyzésnél korábbi naplórészt elemeznünk, s – ha más okból már nincs szükségünk rá – biztonsággal törölhetjük vagy felülírhatjuk.

8.5. példa: Tegyük fel, hogy a napló így kezdődik:

<START T_1>
<T_1,A,5>
<START T_2>
<T_2,B,10>

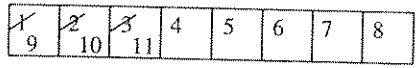
S ekkor döntünk ellenőrzőpont létrehozásáról. Minthogy T_1 és T_2 aktív (nem befejezett) tranzakciók, meg kell várnunk befejeződésüket, mielőtt a <CKPT> bejegyzést a naplóba írnánk.

[9] CKPT – Checkpoint (ellenőrzőpont) rövidítése. *A lektor megjegyzése.*

Az utolsó naplóbejegyzés megtalálása

A napló lényegében egy fájl, melynek blokkjai tartalmazzák a naplóbejegyzéseket. A blokk még ki nem töltött részeit „üres"-ként jelölik. Ha a bejegyzéseket soha nem írjuk felül, akkor a helyreállítás-kezelő az utolsó bejegyzést úgy keresi meg, hogy megkeresi az első üres bejegyzést, és az ezt megelőző bejegyzés a fájl utolsó érvényes bejegyzése.

Ha a régi naplóbejegyzéseket felülírjuk, akkor a naplóbejegyzéseket az alábbi módon:



egyedi, növekvő sorszámmal kell ellátnunk. Ekkor azt a bejegyzést kell megtalálnunk, melynek nagyobb a sorszáma, mint a következőé; ez a bejegyzés a napló pillanatnyi vége, s a naplóbejegyzései sorszámuk szerinti sorban keletkeztek.

A gyakorlatban a nagyméretű napló több fájl egyesítése is lehet. Logikailag ekkor is egy fájlnak tekintjük, s a végét a megfelelő részfájlban keressük.

A napló egy lehetséges folytatását a 8.4. ábra mutatja. Tegyük fel, hogy e ponton lép fel rendszerhiba. A naplót a végétől visszafelé elemezve, T_3-at fogjuk az egyetlen be nem fejezett tranzakciónak találni, és így E és F korábbi értékeit, 25-öt és 30-at kell csak visszaállítanunk. Amikor megtaláljuk a <CKPT> bejegyzést, tudjuk, hogy nem kell a korábbi naplóbejegyzéseket elemeznünk, és tudjuk, hogy az adatbázis állapotának helyrehozásával végeztünk.
□

8.2.5. Ellenőrzőpont-képzés a rendszer működése közben[10]

A 8.2.4. részben bemutatott ellenőrzőpont-képzési technika problémája, hogy gyakorlatilag le kell állítani a rendszer működését (nem engedni új tranzakciók indítását) az ellenőrzőpont elkészültéig. Minthogy az aktív tranzakciók még hosszabb időt igényelhetnek a normális vagy abnormális befejeződésükig, így a felhasználó számára a rendszer leállítottnak tűnhet. Egy jóval bonyolultabb módszerrel, a *működés közbeni ellenőrzőpont-képzéssel* elérjük, hogy az ellenőrzőpont-képzés alatt új tranzakciók indítását ne kelljen szüneteltetni. E módszer lépései:

1. <START CKPT (T_1, ..., T_k)> naplóbejegyzés készítése és a naplóbejegyzés lemezre írása (FLUSH). T_1, ..., T_k az éppen *aktív* tranzakciók nevei.

[10] Az eredeti műben „nonquiescent checkpointing", azaz „nem nyugalmi ellenőrzőpont-képzés"-ként szerepel. *A fordító megjegyzése.*

2. Meg kell várni a T_1, ..., T_k tranzakciók mindegyikének normális vagy abnormális befejeződését, nem tiltva közben újabb tranzakciók indítását.
3. Ha a T_1, ..., T_k tranzakciók mindegyike befejeződött, akkor <END CKPT> naplóbejegyzés elkészítése és a naplóbejegyzés lemezre írása (FLUSH).

Az ilyen típusú napló felhasználásával a következőképpen tudunk rendszerhiba után helyreállítani: a naplót a végétől visszafelé elemezve megtaláljuk az összes nem befejezett tranzakciót, régi értékére visszaállítjuk az ezen tranzakciók által megváltoztatott adatbáziselemek tartalmát. Két eset fordulhat elő aszerint, hogy visszafelé olvasván a naplót, az <END CKPT> naplóbejegyzést vagy a <START CKPT (T_1, ..., T_k)> naplóbejegyzést találjuk előbb.

- Ha előbb az <END CKPT> naplóbejegyzéssel találkozunk, akkor tudjuk, hogy az összes még be nem fejezett tranzakcióra vonatkozó naplóbejegyzést a legközelebbi korábbi <START CKPT (T_1, ..., T_k)> naplóbejegyzésig megtaláljuk. Ennél a <START CKPT (T_1, ..., T_k)> naplóbejegyzésnél megállhatunk, a még korábbiakat már nem kell használnunk, azokat el is dobhatjuk.
- Amennyiben a <START CKPT (T_1, ..., T_k)> naplóbejegyzéssel találkozunk előbb, az azt jelenti, hogy a katasztrófa az ellenőrzőpont-képzés közben fordult elő. Ennek következtében T_1, ..., T_k tranzakciók nem fejeződtek be (legalábbis nem tudtuk a befejeződést regisztrálni) a hiba fellépéséig. Ekkor a be nem fejeződött tranzakciók közül a legkorábban kezdődött tranzakció indulásáig kell a naplóban visszakeresnünk, annál korábbra nem. Az ezt megelőző START CKPT bejegyzés biztosan megelőzi a keresett összes tranzakció indítását leíró bejegyzéseket.[11] Ezenfelül, ha ugyanazon tranzakció naplóbejegyzéseire nézve láncokat is használunk, akkor nem kell a napló minden bejegyzését átnéznünk ahhoz, hogy megtaláljuk a még be nem fejezett tranzakciókra vonatkozó bejegyzéseket, elegendő csak az adott tranzakció bejegyzései láncán visszafelé haladnunk.

Általános szabályként, ha egy <END CKPT> naplóbejegyzést kiírunk lemezre, akkor a megelőző START CKPT bejegyzésnél korábbi naplóbejegyzéseket törölhetjük.

8.6. példa: Tegyük fel, hogy a napló, mint a 8.5. példában is, így kezdődik:

<START T_1>
<T_1,A,5>
<START T_2>
<T_2,B,10>

S most úgy döntünk, hogy működés közbeni ellenőrzőpontot hozunk létre. Mint-

[11] Mivel működés közbeni ellenőrzőpont-képzéssel dolgozunk, előfordulhat, hogy a be nem fejeződött tranzakciók némelyike az előző ellenőrzőpont-képzés kezdete és befejezése között indult el.

hogy e pillanatban T_1 és T_2 aktív (nem befejezett) tranzakciók, ezért a következő naplóbejegyzést kell felírnunk:

<START CKPT (T_1, T_2)>

Tegyük fel, hogy amíg T_1 és T_2 befejeződésére várunk, azalatt egy másik tranzakció, T_3 elkezdődik. A napló egy lehetséges folytatását a 8.5. ábrán mutatjuk be.

Tételezzük fel, hogy most lépett fel valamilyen hiba. A naplót a végétől visszafelé vizsgálva, úgy fogjuk találni, hogy T_3 egy be nem fejezett tranzakció, s 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 tranzakciók a megelőző START CKPT naplóbejegyzés után indulhattak csak el. Tovább visszafelé elemezve, megtaláljuk a <T_3,E,25> bejegyzést, mely megmondja nekünk, hogy 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, és további adatbázis-módosításra vonatkozó bejegyzést nem találunk, így az adatbázison mást már nem kell megváltoztatnunk.

Tegyük fel most, hogy az ellenőrzőpont képzése közben történt katasztrófa, s a

<START T_1>
<T_1,A,5>
<START T_2>
<T_2,B,10>
<START CKPT (T_1, T_2)>
<T_2,C,15>
<START T_3>
<T_1,D,20>
<COMMIT T_1>
<T_3,E,25>
<COMMIT T_2>
<END CKPT>
<T_3,F,30>

8.5. ábra. *Napló működés közbeni ellenőrzőpont-képzéssel*

<START T_1>
<T_1,A,5>
<START T_2>
<T_2,B,10>
<START CKPT (T_1, T_2)>
<T_2,C,15>
<START T_3>
<T_1,D,20>
<COMMIT T_1>
<T_3,E,25>

8.6. ábra. *Napló ellenőrzőpont-képzés közben történt rendszerkatasztrófa során*

napló vége a 8.6. ábrán bemutatott. Visszafelé elemezve a naplót, azonosítjuk a T_3, majd a T_2 tranzakciókat, melyek nincsenek befejezve, s helyreállító módosításokat kell tennünk. Amikor megtaláljuk a <START CKPT (T_1, T_2)> naplóbejegyzést, megtudjuk, hogy az egyetlen további olyan tranzakció, mely lehetséges, hogy nincs befejezve, a T_1. Minthogy azonban a <COMMIT T_1> bejegyzést már láttuk, ebből tudjuk, hogy T_1 *nem* be nem fejezett tranzakció. Láttuk már továbbá a <START T_3> bejegyzést is, s így már tudjuk, hogy csak addig kell folytatnunk a napló visszafelé elemzését, amíg T_2 START bejegyzését meg nem találjuk. Eközben még a B adatbáziselem értékét is visszaállítjuk 10-re. □

8.2.6. Feladatok

8.2.1. feladat: Adjuk meg a 8.1.1. feladatban szereplő tranzakciók (nevezzük mindet T-nek) semmisségi (undo) naplóbejegyzéseit. Tegyük fel, hogy kezdetben $A = 5$ és $B = 10$.

8.2.2. feladat: Az alábbi naplóbejegyzés-sorozatok valamely T tranzakció tevékenységeit tükrözik. Állapítsa meg a semmisségi (undo) naplózás szabályainak megfelelően a naplóbejegyzések és az adatbáziselemeket tartalmazó blokkok lemezre írási lehetőségeit, figyelembe véve, hogy naplóbejegyzést nem lehet addig a lemezre írni, amíg a megelőző bejegyzés nem került lemezre.

* a) <START T>; <T,A,10>; <T,B,20>; <COMMIT T>;
b) <START T>; <T,A,10>; <T,B,20>; <T,C,30>; <COMMIT T>;

! **8.2.3. feladat:** A 8.2.2. feladatban szereplő naplórészleteket bővítsük ki olyan tranzakciók naplóivá, melyek n számú adatbáziselemnek új értéket adnak. Mennyi szabályos, naplózott esemény lesz a tranzakciókban, ha betartjuk a semmisségi naplózás szabályait?

8.2.4. feladat: A következő naplóbejegyzés-sorozat a T és U két tranzakcióra vonatkozik: <START T>; <T,A,10>; <START U>; <U,B,20>; <T,C,30>; <U,D,40>; <COMMIT U>; <T,E,50>; <COMMIT T>. Adjuk meg a helyreállítás-kezelő tevékenységeit, beleértve a lemezen és a naplóban tett módosításait, ha katasztrófa lépett fel, és az utolsó lemezre került naplóbejegyzés:

a) <START U>.
* b) <COMMIT U>.
c) <T,E,50>.
d) <COMMIT T>.

8.2.5. feladat: A 8.2.4. feladatban leírt helyzetek mindegyikére adjuk meg, hogy a T és U által lemezre írott értékek közül melyeknek *kell* megjelenni a lemezen, és melyek jelen*het*nek meg a lemezen?

*! **8.2.6. feladat:** Tegyük fel, hogy a 8.2.4. feladatban szereplő U tranzakciót úgy módosítjuk, hogy az <U,D,40> bejegyzés helyett az <U,A,40> keletkezzen. Mi annak a hatása az A lemezen tárolt értékére, ha a 8.2.4. feladatban megadott pillanatokban lép fel katasztrófa? Mit mutat ez a példa a naplózás lehetőségeiről a tranzakciók atomossága megőrzésében?

8.2.7. feladat: Tegyük fel, hogy a napló a következő bejegyzéssorozatot tartalmazza: <START S>; <S,A,60>; <COMMIT S>; <START T>; <T,A,10>; <START U>; <U,B,20>; <T,C,30>; <START V>; <U,D,40>; <V,F,70>; <COMMIT U>; <T,E,50>; <COMMIT T>; <V,B,80>; <COMMIT V>. Tegyük fel továbbá, hogy a működés közbeni ellenőrzőpontképzést kezdjük alkalmazni, közvetlenül az alábbi naplóbejegyzések (memóriában való) megjelenésétől kezdve:

a) <S,A,60>.
* b) <T,A,10>.
c) <U,B,20>.
d) <U,D,40>.
e) <T,E,50>.

Mindegyik fenti esetre adjuk meg, hogy:

i) Mikor íródik fel az <END CKPT> naplóbejegyzés, és
ii) Bármelyik lehetséges pillanatban, ha hiba lép fel, meddig kell a naplóban visszafelé tekinteni ahhoz, hogy minden befejezetlen tranzakciókra vonatkozó bejegyzést megtaláljunk.

8.3. Helyrehozó naplózás (redo logging)

A semmisségi naplózás (undo logging) természetes és egyszerű stratégiát valósít meg a napló kezelésére és a rendszerhibák esetén való visszaállításra, de a probléma megoldásának nem ez az egyetlen lehetséges megközelítése. A semmisségi naplózás potenciális problémája az, hogy csak azután tudjuk befejezni a tranzakciót, ha az összes adatbázis-módosításai már lemezre íródtak. Olykor a lemezműveletekkel tudnánk takarékoskodni, ha megengednénk, hogy az adatbázis-módosításokat csak a memóriában végezzék a tranzakciók, miközben a napló az eseményeket rögzíti, azért, hogy katasztrófa esetében is biztonságban legyen az adatbázis.

Az adatbáziselemek lemezre való azonnali visszaírásának kényszerét elkerülhetjük, ha a *helyrehozó naplózás* (redo logging) módszerét választjuk. Az alapvető különbségek a semmisségi és a helyrehozó naplózás között az alábbiak:

1. Amíg a semmisségi naplózás a helyreállítás során a be nem fejezett tranzakciók hatásait semmissé teszi, a befejezett tranzakciók hatásait pedig nem módosítja, ad-

dig a helyrehozó naplózás figyelmen kívül hagyja a be nem fejezett tranzakciókat, és megismétli a normálisan befejezettek által végrehajtott változtatásokat.

2. A semmisségi naplózás megkívánja az adatbáziselemek lemezen való módosítását a COMMIT naplóbejegyzés lemezre írása előtt, addig a helyrehozó naplózás a COMMIT naplóbejegyzés lemezre írását várja el, mielőtt bármit is változtatna a lemezen lévő adatbázisban.
3. A semmisségi naplózás U_1 és U_2 szabályainak betartása mellett csak a módosított adatbáziselemek régi tartalmát kell megőriznünk az esetleges visszaállítás biztosításához, a helyrehozó naplózással történő helyreállításhoz a módosított elemek új értékére van szükség. Emiatt a helyrehozó naplózás naplóbejegyzései ugyanolyan formájúak, de más a jelentésük, mint a semmisségi naplózásnál alkalmazottaké.

8.3.1. A helyrehozó naplózás szabályai

A helyrehozó naplózás az adatbáziselemek módosítását a naplóbejegyzésben az új értékkel képviseli (nem pedig a régivel, mint a semmisségi naplózásnál). Ez a bejegyzés ugyanúgy néz ki, mint a semmisségi naplózásnál használt: <*T*,*X*,*v*>, a jelentése azonban más. E bejegyzés jelentése: „a *T* tranzakció az *X* adatbáziselemnek a *v* értéket adta". E bejegyzésben az *X* régi értékét nem jelzi semmi. Mindig, ha a *T* tranzakció módosítja az *X* adatbáziselem értékét, akkor egy <*T*,*X*,*v*> bejegyzést kell a naplóba írni.

Annak sorrendjét, hogy az adat- és naplóbejegyzések hogyan kell hogy lemezre kerüljenek, az alábbi egyszerű „helyrehozó naplózási szabály", az úgynevezett *írj korábban naplózási szabály* írja le.

R_1: Mielőtt az adatbázis bármely *X* elemét a lemezen módosítanánk, szükséges, hogy az *X* ezen módosítására vonatkozó összes naplóbejegyzése, azaz <*T*,*X*,*v*> és <COMMIT *T*>, a lemezre kerüljenek.

Minthogy a COMMIT bejegyzést csak akkor írhatjuk a naplóba, ha a tranzakció teljesen és hibamentesen befejeződött, így a COMMIT bejegyzés csak a módosításokat leíró bejegyzések után állhat, ezért úgy is összegezhetjük az R_1 szabálya hatását, hogy: ha helyrehozó naplózást használunk, akkor az egy tranzakcióra vonatkozó lemezre írásoknak a következő sorrendben kell megtörténniük:

1. Az adatbáziselemek módosítását leíró naplóbejegyzések lemezre írása.
2. A COMMIT naplóbejegyzés lemezre írása.
3. Az adatbáziselemek értékének tényleges cseréje a lemezen.

8.7. példa: Tanulmányozzuk ugyanazt a tranzakciót, amelyiket a 8.3. példában is elemeztünk. A 8.7. ábrán látható ezen tranzakcióra vonatkozó események lehetséges sorrendje.

A főbb különbségek a 8.7. és 8.3. ábrák között a következők: először nézzük a 8.7. ábra 4) és 7) sorait, ezekben a módosítást leíró naplóbejegyzésben az *A* és *B* adatbáziselemek új értéke szerepel (s nem a régi, mint a 8.3. ábrán). A másik különbség, hogy a COMMIT bejegyzés korábbra került, a 8) lépésbe. Ezt követően a napló lemezre írását kiváltó FLUSH LOG következik, s így a *T* tranzakció által végrehajtott módosításokat leíró összes naplóbejegyzés lemezre íródik. Csak ezt követően kerül lemezre az *A* és *B* új, módosított értéke. Az ábrán ezen új értékek kiírását a közvetlenül következő 10) és 11) sorokban láthatjuk, bár a gyakorlatban ezekre esetleg csak később kerül sor. □

Lépés	*Tevékenység*	*t*	*M*-A	*M*-B	*D*-A	*D*-B	*Napló*
1)							<START *T*>
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)							<COMMIT *T*>
9)	FLUSH LOG						
10)	OUTPUT(A)	16	16	16	16	8	
11)	OUTPUT(B)	16	16	16	16	16	

8.7. ábra. *Tevékenységek és naplóbejegyzéseik helyrehozó naplózás használatakor*

8.3.2. Helyreállítás a helyrehozó naplózás használatával

A helyrehozó naplózás R_1 szabályának fontos következménye, hogy ha a naplóban nincs <COMMIT *T*> bejegyzés, akkor tudjuk, hogy a *T* tranzakció nem hajtott végre az adatbázisban módosítást a lemezen. Így a be nem fejezett (nem teljes) tranzakciók a helyreállítás során úgy tekinthetők, mintha meg sem történtek volna. Problémát a befejezett (COMMIT) tranzakciók jelenthetnek, mert nem tudjuk, hogy az általuk elvégzett adatbázis-változtatások közül melyik íródott már lemezre. Szerencsére a helyrehozó naplózás naplója éppen azon információkat – az új értékeket – tartalmazza, melyekre szükségünk van a helyreállításhoz. Ezen új értékeket kell lemezre írnunk, attól függetlenül, hogy esetleg már korábban is kiírodtak. A rendszerkatasztrófa bekövetkezése után a helyrehozó naplózással történő helyreállításhoz a következőket kell tennünk:

1. Meghatározni a befejezett (COMMIT) tranzakciókat.
2. Elemezni a naplót az elejétől kezdve. Minden <*T*,*X*,*v*> naplóbejegyzés megtalálásakor:

   a) Ha *T* nem befejezett tranzakció, akkor nem kell tenni semmit.
   b) Ha *T* befejezett tranzakció, akkor *v* értéket kell az *X* adatbáziselembe írni.

3. Minden *T* be nem fejezett tranzakcióra vonatkozóan <ABORT *T*> naplóbejegyzést kell a naplóba írni, és a naplót ki kell írni lemezre (FLUSH LOG).

8.8. példa: Tegyük fel, hogy a napló a 8.7. ábrának megfelelő, nézzük meg hogyan lehet a helyreállítást elvégezni a különböző pillanatokban bekövetkező katasztrófák esetében.

1. Ha a katasztrófa a 9) lépés után bármikor következik be, akkor a <COMMIT T> 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. Megjegyezzük, hogy ha a katasztrófa a 10) és 11) lépések között következett be, akkor A újraírása redundáns ugyan, de B írása (korában nem történt meg) lényeges lépés az adatbázis konzisztens állapotának eléréséhez. Amennyiben a hiba a 11) lépést követően keletkezett, akkor mindkét adatbáziselem új értékének lemezre írása redundáns ugyan, de semmi gondot nem okoz.
2. Ha a hiba a 8) és 9) lépések között jelentkezik, akkor bár a <COMMIT T> bejegyzés már a naplóba került, de nem biztos, hogy lemezre íródott (ez attól függ, hogy esetleg valami más okból sor került-e a napló lemezre írására). Ha lemezre került, akkor a helyreállítási eljárás az 1) esetnek megfelelően történik. Ha pedig a napló még nem került lemezre, akkor a helyreállítás a következő, 3) esettel megegyező.
3. Ha a katasztrófa a 8) lépést megelőzően keletkezik, akkor <COMMIT T> 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, nincs mit helyreállítani, s végül egy <ABORT T> bejegyzést írunk a naplóba.

□

8.3.3. Helyrehozó naplózás ellenőrzőpont-képzés használatával

A semmisségi naplózásnál látottakhoz hasonlóan a helyrehozó naplózás naplójába is illeszthetünk ellenőrzőpontokat. A helyrehozó naplózásnál azonban új probléma jelentkezik: Minthogy a befejeződött tranzakciók módosításainak lemezre írása a befejeződés után sokkal később is történhet, így az e vonatkozásban ugyanazon pillanatban aktív tranzakciók számát nem tudjuk korlátozni, azon pillanatban sem, amikor az ellenőrzőpont létrehozásáról döntünk. Tekintet nélkül arra, hogy az ellenőrzőpont-képzés alatt tranzakciók indulását megengedjük vagy sem, a kulcsfeladat – amit meg kell tennünk az ellenőrzőpont-készítés kezdete és befejezése közötti időben – azon összes adatbáziselem lemezre való kiírása, melyeket befejezett tranzakciók módosítottak, és még nem voltak lemezre kiírva. Ennek megvalósításához a pufferkezelőnek nyilván kell tartania a *piszkos* puffereket, melyekben már végrehajtott, de lemezre még ki nem írt módosításokat tárol. Azt is tudnunk kell, mely tranzakciók mely puffereket módosították.

Más oldalról 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:

A helyrehozó naplózás eseményeinek sorrendje

Mivel sok befejezett tranzakció is adhatott új értéket ugyanazon X adatbázis-elemnek, ezért a helyrehozó naplózás alkalmazásakor a naplót a korábbi bejegyzésektől a későbbiek felé időrendben haladva kell elemeznünk. Így érhető el, hogy X adatbázisbeli végső értéke – ahogy kell – a normálisan befejeződött tranzakciók által utoljára adott legyen. Ugyanazt az állapotot érjük el tehát, mint ami a semmisségi naplózásnál a napló visszafelé elemzésével volt elérhető.

Ha az adatbázisrendszerünk az atomosságot követeli meg, akkor a semmisségi naplózásnál nem tudtuk pontosan megállapítani, két be nem fejeződött tranzakció esetében, hogy azok módosították-e ugyanazon adatbáziselemet. Ezzel szemben a helyrehozó naplózás alkalmazásával a befejeződött tranzakciókra figyelünk, ha szükséges, ezek módosításait megismételve állítjuk helyre az adatbázis konzisztens állapotát. Ez teljesen rendben van, két rendben befejezett (COMMIT) tranzakció esetében akkor is, ha mindkettő ugyanazon adatbáziselemet módosította különböző pillanatokban. A helyreállítás helyes sorrendje itt mindig fontos, nem úgy, mint a semmisségi naplózás esetében volt (amennyiben a konkurenciafelügyelet megfelelő formája működött).

1. <START CKPT (T_1, ..., T_k)> naplóbejegyzés elkészítése és kiírása lemezre, ahol T_1, ..., T_k az összes éppen aktív (még be nem fejezett) 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 (FLUSH LOG).

<START T_1>
<T_1,A,5>
<START T_2>
<COMMIT T_1>
<T_2,B,10>
<START CKPT (T_2)>
<T_2,C,15>
<START T_3>
<T_3,D,20>
<END CKPT>
<COMMIT T_2>
<COMMIT T_3>

8.8. ábra. *A helyrehozó naplózás naplója*

8.9. példa: A 8.8. ábra egy lehetséges naplót mutat, melynek közepén ellenőrzőpont található. Amikor az ellenőrzőpont-képzés elkezdődött, csak T_2 volt aktív, de a T_1 által A-ba írott érték még csak esetleg került lemezre. Ha még nem, akkor A-t lemezre

kell másolnunk, mielőtt az ellenőrzőpont-képzést befejezhetnénk. A napló érzékelteti, hogy az ellenőrzőpont-képzés befejezéséig más események is bekövetkezhetnek: T_2 a C adatbáziselem tartalmát módosítja, elindul T_3 új tranzakció, és módosítja D értékét. Az ellenőrzőpont-képzés befejezése után már csak T_2 és T_3 tranzakciók befejeződése történt meg. □

8.3.4. Visszaállítás az ellenőrzőponttal kiegészített helyrehozó típusú naplózással

Mint a semmisségi naplózásnál, most is, az ellenőrzőpontok naplóba illesztése segít a naplóátvizsgálás korlátozásában, amikor adatbázis-helyreállítás szükséges. Szintén a semmisségi naplózáshoz hasonlóan két eset fordulhat elő, attól függően, hogy az utolsó ellenőrzőpont-bejegyzés a START vagy az END.

- Tegyük fel először, hogy a katasztrófa előtt a naplóba feljegyzett utolsó ellenőrzőpont-bejegyzés <END CKPT>. Ekkor tudjuk, hogy az olyan értékek, melyeket olyan tranzakciók írtak, melyek a <START CKPT (T_1, ..., T_k)> naplóbejegyzés megtétele előtt befejeződtek, már biztosan lemezre kerültek, s így nem kell velük foglalkoznunk helyreállítandó ezen tranzakciók hatását. Foglalkoznunk kell viszont a T_i-k közé tartozó, valamint az ellenőrzőpont kialakításának megkezdése után induló tranzakciókkal, ezeknek lehetnek olyan adatbázis-módosításaik, melyek még nem kerültek lemezre, pedig a tranzakció már befejeződött. Ekkor olyan visszaállítást kell tennünk, amilyenről a 8.3.2. részben már szó volt, azzal a különbséggel, hogy figyelmünket azon tranzakciókra korlátozhatjuk, melyek az utolsó <START CKPT (T_1, ..., T_k)> naplóbejegyzésben a T_i-k között szerepelnek, vagy ezen naplóbejegyzést követően indultak el. A naplóban való keresés során a legkorábbi <START T_i> naplóbejegyzésig kell visszamennünk, annál korábbra már nem. Megjegyezzük, hogy ezek a START naplóbejegyzések akárhány korábbi ellenőrzőpontnál korábban is felbukkanhatnak. Ahogy a semmisségi naplózásnál is láttuk, az adott tranzakcióra vonatkozó naplóbejegyzések visszafelé keresése segít megtalálni a számunkra éppen fontos bejegyzéseket.
- Tegyük fel most, hogy a naplóba feljegyzett utolsó ellenőrzőpont-bejegyzés a <START CKPT (T_1,..., T_k)> naplóbejegyzés. Nem lehetünk abban biztosak, hogy az ezt megelőzően befejezett tranzakciók által módosított adatbáziselemek már lemezre íródtak. Ezért az előző <END CKPT> bejegyzéshez tartozó <START CKPT (S_1, ..., S_m)> naplóbejegyzésig[12] vissza kell keresnünk, és helyre kell állítanunk az olyan befejeződött tranzakciók tevékenységének eredményeit, melyek ez utóbbi <START CKPT (S_1, ..., S_m)> naplóbejegyzés után indultak, vagy az S_i-k közül valók.

8.10. példa: Tekintsük ismét a 8.8. ábrán bemutatott naplót. Ha a katasztrófa a végén lép fel, akkor az <END CKPT> bejegyzésig kell visszakeresnünk. Ekkor tudjuk, hogy a helyreállítás szempontjából elegendő csak azon tranzakciókat figyelembe venni, melyek egyrészt a <START CKPT (T_2)> bejegyzés felírását követően indultak, vagy szerepelnek e bejegyzés listájában (most csak T_2). Így a vizsgálandó tranzakcióhalmazunk {T_2, T_3}. <COMMIT T_2> és <COMMIT T_3> bejegyzéseket találunk, s ebből tudjuk, hogy mindkettő tranzakció hatását helyre kell állítanunk. A naplóban visszafelé meg kell keresnünk a <START T_2> bejegyzést, s innen már időrendben haladva a naplóban a következő – T_2, T_3 befejezett tranzakciókra vonatkozó – módosítást leíró bejegyzéseket találjuk: <T_2,B,10>, <T_2,C,15> és <T_3,D,20>. Mivel azt nem tudjuk, hogy ezen változtatások a lemezen már megtörténtek-e, ezért most a lemezre újraírjuk a B, C és D tartalmát, megfelelően 10, 15 és 20 értékeket adva nekik.

Tegyük fel most, hogy a katasztrófa a <COMMIT T_2> és <COMMIT T_3> bejegyzések között történt. A helyreállítás az előbbi esethez hasonló, azzal a különbséggel, hogy T_3 nem befejezett tranzakció, ennek megfelelően a <T_3,D,20> helyreállítást *nem* kell végrehajtani. D értékét a helyreállítás során nem változtatjuk meg, hacsak a vizsgált naplórészben található, más tranzakció bejegyzése miatt meg nem kell változtatnunk. A helyreállítást követően egy <ABORT T_3> bejegyzést írunk a naplóba.

Végül, ha a hiba az <END CKPT> bejegyzést megelőzően lépett fel, akkor az utolsó előtti START CKPT bejegyzést kell megkeresnünk (melynek már van <END CKPT> párja), és annak listájából tudjuk meg, melyek az aktív tranzakciók. Ha nem találunk korábbi ellenőrzőpont-bejegyzést, akkor mindenképpen a napló elejére kell mennünk. Így esetünkben az egyedüli befejezett tranzakciónak T_1-et fogjuk találni, s ezért a <T_1,A,5> tevékenységét helyreállítjuk. A helyreállítást követően <ABORT T_2> és <ABORT T_3> bejegyzéseket írunk a naplóba. □

Minthogy a tranzakciók több ellenőrzőpont készítésekor is aktívak lehetnek, célszerű lehet, hogy a <START CKPT (T_1, ..., T_k)> naplóbejegyzésbe nem csak az aktív tranzakciók neveit, hanem olyan mutatókat is elhelyezzünk, melyek az aktív tranzakciók indulását leíró bejegyzések naplóbeli helyét adják meg. Így eljárva, biztonsággal meg tudjuk állapítani, hogy a napló mely korábbi részeit törölhetjük. Amikor <END CKPT> bejegyzést írunk a naplóba, akkor tudjuk, hogy a naplóban már sosem kell korábbra visszatekintenünk, mint ahol a T_i aktív tranzakcióra vonatkozó, legkorábbi <START T_i> bejegyzést találjuk. Következésképpen az ezen START bejegyzést megelőző bejegyzések mindegyike törölhető.

[12] Előzetes hiba miatt előfordulhat, hogy a START CKPT bejegyzésnek nincs <END CKPT> párja. Ezért kell úgy eljárnunk, hogy nem csak a korábbi START CKPT bejegyzést keressük, hanem előbb egy <END CKPT>-t, majd az ezt megelőző START CKPT-t.

8.3.5. Feladatok

8.3.1. feladat: Adjuk meg a 8.1.1. feladatban szereplő tranzakciók (nevezzük mindet T-nek) helyreállítási típusú naplóbejegyzéseit. Tegyük fel, hogy kezdetben $A = 5$ és $B = 10$.

8.3.2. feladat: Ismételjük meg a 8.2.2. feladatot, helyreállítási típusú naplózást használva.

8.3.3. feladat: Ismételjük meg a 8.2.4. feladatot, helyreállítási típusú naplózást használva.

8.3.4. feladat: Ismételjük meg a 8.2.5. feladatot, helyreállítási típusú naplózást használva.

8.3.5. feladat: A 8.2.7. feladat adatait használva az a)–e) helyzetek mindegyikére válaszoljuk meg az alábbi kérdéseket:

i) Mely pontokban fordulhat elő az <END CKPT> felírása, és
ii) Minden lehetséges hibabekövetkezési ponthoz adjuk meg, hogy a naplóban meddig kell visszatekintenünk ahhoz, hogy megtaláljuk az összes befejezetlen tranzakciót. Vegyük figyelembe mindkét lehetőséget, azt is, hogy a hibát megelőzően az <END CKPT> felíródott a naplóba és azt is, ha nem.

8.4. A semmisségi/helyrehozó (undo/redo) naplózás

Láthattuk, hogy a naplózás két különböző megközelítése abban mutat eltérést, hogy a napló az adatbáziselemek értékének módosítása esetén a régi (módosítás előtti) vagy az új (módosítás utáni) érteket tartalmazza. Mindkét módszernek vannak bizonyos hátrányai is:

- A semmisségi (undo) naplózás alkalmazása megköveteli, hogy az adatokat a tranzakció befejezésekor nyomban lemezre írjuk, ezzel (esetleg jelentősen) növeljük a végrehajtandó lemezműveletek számát.
- Másik oldalról, a helyrehozó (redo) naplózás minden módosított adatbázisblokk pufferben tartását igényli, egészen a tranzakció rendes és teljes befejezéséig (commit), a napló kezelésével együtt (esetleg jelentősen) növeli a tranzakciók átlagos pufferigényét.
- Mindkét naplózási módszer az ellenőrzőpont képzése közben ellentétes igényeket támaszt a pufferek lemezre írása szempontjából, kivéve, ha az adatbáziselemek teljes blokkok vagy blokkok sokasága. Például, ha a puffer tartalmaz egy *A* adatbáziselemet, melyet egy rendesen és teljesen befejezett tranzakció módosított, és tartalmaz egy *B* adatbáziselemet is, melyet olyan tranzakció módosított, melyre vonatkozóan a COMMIT bejegyzés még nem került lemezre, akkor az R_1 szabálynak megfelelően, a puffer lemezre másolását igényeljük *A* miatt, viszont tiltjuk ennek megtételét *B* miatt.

Most a *semmisségi/helyrehozó* (undo/redo)-nak nevezett naplózást vizsgáljuk meg. Ez a módszer a tevékenységek elvégzési sorrendjének rugalmasságát növeli azáltal, hogy bővíti a naplózott információk körét.

8.4.1. A semmisségi/helyrehozó (undo/redo) naplózás szabályai

A semmisségi/helyrehozó naplózás, egyetlen különbséggel, ugyanolyan típusú naplóbejegyzéseket használ, mint a naplózás többi módszere. E módszerben az adatbáziselem értékének módosítását leíró naplóbejegyzés négykomponensű. A <*T*,*X*,*v*,*w*> naplóbejegyzés azt jelenti, hogy a *T* tranzakció az adatbázis *X* elemének korábbi *v* értékét *w*-re módosította. A semmisségi/helyrehozó naplózást alkalmazó rendszer a következő előírást kell hogy betartsa:

UR_1: Mielőtt az adatbázis bármely *X* elemének értékét – valamely *T* tranzakció által végzett módosítás miatt – a lemezen módosítanánk, ezt megelőzően a <*T*,*X*,*v*,*w*> módosítást leíró naplóbejegyzésnek lemezre kell kerülnie.

A semmisségi/helyrehozó naplózás, UR_1 szabálya csak azokat a feltételeket kényszeríti, amelyek a semmisségi és a helyrehozó naplózási szabályok mindegyikében szerepelnek. Speciálisan, a <COMMIT *T*> bejegyzés megelőzheti és követheti is az adatbáziselemek lemezen történő bármilyen megváltoztatását.

8.11. példa: A 8.9. ábra, az utoljára a 8.7. példában látott, *T* tranzakcióhoz tartozó naplóbejegyzések sorrendjének egy változatát mutatja. Megjegyezzük, hogy a módosítást leíró naplóbejegyzések már az *A* és *B* adatbáziselemeknek mind a régi, mind az új értékét tartalmazzák. Ebben a sorozatban a <COMMIT *T*> naplóbejegyzés kiírását az *A* és *B* adatbáziselemek lemezre való írása közé tettük. A 10) lépés kerülhetett volna a 9) lépés elé vagy a 11) lépés mögé is. □

Lépés	*Tevékenység*	*t*	*M*-A	*M*-B	*D*-A	*D*-B	*Napló*
1)							<START *T*>
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)							<COMMIT *T*>
11)	OUTPUT(B)	16	16	16	16	16	

8.9. ábra. *Tevékenységek és naplóbejegyzéseik lehetséges sorrendje semmisségi/helyrehozó naplózás használatakor*

8.4.2. Helyreállítás a semmisségi/helyrehozó (undo/redo) naplózás használatakor

Amikor a semmisségi/helyrehozó naplózást használjuk, és helyreállításra kényszerülünk, akkor a módosítást leíró naplóbejegyzésben megtaláljuk mind a *T* tranzakció hatásainak semmissé tételéhez szükséges régi, mind a *T* tranzakció hatásainak helyreállításához szükséges új adatbáziselem-értékeket. A semmisségi/helyrehozó módszer alapelvei:

A késleltetett véglegesítés problémája

A semmisségi naplózáshoz hasonlóan a semmisségi/helyrehozó naplózás is olyan viselkedést mutat, hogy a tranzakció a felhasználó számára korrekten befejezettnek tűnik (például: az ügyfél számítógép-hálózaton vásárolt egy repülőjegyet, majd levált a hálózatról), s még a <COMMIT *T*> 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 szabály használatát javasoljuk:

UR_2 A <COMMIT *T*> naplóbejegyzést nyomban lemezre kell írni, amint megjelenik a naplóban.

Ennek teljesítéséért a 8.9. példánkban a 10) lépés után egy FLUSH LOG lépést kell beiktatnunk.

1. A legkorábbitól kezdve állítsuk helyre minden befejezett tranzakció hatásait.
2. A legutolsótól kezdve tegyük semmissé minden be nem fejezett tranzakció cselekedeteit.

Megjegyezzük, hogy mindkét eljárásra szükségünk van. A rugalmasság lehetővé teszi, hogy a COMMIT bejegyzés és a lemezen végrehajtott adatbázis-módosítások egymáshoz viszonyított sorrendje kötetlen legyen, így előfordulhat az is, 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.

8.12. példa: Tegyük fel, hogy az események a 8.9. ábrán látható sorrendben történtek. A hiba fellépésének időpontja függvényében különböző helyreállítási lehetőségeink vannak.

1. Feltéve, hogy a katasztrófa a <COMMIT *T*> naplóbejegyzés lemezre írását követően fordul elő, ekkor *T*-t befejezett tranzakciónak tekintjük. 16-ot írunk mind az *A*, mind *B* adatbáziselemekbe. Az események jelenlegi sorrendjében *A*-nak már 16 a tartalma, de *B*-nek lehet, hogy nem, aszerint, hogy a hiba a 11) lépés előtt vagy után következett be.
2. Ha a katasztrófa a <COMMIT *T*> naplóbejegyzés lemezre írását megelőzően következett be, akkor *T* befejezetlen tranzakciónak számít. Ez esetben az *A* és *B* adatbáziselemek korábbi értéke, 8 íródik lemezre. Ha a hiba a 9) és 10) lépések között következett be, akkor *A* értéke már 16 volt a lemezen, és emiatt a 8-ra való visszaállítás feltétlenül szükséges. Ebben a konkrét példában a *B* értéke nem igényelne visszaállítást (mert még meg sem változott), ha pedig a hiba a 9) lépés előtt következik be, akkor *A* sem igényelné a visszaállítást. Mivel általában nem lehetünk biztosak abban, vajon a visszaállítás szükséges-e vagy sem, így (a biztonság kedvéért) mindig végre kell hajtanunk a visszaállítást.
□

8.4.3. Semmisségi/helyrehozó naplózás ellenőrzőpont-képzéssel

A működés közbeni ellenőrzőpont-képzés valamivel egyszerűbb a semmisségi/helyrehozó naplózás alkalmazásakor, mint más naplózási módszereknél volt. Csak a következőket kell tennünk:

1. Írjunk a naplóba <START CKPT (T_1, ..., T_k)> naplóbejegyzést, ahol T_1, ..., T_k-k az összes éppen 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, nemcsak a már befejezett tranzakciók által módosítottakat.
3. Írjunk <END CKPT> naplóbejegyzést a naplóba, majd írjuk a naplót lemezre.

A 2) ponttal kapcsolatban megjegyezzük, hogy a semmisségi/helyrehozó naplózás által, a lemezre írások sorrendjére vonatkozóan biztosított rugalmasság miatt, megengedhetjük a be nem fejezett tranzakciók adatainak lemezre való kiírását. Így megen-

A tranzakciók különös viselkedése a helyreállítás alatt

A figyelmes olvasó észrevehette, hogy nem adtuk meg azt, hogy a semmisségi/helyrehozó (undo/redo) naplózás alkalmazásakor a helyreállítás során a semmisségi (undo) vagy a helyrehozó (redo) lépést tesszük meg előbb. Valóban, azt, hogy a semmisségi vagy a helyrehozó lépéseket tesszük-e meg előbb, nyitva hagytuk a következő szituáció miatt: előfordulhat, hogy a *T* tranzakció rendben és teljesen befejeződött, s emiatt helyreállítása során az általa kialakított *X* értéket rekonstruáljuk, melyet viszont egy be nem fejezett, és ezért visszaállítandó *U* tranzakció korábban módosított. A probléma nem az, hogy először helyreállítjuk *X* értékét, és aztán visszaállítjuk *U* előttire, vagy pedig előbb visszaállítjuk, és utána a *T* által írottra rekonstruáljuk. E szituációban egyik út sem helyes, mert a végső adatbázis-állapot nem felel meg egyik – atomosnak elvárt – tranzakció hatásának sem.

A gyakorlatban az adatbázisrendszereknek a módosítások naplózásánál többet kell tenniük. Biztosítaniuk kell, hogy ilyen szituációk ne fordulhassanak elő. A konkurencia kérdéseivel foglalkozó fejezetben vizsgáljuk azt is, mit jelent a *T* és *U* tranzakciók elkülönítése, amivel az ugyanazon *X* adatbáziselemen való kölcsönhatásuk előfordulása elkerülhető. A 10.1. részben kifejezetten az olyan helyzetek megelőzésével foglalkozunk, amikor a *T* tranzakció egy piszkos – más tranzakció által módosított, de még nem véglegesített – *X* adatbáziselemet használna.

gedhetjük a teljes blokknál kisebb adatbáziselemek használatát is, melyek közös pufferbe kerülnek. A tranzakciókra vonatkozóan egyetlen előírást kell tennünk:

- A tranzakció semmilyen értéket nem írhat (még a memóriapufferbe sem), amíg biztosak nem vagyunk abban, hogy nem abortál.

Amint a 10.1. részben látni fogjuk, ezt a megszorítást szinte mindig be kell tartani ahhoz, hogy elkerülhessük a tranzakciók közötti inkonzisztens kölcsönhatást. Megjegyezzük, hogy a helyrehozó naplózás használatakor a fenti feltétel nem elégséges, éppen ezért írja elő az R_1 szabály, hogy ha egy tranzakció B-t módosítja, akkor a tranzakcióra vonatkozó COMMIT naplóbejegyzésnek előbb kell lemezre íródnia, s csak azután írhatjuk B-t lemezre.

8.13. példa: A 8.10. ábra a semmisségi/helyrehozó naplózás alkalmazását mutatja egy, a 8.8. ábrán (helyrehozó naplózás) látottal megegyező esetre. Csak a módosításokat leíró naplóbejegyzéseket cseréltük, megadva bennük a régi és új értékeket. Az egyszerűség kedvéért feltételeztük, hogy a régi érték mindig eggyel kisebb az új értéknél.

Amint a 8.9. példában is, az ellenőrzőpont képzésének kezdetekor T_2 az egyetlen aktív tranzakció. Minthogy ez a napló semmisségi/helyrehozó napló, így lehetséges, hogy T_2 által B-nek adott új érték, 10, lemezre íródik, ami nem volt megengedett 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 T_1 tranzakció alakított ki – is lemezre fogjuk írni, ha még nem került oda.

Ha a katasztrófa ezen eseménysorozat végén jelentkezik, akkor a T_2-t és T_3-at teljesen és rendesen befejezett (COMMIT) tranzakciónak tekintjük. T_1 tranzakció az ellenőrzőpontnál korábbi. Minthogy <END CKPT> bejegyzést találunk a naplóban, így T_1-ről biztosan tudjuk, hogy teljesen és rendesen befejeződött, valamint az általa okozott módosítások lemezre íródtak. Ezért, mint a 8.9. példában is, a T_2 és T_3 által végzett módosítások helyreállítandók, T_1 pedig figyelmen kívül hagyható. Amikor olyan tranzakció hatásait állítjuk helyre, mint amilyen a T_2 is, akkor a naplóban nem kell a

<START T_1>
<T_1,A,4,5>
<START T_2>
<COMMIT T_1>
<T_2,B,9,10>
<START CKPT (T_2)>
<T_2,C,14,15>
<START T_3>
<T_3,D,19,20>
<END CKPT>
<COMMIT T_2>
<COMMIT T_3>

8.10. ábra. *A semmisségi/helyrehozó (undo/redo) naplózás naplója*

<START CKPT (T_2)> bejegyzésnél korábbra visszatekinteni, mert tudjuk, hogy az ellenőrzőpont-képzést megelőzően T_2 által végzett módosítások az ellenőrzőpont képzése alatt lemezre íródtak.

Másik példaként, tegyük fel, hogy a katasztrófa éppen <COMMIT T_3> bejegyzés lemezre írását megelőzően fordult elő. Ekkor T_2-t befejezett, T_3-at pedig befejezetlen tranzakciónak kell tekintenünk. T_2 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 eltérő módon pedig a T_3 hatásait semmissé tesszük, azaz a lemezen D tartalmát 19-re írjuk. Ha T_3 az ellenőrzőpont-képzés kezdetekor már aktív tranzakció lett volna, akkor a naplóban a megelőző START CKPT bejegyzésig kellene visszakeresnünk, azért hogy megtaláljuk T_3 semmissé teendő tevékenységeit (az azokat leíró naplóbejegyzéseket). □

8.4.4. Feladatok

8.4.1. feladat: Adjuk meg a 8.1.1. feladatban szereplő tranzakciók (nevezzük mindet T-nek), semmisségi/helyrehozó (undo/redo) típusú naplóbejegyzéseit. Tegyük fel, hogy kezdetben $A = 5$ és $B = 10$.

8.4.2. feladat: Az alábbi naplóbejegyzés-sorozatok valamely T tranzakció tevékenységeit tükrözik. Állapítsuk meg a semmisségi/helyrehozó (undo/redo) naplózás szabályainak megfelelően a naplóbejegyzések és az adatbáziselemeket tartalmazó blokkok lemezre írási lehetőségeit, figyelembe véve, hogy naplóbejegyzést nem lehet addig a lemezre írni, amíg a megelőző bejegyzés nem került lemezre.

* a) <START T>; <T,A,10,11>; <T,B,20,21>; <COMMIT T>;
b) <START T>; <T,A,10,21>; <T,B,20,21>; <T,C,30,31>; <COMMIT T>;

8.4.3. feladat: A következő semmisségi/helyrehozó naplóbejegyzés-sorozat a T és U két tranzakcióra vonatkozik: <START T>; <T,A,10,11>; <START U>; <U,B,20,21>; <T,C,30,31>; <U,D,40,41>; <COMMIT U>; <T,E,50,51>; <COMMIT T>. Adjuk meg a helyreállítás-kezelő tevékenységeit, beleértve a lemezen és a naplóban tett módosításait, ha katasztrófa lépett fel, és az utolsó lemezre került naplóbejegyzés:

a) <START U>.
* b) <COMMIT U>.
c) <T,E,50,51>.
d) <COMMIT T>.

8.4.4. feladat: A 8.4.3 feladatban leírt helyzetek mindegyikére adjuk meg, hogy a T és U által lemezre írott értékek közül melyeknek *kell* megjelenni a lemezen, és melyek jelen*het*nek meg a lemezen?

8.4.5. feladat: Tegyük fel, hogy a napló a következő bejegyzéssorozatot tartalmazza: <START *S*>; <*S*,*A*,60,61>; <COMMIT *S*>; <START *T*>; <*T*,*A*,61,62>; <START *U*>; <*U*,*B*,20,21>; <*T*,*C*,30,31>; <START *V*>; <*U*,*D*,40,41>; <*V*,*F*,70,71>; <COMMIT *U*>; <*T*,*E*,50,51>; <COMMIT *T*>; <*V*,*B*,21,22>; <COMMIT *V*>. Tegyük fel továbbá, hogy a működés közbeni ellenőrzőpont-képzést kezdjük alkalmazni, közvetlenül az alábbi bejegyzések (memóriában való) megjelenésétől kezdve:

a) <*S*,*A*,60,61>.
* b) <*T*,*A*,61,62>.
c) <*U*,*B*,20,21>.
d) <*U*,*D*,40,41>.
e) <*T*,*E*,50,51>.

Mindegyik fenti esetre adjuk meg, hogy:

i) Mikor írható fel az <END CKPT> naplóbejegyzés, és
ii) Bármelyik lehetséges pillanatban, ha hiba lép fel, meddig kell a naplóban visszafelé tekinteni, ahhoz, hogy minden, be nem fejezett tranzakciókra vonatkozó bejegyzést megtaláljunk. Fontoljuk meg mindkét lehetőséget is, hogy a hiba az <END CKPT> naplóbejegyzés felírása előtt vagy az után jelentkezik.

8.5. Az eszközök meghibásodása elleni védekezés

A naplózással a rendszerhibák ellen védekezhetünk. Gondoskodhatunk arról, hogy rendszerhiba következtében legfeljebb csak a memóriában tárolt ideiglenes adatok vesznek el, de a lemezről semmi nem veszhet el. Ugyanakkor, amint a 8.1.1. részben már foglalkoztunk vele, sok komoly hibát okoz egy vagy több lemez elvesztése (tönkremenetele). Az adatbázist a naplóból elméletileg akkor tudjuk rekonstruálni, ha:

a) A naplót tároló lemez különbözik az adatokat (adatbázist) tartalmazó lemez(ek)től.
b) A naplót sosem dobjuk el az ellenőrzőpont-képzést követően, és
c) A napló helyrehozó (redo) vagy semmisségi/helyrehozó (undo/redo) típusú, s így az új értékeket tárolja.

Ugyanakkor, amint már említettük, a napló esetleg az adatbázisnál is gyorsabban növekedhet, s így nem praktikus a naplót örökre megőrizni.

8.5.1. Az archívmentés

Az eszközök meghibásodása elleni védekezés egyik megoldása az *archiválás* – az adatbázis másolatának elkészítése egy (több), az adatbázisétól különböző adathordozón. Ha lehetséges, lezárjuk az adatbázist addig, amíg elkészítjük a biztonsági másolatot (backup) valamely tárolóeszközön (például optikai lemezen vagy mágnesszalagon), majd a biztonsági másolatot az adatbázistól távol, biztonságos helyen tároljuk. A biztonsági másolat megőrzi az adatbázis mentéskori állapotát, s ha eszközhiba lép fel, akkor a mentésből az adatbázis ezen (mentéskori) állapotát vissza tudjuk állítani.

Miért nem csak a naplót mentjük?

Felmerülhet bennünk a kérdés, milyen gyakran kell elkészíteni a biztonsági mentést, hiszen a napló használatával – ha nem akadunk el – egy régi mentésből is helyreállíthatnánk az adatbázist. A válasz nem nyilvánvaló, az adatbázis méretén és tipikus módosítási fokán múlik. Amíg az adatbázisnak naponta esetleg csak kis része változik, addig a naplózandó módosítások tömege egy egész év folyamán sokkal nagyobb lehet, mint maga az adatbázis. Ha soha nem archiválunk, akkor a napló soha nem csonkolható, és a napló tárolási/kezelési költsége hamar túllépheti az adatbázis másolatának tárolási költségét.

A napló használatával sokkal frissebb állapotot tudunk rekonstruálni. Ha a biztonsági másolat készítéséről keletkező naplót megőrizzük, és az túlélte az eszköz meghibásodását, akkor a hiba után (esetleg másik lemezen) visszaállítva a biztonsági másolatból, a napló felhasználásával a mentés óta történt adatbázis-változásokat is át tudjuk vezetni az adatbázison. A napló keletkezése közben, amilyen gyorsan csak lehet, távoli másolatot készítünk róla. Ezzel a napló elvesztése ellen védekezhetünk. Így, ha a napló, az adatokkal együtt elveszik is, akkor még mindig használhatjuk az adatbázis mentését és a napló távoli másolatát az adatbázis visszaállításra egészen addig a pillanatig, amikor a napló utolsó átvitele történt a távoli másolatára.

Ha az adatbázis nagy, akkor a biztonsági mentés elkészítése (kiírása) hosszas folyamat, általánosan bevált, hogy nem mentik a teljes adatbázist minden archiváló alkalommal. Ezért a mentésnek két szintjét különböztetjük meg:

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

Lehetséges a mentésnek több szintjét is használni, a teljes mentést „0-dik szintűnek" tekintve, az „i-edik szintű" mentésen pedig azt értve, mely mentés az előző „i-edik szintű", vagy alacsonyabb szintű mentések óta megváltozott elemek másolatát tartalmazza.

Az adatbázist a teljes mentésből és a megfelelő növekményes mentésekből (a helyreállító vagy a semmisségi/helyreállító naplók rendszerhiba utáni visszaállítási folyamatához hasonló) módszerrel tudjuk rekonstruálni. Visszamásoljuk a teljes mentést, majd az ezt követő, legkorábbi növekményes mentéstől kezdve végrehajtjuk a növekményes mentésekben tárolt változtatásokat. A növekményes mentések az adatok-

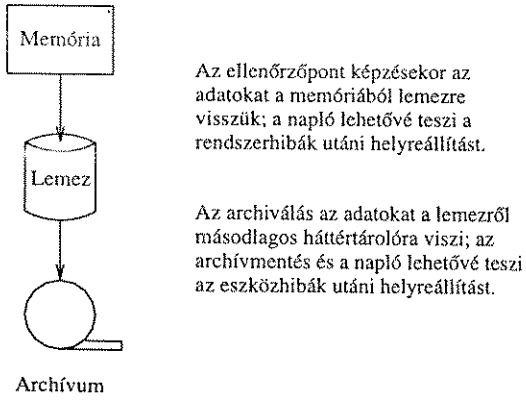
nak (a teljes adatbázishoz viszonyítva) csak azt a kis részét érintik, amely az utolsó mentés óta változott meg, s így ezek kevesebb helyet igényelnek és gyorsabban menthetők, mint a teljes mentés hely- és időigénye.

8.5.2. Archiválás működés közben

A 8.5.1. részben bemutatott, egyszerűnek látszó archiválással az a probléma, hogy sok adatbázist nem lehet lezárni arra az időre (lehet, hogy órákra), amíg a biztonsági mentését elkészítjük. Így, a működés közbeni ellenőrzőpont-képzéshez hasonlóan, meg kell oldanunk a *működés közbeni archiválást* is. Visszagondolunk arra, ahogy a működés közbeni ellenőrzőpont-képzés megkísérli az indulásakori adatbázis-állapot (megközelítő) másolatát létrehozni a lemezen. Az ellenőrzőpont létrehozásának környékén keletkezett kis naplórészletre támaszkodva az adatbázis állapotában történt minden olyan eltérést rendbe tudunk hozni, melyet az okozott, hogy az ellenőrzőpont képzése alatt új tranzakciók indulhattak és lemezírások történhettek.

Ehhez hasonlóan a működés közbeni archiválás megbízhatóan tud az adatbázisról olyan másolatot készíteni, ami az archiválás kezdetének megfelelő adatbázis-állapotot rögzíti, ugyanakkor a mentés alatti percekben vagy órákban az adatbázis működése sok adatbáziselemet cserélhet. Ha az adatbázis mentésből való visszaállítása szükséges, akkor a mentés alatt keletkezett naplóbejegyzések felhasználásával az adatbázis konzisztens állapota állítható elő. A hasonlóságot a 8.11. ábrán mutatjuk be.

A működés közbeni archiválás az adatbázis elemeit valamely fix sorrendben másolja, mialatt megeshet, hogy ezen elemeket az éppen végrehajtott tranzakciók módosítják. Ennek eredményeként megtörténhet, hogy a biztonsági mentésbe másolt adatbáziselem értéke nem ugyanaz, mint a mentés megkezdésekor volt. Amíg a mentés alatt keletkezett naplót megőrizzük, addig az eltérések a napló felhasználásával korrigálhatók.



8.11. ábra. *Az ellenőrzőpont-képzés és az archiválás közötti hasonlóság*

8.14. példa: Nagyon egyszerű példaként, tegyük fel, hogy adatbázisunk 4 elemből, *A*, *B*, *C* és *D*-ből áll. Ezek értéke a biztonsági mentés (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, az események sorrendje pedig legyen a 8.12. ábrának megfelelő. Ekkor noha az adatbázis tartalma a mentés kezdetekor 1, 2, 3, 4 volt, a mentés végére 5, 7, 6, 4 lett, a mentett archívumba 1, 2, 6, 4 került, jóllehet ilyen adatbázis-állapot a mentés ideje alatt nem is fordult elő. □

Lemez	*Mentés*
	A másolása az archívumba
A := 5	
	B másolása az archívumba
C := 6	
	C másolása az archívumba
B := 7	
	D másolása az archívumba

8.12. ábra. *Események a működés közbeni archiválás alatt*

Részletesebben a biztonsági mentés (archívum) elkészítése a következő lépésekből áll. Feltételezzük, hogy az alkalmazott naplózási módszer a helyrehozó (redo) vagy a semmisségi/helyrehozó (undo/redo) módszerek valamelyike; a semmisségi (undo) naplózás nem alkalmas a működés közbeni archiválással való használatra.

1. A <START DUMP> bejegyzés naplóba írása.
2. Az alkalmazott naplózási módnak megfelelő ellenőrzőpont kialakítása.
3. A kívánt adatlemez(ek) teljes vagy növekményes mentésének végrehajtása, arra ügyelve, hogy az adatok másolata (a mentés) biztonságos távoli helyre kerüljön.
4. Gondoskodjunk arról is, hogy a napló szükséges részéről is másolat készüljön, és az is biztonságos, távoli helyre kerüljön. 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.

A mentés befejezésekor biztonsággal eldobhatjuk a napló 2. pontban végrehajtott ellenőrzőpont-képzést *megelőzően keletkezett* részét.

8.15. példa: Tegyük fel, hogy a 8.14. példabeli egyszerű adatbázis mentés közbeni módosításait két tranzakció, T_1 (mely *A*-t és *B*-t módosította) és T_2 (amely *C*-t módosította) végezte, melyek a mentés kezdetekor aktívak voltak. A 8.13 ábrán látjuk a mentés alatti események lehetséges naplóbejegyzéseit, semmisségi/helyrehozó (undo/redo) naplózási módszert alkalmazva.

Megjegyezzük, hogy nem tüntettük fel T_1 befejezését. Az eléggé valószínűtlen, hogy egy tranzakció a teljes mentés egész ideje alatt aktív maradjon, de ez a lehetőség nem befolyásolja a következőként bemutatandó helyreállítási módszer helyességét. □

<START DUMP>
<START CKPT (T_1, T_2)>
<T_1,A,1,5>
<T_2,C,3,6>
<COMMIT T_2>
<T_1,B,2,7>
<END CKPT>
mentés befejezése
<END DUMP>

8.13. ábra. *A mentés közben keletkező napló*

8.5.3. Helyreállítás az archívmentés és a napló használatával

Tegyük fel, hogy készülékhiba lépett fel, s az adatbázist rekonstruálnunk kell. A helyreállítást a legutolsó biztonsági mentés (archívmentés), és a napló – katasztrófa során el nem veszett – távoli mentése felhasználásával végezzük. A következő lépéseket hajtjuk végre:

1. Az adatbázis visszaállítása a biztonsági (archív) mentésből.

   a) Meg kell keresni a legutolsó teljes mentést, belőle rekonstruálni az adatbázist. (Azaz a mentést az adatbázisba másoljuk.)
   b) Ha van(nak) későbbi növekményes mentés(ek), akkor ezeket időrendi sorrendben használva, módosítjuk az adatbázist.

2. Módosítjuk az adatbázist a napló katasztrófát túlélt részével. (Természetesen a naplózási módszernek megfelelő helyreállítási eljárást kell alkalmaznunk.)

8.16. példa: Tegyük fel, hogy a 8.15. példában szerelő biztonsági mentés elkészítését követően történik eszközmeghibásodás, és a 8.13 ábrán látott napló ezt túlélte. Azért, hogy az eljárást érdekesebbé tegyük – a 8.13. ábrának megfelelően – tekintsük úgy, hogy a napló katasztrófát túlélt részében nincs <COMMIT T_1> bejegyzés, jóllehet <COMMIT T_2> bejegyzés van. Az adatbázist először a biztonsági mentésből visszatöltjük, s így A, B, C, D elemei rendre az 1, 2, 6, 4 értékeket kapják.

Ezután a naplót vesszük elő. Minthogy T_2 befejezett tranzakció, helyreállítjuk (redo) azon lépés hatását, mely C értékét 6-ra módosította. Példánkban C értéke már 6, de előfordulhatna, hogy:

a) C mentése azt megelőzően történt, hogy C értékét T_2 tranzakció módosította volna, vagy
b) A mentésben C-nek később kapott értéke van, mely értéket olyan tranzakció állított be, melyre vonatkozó COMMIT bejegyzést a napló – katasztrófát túlélt részében – vagy találunk, vagy nem. C értékét a mentésben talált értékre akkor állítjuk, *ha* az ezt beállító tranzakció COMMIT bejegyzését megtaláljuk.

Minthogy T_1 gyaníthatóan nem befejezett tranzakció (mert COMMIT bejegyzését nem találjuk), így T_1 hatásait semmissé kell tennünk (undo). A T_1-re vonatkozó naplóbejegyzések használatával meg tudjuk állapítani, hogy A értékét 1-re, B értékét 2-re kell visszaállítanunk. Előfordulhat persze, hogy a mentésen ez az értékük, de a pillanatnyi mentésben ettől eltérő értékeik is lehetnek, ha A és/vagy B módosított értéke archiválódott. (Ez a módosításnak és a mentésnek az időbeli sorrendjétől függ.) □

8.5.4. Feladatok

8.5.1. feladat: Ha semmisségi/helyrehozó (undo/redo) naplózás helyett a helyrehozó (undo) naplózást használjuk a 8.15. és 8.16. példákban, akkor:

a) Hogyan fog kinézni a napló?
*! b) Ha a mentést és a naplót használjuk a helyreállításhoz, mi lesz annak következménye, hogy T_1 nem befejezett?
c) Mi lesz az adatbázis állapota a helyreállítás után?

8.6. Összefoglalás

- *Tranzakciókezelés:* A tranzakciókezelő két tipikus feladata: naplózással biztosítani az adatbázisban végrehajtott tevékenységek hatásainak helyreállíthatóságát, és az ütemezőn keresztül biztosítani tranzakciók korrekt párhuzamos működését. (Utóbbit e fejezetben nem tárgyaltuk.)
- *Adatbáziselemek:* Az adatbázist elemekre osztottaknak tekintjük. Az adatbáziselemek tipikusan lemezblokkok, de lehetnek sorok, osztályok extentjei, vagy más egységek. Az adatbáziselemek a naplózás és ütemezés egységei.
- *Naplózás:* A naplóban a tranzakciók összes fontosabb ténykedéseit – a működés megkezdése, adatbáziselemek módosítása, normális vagy abnormális befejeződés – leíró naplóbejegyzéseket tároljuk. A naplót – az általa leírt adatbázis-módosítások lemezre mentése környékén lemezre kell menteni. A napló lemezre mentésének pontos ideje az alkalmazott naplózási módszer függvénye.
- *Helyreállítás:* Rendszerhiba fellépésekor, a naplót használva, az adatbázis konzisztens állapota helyreállítható.
- *Naplózási módszerek:* A naplózás három tipikus módszere a semmisségi (undo), a helyreállító (redo) és a semmisségi/helyreállító (undo/redo), nevüket az alkalmazott helyreállítási eljárásnak megfelelően kapták.
- *Semmisségi (undo) naplózás:* Ez a naplózási módszer az adatbáziselemek értékének megváltoztatásakor csak a régi értéket tárolja a naplóbejegyzésekben. A semmisségi naplózás alkalmazásakor az adatbáziselem új értéke csak azt követően íródik lemezre, miután a változást leíró naplóbejegyzés már lemezre került, ugyanakkor az adatbáziselem lemezre írásának meg kell előznie a tranzakció normál befe-

jezését leíró COMMIT naplóbejegyzés lemezre írását. A helyreállítás a befejezetlen tranzakciók által módosított adatbáziselemek régi értékének visszaállításával történik, azaz a be nem fejezett tranzakciók esetleges hatásainak semmissé tételével.

- *Helyreállító (redo) naplózás:* Ebben a naplózási módszerben a módosított adatbáziselemeknek csak az új értékét tároljuk a módosítást leíró naplóbejegyzésekben. A naplózás eme módszerében az adatbáziselemek értéke csak azt követően íródik lemezre, miután a módosítást végző tranzakció összes változtatást leíró, valamint a véglegesítést jelentő naplóbejegyzése már lemezre került. A helyreállítás a befejezett tranzakciók által módosított adatbáziselemek új értékének újra (adatbázisba) írásával történik.
- *Semmisségi/helyreállító (undo/redo) naplózás:* Ezzel a naplózási módszerrel mind a régi, mind az új értékek naplózódnak. A semmisségi/helyreállító naplózás a többinél sokkal rugalmasabb módszer abban, hogy csak annyit követel meg, hogy a változást leíró naplóbejegyzés a tényleges adatbázisbeli módosítást megelőzően kerüljön lemezre. Arra vonatkozóan nincs kikötése, hogy a tranzakció befejezését leíró bejegyzés mikor kerül lemezre. A helyreállítás a befejezett tranzakciókra a „helyreállító", a be nem fejezett tranzakciókra a „semmisségi" módszer szerint történik.
- *Ellenőrzőpont-képzés:* Az összes helyreállítási módszer elvileg a teljes napló visszamenőleges elemzését igényli. Az adatbázisrendszerek a naplóban alkalmankénti ellenőrzőpont-képzéssel biztosítani tudják, hogy a helyreállítás során az ellenőrzőpontnál korábban felírt naplóbejegyzésekre már ne legyen szükség. Így a régi naplórész törölhető, s a lemez területe újra felhasználható.
- *Működés közbeni ellenőrzőpont-képzés:* Az ellenőrzőpont képzése közben az adatbázisrendszer más működését (esetleg hosszú időre) le kellene állítani. Ennek elkerülésére az összes naplózási módszerhez megalkották a működés közbeni ellenőrzőpont-képzés technikáját. Ez lehetővé teszi, hogy az ellenőrzőpont képzése közben a rendszer működjön, és adatbázis-módosítások is megtörténjenek. Ennek egyetlen pluszköltsége az, hogy a helyreállítás során néhány, az ellenőrzőpont-képzést megelőzően keletkezett, naplóbejegyzést is át kell vizsgálni.
- *Archiválás – biztonsági mentés:* A naplózás csak a memóriatartalom elvesztésével járó rendszerhibák utáni helyreállítást teszi lehetővé. Az archiválással tudunk védekezni az adatbázist hordozó lemez tartalmának sérülése ellen. Az archívum az adatbázis biztonságos helyen tárolt másolata.
- *Növekményes mentés:* A teljes adatbázis rendszeres másolása-mentése helyett, a teljes másolatot követően néhány növekményes mentést készíthetünk. A növekményes mentés során csak az utolsó mentés óta megváltozott adatokat mentjük, ezzel mentési időt és helyet takarítunk meg.
- *Működés közbeni archiválás:* Az a módszer, amikor az adatbázis a biztonsági mentés közben működésben van. E módszer használata megköveteli az archiválás közben a napló használatát és ellenőrzőpont-képzését is.
- *Készülék meghibásodás utáni helyreállítás:* Ha a lemez megy tönkre, akkor az adatbázis konzisztens állapotát egy teljes biztonsági mentés visszatöltése, majd a későbbi növekményes mentések, végül a napló mentett másolatának használatával helyreállíthatjuk.

8.7. Irodalomjegyzék

A legjelentősebb mű, mely a tranzakciók összes vonatkozásaival, közte a naplózással és a helyreállítással is foglalkozik, Gray és Reuter [5] könyve. Ez a könyv a tranzakciókra vonatkozó meggondolásokat részben Jim Gray [3] széles körben használt cikkére alapozza. Később a naplózási és helyreállítási módszerek elsődleges forrásai [4] és [8].

[2] a tranzakciófeldolgozás egy korai, nagyon tömör leírása. [7] a téma egy sokkal újabb feldolgozása.

Két korai áttekintés, [1] és [6] a helyreállítással foglalkozó alapművek, a naplózás azon három alaptípusának rendszerezői, mely felosztást mi is követtünk.

1. P. A. Bernstein, N. Goodman, and V. Hadzilacos, „Recovery algorithms for database systems", *Proc. 1983 IFIP Congress,* North Holland, Amsterdam, pp. 799–807.
2. P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems,* Addison-Wesley, Reading MA, 1987.
3. J. N. Gray, „Notes on database-operating systems", in *Operating Systems: an Advanced Course,* pp. 393–481, Springer-Verlag, 1978.
4. J. N. Gray, P. R. McJones, and M. W. Blasgen, „The recovery manager of the System R database manager", *Computing Surveys* **13**:2 (1981), pp. 223–242.
5. J. N. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques,* Morgan-Kaufmann, San Francisco, 1993.
6. T. Haerder and A. Reuter, „Principles of transaction-oriented database recovery – a taxonomy", *Computing Surveys* **15**:4 (1983), pp. 287–317.
7. V. Kumar and M. Hsu, *Recovery Mechanisms in Database Systems,* Prentice-Hall, Englewood Cliffs NJ, 1998.
8. C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. Schwarz, „ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging", *ACM Trans. On Database Systems* **17**:1 (1992), pp. 94–162.

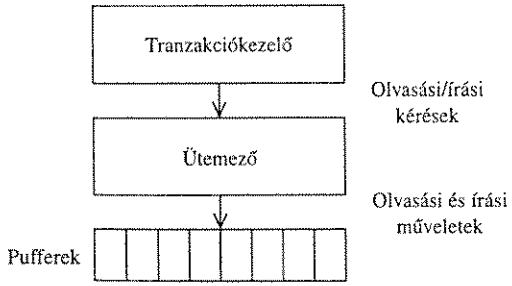
9. fejezet

Konkurenciavezérlés

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

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

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



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

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

9.1. Soros és sorba rendezhető ütemezések

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

9.1.1. Ütemezések

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

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

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

9.2. ábra. *Két tranzakció*

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

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

9.1.2. Soros ütemezések

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

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

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

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

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

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

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

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

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

9.1.3. Sorba rendezhető ütemezések

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

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

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

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

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

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

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

9.6. ábra. *Nem sorba rendezhető ütemezés*

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

9.1.4. A tranzakció szemantikájának hatása

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

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

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

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

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

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

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

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

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

9.1.5. A tranzakciók és ütemezések jelölése

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

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

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

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

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

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

Pontosítva a jelölést:

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

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

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

9.1.6. Feladatok

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

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

Fejezzük ki a T_1 tranzakciót r és w műveletek sorozataként!

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

9.2. Konfliktus-sorbarendezhetőség

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

9.2.1. Konfliktusok

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

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

Másrészt három esetben nem cserélhetjük fel a műveletek sorrendjét:

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

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

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

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

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

9.6. példa: Legyen az ütemezés

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

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

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

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

9.2.2. Megelőzési gráfok és teszt a konfliktus-sorbarendezhetőségre

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

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

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

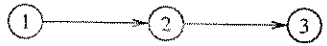
Megjegyezzük, hogy ezek pontosan azok a feltételek, amikor nem lehet felcserélni az A_1 és A_2 sorrendjét. Tehát, A_1 az A_2 előtt szerepel bármely az S-sel konfliktusekvivalens ütemezésben. Ennek eredményeként, ha ezek közül az ütemezések közül az egyik soros ütemezés, akkor ebben T_1-nek meg kell előznie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

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

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

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

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



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

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

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

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

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

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

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

Miért nem szükséges konfliktus-sorbarendezhetőség a sorbarendezhetőséghez?

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

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

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

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

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

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

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

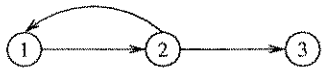
S'': $r_1(B)$; $w_1(B)$; $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$; $r_3(A)$; $w_3(A)$;

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

9.9. példa: Tekintsük az alábbi ütemezést:

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

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



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

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

9.2.3. Miért működik a megelőzési gráfon alapuló tesztelés?

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

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

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

Indukció: Legyen S

$T_1, T_2, \ldots, T_n$

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

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

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

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

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

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

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

9.2.4. Feladatok

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

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

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

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

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

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

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

Válaszoljunk a következő kérdésekre:

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

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

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

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

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

9.2.4. feladat: Tekintsük az alábbi ütemezéseket:

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

Válaszoljunk mindegyik esetében a következő kérdésekre:

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

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

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

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

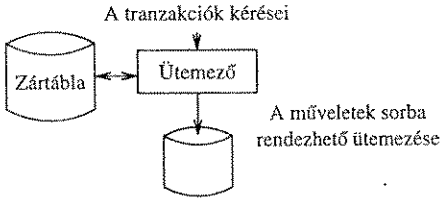
9.3. A sorbarendezhetőség biztosítása zárakkal

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

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

9.3.1. Zárak

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



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

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

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

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

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

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

Kibővítjük a jelöléseinket a zárolás és a feloldás műveletekkel:

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

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

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

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

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

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

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

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

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

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

9.3.2. A zárolási ütemező

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

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

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

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

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

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

9.3.3. A kétfázisú zárolás

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

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

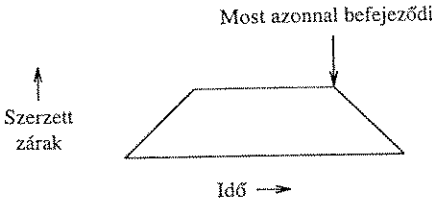
A „kétfázis", amelyre a 2FZ-vel hivatkozunk, abból adódik, hogy az első fázisban csak zárolásokat adunk ki, a második fázisban pedig csak megszüntetünk zárolásokat. A kétfázisú zárolás a konzisztenciához hasonlóan, a tranzakcióban a műveletek sorrendjére egy feltétel. Azokat a tranzakciókat, amelyek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

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

9.3.4. Miért működik a kétfázisú zárolás?

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

9.3.5. Feladatok

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

A holtpont kockázata

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

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

A tranzakciós műveletek egy lehetséges végrehajtása:

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

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

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

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

* a) Adjunk példát a zárolások miatt tiltott ütemezésre!

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

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

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

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

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

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

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

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

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

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

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

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

9.4. Különböző zármódú zárolási rendszerek

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

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

9.4.1. Osztott és kizárólagos zárak

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.4.2. Kompatibilitási mátrixok

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

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

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

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

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

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

9.4.3. Zárak felminősítése

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

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

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

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

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

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

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

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

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

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

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

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

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

9.4.4. Módosítási zárak

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

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

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

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

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

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

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

A 9.18. ábrának megfelelő eseménysorozatot a 9.20. ábrán láthatjuk. Itt a T_2-t el-

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

utasítjuk, amelyik másodikként kérte az A módosítási zárolását. A T_1-nek megengedjük, hogy befejeződjön, és ezután folytatódhat a T_2. A zárolási rendszer hatékonyan megakadályozta a T_1 és T_2 konkurens végrehajtását, ebben a példában viszont lényeges mennyiségű konkurens végrehajtás vagy holtpontot vagy inkonzisztens adatbázis-állapotot eredményez. □

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

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

9.4.5. Növelési zárak

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.4.6. Feladatok

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

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

Végezzük el a következőket:

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

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

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

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

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

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

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

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

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

Válaszoljunk a következőkre:

* a) Ezeknek a tranzakcióknak mennyi átlapolása (ütemezése) sorba rendezhető?

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

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

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

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

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

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

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

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

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

Válaszoljunk az alábbi kérdésekre:

* a) Mely műveletpárok felcserélhetők? Például, ha elforgatjuk a vektort úgy, hogy a hajlásszöge 120° legyen, és azután megváltoztatjuk az x koordinátáját 10-re, ez ugyanaz-e, mintha először változtatnánk meg az x koordinátáját 10-re, és azután változtatnánk a hajlásszöget 120°-ra?

b) Az a) válasz alapján mi a négy zártípus kompatibilitási mátrixa?

!! c) Tegyük fel, hogy megváltoztatjuk a négy műveletet úgy, hogy ahelyett, hogy új értéket adnánk egy mértéknek, inkább növeljük a mértéket (pl. „adjunk hozzá 10-et az x koordinátához", vagy „forgassuk el a vektort 30°-kal az óramutató irányába"). Mi lenne ekkor a kompatibilitási mátrix?

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

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

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

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

9.5. A zárolási ütemező felépítése

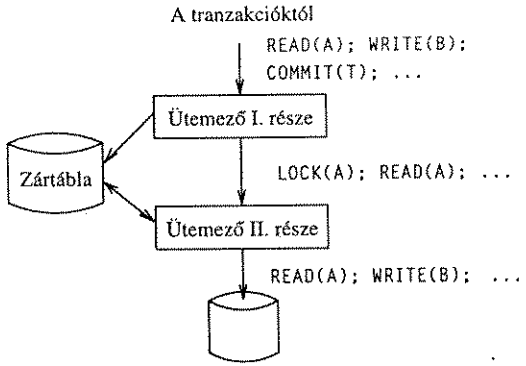
Eddig már számos zárolási sémát láttunk, most a leginkább arra van szükségünk, hogy megnézzük, hogyan működik egy olyan ütemező, amely ezek közül a sémák közül használja valamelyiket. Itt csak néhány megadott elven alapuló egyszerű ütemező felépítését tekintjük:

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

9.5.1. Zárolási műveleteket beszúró ütemező

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

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



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

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

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

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

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

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

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

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

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

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

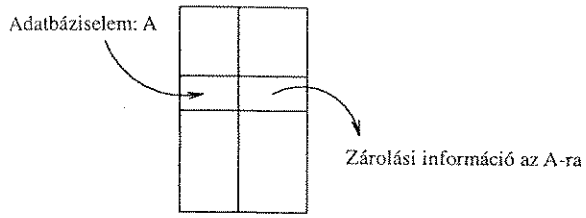
Ezután az $r_1(B)$ művelet érkezik be az ütemezőhöz azzal a figyelmeztetéssel, hogy ezt a zárat fel lehet minősíteni. Az ütemező I. része ekkor kibocsátja $ul_1(B)$; $r_1(B)$-t a II. résznek. Az utóbbi megnézi a zártáblát, és azt találja, hogy a T_1 engedélyezheti a módosítási zárat B-re, ugyanis csak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

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

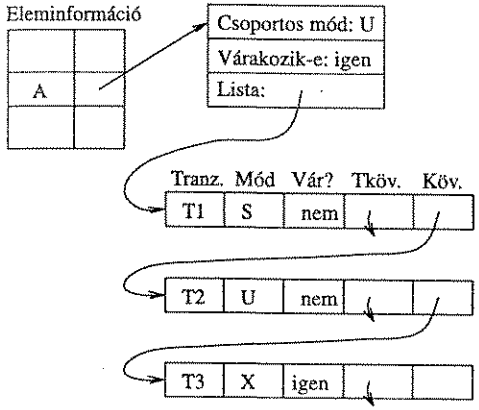
9.5.2. A zártábla

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

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



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



9.26. ábra. *Zártábla-bejegyzések szerkezete*

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

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

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

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

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

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

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

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

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

Zárolási kérések kezelése

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

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

Zárfeloldások kezelése

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

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

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

9.5.3. Feladatok

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

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

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

---

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

9.6. Adatbáziselemekből álló hierarchiák kezelése

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

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

9.6.1. Többszörös szemcsézettségű zárak

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

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

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

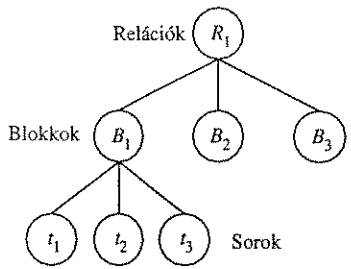
Bizonyos alkalmazások mind a nagy, mind a kis szemcsézettségű zárakat is tudják alkalmazni. Például a 9.20. példában tárgyalt banki adatbázisnál világos, hogy blokk- vagy sorszintű zárolás is szükséges, de néhány esetben a teljes számlareláció zárolása is szükséges lehet, annak érdekében, hogy ellenőrizzük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

9.6.2. A figyelmeztető zárak

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

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

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



9.27. ábra. *Hierarchikusan szervezett adatbáziselemek*

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

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

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

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

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

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

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

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

9.21. példa: Tekintsük a következő relációt

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

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

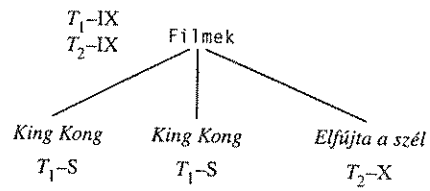
```
SELECT *
FROM Film
WHERE filmCím = 'King Kong';
```

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

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

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

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



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

Csoportos mód a szándékzárolásokhoz

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

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

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

9.6.3. Fantomok és a beszúrások helyes kezelése

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

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

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


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

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

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

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

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

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

9.6.4. Feladatok

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

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

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

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

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

9.7. Faprotokoll

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

9.7.1. Fa alapú zárolások idítékai

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

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

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

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

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

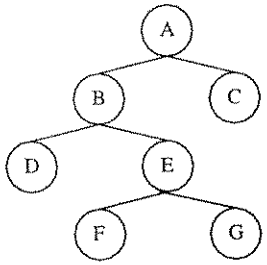
9.7.2. Faszerkezetű adatok hozzáférési szabályai

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

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

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

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



9.30. ábra. *Zárolható elemekből álló fa*

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

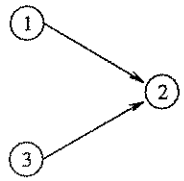
9.31. ábra. *A faprotokollt követő három tranzakció*

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

9.7.3. Miért működik a faprotokoll?

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

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



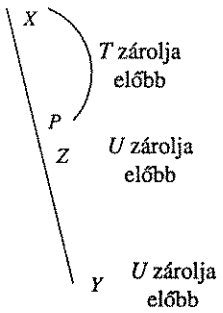
9.32. ábra. *A 9.31. ábra ütemezéséből származó megelőzési gráf*

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

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

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

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



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

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

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

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

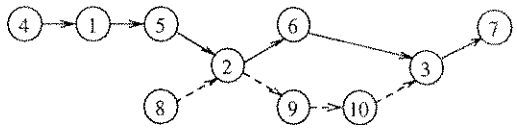
- Ha a T_i előbb zárolja a gyökeret, mint a T_j, akkor a T_i minden T_j-vel közös $T_i <_S T_j$ csomópontot előbb zárol, mint a T_j. Vagyis $T_i <_S T_j$, de nem $T_j <_S T_i$.

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

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

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

Az indukciós feltevés szerint, van soros sorrend az összes olyan tranzakcióhoz, amelyek ugyanabban a tetszőleges részfában zárolnak csomópontokat. Csupán egybe kell olvasztanunk a különböző részfákhoz tartozó soros sorrendeket. Mivel a tranzakcióknak ezekben a listáiban csak azok a tranzakciók közösek, amelyek a gyökeret záró tranzakciók, és megállapítottuk, hogy ezek a tranzakciók minden közös csomópontot ugyanabban a sorrendben zárolnak, ahogy a gyökeret zárolják, nem fordulhat elő két, a gyökeret záró tranzakció különböző sorrendben két részlistán. Pontosabban, ha T_i és T_j előfordul a gyökér valamely C gyermekéhez tartozó listán, akkor ezek a C-t ugyanabban a sorrendben zárolják, ahogyan a gyökeret zárolják, és emiatt a listán is ebben a sorrendben fordulnak elő. Így felépíthetjük a soros sorrendet a teljes tranzakcióhalmazhoz azokból a tranzakciókból kiindulva, amelyek a gyökeret zárolják, a megfelelő sorrendjükben, és beleolvasztjuk azokat a tranzakciókat, amelyek nem zárolják a gyökeret, a részfáik soros sorrendjével konzisztens tetszőleges sorrendben.



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

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

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

9.7.4. Feladatok

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

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

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

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

Válaszoljunk a következőkre:

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

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

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

9.8. Konkurenciavezérlés időbélyegzőkkel

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

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

Mindkét megközelítés *optimista* abban az értelemben, hogy feltételezik: nem fordul elő nem sorba rendezhető viselkedés, és csak akkor tisztázza a helyzetet, amikor a megszegés nyilvánvaló. Ezzel ellentétben, minden zárolási módszer azt feltételezi, hogy a dolgok rosszra fordulnak, hacsak a tranzakciókat azonnal meg nem akadályozzák a nem sorba rendezhető viselkedésbe kerülésben. Az optimista megközelítések abban különböznek a zárolásoktól, hogy az egyetlen ellenszerük, amikor valami rossz-

ra fordul, hogy azt a tranzakciót, amely nem sorba rendezhető viselkedésbe próbált kerülni, abortáljuk (leállítjuk), és aztán újraindítjuk. A zárolási ütemezők ezzel ellentétben késleltetik a tranzakciókat, de nem abortálják őket.[11] Általában az optimista ütemezők akkor jobbak a zárolásinál, amikor sok tranzakció csak olvasási, ugyanis az ilyen tranzakciók önmagukban soha nem okozhatnak nem sorba rendezhető viselkedést.

9.8.1. Időbélyegzők

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

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

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

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

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

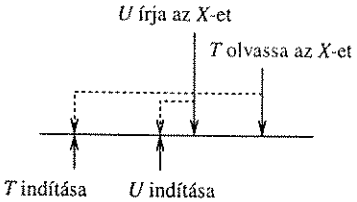
[11] Ez nem azt jelenti, hogy az a rendszer, amely zárolási ütemezőt használ, soha nem abortáltatja a tranzakciót. Például a 10.3. részben tárgyaljuk a holtpontok feloldását szolgáló tranzakcióabortáltatást. Egy zárolási ütemező viszont soha nem használ tranzakcióabortálást egyszerűen mint egy választ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

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

9.8.2. Fizikailag nem megvalósítható viselkedések

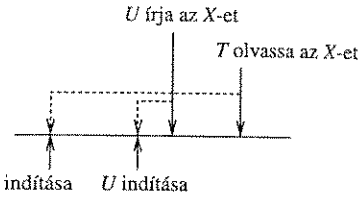
Azért, hogy megértsük az időbélyegzőn alapuló ütemező felépítését és szabályait, emlékeztetnünk kell arra, hogy az ütemező feltételezi, hogy a tranzakciók időbélyegző szerinti sorrendje egyúttal olyan soros sorrend, amely a végrehajtás sorrendjét is jelenti. Így az ütemező feladata azon túl, hogy hozzárendeli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

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



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

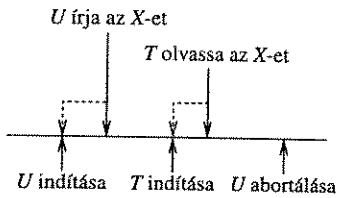
[12] Bár a piaci rendszerek általában a felhasználóra bízzák, hogy megengedhetők-e a piszkos olvasások.



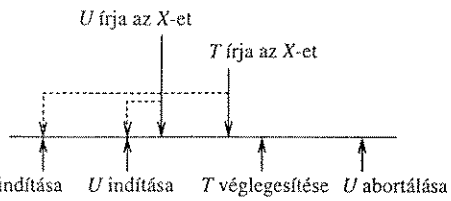
9.36. ábra. *A T tranzakció túl késői írást próbál végezni*

9.8.3. Piszkos adatok problémái

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



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



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

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

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

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

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

9.8.4. Az időbélyegzőn alapuló ütemezések szabályai

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

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

A szabályok a következők:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.8.5. Többváltozatú időbélyegzők

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

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

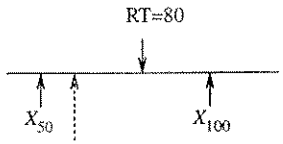
9.27. példa: Tekintsük a 9.40. ábrán szereplő, az A adatbáziselemhez hozzáférő tranzakciók halmazát. Ezek a tranzakciók egy közönséges időbélyegzőn alapuló ütemező alatt működnek, és amikor a T_3 megpróbálja az A-t olvasni, azt találja, hogy a WT(A) nagyobb, mint a saját időbélyegzője, és abortálni kell.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

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

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

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

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



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

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

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

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

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

9.42. ábra. *Többváltozatú konkurenciavezérlést alkalmazó tranzakciók végrehajtása*

9.8.6. Az időbélyegzők és zárolások

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

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

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

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

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

9.8.7. Feladatok

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

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

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

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

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

9.9. Konkurenciavezérlés érvényesítéssel

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

9.9.1. Érvényesítésen alapuló ütemező felépítése

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

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

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

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

1. *KEZD*, a már elindított, de még nem teljesen érvényesített tranzakciók halmaza. Ebben a halmazban minden T tranzakcióhoz az ütemező karbantartja a $\mathtt{KEZD}(T)$-t a T indításának időpontjában.

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

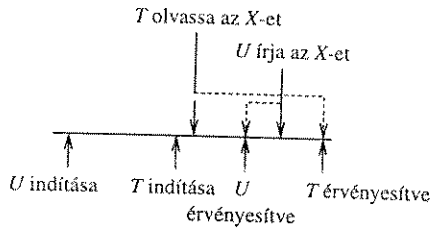
9.9.2. Az érvényesítési szabályok

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

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

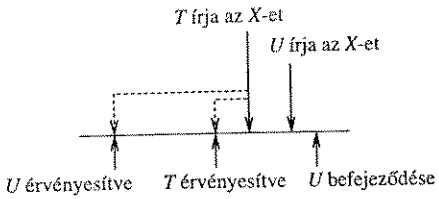
   a) U az *ÉRV*-ben vagy a *BEF*-ben van, vagyis az U-t már érvényesítettük.
   b) $\mathtt{BEF}(U) > \mathtt{KEZD}(T)$, vagyis az U nem fejeződött be a T indítása előtt.[13]
   c) $\mathtt{RS}(T) \cap \mathtt{WS}(U)$ nem üres, legyen X a metszetben levő adatbáziselem.

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



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

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



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

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

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

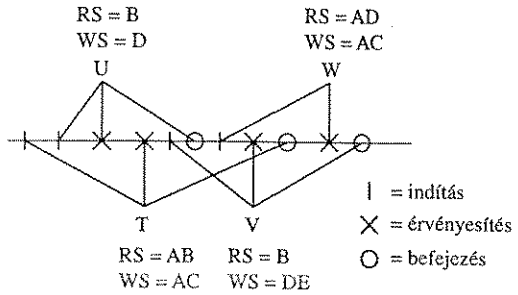
   a) U az *ÉRV*-ben van; vagyis az U-t már sikeresen érvényesítettük.
   b) $\mathtt{BEF}(U) > \mathtt{ÉRV}(T)$; vagyis az U-t nem fejeztük be mielőtt a T az érvényesítési fázisába lépett volna.
   c) $\mathtt{WS}(T) \cap \mathtt{WS}(U) \neq \emptyset$ legyen X mind a két írási halmazban.

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

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

- Összehasonlítjuk az $\mathtt{RS}(T)$-t a $\mathtt{WS}(U)$-val, és ellenőrizzük, hogy $\mathtt{RS}(T) \cap \mathtt{WS}(U) = \emptyset$, minden olyan U-ra, amely még nem fejeződött be a T elindítása előtt, vagyis, ha $\mathtt{BEF}(U) > \mathtt{KEZD}(T)$, és az U egy korábban érvényesített tranzakció.
- Összehasonlítjuk a $\mathtt{WS}(T)$-t a $\mathtt{WS}(U)$-val, és ellenőrizzük, hogy $\mathtt{WS}(T) \cap \mathtt{WS}(U) = \emptyset$, minden olyan U-ra, amely még nem fejeződött be a T-t érvényesítése előtt, vagyis, ha $\mathtt{BEF}(U) > \mathtt{ÉRV}(T)$, és az U egy korábban érvényesített tranzakció.

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



9.45. ábra. *Négy tranzakció és az érvényesítéseik*

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

   - $\mathtt{RS}(V) \cap \mathtt{WS}(T) = \{B\} \cap \{A, C\} = \emptyset$.
   - $\mathtt{WS}(V) \cap \mathtt{WS}(T) = \{D, E\} \cap \{A, C\} = \emptyset$.
   - $\mathtt{RS}(V) \cap \mathtt{WS}(U) = \{B\} \cap \{D\} = \emptyset$.

   Így a V-t sikeresen érvényesítjük.

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

   - $\mathtt{RS}(W) \cap \mathtt{WS}(T) = \{A, D\} \cap \{A, C\} = \{A\}$.
   - $\mathtt{RS}(W) \cap \mathtt{WS}(V) = \{A, D\} \cap \{D, E\} = \{D\}$.
   - $\mathtt{WS}(W) \cap \mathtt{WS}(V) = \{A, C\} \cap \{D, E\} = \emptyset$.

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

□

Csak egy pillanat

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

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

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

9.9.3. Három konkurenciavezérlés működésének összehasonlítása

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

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

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

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

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

9.9.4. Feladatok

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

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

9.10. Összefoglalás

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

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

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

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

9.11. Irodalomjegyzék

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

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

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

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

10. fejezet

Bővebben a tranzakciókezelésről

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

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

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

10.1. Tranzakciók, melyek nem véglegesített adatokat olvasnak

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

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

felejtő központi memória és a tranzakció memóriaterülete között mozognak. A különféle naplózási módszerek segítségével egy esetleges rendszerhiba esetén biztosan helyreállíthatók voltak a véglegesített tranzakciók műveletei és ezek hatása az adatbázislemezen tárolt példányán. A naplózási rendszerek azonban nem tesznek kísérletet a sorbarendezhetőség támogatására; az adatbázis állapotát még akkor is visszaállítják az eredeti formájára, ha az nem sorba rendezhető műveletek eredményeképpen jött létre. Ami azt illeti, a forgalomban lévő adatbázisrendszerek sem mindig ragaszkodnak a sorbarendezhetőséghez, és néhányuk csak a felhasználó kifejezett kérésére tesz erre irányuló törekvéseket.

A 9. fejezetben viszont egyedül a sorbarendezhetőségről volt szó. A fejezet alapelvei szerint tervezett ütemező olyan dolgokat művelhet, amelyeket a naplóvezérlő nem tűrhet el. Például a sorbarendezhetőség definíciójában semmi sem gátolja meg a tranzakciót abban, hogy a véglegesítés előtt a megfelelő zár birtokában A-nak új értéket adjon az adatbázisban, és ezzel megsértse a naplózási politikát. Még rosszabb esetben a tranzakció az adatbázisba írás után abortál, ami könnyen eredményezhet inkonzisztens állapotot még rendszerhiba nélkül is, annak ellenére, hogy az ütemező elméletileg támogatja a sorbarendezhetőséget.

10.1.1. A piszkos adat probléma

Már volt arról szó, hogy az adatot „piszkosnak" nevezzük, ha egy olyan tranzakció írta, amely még nem fejeződött be. Piszkos adat feltűnhet a pufferben, a lemezen vagy mindkét helyen; ezek közül bármelyik problémát okozhat.

10.1. példa: Vegyük újra a 9.13. ábra sorba rendezhető ütemezését, de tegyük fel, hogy T_1 a B olvasása után valamilyen oknál fogva abortál. Ekkor a 10.1. ábrán látható események követik egymást. T_1 abortáltatása után az ütemező feloldja a B-re T_1-nek kiosztott zárat. Ez a lépés elkerülhetetlen, mert különben egyik tranzakció sem zárolhatná B-t.

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

10.1. ábra. *T_1 piszkos adatot ír, majd abortál*

T_2 azonban most olyan adatot olvasott, amely egy inkonzisztens adatbázisállapothoz tartozik. Azaz T_2 a T_1 által megváltoztatott A értéket és a T_1 működése előtti B-t olvasta. Ebben az esetben nem számít, hogy a T_1 által írt $A = 125$ érték kikerült-e a lemezre vagy sem, T_2 úgyis a pufferből kapja az értékeket. Az inkonzisztens állapot olvasása következtében T_2 inkonzisztens állapotban hagyja az adatbázist (a lemezen), ahol $A \neq B$.

A 10.1. ábra ütemezésében az okozza a problémát, hogy a T_1 által írt A piszkos, akár a pufferben van, akár a lemezen. Az a tény, hogy T_2 olvasta és a saját számításaihoz felhasználta A-t, megkérdőjelezi a tranzakció helyes működését. A 10.1.2. részben látni fogjuk, hogy ha ilyen eset előfordulhat, akkor T_2-t és T_1-et is abortáltatni kell, és vissza kell görgetni (rollback). Ha még azt is megengedjük, hogy a T_1 által írt A érték a lemezen is megjelenjen, az elég sokba kerül, hiszen a változtatás meg nem történtté tevéséhez a naplót kell felhasználnunk. Ezért olyan szabályokat kell kidolgoznunk, amelyek segítségével megelőzhető, hogy a piszkos adat kikerüljön a lemezre. □

10.2. példa: Most nézzük a 10.2. ábrán látható eseménysorozatot, amely a 9.8. részben megismert időbélyegző alapú ütemező felügyelete alatt játszódik le. Tegyük fel azonban, hogy ez az ütemező nem használja a 9.8.1. részben bevezetett véglegesítés bitet. Abban a részben láthattuk, hogy ennek a bitnek a segítségével megelőzhető, hogy egy tranzakció olyan adatot olvasson, amelyet egy még folyamatban lévő másik tranzakció írt. Így, amikor a második lépésben T_1 B-t olvassa, nincs véglegesítés bit ellenőrzés, ami késleltethetné T_1-et. T_1 továbbléphet, akár a lemezre is írhat, és véglegessé válhat. B olvasása utáni műveleteit nem tüntettük fel az ábrán.

Végül T_2 fizikailag nem megvalósítható módon próbálja C-t olvasni, ezért abortál. T_2-nek a B-n végrehajtott változtatásait érvénytelenítjük, B értékét és írási idejét visszaállítjuk arra, amit T_2 felülírt. Mindezek ellenére T_1 hozzáférhetett B érvénytelen értékéhez, és ezt bármire felhasználhatja, például új értékeket számolhat A-nak, B-nek és/vagy C-nek, és ezeket kiírhatja a lemezre. Vagyis mivel T_1 B piszkos értékét olvasta, inkonzisztens állapotba hozta az adatbázist. Vegyük észre, hogyha használtuk volna a véglegesítés bitet, akkor a 2. lépésben B olvasását késleltethettük volna egészen addig, amíg T_2 nem abortál, és az általa írt B értéket vissza nem állítjuk az előző (feltehetően véglegesített) értékére. □

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

10.2. ábra. *T_1 a T_2-ből olvas piszkos adatot, ezért T_2-vel együtt őt is abortáltatni kell*

Az elkülönítés szintjei SQL-ben

Az SQL2-szabvány nem teszi fel, hogy a tranzakciók sorba rendezhető módon futnak, ennek megfelelően a forgalomban lévő rendszerek a felhasználóra bízzák a kívánt konkurenciavezérlési szint beállítását. SQL2-ben a legmagasabb „elkülönítési szint" a *sorba rendezhető* (serializable), amely pontosan azt jelenti, amit a szó takar: ezen a szinten a tranzakciónak úgy kell lefutnia, mintha egy pillanat alatt történt volna, és mintha minden más tranzakció teljes egészében vagy előtte, vagy utána ment volna végbe.

Az „olvasásbiztos" elkülönítési szint nem követeli meg a sorbarendezhetőséget, viszont megtiltja a piszkos adat olvasását. Lehetséges azonban, hogy ha ezen a szinten egy T tranzakció kétszer olvassa A-t, akkor két különböző értéket kap, amelyeket két különböző véglegesített tranzakció írt. Ehhez kapcsolódik a „megismételhető olvasás" szintje, amely egy kicsit erősebb feltételt szab: nemcsak hogy ha T olvassa A-t, akkor A nem lehet piszkos, de ha A egy reláció, akkor A minden további olvasásánál csak bővebb részhalmazt kaphatunk; vagyis amíg T aktív, A-nak semmilyen része sem tűnhet el.

A negyedik, „nem olvasásbiztos" szinten semmilyen megszorítás sincs arra vonatkozólag, hogy a tranzakció milyen adatokhoz férhet hozzá. Csak ez a szint engedi meg a piszkos adat olvasását.

10.1.2. Továbbgyűrűző visszagörgetés

A fenti példákból látszik, hogy ha a tranzakciók hozzáférhetnek piszkos adatokhoz, akkor néha végre kell hajtanunk egy *továbbgyűrűző visszagörgetést* (cascading rollback). Ez azt jelenti, hogy amikor T abortál, meg kell határoznunk, hogy milyen tranzakciók olvasták a T által írt adatokat, ezeket abortáltatnunk kell, és rekurzívan az összes olyan tranzakciót is, amelyek az ezek által írt adatokat olvasták. Tehát meg kell találnunk minden olyan U tranzakciót, amely a T által írt piszkos adato(ka)t olvasta, abortáltatnunk kell U-t, majd meg kell találnunk minden olyan V tranzakciót, amely az U által írt piszkos adatot olvasta, abortáltatnunk kell V-t és így tovább. Hogy az abortáltatott tranzakció hatását érvénytelenítsük az adatbázison, használhatjuk a naplót, ha megtalálhatók benne a változók előző értékei (semmisségi vagy semmisségi/helyrehozó naplózás). Ha a piszkos adat hatása még nem érte el a lemezt, akkor az adatok helyreállításához az adatbázis lemezen található változata is megfelelő alapot nyújt. Ezekről a módszerekről részletesebben a következő részben lesz szó.

Mint azt már megjegyeztük, az időbélyegző alapú ütemező a véglegesítés bit használatával megakadályozza az olyan tranzakció továbblépését, amely piszkos adatot olvasott, azaz ebben az esetben nem lesz szükség visszagörgetésre. Az érvényesítés alapú ütemező is elkerüli a továbbgyűrűző visszagörgetést, hiszen az adatbázisba írás (sőt már a pufferbe írás is) csak azután történik meg, miután az ütemező megállapította, hogy a tranzakció végleges lesz.

10.1.3. A visszagörgetés kezelése

Most nézzük meg, hogyan kezelhető a továbbgyűrűző visszagörgetés problémája a zárolás alapú ütemezők esetén. Egyszerűen biztosítható, hogy ne legyen szükség visszagörgetésre:

- *Szigorú zárolás:* A tranzakció egészen addig nem engedheti el az írás zárjait (illetve az olyan típusú zárjait, például növelési zár, amelyek birtokában megváltoztathatja az adat értékét), amíg nem lett végleges vagy nem abortált, és az ennek megfelelő naplóbejegyzés nem került ki a lemezre.

Vegyük észre, hogy a feltétel, amely szerint a tranzakció befejezésére vonatkozó feljegyzés lemezre írása megelőzi a zárakról való lemondást, biztosítja a következőt: ha rendszerhiba következik be, miután a T tranzakció eleresztette a zárjait, és egy másik tranzakció olvasta a T által írt adatokat, akkor ezek az adatok nem lehetnek piszkosak amiatt, hogy a helyreállító eljárás abortáltatja T-t. A tranzakciók olyan ütemezését, amely megfelel a szigorú zárolás szabályának, *helyreállíthatónak* (recoverable) nevezzük.

Világos, hogy egy helyreállítható ütemezésben a tranzakciók nem olvashatnak piszkos adatot, hiszen egy még folyamatban lévő tranzakció által pufferbe írt adat egészen addig zár alatt marad, amíg a tranzakció be nem fejeződik. A pufferben lévő adatok helyreállítása azonban még mindig problémát jelent abban az esetben, ha a tranzakció abortál, hiszen a végrehajtott változtatások hatását érvényteleníteni kell. Hogy ez mennyire bonyolult, az attól is függ, hogy az adatbáziselemek blokk méretűek vagy annál kisebbek. Most mindkét esetet megvizsgáljuk.

Blokkok visszagörgetése

Ha a zárolható adatbáziselemek blokkok, akkor létezik a visszagörgetésnek egy egyszerű módja, amelyhez nem kell a naplót felhasználnunk. Tegyük fel, hogy T tranzakció rendelkezik az A blokk kizárólagos zárjával. A új értékét beírta a pufferbe, majd abortálnia kellett. Mivel A zárolva volt, mióta T felülírta, más tranzakció nem férhetett hozzá az értékéhez. Könnyű visszaállítani A régi értékét, ha a következő szabály szerint járunk el:

- A még folyamatban lévő tranzakciók által írt blokkok a központi memóriában vannak csatolva, ami azt jelenti, hogy ezeket a puffereket nem írhatjuk ki a lemezre.

Ilyenkor T abortálása után a „visszagörgetés" annyiból áll, hogy utasítjuk a pufferkezelőt, hogy ne vegye figyelembe A értékét. Vagyis az A által elfoglalt puffer nem íródik ki sehova, hanem az elérhető „szabad" pufferek közé kerül. Biztosak lehetünk benne, hogy A értéke a lemezen a legutóbbi végleges tranzakció eredménye, amely pontosan az, amit szeretnénk.

Ha a 9.8.5. és 9.8.6. részben megismert többverziós időbélyegzős rendszert használjuk, akkor is könnyű dolgunk van a visszagörgetéssel. Ebben az esetben is fel kell tennünk, hogy a még folyamatban lévő tranzakciók által írt blokkok a memóriában csatolva vannak. Ekkor az A értékeit tartalmazó listából egyszerűen töröljük azt, amely T hatására jött létre. Vegyük észre, hogy mivel T értékváltoztató tranzakció volt, ezért az általa írt A érték egészen addig el volt zárva a többi tranzakció elől, amíg T nem abortált (feltéve, hogy a 9.8.6. rész időbélyegző/zár sémáját használtuk).

Kis adatbáziselemek visszagörgetése

Amikor a zárolható adatbáziselemek nem egész blokkok, csak blokkrészletek (például sorok vagy objektumok), akkor az abortált tranzakciók által megváltoztatott pufferek helyreállítása egyszerű módszerrel nem oldható meg. Az a probléma ugyanis, hogy egy pufferben több adatelem is megtalálható, és ezeket két vagy több tranzakció is írhatja, vagyis ha ezek közül az egyik abortál, a többi által véghezvitt változtatásokat még meg kell őriznünk. Egy abortált tranzakció által írt kis A adatbáziselem régi értékének visszaállítására több lehetőségünk is van.

1. A eredeti értékét a pufferben a lemezen található adatbázis alapján állítjuk vissza.
2. Semmisségi vagy semmisségi/helyrehozó naplózás esetén A előző értékét a napló alapján állítjuk vissza.
3. Készíthetünk minden tranzakcióhoz a központi memóriában egy külön naplót arra az időre, amíg a tranzakció aktív. Ebbe a tranzakciók által végrehajtott változtatásokat jegyezzük fel, tehát A régi értéke is megtalálható a megfelelő „naplóban".

Egyik módszer sem ideális. Az elsőben biztosan a lemezen tárolt információhoz kell fordulnunk. A második esetben, ha a napló megfelelő része még mindig a pufferben van, akkor lehet, hogy nem kell a lemezhez fordulnunk. Rosszabb esetben azonban alaposan át kell vizsgálnunk a lemezen tárolt naplót, hogy az A-ra vonatkozó bejegyzést megtaláljuk. Az utolsó módszerrel nem kell a lemezről olvasnunk, viszont a központi memória „naplók" sok helyet foglalhatnak a memóriában.

10.1.4. Csoportos véglegesítés

Bizonyos körülmények között akkor is elkerülhető a piszkos adat olvasása, ha nem írjuk ki azonnal a lemezre a naplóban található commit rekordokat. Ha a naplóbejegyzéseket abban a sorrendben írjuk ki a lemezre, ahogy a naplóban szerepelnek, akkor amint a napló pufferben lévő részébe kerül egy commit rekord, feloldhatjuk a megfelelő zárakat.

10.3. példa: Tegyük fel, hogy T_1 tranzakció az X írása után véglegessé válik, kiírja a commit rekordját a naplóba, de ez a pufferben marad. Bár T_1 nem végleges abban az

Mikor végleges igazán egy tranzakció?

A csoportos véglegesítés kifinomultságáról eszünkbe juthat, hogy egy lezárult tranzakció több különböző állapotban lehet a műveletei befejezése és a valódi „végleges" állapota között, ahol ez utóbbi azt jelenti, hogy a tranzakció hatása semmilyen körülmények között, még a rendszer összeomlása esetén sem vész el. Ahogy a 8. fejezetben megjegyeztük, lehetséges, hogy egy tranzakció véget ér, és még a `COMMIT` feljegyzés is bekerül a központi memória egy pufferébe, de ha ez nem éri el a lemezt, rendszerhiba esetén mégis elvész a tranzakció hatása. Sőt a 8.5. részben azt láttuk, hogy ha a `COMMIT` rekord már el is érte a lemezt, de mentés még nem készült róla, az adathordozó hibája esetén a tranzakció ugyanúgy semmissé válhat, és elveszthetí a hatását.

Ha a hiba lehetőségét kizárjuk, ezek az állapotok mind ekvivalensek abban az értelemben, hogy minden tranzakció a lezárulását követően biztosan eléri a tartósság szintjét, azaz még az adathordozó hibája esetén sem vész el a hatása. Ha azonban a hibákat és ezek helyreállítását is számításba kell vennünk, fontos felismernünk a fenti állapotok közti különbséget, amelyekre egyébként mindre „véglegesként" hivatkozhatnánk.

értelemben, hogy a commit rekordja túlélne egy rendszerhibát, mégis feloldjuk a zárjait. Ezután T_2 olvassa X-et és „véglegessé válik", de a commit rekord, amely T_1 commit rekordját követi a naplóban, szintén a pufferben marad. Mivel a naplóbejegyzések az eredeti sorrendben kerülnek ki a lemezre, a helyreállítás-kezelő csak akkor foghatja fel T_2-t végleges tranzakciónak (a lemezen talált commit rekord következtében), ha T_1-et is véglegesként kezeli. Így a helyreállítás-kezelő szempontjából három eset különböztethető meg:

1. Sem T_1, sem T_2 commit rekordja nem kerül ki a lemezre. Ekkor a helyreállítás-kezelő mindkettőt abortáltatja, és az a tény, hogy T_2 a le nem zárult T_1 által írt X-et olvasta, lényegtelen.
2. T_1 végleges, T_2 nem. Két okból kifolyólag sincs probléma: T_2 tranzakció X-et egy véglegesített tranzakció után olvasta, vagyis X nem volt piszkos, másrészt T_2-t egyébként is abortáltatjuk az adatbázisra való hatás nélkül.
3. Mindkettő végleges. Ekkor T_2 nem piszkos adatot olvasott.

Ezzel szemben tegyük fel, hogy a T_2 commit rekordját tartalmazó puffer kikerült a lemezre (például, mert a pufferkezelő ezt a puffert valami másra akarta használni), T_1 commit rekordja viszont nem. Ha ezen a ponton a rendszer összeomlik, a helyreállítás-kezelő számára úgy fog tűnni, mintha T_2 véglegesített lett volna, T_1 pedig nem. T_2 hatása végleges marad az adatbázison, annak ellenére, hogy ez az X piszkos adat olvasásán alapult. □

Konkurencia és helyreállíthatóság

Talán feltűnt, hogy a 8. fejezetben megismert három naplózási módszer nem mindig felel meg a helyreállítható ütemezés követelményeinek. Az olyan rendszereket, amelyek mind a konkurencia, mind a helyreállíthatóság elvárásainak megfelelnek, gyakran nevezzük ACID- (atomosság, konzisztencia, elkülönítés, tartósság) rendszereknek (lásd az 1.2.4. rész idevágó részeit). Itt a megfelelő alkalom tehát, hogy megjegyezzük: a naplózás és a zárolás (vagy akármilyen más konkurenciavezérlő módszer) egymás ellen ható mechanizmusok.

Például lehet egy olyan rendszerünk, amely naplózza a tranzakciók műveleteit, hogy biztosítsa az atomosságukat, még akkor is, ha semmi szükségünk a sorbarendezhetőségre vagy a piszkos adatok olvasásának az elkerülésére. Ami ezt illeti, a forgalomban lévő rendszerek gyakran a felhasználóra bízzák, hogy igényli-e a tranzakciók sorbarendezhetőségét, illetve a piszkos olvasás megakadályozását; ahogy ezt a 10.1.1. részben az SQL2 „elkülönítési szintjeit" tárgyaló dobozban is említettük. Megfordítva, a megszokott ABKR-ekben lehetőség van a naplózás kikapcsolására, de ezzel együtt az összes vagy néhány tranzakció futása sorba rendezhető. Ennek az a következménye, hogy egy esetleges rendszerhiba esetén a sorbarendezhetőség és az adatbázis konzisztenciája nincs biztosítva, de a felhasználónak esetleg más módszerei is lehetnek a konzisztencia visszaállítására.

A 10.3. példa alapján levonhatjuk azt a következtetést, hogy a zárakat még azelőtt feloldhatjuk, mielőtt a tranzakció commit rekordja a lemezen megjelenne. Ez az eljárás, amit gyakran *csoportos véglegesítésnek* (group commit) nevezünk, a következőkben foglalható össze:

- Ne oldjuk fel a zárakat, amíg a tranzakció be nem fejeződik, és amíg a commit naplóbejegyzés legalább a pufferben fel nem tűnik.
- A naplóblokkokat abban a sorrendben írjuk ki a lemezre, amelyben keletkeztek.

A csoportos véglegesítés elve éppúgy, mint a 10.1.3. részben tárgyalt „helyreállítható ütemezés" elve, biztosítja, hogy piszkos adat olvasására soha ne kerüljön sor.

10.1.5. Logikai naplózás

A 10.1.3. részben láthattuk, hogy a „piszkos olvasásokat" könnyebb megelőzni, illetve ha az olvasás már megtörtént, akkor könnyebb kijavítani abban az esetben, ha a zárolható egységek blokkok, illetve lapok. Azonban még ekkor is legalább két probléma adódik:

1. Minden naplózási módszer számon tartja az adatbáziselemek régi és/vagy új értékét. Ha egy blokkon belül csak kis változtatás történik, például egy sor egy mezőjét átírjuk, vagy beszúrunk/törlünk egy sort, akkor a naplóba nagy mennyiségű redundáns adat kerül be.
2. Az a követelmény, hogy az ütemezés helyreállítható legyen, azaz a zárak feloldása csak a tranzakció véglegesítése után következzen be, a konkurenciát komolyan gátolhatja. Például emlékezzünk vissza a 9.7.1. részre, ahol éppen a korai zárfeloldás előnyeit tárgyaltuk B-fa-index használata esetén. Ha megköveteljük, hogy a zárakat egészen a tranzakció véglegesítéséig fenn kell tartani, akkor ez az előny nem használható ki, és gyakorlatilag egyszerre csak egy értékváltoztató tranzakciónak engedjük meg, hogy a B-fához hozzáférjen.

Mindkét probléma indokolja a *logikai naplózás* használatát, ahol csak a blokkok változtatásai vannak feltüntetve. A változtatás természetétől függően a bonyolultságnak különböző fokai lehetnek.

1. Az adatbáziselemeknek csak néhány bájtján változtatunk, például egy rögzített hosszúságú mezőt módosítunk. Ebben a helyzetben egy kézenfekvő módszer, ha csak a megváltoztatott bájtokat és azok pozícióját jegyezzük fel. A 10.4. példában bemutatjuk ezt az esetet, és a módosítást leíró naplórekord megfelelő formáját.
2. Az adatbáziselemen végrehajtott változtatás egyszerűen leírható és könnyen visszaállítható, de az adatbáziselem legtöbb/összes bájtját érinti. A 10.5. példában egy gyakori szituáció kerül elő, ahol egy változó hosszú mezőn változtatunk, és a mező rekordjait és még más rekordokat is el kell csúsztatnunk a blokkon belül. A blokk új és régi értéke nagyon különböző lehet, ha nincs a tudatunkban, és nem jelöljük a változás egyszerű okát.
3. Az adatbáziselemen sok bájtot érintő változtatást hajtunk végre, és a további módosítások megakadályozhatják a meg nem történtté tevését. Ez valódi „logikai" naplózás, mert a semmisségi/helyrehozó folyamatot még csak nem is az adatbáziselemeken, hanem az általuk képviselt magasabb szintű „logikai" struktúrán keresztül látjuk. A 10.6. példában B-fák (a lemezblokkoknak mint adatbáziselemeknek megfelelő logikai struktúra) segítségével szemléltetjük a logikai naplózásnak ezt a komplex formáját.

10.4. példa: Tegyük fel, hogy az adatbáziselemek olyan blokkok, amelyek mindegyike tartalmaz néhány sort valamilyen relációból. Egy attribútum módosítását egy ehhez hasonló naplóbejegyzéssel fejezhetjük ki: „t sor a attribútumát v_1-ről v_2-re változtattuk". Ha a blokk egy üres részére beszúrunk egy új sort, azt a következőképpen fejezhetjük ki: „a p eltolási érték pozíciótól kezdve beszúrtunk egy t sort az $(a_1, a_2, \ldots, a_k)$ értékkel". Ha a módosított attribútum vagy a beszúrt sor méretben nem összemérhető a blokk nagyságával, akkor ezek a feljegyzések sokkal kisebb helyet foglalnak el, mint maga a blokk. Ezenfelül a semmissé tevő és helyrehozó műveletek végrehajtását is támogatják.

Vegyük észre, hogy a naplóba feljegyzett mindkét művelet idempotens: ha egy blokkon többször is végrehajtjuk őket, ugyanazt kapjuk, mintha csak egyszer alkal-

maztuk volna a műveletet. Hasonlóan az inverz műveleteik ($t[a]$ visszaállítása v_2-ről v_1-re, illetve t sor törlése) is idempotensek. Vagyis az ilyen típusú bejegyzések a helyreállítás során pontosan ugyanúgy használhatók, mint a 8. fejezetben használt módosítást leíró rekordok. □

10.5. példa: Ismét tegyük fel, hogy az adatbáziselemek sorokat tartalmazó blokkok, de most a sorok valamilyen változó hosszúságú mezőt is tartalmaznak. Ha a 10.4. példában említett módosításhoz hasonlóra kerül sor, akkor lehet, hogy a blokk nagy részét el kell csúsztatnunk ahhoz, hogy a hosszabb mezőnek helyet biztosítsunk, vagy hogy helyet takarítsunk meg, ha a mező megrövidül. Rendkívüli esetben túlcsordulásblokkot kell létrehoznunk az eredeti blokk egy részének tárolására (lásd 3.5. részt), illetve a túlcsordulásblokkot el kell távolítanunk, ha egy rövidebb mező miatt lehetővé válik két blokk egyesítése.

Ha a blokkot a túlcsordulásblokkjaival együtt egy adatbáziselem részeként tekintjük, akkor kézenfekvő a módosított mező régi és/vagy új értékeinek a használata a változtatás meg nem történtté tevéséhez illetve újra elvégzéséhez. A blokk-plusz-túlcsordulásblokk(ok)ra azonban úgy kell gondolnunk, mintha az adott sorokat „logikai" szinten tartalmaznák. Talán nem is leszünk képesek a blokkok bájtjait az eredeti állapotukba visszaállítani egy semmissé tevés vagy helyrehozás után, hiszen olyan módosítások miatt, amelyek más mezők hosszúságát változtatták, a blokkok újraszervezésére is sor kerülhetett. Ha viszont az adatbáziselemekre úgy gondolunk, mint blokkok gyűjteményére, amelyek együtt jelentenek bizonyos sorokat, akkor a helyrehozás, illetve semmissé tevés csakugyan visszaállítja az elem logikai „állapotát". □

Nem mindig lehet azonban a blokkokat a túlcsordulásblokkok mechanizmusán keresztül kiterjeszthető egységként kezelni, mint ahogy ezt a 10.5. példában láttuk. Ezért lehet, hogy a műveletek semmissé tevésére, illetve helyrehozására csak a blokkoknál magasabb szinten van lehetőség. A következő példa a B-fa-indexek fontos esetét tárgyalja, ahol a blokkok kezelése nem teszi lehetővé a túlcsordulásblokkok alkalmazását, és a semmissé tevés, illetve helyrehozás műveletekre úgy kell gondolnunk, mintha a blokkok helyett a B-fa logikai szintjén történnének meg.

10.6. példa: Nézzük meg, hogyan lehetne a B-fán történt változtatásokat a logikai naplózás módszerével nyilvántartani. Az egész csúcspont (blokk) régi és/vagy új értéke helyett csak egy rövid bejegyzést írunk a naplóba, amely a következő módosítások valamelyikét írja le:

1. Kulcs-mutató pár beszúrása vagy törlése a leszármazott (gyerek) csúcspontba vagy csúcspontból.
2. A mutatóhoz rendelt kulcs megváltoztatása.
3. Csúcspontok szétválasztása vagy egyesítése.

A fenti módosítások mindegyike leírható egy rövid naplóbejegyzéssel. Még a szétválasztás műveletéhez is csak azt kell megmondanunk, hogy melyik csúcs osztódik és

melyek az új csúcsok. Hasonlóan az egyesítés műveleténél is csak az érintett csúcspontokat kell megadnunk, hiszen az egyesítés módját a használt B-fa-kezelő algoritmus határozza meg.

A logikai módosító rekordok használatával lehetőségünk nyílik arra, hogy a zárakat korábban feloldjuk, mint ahogy erre egy helyreállítható ütemezés szerint sor kerülhetne. Ennek az az oka, hogy a piszkos B-fa-blokkok olvasása nem jelent problémát a tranzakció részéről, feltéve, hogy az olvasás egyetlen célja a tranzakció számára szükséges adatok helymeghatározása.

Tegyük fel például, hogy T tranzakció az N levelet olvassa, de az U tranzakció, amely N-t utoljára írta, abortál, ezért az N-en történt változtatásokat (például az U által beszúrt sor miatt az N-be beszúrtunk egy új kulcs-mutató párt) érvényteleníteni kell. Ha T is beszúrt egy kulcs-mutató párt N-be, akkor N-et már nem lehet az U működése előtti állapotába visszavinni. U-nak az N-re gyakorolt hatását azonban érvényteleníteni tudjuk: ebben a példában az U által beszúrt kulcs-mutató párt töröl-nénk. Az így kapott N persze nem lesz ugyanaz, mint ami U előtt volt, hiszen tartalmazni fogja a T által beszúrt kulcs-mutató párt. Az adatbázis azonban konzisztens maradt, mivel a B-fa továbbra is csak a végleges tranzakciók hatását tükrözi. A B-fát tehát visszaállítottuk logikai szinten, de fizikai szinten nem. □

10.1.6. Feladatok

* **10.1.1. feladat:** Hogyan lehetne az

$r_1(A)$; $r_1(B)$; $w_1(A)$; $w_1(B)$;

műveletsorozatba a 9.3. részben megismert egyszerű zárolásokat beilleszteni úgy, hogy a T_1 tranzakció alkalmazza:

a) A kétfázisú zárolás és a szigorú zárolás módszerét?
b) A kétfázisú zárolás módszerét, de ne alkalmazzon szigorú zárolást?

Adjuk meg az összes lehetséges megoldást!

10.1.2. feladat: Tegyük fel, hogy a következő ütemezések mindegyike a T tranzakció abortálásával zárul. Mely tranzakciókat kell visszagörgetnünk?

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

10.1.3. feladat: Vegyük a 10.1.2. feladat ütemezéseit, de most tegyük fel, hogy mindhárom tranzakció véglegessé válik, és az utolsó műveletük után rögtön be is írják a

naplóba a commit rekordjukat. A rendszer összeomlása miatt azonban a napló vége már nem kerül ki a lemezre, így az utolsó néhány rekord elvész. Válaszoljunk a következő kérdésekre attól függően, hogy a napló mekkora része élte túl a rendszerhibát!

i) Mely tranzakciók nem tekinthetők véglegesnek?
ii) Létrejön-e piszkos olvasás a helyreállítás közben? Ha igen, mely tranzakciókat kell visszagörgetni?
iii) Milyen további piszkos olvasások jöhettek volna létre, ha a bejegyzések nem a napló végéről, hanem valahonnan a közepéről vesztek volna el?

10.2. Nézet-sorbarendezhetőség

A 9.1.4. részben azt mondtuk, hogy az ütemezők tervezésekor az a valódi célunk, hogy csak sorba rendezhető ütemezések jöhessenek létre. Azt is láttuk, hogy egy ütemezés sorbarendezhetősége a tranzakciók adatokon végzett műveleteitől is függhet. A 9.2. részben pedig azt is megtudtuk, hogy az ütemezők rendszerint a „konfliktus-sorbarendezhetőséget" juttatják érvényre, amely a tranzakciók adatokon végrehajtott módosításaira való tekintet nélkül garantálja a sorbarendezhetőséget.

Azonban a konfliktus-sorbarendezhetőségnél gyengébb feltételek is biztosíthatják a sorbarendezhetőséget. Ebben a részben egy ilyen feltétellel, a „nézet-sorbarendezhetőséggel" foglalkozunk. A nézet-sorbarendezhetőség veszi az összes olyan esetet T és U tranzakció kapcsán, amikor T írja az U által olvasott adatbáziselemet. A kétféle sorbarendezhetőség közötti lényeges különbség akkor jön elő, amikor T tranzakció írja A-t, de azt az értéket nem olvassa más tranzakció (mert később egy másik tranzakció felülírja A-t). Ilyenkor a $w_T(A)$ művelet az ütemezés egyéb olyan pozícióira is kerülhet, ahol éppúgy sohasem olvassák, viszont a konfliktus-sorbarendezhetőség definíciója mellett nem lennének megengedhetők. Ebben a részben pontosan definiáljuk a nézet-sorbarendezhetőség fogalmát, majd megadunk egy eljárást, amely segítségével ellenőrizhető, hogy egy adott ütemezés sorba rendezhető-e ilyen értelemben.

10.2.1. Nézetekvivalencia

Tegyük fel, hogy ugyanazokhoz a tranzakciókhoz van két ütemezés: S_1 és S_2. Az ütemezések legelejére, illetve legvégére vegyünk fel további két hipotetikus tranzakciót, T_0-t, illetve T_f-t. Úgy képzeljük, hogy a többi tranzakció által olvasott adatbáziselemek kezdeti értékét T_0 írta, az általuk írt értékeket pedig T_f olvassa az ütemezés végén. Ekkor minden $r_i(A)$ olvasáshoz megadható az a $w_j(A)$ írásművelet, amely megelőzi $r_i(A)$-t, és a legközelebb esik hozzá.[2] Ilyenkor a T_j tranzakciót az $r_i(A)$ olvasás *forrásának* nevezzük. Vegyük észre, hogy T_j lehet a hipotetikus T_0 kezdeti tranzakció, illetve T_i lehet a T_f tranzakció is.

[2] Bár az eddigiekben nem akadályoztuk meg a tranzakciókat abban, hogy ugyanazt az elemet kétszer is írják, erre általában nincs szükségük. A továbbiakban hasznos feltennünk, hogy a tranzakciók az egyes elemeket csak egyszer írják.

Ha az olvasásműveletek forrása mindkét ütemezésben ugyanaz, akkor azt mondjuk, hogy az S_1 és az S_2 ütemezések nézetekvivalensek. Kétségtelen, hogy a nézetekvivalens ütemezések valóban ekvivalensek: akármilyen adatbázis-állapoton hajtjuk őket végre, azonos lesz a hatásuk. Ha az S ütemezés nézetekvivalens egy soros ütemezéssel, akkor S-t *nézet-sorbarendezhető* ütemezésnek nevezzük.

10.7. példa: Vegyük a következő S ütemezést:

T_1:			$r_1(A)$		$w_1(B)$				
T_2:	$r_2(B)$	$w_2(A)$				$w_2(B)$			
T_3:				$r_3(A)$			$w_3(B)$		

A tranzakciók műveleteit most függőleges irányban tagoltuk, hogy jobban lehessen követni, melyik tranzakció mit csinál; az ütemezés a megszokott módon balról jobbra olvasandó.

S-ben T_1 és T_2 is írja B-t, de ezek az értékek elvesznek (felülíródnak), csak a T_3 által írt B érték éri meg az ütemezés végét, ahol a hipotetikus T_f tranzakció „kiolvassa". S nem konfliktus-sorbarendezhető. Hogy lássuk, miért, először vegyük észre, hogy T_2 azelőtt írja A-t, mielőtt azt T_1 olvasná, azaz egy feltételes konfliktus ekvivalens soros ütemezésben T_2-nek meg kell előznie T_1-et. Az a körülmény viszont, hogy a $w_1(B)$ művelet megelőzi $w_2(B)$-t, azt vonja maga után, hogy a feltételes konfliktus ekvivalens soros ütemezésben T_1-nek kell megelőznie T_2-t. Pedig sem $w_1(B)$-nek, sem $w_2(B)$-nek nincs hosszú távú hatása az adatbázisra. A nézet-sorbarendezhetőség az ilyen típusú, lényegtelen írásműveleteket hagyja figyelmen kívül, amikor megállapítja, hogy melyek azok a tényleges megszorítások, amelyekhez igazodnia kell egy ekvivalens soros ütemezésnek.

Formálisabban, tekintsük S-ben az egyes olvasásműveletek forrásait:

1. $r_2(B)$ forrása T_0, mivel ezelőtt nem írjuk B-t S-ben.
2. $r_1(A)$ forrása T_2, mivel az olvasás előtt T_2 írta legutóbb A-t.
3. Hasonlóan, $r_3(A)$ forrása T_2.
4. A hipotetikus $r_{T_f}(A)$ művelet forrása T_2.
5. A hipotetikus $r_{T_f}(B)$ művelet forrása T_3, amely legutoljára írta B-t.

Természetes, hogy T_0, illetve T_f a valódi tranzakciók előtt, illetve után tűnik fel, akármilyen ütemezést is veszünk. Ha a valódi tranzakciók (T_2, T_1, T_3) sorrendjét vesszük, akkor minden olvasás forrása ugyanaz, mint S-ben. Azaz T_2 olvassa B-t és biztosan T_0 az utolsó „író" tranzakció. T_1 olvassa A-t, de T_2 már írta A-t, vagyis $r_1(A)$ forrása T_2, ahogy S-ben. T_3 is olvassa A-t, de mivel az ezt megelőző T_2 írta A-t, $r_3(A)$ forrása T_2, mint S-ben. Végül a hipotetikus T_f olvassa A-t és B-t, de A, illetve B utolsó írói a (T_2, T_1, T_3) ütemezésben rendre T_2, illetve T_3, szintén, mint S-ben. Megállapíthatjuk, hogy S egy nézet-sorbarendezhető ütemezés, és hogy (T_2, T_1, T_3) egy vele nézetekvivalens ütemezés. □

10.2.2. Poligráfok és nézet-sorbarendezhetőségi teszt

A 9.2.2. részben a konfliktus-sorbarendezhetőség tesztelésére a megelőzési gráfot használtuk. Ez a gráf általánosítható úgy, hogy a nézet-sorbarendezhetőség definíciójában megkövetelt összes megelőzési megszorítást tükrözze. Az ütemezéshez tartozó *poligráfot* definíció szerint a következő alkotóelemekből kell összeállítani:

1. Minden tranzakcióhoz vegyünk egy-egy csomópontot, és még további kettőt a hipotetikus T_0-nak és T_f-nek.
2. Ha $r_i(X)$ művelet forrása T_j, vegyük fel a T_j-ből T_i-be mutató irányított élt.
3. Tegyük fel, hogy az $r_i(X)$ olvasás forrása T_j, de T_k is írja X-t. Nem engedhetjük meg, hogy T_k tranzakció T_j és T_i közé essen, tehát vagy T_j előtt vagy T_i után kell állnia. Ezt a feltételt a T_k-ból T_j-be és a T_i-ből T_k-ba vezető élpár felvételével (szaggatott vonallal rajzolva) adhatjuk meg. Az élpár egyik fele „valódi", de nem foglalkozunk vele, hogy melyik. Amikor majd körmentessé próbáljuk a gráfot tenni, az élpárnak azt a felét tartjuk meg, amelyik ebben jobban segít. Vannak azonban fontos speciális esetek, amikor az élpár csak egyetlen élből áll:

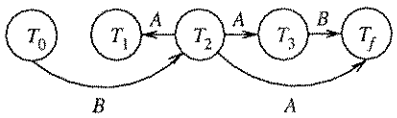
   a) Ha $T_j = T_0$, akkor T_k-nak nincs lehetősége arra, hogy T_j előtt szerepeljen. Ilyenkor az élpár helyett a $T_i \rightarrow T_k$ élt használjuk.
   b) Ha $T_i = T_f$, akkor T_k nem követheti T_i-t. Ilyenkor az élpár helyett a $T_k \rightarrow T_j$ élt használjuk.

10.8. példa: Vegyük a 10.7. példa ütemezését. A 10.3. ábrán látható, hogyan kezdtük el az S-hez tartozó poligráfot felépíteni. A csúcsokon kívül csak a 2. szabály alapján berajzolható élek vannak meg; ezekre azt az adatelemet is ráírtuk, ami miatt bekerültek a gráfba. A-t tehát T_2 továbbadja T_1-nek, T_3-nak és T_f-nek, míg B T_0-tól T_2-höz, illetve T_3-tól T_f-hez kerül.

Most meg kell gondolnunk, hogy milyen tranzakciók léphetnek közbe azáltal, hogy ugyanazt az adatelemet írják, mint ami összeköttetésekben szerepel. Ezeket a lehetséges beavatkozásokat zárjuk ki a 3. szabály élpárjaival, amelyek ebben a példában mind egyetlen élt jelentenek majd valamelyik speciális eset miatt.

Vegyük a $T_2 \rightarrow T_1$ irányított élt. Az A adatelemet csak T_0 és T_2 írja, és ezek közül egyik sem kerülhet T_2 és T_1 közé, mivel T_0 nem változtathatja meg a pozícióját, T_2 pedig már ott szerepel az él egyik végén. Vagyis nem kell új élt berajzolnunk. Hasonló érvelés alapján a $T_2 \rightarrow T_3$ és a $T_2 \rightarrow T_f$ esetén sem kell újabb éleket rajzolnunk ahhoz, hogy az A-t író tranzakciókat kívül tartsuk ezektől az élektől.

Most nézzük a B-hez tartozó éleket. Vegyük észre, hogy T_0, T_1, T_2 és T_3 is írja B-t.

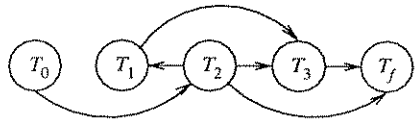


10.3. ábra. *Poligráf kezdemény a 10.8. példához*

Vegyük először a $T_0 \rightarrow T_2$-t. Ahogy az előbb láttuk, T_0-val és T_2-vel most nem kell foglalkoznunk, hiszen az él végpontjai nem okozhatnak problémát; a B adatelemet író tranzakciók közül tehát marad T_1 és T_3. Mivel T_1 nem szerepelhet T_0 és T_2 között, elméletileg be kellene rajzolnunk a $(T_1 \rightarrow T_0, T_2 \rightarrow T_1)$ élpárt. T_0-t azonban semmi sem előzheti meg, vagyis a $T_1 \rightarrow T_0$ lehetőség valójában nem lehetőség. Ebben a speciális esetben elég csak a $T_2 \rightarrow T_1$ irányított él felvétele. De ez az él már megvan A miatt, vagyis gyakorlatilag semmit sem változtatunk a poligráfon annak érdekében, hogy T_1 ne kerülhessen T_0 és T_2 közé.

T_3 sem kerülhet T_0 és T_2 közé. Az előző esethez hasonlóan itt is az élpár helyett csak a $T_2 \rightarrow T_3$ élt kellene felvennünk a gráfba, de ez ugyancsak szerepel már A miatt. Vagyis most sem változtatunk semmin.

Most nézzük a $T_3 \rightarrow T_f$ irányított élt. Mivel T_3-on kívül B-t még T_0, T_1 és T_2 írja, ezeket mind kívül kell tartanunk ezen az élen. T_0 nem eshet T_3 és T_f közé, de T_1 vagy T_2 igen. Mivel semelyikük sem tehető T_f mögé, biztosítanunk kell, hogy T_1 és T_2 mind T_3 előtt jelenjenek meg. A $T_2 \rightarrow T_3$ él már szerepel a gráfban, de a $T_1 \rightarrow T_3$-at hozzá kell adnunk. Ez az egyetlen változtatás, amit végre kell hajtanunk a poligráfon. Az S-hez tartozó végleges poligráf a 10.4. ábrán látható. □



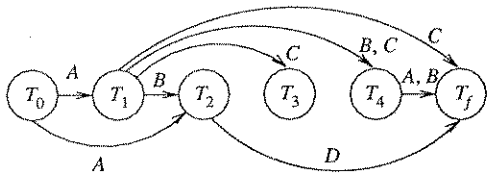
10.4. ábra. *Kész poligráf a 10.8. példához*

10.9. példa: A 10.8. példában minden élpárról kiderült, hogy valamelyik speciális eset miatt tulajdonképpen csak egyetlen élből áll. A 10.5. ábrán egy olyan ütemezést láthatunk, amely poligráfjában egy valódi élpár is szerepel.

T_1	T_2	T_3	T_4
	$r_2(A)$;		
$r_1(A)$; $w_1(C)$;			
		$r_3(C)$;	
$w_1(B)$;			
			$r_4(B)$;
		$w_3(A)$;	
			$r_4(C)$;
	$w_2(D)$; $r_2(B)$;		
			$w_4(A)$; $w_4(B)$;

10.5. ábra. *Az ütemezéshez tartozó poligráf valódi élpárt tartalmaz*

A 10.6. ábrán látható poligráf csak a forrás-olvasó kapcsolatokat mutatja. Mint a 10.3. ábrán, itt is ráírtuk az élekre, hogy melyik adatelem miatt kellett berajzolnunk. Ezek után végig kell mennünk az összes lehetséges eseten, amikor élpárt kéne felven-



10.6. ábra. *Poligráfkezdemény a 10.9. példához*

nünk. Ahogy a 10.8. példában láttuk, többféleképpen is egyszerűsíthető ez az eljárás. A $T_j \to T_i$ él vizsgálata esetén T_k szerepében (amelyek nem lehetnek középen) csak a következő tranzakciókat kell megvizsgálnunk:

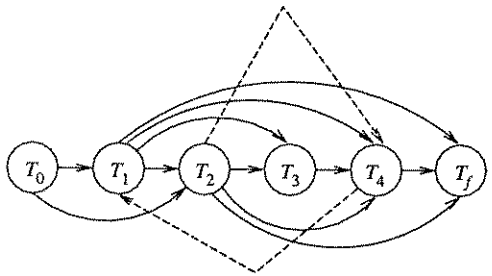
- A $T_j \to T_i$ élhez tartozó adatelemek írói.
- De nem T_0 vagy T_f, amelyek sohasem vehetik fel T_k szerepét, és
- Nem T_i vagy T_j, a kérdéses él végpontjai.

Ezek fényében nézzük végig az A adatbáziselemhez tartozó éleket. Az A-t író tranzakciók: T_0, T_3, T_4. Ezek közül T_0-val egyáltalán nem kell foglalkoznunk. T_3 nem kerülhet T_4 és T_f közé, ezért felvesszük a $T_3 \to T_4$ élt a gráfba; az élpár másik fele: a $T_f \to T_3$ nem jöhet számításba. Hasonlóan T_3 nem kerülhet T_0 és T_1, illetve a T_0 és T_2 közé sem, így a gráf a $T_0 \to T_3$, illetve $T_2 \to T_3$ élekkel bővül tovább.

Most nézzük meg, milyen megszorítások vonatkoznak T_4-re A írása miatt. T_4 a $T_4 \to T_f$ él egyik végpontja, azaz ezzel az éllel nem kell foglalkoznunk. T_4 nem kerülhet T_0 és T_1, illetve T_0 és T_2 közé, tehát fel kell vennünk a gráfba a $T_1 \to T_4$, illetve a $T_2 \to T_4$ éleket.

Következő lépésként vegyük a B-hez tartozó éleket. A B-t író tranzakciók: T_0, T_1, T_4. T_0-t most is figyelmen kívül hagyhatjuk. B miatt a következő éleket kell sorba vennünk: $T_1 \to T_2$, $T_1 \to T_4$, $T_4 \to T_f$. T_1-nek nincsen jelentősége az első két éllel kapcsolatban, viszont a harmadik miatt fel kell vennünk a $T_1 \to T_4$ élt.

T_4-nek csak a $T_1 \to T_2$ élre lehet hatása. Mivel ennek egyik végpontja sem T_0 vagy T_f, egy valódi élpárral bővül a gráf: ($T_4 \to T_1$, $T_2 \to T_4$)-gyel. Az új élekkel kibővített poligráf a 10.7. ábrán látható.



10.7. ábra. *Kész poligráf a 10.9. példához*

A C adatelemet T_0 és T_1 írja. Az előbbiekhez hasonlóan T_0 most sem jelenthet problémát. T_1 viszont minden C-hez tartozó élnek része, tehát vele sem kell foglalkoznunk. A helyzet gyakorlatilag D-vel kapcsolatban is ugyanez, vagyis megállapíthatjuk, hogy nincs szükség további élek felvételére. A végleges poligráf tehát megegyezik azzal, amelyet a 10.7. ábrán láthatunk. □

10.2.3. A nézet-sorbarendezhetőség tesztelése

Mivel minden élpárból csak az egyik élt kell kiválasztanunk, az S ütemezéshez pontosan akkor adható meg vele ekvivalens soros ütemezés, ha az S-hez tartozó poligráf az élpárok egy-egy tagjának elhagyásával körmentessé tehető. Először tegyük fel, hogy van egy ilyen körmentes gráfunk. Ekkor a gráf egy topologikus sorrendje a tranzakciók egy olyan rendezését adja, amelyben a forrás és az olvasó tranzakció közé nem ékelődhet be író tranzakció, továbbá minden író tranzakció megelőzi a hozzá tartozó olvasó tranzakciókat. Így a soros ütemezésben az olvasó-forrás kapcsolatok pontosan ugyanazok, mint S-ben; a két ütemezés nézetekvivalens, tehát S nézet-sorbarendezhető.

Megfordítva, ha S nézet-sorbarendezhető, akkor létezik hozzá egy S' nézetekvivalens soros ütemezés. Az S-hez tartozó poligráfban minden ($T_k \to T_j$, $T_i \to T_k$) élpár esetén az S' ütemezésben T_k vagy megelőzi T_j-t vagy T_i után következik. Ellenkező esetben T_k írásművelete megszakítaná a T_j és T_i közötti kapcsolatot, ami azt jelentené, hogy S és S' nem nézetekvivalens. Hasonlóan, a poligráf minden irányított éle is megjelenik az S'-beli tranzakciók sorrendjében. Ebből az következik, hogy a poligráf minden élpárjából ki tudjuk választani az egyik élt úgy, hogy az így kapott gráf irányított éleinek megfelel az S' soros ütemezése. Tehát a gráf körmentes.

10.10. példa: Vegyük a 10.4. ábrán látható poligráfot. Ez már ebben az állapotában is körmentes gráf. Ehhez egyetlen topologikus sorrend adható, így a 10.8. példában bemutatott ütemezés nézetekvivalens soros ütemezése (T_2, T_1, T_3).

Most nézzük a 10.7. ábra poligráfját. Két esetet kell megvizsgálnunk aszerint, hogy az élpárból melyik él marad meg. Ha a $T_4 \to T_1$ élt választjuk, akkor létrejön egy irányított kör a gráfban. A $T_2 \to T_4$ választása esetén azonban a gráf körmentes lesz. A gráfhoz megadható egyetlen topologikus sorrend (T_1, T_2, T_3, T_4), amely egyben nézetekvivalens soros ütemezésként is szolgál, és igazolja, hogy az eredeti ütemezés nézet-sorbarendezhető. □

10.2.4. Feladatok

10.2.1. feladat: Adjuk meg a következő ütemezésekhez tartozó poligráfokat és az összes nézetekvivalens soros ütemezést:

* a) $r_1(A)$; $r_2(A)$; $r_3(A)$; $w_1(B)$; $w_2(B)$; $w_3(B)$;
b) $r_1(A)$; $r_2(A)$; $r_3(A)$; $r_4(A)$; $w_1(B)$; $w_2(B)$; $w_3(B)$; $w_4(B)$;

c) $r_1(A)$; $r_3(D)$; $w_1(B)$; $r_2(B)$; $w_3(B)$; $r_4(B)$; $w_2(C)$; $r_5(C)$; $w_4(E)$; $r_5(E)$; $w_5(B)$;
d) $w_1(A)$; $r_2(A)$; $w_3(A)$; $r_4(A)$; $w_5(A)$; $r_6(A)$;

! **10.2.2. feladat:** Határozzuk meg, hogy az alábbi soros ütemezésekkel az előző feladat ütemezései közül hány i) konfliktusekvivalens, illetve ii) nézetekvivalens:

* a) $r_1(A)$; $w_1(B)$; $r_2(A)$; $w_2(B)$; $r_3(A)$; $w_3(B)$; azaz mindhárom tranzakció először olvassa A-t, majd írja B-t.
b) $r_1(A)$; $w_1(B)$; $w_1(C)$; $r_2(A)$; $w_2(B)$; $w_2(C)$; azaz mindhárom tranzakció először olvassa A-t, majd írja B-t és C-t.

10.3. Holtpontkezelés

Már többször megfigyelhettük, hogy az egyszerre végrehajtott tranzakciók versenyezhetnek egymással az erőforrásokért, és ezáltal *holtpontra* (deadlock) juthatnak, ami azt jelenti, hogy minden tranzakció egy másik tranzakció által birtokolt erőforrásra vár, és egyik sem tud továbblépni.

- A 9.3.4. részben láttuk, hogy a kétfázisú zárolást használó tranzakciók szokásos műveletei még mindig vezethetnek holtponthoz, ha mindegyik tranzakció valami olyasmit zárolt, amelyre egy másiknak is szüksége van.
- A 9.4.3. részben pedig azt láttuk, hogy az a lehetőség, hogy az osztott zár kizárólagos zárrá minősíthető fel, szintén okozhat holtpontot, ha minden tranzakció rendelkezik ugyanarra az elemre egy osztott zárral, és ezt egyszerre akarják felminősíteni.

A holtpontkezelés problémája két fő irányból közelíthető meg. Vagy valahogy rájövünk, hogy néhány tranzakció holtpontra jutott, és ebből a helyzetből keresünk kiutat, vagy már eleve úgy kezeljük a tranzakciókat, hogy soha ne juthassanak holtpontra.

10.3.1. Holtpontérzékelés időkorláttal

Ha a holtpont már bekövetkezett, akkor általában nem lehet a helyzeten úgy javítani, hogy minden tranzakció továbbléphessen. Azaz legalább egy tranzakciót vissza kell görgetni – abortáltatni kell, majd újraindítani.

A holtpontok érzékelésére és feloldására a legegyszerűbb megoldást az *időtúllépés* (timeout) módszere adja. Időkorlátokat vezetünk be, amely arra vonatkozik, hogy az egyes tranzakciók mennyi ideig lehetnek aktívak, és ha ezt a határt túllépik, akkor visszagörgetjük őket. Például egy egyszerű rendszerben, ahol a tipikus tranzakciók ezredmásodpercek alatt lefutnak, az egyperces időkorlátnak tényleg csak a holtpontra jutott tranzakcióra lenne hatása. De ha van néhány összetettebb tranzakció is, akkor az időtúllépés bekövetkezéséhez hosszabb időt választhatunk.

Vegyük észre, hogyha a holtpontra jutott tranzakció túllépi az időkorlátját, akkor a többi erőforrással együtt az eddig birtokolt zárjairól is lemond. Így tehát van esély arra, hogy a holtponton álló többi tranzakció még azelőtt be tudja fejezni a tevékenységét, mielőtt kifutna az időből. De mivel a holtpontra jutott tranzakciók valószínűleg körülbelül ugyanabban az időpontban indultak (különben az egyik befejeződött volna, még mielőtt a másik elkezdődött volna), az is lehetséges, hogy a rendszer hamis időtúllépéseket érzékel, azaz úgy görgeti vissza a tranzakciókat, hogy azok már túljutottak a közös holtponton.

10.3.2. A várakozási gráf

Azok a helyzetek, amikor a holtpont azért alakul ki, mert az egyik tranzakció a másik birtokában lévő zárakra vár, jól kezelhetők a *várakozási gráffal* (waits-for graph). Ebben a gráfban azt tartjuk nyilván, hogy melyik tranzakció melyikre vár. A várakozási gráf segítségével érzékelhetővé válnak a már kialakult holtpontok, de meg is előzhető a kialakulásuk. Mi az utóbbival próbálkozunk, ami azzal jár, hogy a várakozási gráfot egész idő alatt nyilván kell tartanunk, és az olyan műveleteket, amelyek következtében a gráfban kör alakulna ki, nem szabad megengednünk.

Idézzük fel a 9.5.2. rész alapján, hogy a zártáblában minden X adatbáziselemhez létezik egy lista, amelyben azon tranzakciók mellett, amelyek arra várnak, hogy zárolhassák X-t, azok is fel vannak sorolva, amelyek rendelkeznek X zárjával. A várakozási gráf csúcsai a listában található tranzakcióknak felelnek meg. A gráfban irányított él fut T-ből U-ba, ha létezik olyan A adatbáziselem, hogy:

1. U zárolja A-t.
2. T arra vár, hogy zárolhassa A-t, és
3. T csak akkor kapja meg a számára megfelelő módban A zárját, ha először U lemond róla.[3]

Ha nincsen (irányított) kör a gráfban, akkor végül minden tranzakció be tudja fejezni a működését. Lesz legalább egy olyan tranzakció, amelyik nem vár semelyik másikra, így ez biztosan befejeződhet. Ekkor viszont megint lesz legalább egy tranzakció, amelyik nem várakozik, ezért továbbléphet és így tovább.

Ha azonban a gráf nem körmentes, akkor a körben részt vevő tranzakciók nem léphetnek tovább, azaz holtpontra jutottak. A holtpont-megelőzési stratégia tehát abból áll, hogy minden olyan tranzakciót visszagörgetünk, amelynek valami olyan igénye van, ami kört idézne elő a várakozási gráfban.

[3] Egyszerű esetekben, mint amikor osztott és kizárólagos zárakat használunk, minden egyes várakozó tranzakciónak meg kell várnia, amíg *minden* zárral rendelkező tranzakció feloldja a saját zárját. Vannak azonban olyan zármód rendszerek is, ahol egy tranzakció már akkor is megkaphatja a zárat, ha még csak néhány zárbirtokos mondott le róla. Lásd 10.3.6. feladat.

10.11. példa: Tegyük fel, hogy a következő négy tranzakcióval rendelkezünk:

T_1: $l_1(A)$; $r_1(A)$; $l_1(B)$; $w_1(B)$; $u_1(A)$; $u_1(B)$;
T_2: $l_2(C)$; $r_2(C)$; $l_2(A)$; $w_2(A)$; $u_2(C)$; $u_2(A)$;
T_3: $l_3(B)$; $r_3(B)$; $l_3(C)$; $w_3(C)$; $u_3(B)$; $u_3(C)$;
T_4: $l_4(D)$; $r_4(D)$; $l_4(A)$; $w_4(A)$; $u_4(D)$; $u_4(A)$;

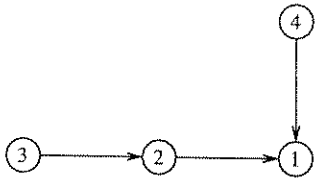
Mindegyik az egyik elemet olvassa, a másikat írja. Egy egyszerű zárrendszert használunk egyféle zárral, bár ugyanezt a jelenséget figyelhetnénk meg, ha egy osztott/kizárólagos rendszerben a szokásos módon osztanánk a zárakat: osztott zárat az olvasáshoz, kizárólagos zárat az íráshoz.

A 10.8. ábrán egy lehetséges ütemezés kezdeti szakasza látható. Az első négy lépésben mindegyik tranzakció zárolja azt az elemet, amelyet olvasni szeretne. Az 5) lépésben T_2 megpróbálja zárolni A-t, de nem tudja, mert a zár már T_1 birtokában van. T_2 tehát várakozik T_1-re, ezért a várakozási gráfba berajzolunk egy élt a T_2-nek megfelelő csúcsból a T_1-nek megfelelő csúcs felé.

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$			
2)		$l_2(C)$; $r_2(C)$;		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(D)$; $r_4(D)$;
5)		$l_2(A)$; **Elutasítva**		
6)			$l_3(C)$; **Elutasítva**	
7)				$l_4(A)$; **Elutasítva**
8)	$l_1(B)$; **Elutasítva**			

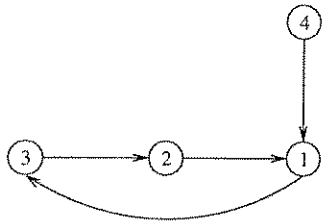
10.8. ábra. *Egy ütemezés első néhány lépése, amelyben holtpont alakul ki*

Hasonlóan, a 6) lépésben T_3 nem tudja zárolni C-t T_2 miatt, a 7) lépésben pedig T_4 vall kudarcot A zárolásával T_1 miatt. Az ebben az állapotban egyelőre körmentes várakozási gráf a 10.9. ábrán látható.



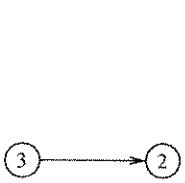
10.9. ábra. *A várakozási gráf az ütemezés 7) lépése után*

A 8) lépésben T_1-nek várnia kell B zárolásával T_3 miatt. Ha megengednénk T_1-nek, hogy várjon erre a zárra, akkor T_1, T_2 és T_3 mentén kör jönne létre a várakozási gráfban, ahogy ezt a 10.10. ábra is mutatja. Mivel a körben mindegyik tranzakció arra vár, hogy a másik befejeződjön, egyik sem tud továbblépni, vagyis ennek a három tranzakciónak a részvételével holtpont alakul ki. Véletlen egybeesés, hogy T_4 sem fejeződhet be annak ellenére, hogy nincs benne a körben. Az ő előrejutása azonban T_1 továbblépésén múlik.



10.10. ábra. *A várakozási gráf az ütemezés 8) lépése után kört tartalmaz*

Mivel a kört okozó tranzakciókat visszagörgetjük, így teszünk T_1-gyel is. A várakozási gráf a 10.11. ábrának megfelelően alakul. T_1 feloldja A zárolását, amelyet vagy T_2 vagy T_4 vesz át. Tegyük fel, hogy a zár T_2 birtokába kerül. T_2 befejeződik, ezáltal feloldódik a zár A-n és C-n. Most T_3, amely C-t akarja zárolni, és T_4 is, amely A-t, lezárulhat. Valamivel később T_1-et újraindítjuk, de nem kaphatja meg sem A, sem B zárját, amíg T_2, T_3 és T_4 be nem fejeződött. □



10.11. ábra. *A várakozási gráf T_1 visszagörgetése után*

10.3.3. Holtpontmegelőzés az elemek sorbarendezésével

Most ismerkedjünk meg néhány más módszerrel is, amellyel megelőzhető a holtpont kialakulása. Az elsőben sorrendbe kell raknunk az adatbáziselemeket valamilyen tetszőleges, de előre rögzített rendezés szerint. Ha például az adatbáziselemek blokkok, akkor rendezhetjük őket lexikografikusan a fizikai címük szerint. Emlékezzünk vissza, a 3.3.1. részben volt arról szó, hogy egy blokk fizikai címe általában egy olyan bájtsorozat, amely a blokknak a tárrendszeren belüli helyét írja le.

Ha minden tranzakciótól megkívánjuk, hogy az adatbáziselemeket a megadott sorrendnek megfelelően próbálja meg zárolni (ami egyébként a legtöbb alkalmazásban nem reális feltétel), akkor a (foglalt) zárakra való várakozások miatt nem alakulhat ki holtpont. Ugyanis tegyük fel, hogy T_2 várakozik A_1 zárolásával T_1 miatt, T_3 várakozik A_2 zárolásával T_2 miatt, és így tovább, T_n várakozik A_{n-1} zárolásával T_{n-1} miatt, T_1 pedig A_n zárolásával várakozik T_n miatt. Mivel T_2 zárolta A_2-t, de várnia kell A_1-re,

$A_2 < A_1$ relációnak kell teljesülnie az adott rendezésen. Hasonlóan, $A_i < A_{i-1}$, ahol $i = 3, 4, \ldots, n$. De mivel T_1 zárolta A_1-et és várakozik A_n-re, $A_1 < A_n$ is fennáll. Ekkor a következő teljesül: $A_1 < A_n < A_{n-1} < \ldots < A_2 < A_1$, ami lehetetlen, hiszen ebből $A_1 < A_1$ következne.

10.12. példa: Tegyük fel, hogy az adatbáziselemek alfabetikusan vannak rendezve. Ekkor, ha a 10.11. példa négy tranzakcióját vesszük, akkor T_2-t és T_4-t át kell írnunk, hogy a megfelelő sorrendben zárolják az elemeket. Így a négy tranzakció a következő:

T_1: $l_1(A)$; $r_1(A)$; $l_1(B)$; $w_1(B)$; $u_1(A)$; $u_1(B)$;
T_2: $l_2(A)$; $l_2(C)$; $r_2(C)$; $w_2(A)$; $u_2(C)$; $u_2(A)$;
T_3: $l_3(B)$; $r_3(B)$; $l_3(C)$; $w_3(C)$; $u_3(B)$; $u_3(C)$;
T_4: $l_4(A)$; $l_4(D)$; $r_4(D)$; $w_4(A)$; $u_4(D)$; $u_4(A)$;

A 10.12. ábrán látható, hogy mi történik, ha ugyanazt az ütemezést használjuk, mint ami a 10.8. ábrán adott. Az első lépésben T_1 zárolja A-t. A következő lépésben T_2 próbálja végrehajtani az első műveletét, de A zárolásával T_1 miatt várnia kell. Ezek után T_3 zárolja B-t, majd T_4 is megpróbálja zárolni A-t, de várakoznia kell.

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$			
2)		$l_2(A)$; **Elutasítva**		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(A)$; **Elutasítva**
5)			$l_3(C)$; $w_3(C)$;	
6)			$u_3(B)$; $u_3(C)$;	
7)	$l_1(B)$; $w_1(B)$;			
8)	$u_1(A)$; $u_1(B)$;			
9)		$l_2(A)$; $l_2(C)$;		
10)		$r_2(C)$; $w_2(A)$;		
11)		$u_2(A)$; $u_2(C)$;		
12)				$l_4(A)$; $l_4(D)$;
13)				$r_4(D)$; $w_4(A)$;
14)				$u_4(A)$; $u_4(D)$;

10.12. ábra. *Az elemeket a tranzakciók alfabetikus sorrendben zárolják, ezzel megelőzhető a holtpont kialakulása*

Mivel T_2 elakadt, a 10.8. ábra ütemezése szerint T_3 következik. Sikeresen zárolja C-t, majd a 6) lépésben lezárul. Mivel ezzel egy időben elengedi B-t és C-t, T_1 is befejeződhet, ami meg is történik a 8) lépésben. Ezen a ponton A felszabadul a zárolás alól, és feltehető, hogy FIFO alapon T_2 birtokába kerül a zár. Most T_2 mindkét szükséges adatelemét zárolja, majd a 11) lépésben lezárul. Végül T_4 is megkapja az óhajtott zárakat és befejeződik. □

10.3.4. Holtpontérzékelés időbélyegzővel

A 10.3.2. részben láttuk, hogy a várakozási gráf segítségével észlelni tudjuk a holtpontok kialakulását. Ez a gráf azonban nagy lehet, és minden alkalommal kört keresni benne, valahányszor egy tranzakciónak várnia kell egy zárra, nagyon időigényessé válhat. A várakozási gráf mellett egy másik lehetőség az időbélyegzők bevezetése. Minden egyes tranzakcióhoz hozzárendelünk egy-egy időbélyegzőt, amely:

- Csak a holtpont érzékelésére használatos. Ez nem ugyanaz, mint az időbélyegző, amelyet a 9.8. részben a konkurenciavezérléshez használtunk, még akkor sem, ha éppen az időbélyegző alapú konkurenciavezérlés van érvényben.
- Például, ha egy tranzakciót vissza kell görgetni, akkor egy új, későbbi konkurencia időbélyegzővel kezdi újra a működését, de a holtpontészleléshez használt időbélyegzője sohasem változik.

Az időbélyegzőt akkor használjuk, amikor egy T tranzakciónak az U tranzakció birtokában lévő zárra kell várakoznia. Attól függően, hogy T vagy U az *idősebb* (korábbi időbélyegzővel rendelkező), két különböző dolog történhet. Két különféle eljárásmód használható a tranzakciók kezelésére és a holtpontok érzékelésére.

1. *Megvár-meghal séma:*

   a) Ha T idősebb U-nál (azaz T időbélyegzője korábbi, mint U időbélyegzője), akkor megengedjük, hogy T az U által birtokolt zár(ak)ra várakozzon.
   b) Ha U idősebb T-nél, akkor T „meghal": visszagörgetjük.

2. *Megsebez-megvár séma:*

   a) Ha T idősebb U-nál, akkor „megsebzi" U-t. A „seb" általában végzetes: U-t vissza kell görgetni, és le kell mondania a szükséges zárakról T javára. Egyetlen eset képez kivételt: ha, mire a „sebzésnek" hatása lenne, U befejeződik, és elengedi a zárjait. Ilyenkor U életben marad, és nem kell visszagörgetni.
   b) Ha U idősebb T-nél, akkor T az U birtokában lévő zár(ak)ra várakozik.

10.13. példa: Vegyük a 10.12. példa tranzakcióit, és nézzük meg, hogyan működik a megvár-meghal séma. Feltesszük, hogy T_1, T_2, T_3, T_4 az időbélyegzők sorrendje, azaz T_1 a legidősebb tranzakció. Azt is feltesszük, hogy amikor egy tranzakciót visszagörgetünk, az nem indul újra olyan hamar, hogy még azelőtt aktívvá válna, mielőtt a többi tranzakció lezárulna.

Az események egy lehetséges sorrendje a megvár-meghal sémát használva a 10.13. ábrán látható. A zárját először T_1 kapja meg. Amikor T_2 akarja zárolni A-t, meghal, mert T_1 idősebb nála. A 3) lépésben T_3 zárolja B-t, de a 4) lépésben T_4 akarja zárolni A-t, és meghal, mert a zár birtokosa, T_1, idősebb T_4-nél. A következő lépésben T_3 megkapja C zárját, majd befejeződik. Amikor T_1 folytatja a működését, elérhető lesz számára B zárja, majd a 8) lépésben szintén befejeződik.

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$			
2)		$l_2(A)$; **Meghal**		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(A)$; **Meghal**
5)			$l_3(C)$; $w_3(C)$;	
6)			$u_3(B)$; $u_3(C)$;	
7)	$l_1(B)$; $w_1(B)$;			
8)	$u_1(A)$; $u_1(B)$;			
9)				$l_4(A)$; $l_4(D)$;
10)		$l_2(A)$; **Várakozik**		
11)				$r_4(D)$; $w_4(A)$;
12)				$u_4(A)$; $u_4(D)$;
13)		$l_2(A)$; $l_2(C)$;		
14)		$r_2(C)$; $w_2(A)$;		
15)		$u_2(A)$; $u_2(C)$;		

10.13. ábra. *A tranzakciók műveletei a megvár-meghal sémában érzékelik a holtpontot*

Most újrakezdődik a két visszagörgetett tranzakció: T_2 és T_4. A holtpontkezeléshez használt időbélyegzőjük nem változik; T_2 továbbra is idősebb, mint T_4. Feltesszük azonban, hogy T_4 kezdődik előbb, a 9) lépésben, és amikor a 10) lépésben az idősebb T_2 zárolni akarja A-t, akkor erre várnia kell, de nem abortál. T_4 lezárul a 12) lépésben, majd az utolsó három lépésben T_2 is befejezi a működését. □

10.14. példa: Most a 10.14. ábra segítségével kövessük nyomon, hogy ugyanezek a tranzakciók hogyan viselkednek, ha a megsebez-megvár sémát használjuk. Mint a 10.13. ábrán látható, T_1 itt is A zárolásával indul. Amikor a 2) lépésben T_2 akarja zárolni A-t, várakoznia kell, hiszen T_1 idősebb T_2-nél. Miután a 3) lépésben T_3 zárolja B-t, T_4-nek is várnia kell A zárjára.

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$			
2)		$l_2(A)$; **Várakozik**		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(A)$; **Várakozik**
5)	$l_1(B)$; $w_1(B)$;		**Megsebezve**	
6)	$u_1(A)$; $u_1(B)$;			
7)		$l_2(A)$; $l_2(C)$;		
8)		$r_2(C)$; $w_2(A)$;		
9)		$u_2(A)$; $u_2(C)$;		
10)				$l_4(A)$; $l_4(D)$;
11)				$r_4(D)$; $w_4(A)$;
12)				$u_4(A)$; $u_4(D)$;
13)			$l_3(B)$; $r_3(B)$;	
14)			$l_3(C)$; $w_3(C)$;	
15)			$u_3(B)$; $u_3(C)$;	

10.14. ábra. *A tranzakciók műveletei a megsebez-megvár sémában érzékelik a holtpontot*

Miért működik jól az időbélyegző alapú holtpontérzékelés?

Azt állítjuk, hogy sem a megvár-meghal, sem a megsebez-megvár sémával nem alakulhat ki kör a várakozási gráfban, így holtpont sem jöhet létre. Tegyük fel az ellenkezőjét, vagyis hogy létezik egy $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$ kör. A legidősebb tranzakció legyen mondjuk T_2.

A megvár-meghal sémában mindig csak újabb tranzakciókra lehet várni. Így nem lehetséges, hogy T_1 várjon T_2-re, mivel T_2 biztosan idősebb T_1-nél. A megsebez-megvár sémában csak idősebb tranzakciókra lehet várni. Tehát semmiképp sem lehetséges, hogy T_2 az újabb T_3-ra várakozzon. Ezekből következik, hogy a kör nem létezhet, ezért holtpont sincsen.

Ez után tegyük fel, hogy az 5) lépésben T_1 folytatja a működését, zár alá akarja helyezni B-t. B zárja már T_3 birtokában van, de mivel T_1 az idősebb, „megsebzi" T_3-t. T_3 még nem zárult le, a seb halálos: feloldjuk T_3 zárjait, őt pedig visszagörgetjük, így T_1 befejeződhet.

Amikor T_1 feloldja A zárját, tegyük fel, hogy ez T_2-höz kerül, amely ekkor továbbléphet. T_2 után a zárat T_4 kapja meg, amely ezután befejeződik. Végül T_3 újraindul és közbeavatkozás nélkül lezárul. □

10.3.5. A holtpontkezelő módszerek összehasonlítása

A megvár-meghal és a megsebez-megvár sémában is az idősebb tranzakciók maguknál újabb tranzakciókat végeznek ki. Mivel a tranzakciók mindig az eredeti időbélyegzőjükkel indulnak újra, egyszer mindegyikre teljesülni fog, hogy éppen ő a legidősebb a rendszerben, és mint ilyen, biztosan lezárul. Erre a biztosítékra, hogy végül mindegyik tranzakció befejeződik, úgy hivatkozunk, hogy *nincs kiéheztetés*. Vegyük észre, hogy a fejezetben leírt többi séma nem feltétlenül előzi meg a kiéheztetést; ha nem teszünk további intézkedéseket, a tranzakciókkal újra és újra megtörténhet, hogy az indításuk után holtpontra jutnak, és ezért visszagörgetjük őket. Ezzel kapcsolatban nézzük meg a 10.3.7. feladatot.

A megvár-meghal és a megsebez-megvár sémák működése között azonban van egy finom különbség. A megsebez-megvár sémában valahányszor egy idősebb tranzakciónak egy újabb tranzakció birtokában lévő zárra van szüksége, az újabb tranzakció meghal. Ha feltesszük, hogy a tranzakciók az indítás után nem sokkal már birtokba veszik a szükséges zárakat, akkor ritkán fog az előfordulni, hogy egy újabb tranzakció megszerzi a zárakat az idősebb elől. Ebben a sémában tehát a tranzakciók visszagörgetésére várhatóan csak ritkán kerül sor.

Másrészt, amikor a megvár-meghal séma görget vissza egy tranzakciót, akkor az

még mindig a „zárgyűjtő" stádiumában van, amely feltehetően a tranzakció legkorábbi szakaszát jelenti. Így, habár a megvár-meghal séma talán több tranzakciót görget vissza, mint a megsebez-megvár séma, ezek a tranzakciók csak rövid ideig működtek az abortálás előtt. Ezzel szemben a megsebez-megvár sémában visszagörgetett tranzakció valószínűleg már birtokba vette az összes szükséges zárat, és tekintélyes mennyiségű processzoridőt vett igénybe az eddigi működése. A feldolgozandó tranzakcióktól függően tehát hol az egyik, hol a másik módszer eredményez több fölösleges munkát.

Most vizsgáljuk meg a két séma előnyeit és hátrányait a várakozási gráf kézenfekvő konstrukciójával és használatával szemben is. A következő szempontok a lényegesek:

- A megsebez-megvár és a megvár-meghal sémát is könnyebb megvalósítani, mint egy olyan rendszert, amely folyamatosan nyilvántartja vagy időszakosan létrehozza a várakozási gráfot. A várakozási gráf felépítésével járó hátrány még rendkívülibb, amikor osztott adatbázisokkal van dolgunk, és a gráfot a különböző munkaállomásokról[4] összegyűjtött zártáblák alapján kell elkészíteni. Erről bővebben a 10.6. részben lesz szó.
- A várakozási gráf használatával minimálisra csökkenthető a holtpont miatt abortáltatott tranzakciók száma. A tranzakciót mindig csak akkor abortáltatjuk, ha tényleg holtpontra jutott. Másrészt azonban, a megsebez-megvár és a megvár-meghal sémával is előfordulhat néha, hogy olyankor görget vissza egy tranzakciót, amikor nem jött volna létre holtpont, és a tranzakció életben hagyásával sem alakulhatott volna ki.

10.3.6. Feladatok

10.3.1. feladat: Tegyük fel, hogy az alábbi műveletsorozatokban minden egyes olvasás-, illetve írásműveletet közvetlenül megelőzi az osztott, illetve kizárólagos zár igénylése. Tegyük fel továbbá, hogy a zárak feloldása rögtön a tranzakció utolsó művelete után történik meg. Adjuk meg azokat a műveleteket, amelyeknek a végrehajtását az ütemező megtagadja, és mondjuk meg, hogy létrejön-e holtpont vagy sem! Adjuk meg továbbá, hogy hogyan alakul a műveletek végrehajtása során a várakozási gráf! Ha létrejön egy holtpont, abortáltassuk az egyik tranzakciót, és mutassuk meg, hogyan folytatódik a műveletsorozat!

* a) $r_1(A)$; $r_2(B)$; $w_1(C)$; $r_3(D)$; $r_4(E)$; $w_3(B)$; $w_2(C)$; $w_4(A)$; $w_1(D)$;
b) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(D)$;
c) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(A)$;
d) $r_1(A)$; $r_2(B)$; $w_1(C)$; $w_2(D)$; $r_3(C)$; $w_1(B)$; $w_4(D)$; $w_2(A)$;

10.3.2. feladat: Hogyan játszódnak le a 10.3.1. feladat műveletsorozatai a megsebez-megvár holtpont-megelőzési rendszerben? Tegyük fel, hogy a holtpont-időbélyegzők sorrendje megegyezik a tranzakciók indexével, azaz T_1, T_2, T_3, T_4. Tegyük fel azt is, hogy a tranzakciók esetleges újraindítása a visszagörgetés sorrendjében történik.

[4] Angolul *site*. A hálózatban részt vevő számítógépeket, „helyszíneket" nevezzük így. *A fordító megjegyzése.*

10.3.3. feladat: Hogyan játszódnak le a 10.3.1. feladat műveletsorozatai a megvár-meghal holtpont-megelőzési rendszerben? Használjuk ugyanazokat a feltevéseket, mint a 10.3.2. feladatban!

! **10.3.4. feladat:** Igaz-e, hogy minden $n > 1$ egészhez létezik olyan várakozási gráf, amelyben a legrövidebb kör hossza n? Mi a helyzet $n = 1$, vagyis hurokél esetén?

!! **10.3.5. feladat:** A holtpontok elkerülésére a következő módszer is megadható: minden tranzakció a futása elején bejelenti, hogy mely zárakra lesz szüksége, ezeket vagy mind megkapja, vagy egyiket sem kapja meg, és várakoznia kell. Elkerülhető-e ezzel a módszerrel a zárolások miatt kialakuló holtpont? Ha igen, állításunkat indokoljuk; ha nem, adjunk ellenpéldát!

! **10.3.6. feladat:** Vegyük a 9.6. részben megismert szándékjelölő zárolási rendszert. Írjuk le, hogyan lehetne ehhez a zárrendszerhez várakozási gráfot készíteni! Vegyük például azt a lehetőséget, amikor az A adatbáziselemet különböző tranzakciók S, illetve IX módban zárolják. Milyen éleket rajzolunk a gráfba, ha A zárolásával valamelyik tranzakciónak várnia kell?

*! **10.3.7. feladat:** A 10.3.5. részben rámutattunk arra, hogy a megsebez-megvár, illetve megvár-meghal sémától eltérő holtpontérzékelő módszerek nem feltétlenül előzik meg a tranzakciók kiéheztetését, amikor is a tranzakciót ismételten visszagörgetjük, így az sohasem éri el a végpontját. Adjunk példát arra, hogy hogyan vezethet a tranzakciók kiéheztetéséhez annak a módszernek a használata, amikor minden olyan tranzakciót visszagörgetünk, amely kört hozna létre a várakozási gráfban! Megelőzhető-e a kiéheztetés, ha a tranzakciók csak egy rögzített sorrendben zárolhatják az elemeket? Mit mondhatunk az időtúllépésről, mint holtpontfeloldó mechanizmusról?

10.4. Osztott adatbázisok

Ebben a fejezetben az osztott adatbázisrendszerek alapelemeit tekintjük át. Egy osztott rendszerben sok, viszonylag autonóm processzorral rendelkezünk, amelyek mind részt vehetnek az adatbázis-műveletekben. Az osztott adatbázisok alkalmazásában számos lehetőség rejlik:

1. Mivel egyszerre több gép is dolgozhat ugyanazon a problémán, jobbak a lehetőségek a párhuzamosításra és arra, hogy a lekérdezésekre gyorsan kapjuk meg a választ.
2. Mivel az adatok többszörözés útján számos helyszínen is jelen lehetnek, a rendszernek valószínűleg nem kell leállnia a feldolgozással csak azért, mert az egyik munkaállomás vagy alkotóelem az adott pillanatban nem érhető el.

Másrészt azonban az osztott feldolgozás minden tekintetben növeli az adatbázisrendszer összetettségét, ezért újra kell gondolnunk az ABKR legalapvetőbb elemeinek a tervezését is. Sok osztott környezetben a kommunikációs költség túlnőhet a feldolgozás költségén, így kritikus témává válik az elküldött üzenetek száma. Ebben a fejezetben az osztott adatbázisok fő kérdései kerülnek bevezetésre, az utána következő részekben pedig az osztott adatbázisok két fontos problémájának, az osztott véglegesítésnek és az osztott zárolásnak megoldására összpontosítunk.

10.4.1. Osztott adatok

Az adatok szétosztásának egyik fontos oka lehet, hogy az adatbázist felhasználó szervezet maga is több helyszínre van szétosztva, és minden munkaállomáson megvannak az elsősorban odatartozó adatok. Lássunk néhány példát:

1. Egy banknak lehet sok fiókja. Minden fiók (vagy egy adott város fiókcsoportja) rendelkezik a nála (vagy a városban) vezetett számlák adatbázisával. Az ügyfelek akármelyik fiókban elintézhetik a pénzügyeiket, de általában a „saját" fiókjukba járnak, ahol a számla adatait is tárolják. A banknak lehetnek a központi fiókban tárolt adatai is, például az alkalmazottakról vagy a banki politikáról, amely például az aktuális kamatlábat is jelentheti. Természetesen az egyes fiókokban tárolt adatrekordokról készült biztonsági másolatot (backup) is tárolják valahol, de valószínűleg nem az adott és nem is a központi fiókban.
2. Egy üzlethálózat sok önálló áruházból állhat. Minden üzlet (vagy egy város üzletcsoportja) rendelkezik az adott üzletben történt vásárlásokra vonatkozó adatbázissal és egy raktáradatbázissal. Lehet, hogy van egy központi részleg, ahol az alkalmazottakról, az üzletláncszintű leltárról vagy a hitelkártyával fizető vásárlókról tárolnak adatokat, a szállítókról pedig olyan információkat tartanak nyilván, mint például a nem teljesített megrendelések, vagy hogy mennyivel tartozik még az üzletlánc. Ezenfelül létezhet még egy „adattárház" is, amelyben megtalálható az összes üzletben nyilvántartott vásárlási adatok másolata. Ezt az elemzők arra használják, hogy ad hoc lekérdezések segítségével analizálják, illetve előre jelezzék a várható keresletet (lásd 11.3. részt).
3. On-line könyvekkel és egyéb dokumentumokkal rendelkező egyetemek közös erővel létrehozhatnak egy digitális könyvtárat. Akármelyik munkaállomáson indítunk egy keresést, az az elérhető dokumentumok *uniójának* a katalógusát fogja megvizsgálni, és ha valamelyik munkaállomáson elérhető a keresett dokumentum, a felhasználó megkapja az elektronikus másolatát.

Néhány esetben az a reláció, amire logikailag úgy gondolunk, mintha egyetlen reláció lenne, valójában sok különböző helyen megtalálható részre van szétosztva. Úgy képzelhetjük például, hogy az üzlethálózat egyetlen vásárlási relációval rendelkezik:

```
Eladások(árucikk, dátum, ár, vevő)
```


A kommunikációs költségben szerepet játszó tényezők

Mivel manapság egyre olcsóbban egyre nagyobb sávszélesség érhető el, eltűnődhetünk azon, vajon számításba kell-e vennünk a kommunikációs költséget az osztott adatbázisrendszer tervezése közben. A legnagyobb elektronikusan kezelt objektumok között most az adatok bizonyos fajtái is megtalálhatók, tehát még egy nagyon olcsó kommunikációs megoldás mellett sem hanyagolható el egy terabájt méretű adatdarab továbbításának a költsége. A legtöbb esetben azonban a kommunikációs költség nem csupán a bitek átküldéséből áll – figyelembe kell vennünk a különböző protokollrétegeket is, amelyek előkészítik, majd a vevő oldalon helyreállítják a továbbításra szánt adatokat, és kezelik az egész kommunikációt. Ezen protokollok mindegyike tekintélyes mennyiségű számítást igényel. A számítások elvégzése is egyre kevesebbe kerül, a kommunikáció miatt végrehajtott számítások mennyisége azonban valószínűleg még mindig jelentős marad ahhoz képest, amennyire a legfontosabb adatbázis-műveletek elvégzéséhez a hagyományos, egyprocesszoros rendszerekben szükség van.

Ez a reláció azonban fizikailag nem létezik, csak mint a hálózat üzleteiben tárolt, azonos sémával rendelkező számos reláció uniója. Ezeket a helyi relációkat *töredékeknek* (fragments) hívjuk, a logikai reláció fizikai részekre bontását pedig az `Eladások` reláció *vízszintes irányú (horizontális) dekompozíciójának* nevezzük. A felbontásra azért hivatkozunk „vízszintesként", mert a „nagy" `Eladások` reláció részekre bontását úgy is elképzelhetjük, mintha vízszintes vonalakkal választanánk szét az egyes üzletekhez tartozó sorokat egymástól.

Más helyzetekben úgy tűnik, mintha az osztott adatbázis a relációt „függőlegesen" bontotta volna fel, mégpedig úgy, hogy ami logikailag egy reláció lenne, azt két vagy több, más-más munkaállomáson megtalálható relációra osztja szét, amelyek attribútumai az eredeti attribútumhalmaz részhalmazát képezik. Például, ha meg akarjuk keresni azokat a vásárlásokat a bostoni áruházban, ahol a vevő több mint 90 napos hátralékban van a hitelkártyaszámla kiegyenlítésével, akkor hasznos lenne egy olyan reláció (vagy nézettábla), amely az `Eladások` táblázatból az `árucikk`, `dátum` és `vevő` információkkal együtt a vevő utolsó hitelkártyaszámla befizetésének az időpontját is tartalmazná. Az itt leírt esetben azonban ez a reláció függőlegesen fel van bontva, ezért össze kellene kapcsolnunk a központban tárolt hitelkártyás vásárló relációt a bostoni áruházban nyilvántartott `Eladások` töredékkel.

10.4.2. Osztott tranzakciók

Az adatok megosztásának az az egyik következménye, hogy a tranzakciók különböző helyeken végrehajtandó folyamatokat is magukban foglalnak. Így az eddigi tranzakciómodellünket meg kell változtatni. A tranzakciót nem tekinthetjük többé egy darab kódnak, amit az egyetlen ütemezővel és az egyetlen naplókezelővel kommunikáló

egyetlen processzor egyetlen számítógépen hajt végre. A tranzakció most egymással kapcsolatot tartó, különböző munkaállomásokon megtalálható *tranzakció-alkotórészekből* áll, amelyek mindegyike az adott munkaállomáson működő lokális ütemezővel és naplókezelővel kommunikál. Két fontos kérdést kell újra átgondolnunk:

1. Hogyan kezeljük a véglegesítés/abortálás döntést osztott tranzakciók esetén? Mi történik, ha a tranzakció egyik alkotórésze abortáltatni akarja az egész tranzakciót, de a többi alkotórésznek nincs problémája, és inkább véglegessé szeretne válni? A 10.5. részben megtárgyaljuk a „kétfázisú véglegesítés" technikáját, melynek segítségével helyes döntés hozható, és amely gyakran akkor is figyelembe veszi a még elérhető munkaállomásokat, ha néhány másik már átmenetileg működésképtelenné vált.
2. Hogyan biztosítható az olyan tranzakciók sorbarendezhetősége, amelyek különböző helyeken végrehajtott alkotórészeket foglalnak magukban? A 10.6. részben különös figyelmet szentelünk a zárolás problémájának, és megnézzük, hogyan használhatók a lokális zártáblák az adatbáziselemek globális zárolásához, és így hogyan támogatják egy megosztott környezetben a tranzakciók sorbarendezhetőségét.

10.4.3. Adattöbbszörözés

Az osztott rendszereknek egy fontos előnye az, hogy az adatokat *többszörözni* tudjuk, azaz az adatelemekről másolatokat tarthatunk több helyen is. Ez egyrészt hasznos abban az esetben, ha az egyik munkaállomás meghibásodik, mert ilyenkor egy másik helyen is megtalálhatók azok az adatok, amelyekhez most a meghibásodás miatt nem lehet hozzáférni. Másrészről pedig növelhetjük a válaszadás sebességét azzal, hogy a szükséges adatokból másolatokat tárolunk azon a munkaállomáson, ahol a lekérdezést elindítják.

Például:

1. A bank minden fiókjában tárolhat egy-egy másolatot az aktuális kamatlábakról, így az erre vonatkozó lekérdezéseket nem kell elküldeni a központi fiókba.
2. Az áruház minden egyes üzletében tárolhat egy-egy másolatot a szállítókról nyilvántartott információkról, így ha lokálisan szükség van valamilyen adatra (például az üzletvezető tudni szeretné egy szállító telefonszámát, mert ellenőrizni akarja a szállítmányt), akkor ehhez nem kell a központi részlegbe üzeneteket küldeni.
3. A digitális könyvtár ideiglenesen eltárolhatja (cache-elheti) egy népszerű dokumentum másolatát abban az iskolában, ahol a diákoknak ez kötelező olvasmány.

A többszörözött adatokkal kapcsolatban azonban számos problémával is szembe kell néznünk.

a) Hogyan oldható meg, hogy a másodpéldányok mind azonosak legyenek? A többszörözött adat módosítása lényegében egy olyan osztott tranzakcióvá válik, amely az összes másolatot módosítja.

b) Hogyan döntsük el, hogy hol, mennyi másodpéldányt tároljunk? Minél több másolattal rendelkezünk, annál nagyobb erőfeszítést vesz igénybe az adat módosítása, viszont annál könnyebb lesz a lekérdezések végrehajtása. Egy ritkán módosított relációból például a maximális hatásfok érdekében mindenhol tárolhatunk másolatot, viszont egy gyakran módosított relációból csak egy vagy kettő másodpéldányunk lehetne.
c) Mi történik hálózati kommunikációs hiba esetén, amikor ugyanannak az adatnak különböző másodpéldányai egymástól független változásokon mehetnek keresztül, és amikor a hálózat újra helyreáll, az adat másolatait valahogy össze kell egyeztetni?

10.4.4. Osztott lekérdezésoptimalizálás

A megosztott adatok jelenléte hatást gyakorol a fizikai lekérdezési terv létrehozásakor választható lehetőségekre és a terv összetettségére is (lásd 7.7. részt). Többek között a következő kérdéseket kell eldöntenünk, amikor fizikai tervet választunk:

1. Ha a szükséges R relációnak több másolata is létezik, melyikből vegyük R értékét?
2. Ha egy kétrelációs műveletet, például összekapcsolást alkalmazunk R-re és S-re, akkor számos lehetőség közül kell kiválasztanunk egyet. Néhány ezekből:

   a) Lemásolhatjuk S-t az R-t tároló munkaállomásra, és a számítást ott végezzük el.
   b) Lemásolhatjuk R-t az S-t tároló munkaállomásra, és a számítást ott végezzük el.
   c) Lemásolhatjuk R-t és S-t is egy harmadik helyre, és a számítást ott végezzük el.

   Hogy melyik a legjobb választás, az többek között olyan tényezőktől is függ, hogy melyik helyen van elérhető számítási kapacitás, és hogy a művelet eredményét kombináljuk-e egy harmadik munkaállomáson található adattal. Például ha $(R \bowtie S) \bowtie T$ eredményét kell kiszámítanunk, azt a lehetőséget is választhatjuk, hogy R-t és S-t is továbbítjuk a T-t tároló munkaállomásra, és mindkét összekapcsolást ott végezzük el.

Ha az R reláció az R_1, R_2, ..., R_n töredékek képében van több helyre szétosztva, akkor a logikai lekérdezési terv választásakor R helyett mindenhol az

$$R_1 \cup R_2 \cup \ldots \cup R_n$$

uniót kell írnunk. A lekérdezéstől függően aztán jelentősen egyszerűsíthetjük a kifejezést. Például, ha mindegyik R_i a 10.4.1. részben tárgyalt `Eladások` reláció egy-egy töredéke, és minden töredék egy-egy üzlethez van hozzárendelve, akkor a bostoni áruház eladásaival kapcsolatos lekérdezésekben a bostoni töredéken kívül minden más töredéket elhagyhatunk az unióból.

10.4.5. Feladatok

*!! **10.4.1. feladat:** Ebben a feladatban lehetőségünk lesz néhány olyan probléma felvetésére, amelyek az adattöbbszörözési stratégia kiválasztásakor jönnek elő. Tegyük fel, hogy az R relációhoz n munkaállomás férhet hozzá. Az i-edik munkaállomás másodpercenként q_i lekérdezést és u_i módosítást hajt végre R-en, $i = 1, 2, \ldots, n$. A lekérdezés végrehajtásának a költsége az R relációval rendelkező munkaállomáson c, de ha az adott munkaállomás nem tárol másodpéldányt R-ről, ezért a lekérdezést továbbítani kell egy másikra, a költség $10c$. A módosítás végrehajtása egy „helyi" R-en d, de minden távoli példányon $10d$. Ezen paraméterek függvényében, nagy n esetén, hogyan döntenénk el, hogy mely munkaállomások rendelkezzenek R másodpéldányával és melyek ne?

10.5. Osztott véglegesítés

Ebben a fejezetben arról lesz szó, hogy hogyan biztosítható a különböző helyeken végrehajtott alkotórészekből álló osztott tranzakció atomossága. A következő fejezet az osztott tranzakciók egy másik fontos tulajdonságát, a sorbarendezhetőséget tárgyalja. A következő példán keresztül bemutatjuk, hogy milyen problémák merülhetnek fel.

10.15. példa: Vegyük a 10.4. részben említett üzlethálózatot. Tegyük fel, hogy a hálózat igazgatója minden egyes áruházról tudni akarja, hogy mennyi fogkeféje van raktáron, majd ezek alapján kiegyenlíti a készleteket, azaz néhány áruháznak olyan utasítást fog adni, hogy valamelyik másik áruházba helyezze át a fogkefekészletének egy bizonyos részét. Az egész műveletet egyetlen globális T tranzakció végzi el, amelynek több alkotórésze is van: az i-edik üzletben T_i, az igazgató irodájában pedig T_0. T a következő tevékenységeket hajtja végre:

1. Létrehozza T_0 alkotórészt az igazgató munkaállomásán.
2. T_0 minden áruházba üzenetet küld, utasítva őket, hogy hozzák létre a T_i alkotórészeket.
3. Minden T_i végrehajt egy lekérdezést az i-edik üzletben, amelyből megtudja a raktáron lévő fogkefék számát, majd továbbítja ezt az értéket T_0-nak.
4. T_0 a visszajelzett értékek alapján, valamilyen, itt nem tárgyalt algoritmus segítségével meghatározza, hogy a fogkefeállományt hogyan kell átszervezni. Ezután T_0 olyan üzeneteket küld a megfelelő üzleteknek (ebben az esetben a 7-esnek és a 10-esnek), mint „a 10-es számú áruház szállítson át 500 fogkefét a 7-es számú áruházba".
5. Az üzletek a kapott utasítások alapján módosítják a leltárukat, és végrehajtják a szükséges szállításokat. □

10.5.1. Az osztott atomosság támogatása

A 10.15. példában számos dolog elromolhat, és ezek közül sok eredményezheti azt, hogy sérül T atomossága, azaz néhány műveletét végrehajtja a rendszer, de a többit nem. A naplózás és a helyreállítás mechanizmusa, amelyről feltesszük, hogy minden munkaállomáson jelen van, biztosítja az egyes T_i-k atomosságát, de nem biztosítja, hogy maga T is atomosan fusson le.

10.16. példa: Tegyük fel, hogy a fogkeféket újraosztó algoritmus hibás és ennek következtében a 10-es áruháznak több fogkefét kellene átszállítania, mint amennyi a raktárán van. Ezért T_{10} abortálni fog, és a 10-es üzletből nem szállítanak át egyetlen fogkefét sem, és az üzlet leltárát sem módosítják. T_7 viszont nem talál semmi problémát, és miután a várt fogkefeszállítmánynak megfelelően módosította a 7-es üzlet leltárát, véglegessé válik. Most tehát T nemcsak hogy nem volt atomos (mivel T_{10} soha nem fut le), de ráadásul még inkonzisztens állapotban is hagyta az adatbázist: a leltárban szereplő fogkefék száma nem egyenlő a raktárban ténylegesen megtalálható fogkefék számával. □

A problémák egy másik forrását jelenti annak a lehetősége, hogy egy munkaállomás meghibásodik, vagy leszakad a hálózatról az osztott tranzakció futása közben.

10.17. példa: Tegyük fel, hogy T_{10} még válaszol T_0 első kérdésére, vagyis megküldi a raktáron lévő fogkefék számát, de a 10-es üzlet számítógépe ezután működésképtelenné válik, így T_{10} soha nem kapja meg T_0 utasításait. Véglegessé válhat-e valaha is az osztott T tranzakció? Mit kellene T_{10}-nek tennie, miután a számítógép megjavul? □

10.5.2. Kétfázisú véglegesítés

Annak érdekében, hogy a 10.5.1. részben jelzett problémák elkerülhetők legyenek, az osztott ABKR-ek egy összetett protokollt használnak annak az eldöntésére, hogy egy osztott tranzakció véglegessé váljon-e vagy ne. Ez a *kétfázisú véglegesítés protokoll*[5], és ebben a fejezetben az eljárás alapötletével ismerkedünk meg. Hogy globális döntést hozunk a tranzakció véglegesítéséről, az azt jelenti, hogy vagy minden alkotórésze véglegessé válik, vagy egyetlenegy sem. Ahogy szokásos, most is feltesszük, hogy az egyes munkaállomásokon a lokális alkotórészek vagy véglegessé válnak, vagy nincs hatásuk egyáltalán az adott adatbázisra, vagyis hogy a tranzakció alkotórészei atomosak. Így annak a szabálynak a követésével, hogy az osztott tranzakció vagy minden alkotórésze véglegessé válik, vagy egyik sem, maga az osztott tranzakció is atomossá tehető.

Most következzék néhány kiugróan lényeges dolog a kétfázisú véglegesítés protokollal kapcsolatban:

---

[5] Ne keverjük össze a kétfázisú véglegesítést a kétfázisú zárolással. Ez két, egymástól független elgondolás, egymástól eltérő problémák megoldására.

- Föltesszük, hogy minden munkaállomás naplózza a saját eseményeit, és hogy nincs globális napló.
- Azt is feltesszük, hogy az egyik munkaállomás speciális szerepet játszik annak eldöntésében, hogy az osztott tranzakció véglegessé válhat-e vagy sem. Ezt a munkaállomást *koordinátornak* nevezzük. Koordinátor lehet például az a munkaállomás, ahonnan a tranzakció ered; ez a 10.5.1. rész példáiban a T_0 munkaállomása.
- A kétfázisú véglegesítés protokoll során a koordinátor és a többi munkaállomás bizonyos üzeneteket váltanak egymással. Minden üzenetet küldő munkaállomás naplózza az általa küldött üzeneteket, ezzel segítve az esetleg szükséges helyreállítás műveletét.

Ezeket észben tartva most már jellemezhető a két fázis a munkaállomások közti üzenetcserék leírásával.

Első fázis

A kétfázisú véglegesítés első fázisában a T osztott tranzakció koordinátora eldönti, hogy mikor kísérli meg T véglegesítését. A kísérlet feltehetően akkor történik meg, amikor a koordinátornál futó alkotórész már készen áll a véglegesítésre, de elvben a lépéseket akkor is végre kell hajtani, ha ez az alkotórész abortálni szándékozik (persze nyilvánvaló egyszerűsítések mellett, ahogy ezt majd látni fogjuk). A koordinátor a T tranzakció alkotórészeihez tartozó összes munkaállomást megszavaztatja arról, hogy a véglegesítés vagy az abortálás mellett van-e.

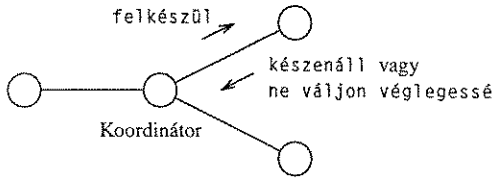
1. A koordinátor a saját naplójába felveszi a <T `Felkészül`> bejegyzést.
2. A koordinátor minden munkaállomásra (elméletileg a sajátjára is) elküldi a T `felkészül` üzenetet.
3. Minden munkaállomás, amely megkapta a T `felkészül` üzenetet, eldönti, hogy a nála található T alkotórészt véglegesíteni vagy abortáltatni akarja. A döntés késleltethető, ha az adott alkotórész még folyamatban van, de a munkaállomásnak végül vissza kell jeleznie.
4. Ha a munkaállomás véglegesíteni akarja az alkotórészt, akkor ennek az *előzetesen véglegesített* állapotba kell lépnie. Az alkotórészt ebben az állapotban a munkaállomás már csak akkor abortáltathatja, ha erre a koordinátortól utasítást kap. Hogy T alkotórésze az előzetesen véglegesített állapotba jusson, a következőket kell megtenni:

   a) Az összes olyan lépést végre kell hajtani, amely annak a biztosításához szükséges, hogy T lokális alkotórészének ne kelljen majd abortálnia, a munkaállomáson bekövetkező rendszerhibát követő helyreállítás során sem. Így nem csak a lokális T műveleteit kell elvégezni, hanem a megfelelő naplózási műveleteket is, hogy egy esetleges helyreállítás során T hatását ne semmisítsük meg, legfeljebb újra futtassuk a tranzakciót. A tényleges műveletek a naplózási módszertől függenek, de a lokális T működéséhez tartozó naplóbejegyzéseket mindeneképpen ki kell írni a lemezre.
   b) Fel kell venni a <T `Készenáll`> bejegyzést a lokális naplóba, és a naplót ki kell írni a lemezre.
   c) El kell küldeni a koordinátornak a T `készenáll` üzenetet.

T alkotórészét azonban még nem véglegesíti ebben a lépésben a munkaállomás; ezzel várnia kell a második fázisig.

1. Ha a munkaállomás inkább abortáltatni akarja az alkotórészt, akkor naplózza a <T `Nem Válik Véglegessé`> bejegyzést és elküldi a T `ne váljon véglegessé` üzenetet a koordinátornak. Biztonságos már ebben a lépésben abortáltatni az alkotórészt, hiszen T is biztosan abortál, még akkor is, ha csak egyetlenegy alkotórész szavazott a véglegesítés ellen.

Az első fázisban váltott üzeneteket a 10.15. ábrán foglaltuk össze.



10.15. ábra. *A kétfázisú véglegesítés első fázisában váltott üzenetek*

Második fázis

A második fázis akkor kezdődik, amikor a koordinátor mindegyik munkaállomástól megkapta a `készenáll` vagy a `ne váljon véglegessé` üzenetet. Lehetséges azonban, hogy valamelyik munkaállomás nem válaszol, talán mert meghibásodott, vagy leszakadt a hálózatról. Ebben az esetben a koordinátor egy megfelelő időkorlát túllépése után úgy fogja venni, mintha ez a munkaállomás a `ne váljon véglegessé` választ küldte volna.

1. Ha a koordinátor a T `készenáll` üzenetet kapta T minden alkotórészétől, akkor a T véglegesítése mellett fog dönteni. A koordinátor

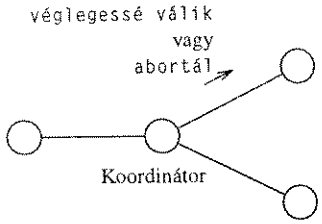
   a) Felveszi a saját naplójába a <`Commit` T> bejegyzést, és
   b) Elküldi a T `véglegessé válik` üzenetet minden T-hez tartozó munkaállomásra.

2. Ha a koordinátor egy vagy több munkaállomástól a `ne váljon véglegessé` üzenetet kapta, akkor

a) Felveszi a saját naplójába az <Abort *T*> bejegyzést, és
b) Elküldi a *T* abortál üzenetet minden *T*-hez tartozó munkaállomásra.

3. Ha egy munkaállomás *T* véglegessé válik üzenetet kap, akkor véglegesíti *T* alkotórészét, és felveszi a naplóba a <Commit *T*> bejegyzést.
4. Ha egy munkaállomás *T* abortál üzenetet kap, akkor abortáltatja *T* alkotórészét, és felveszi a naplóba az <Abort *T*> bejegyzést.

A második fázis üzeneteit a 10.16. ábrán foglaltuk össze.



10.16. ábra. *A kétfázisú véglegesítés második fázisában váltott üzenetek*

10.5.3. Az osztott tranzakciók helyreállítása

A kétfázisú véglegesítés folyamata alatt bármikor meghibásodhat egy munkaállomás. Biztosítanunk kell, hogy ami a helyreállításkor történik, az megfelel annak a döntésnek, amelyet a *T* osztott tranzakcióról hoztunk. Attól függően, hogy az adott helyen mi a *T*-vel kapcsolatos utolsó naplóbejegyzés, a következő eseteket kell végiggondolnunk:

1. Ha a *T*-re vonatkozó utolsó naplóbejegyzés <Commit *T*>, akkor a koordinátor biztosan véglegesítette *T*-t. A naplózási módszertől függően szükség lehet arra, hogy a helyreállítás során *T* alkotórészét újra végrehajtsák.
2. Ha az utolsó feljegyzés <Abort *T*>, akkor az előző esethez hasonlóan itt is ismerjük a globális döntést: *T* abortált. Ha a naplózási módszer igényli, a *T* alkotórész hatását meg kell semmisíteni.
3. Ha az utolsó bejegyzés <*T* Nem válik véglegessé>, akkor a globális döntés biztosan *T* abortálása volt. Ha szükséges, *T* hatását a lokális adatbázison megsemmisítjük.
4. Nehéz a dolgunk, ha *T*-vel kapcsolatban az utolsó feljegyzés <*T* Készenáll>. Ilyenkor a magát helyreállítani próbáló munkaállomás nem tudja, hogy mi lehetett a globális döntés, ezért ezt egy másik munkaállomástól kell megtudnia. Kommunikálhat a koordinátorral, ha az éppen nem működésképtelen, a koordinátor meghibásodása esetén azonban megkérdezhet egy másik munkaállomást is, amely majd a naplója alapján megadja a szükséges információt. Legrosszabb esetben, ha semelyik másik munkaállomással sem tud kapcsolatba lépni, *T* lokális alkotórészét

függőben kell tartania addig, amíg meg nem tudja állapítani, hogy mi volt a *T*-re vonatkozó véglegesítés/abortálás döntés.[6]
5. Az is előfordulhat, hogy a lokális napló egyáltalán nem tartalmaz a kétfázisú véglegesítés protokollal kapcsolatban *T*-re vonatkozó feljegyzést. Ebben az esetben az önhelyreállítást végző munkaállomás egyoldalúan a *T* alkotórész abortáltatása mellett dönthet, ami minden naplózási módszernek megfelel. Lehetséges, hogy a koordinátor egyébként is abortáltatni akarta *T*-t a meghibásodott munkaállomás időkorlát-túllépése miatt. Ha ez csak rövid ideig volt működésképtelen, akkor a többi helyen *T* még mindig aktív lehet, de a lokális *T* alkotórész abortáltatása soha nem okoz problémát, ha a munkaállomás a később kezdődő első fázisban a *T* ne váljon véglegessé választ adja.

A fenti elemzésben feltettük, hogy a meghibásodott munkaállomás nem a koordinátor volt. Ha a koordinátor válik működésképtelenné a kétfázisú véglegesítés folyamata alatt, akkor új problémák merülnek fel.

Először is, a túlélő résztvevőknek vagy meg kell várniuk, amíg a koordinátort helyreállítják, vagy új koordinátort kell választani. Mivel nem lehet előre tudni, hogy a koordinátor mikor lesz újra üzemképes, erős az indíttatás az új vezető választására, legalábbis abban az esetben, amikor a koordinátor még egy rövid várakozási idő eltelte után sem állt helyre.

A *vezetőválasztás* az osztott rendszerek egy méltán összetett problémája – részletesebb vizsgálata túlmutat e könyv célkitűzésein. Egy egyszerű módszer azonban a legtöbb esetben működni fog. Például feltehető, hogy minden résztvevő munkaállomás rendelkezik egy egyedi azonosítóval; az IP-cím sok helyzetben megfelel erre a célra. Minden résztvevő az összes többi résztvevőnek küld egy üzenetet a saját azonosítójával, jelezvén, hogy őt meg lehet választani vezetőnek. Megfelelő idő eltelte után minden résztvevő a legalacsonyabb azonosítószámmal rendelkező munkaállomást ismeri el új koordinátorként azok közül, amelyekről hallott, és erről minden más munkaállomásnak értesítést küld. Ha minden résztvevő azonos üzenetet kap, akkor egyértelmű és mindenki számára ismeretes az új koordinátor személye. Ha az üzenetek nem egyeznek meg egymással, vagy egy résztvevő nem válaszolt, erről szintén mindenki tud majd, és a választást újrakezdhetik.

Most az új vezető szavazás formájában információt gyűjt a munkaállomásoktól a *T* osztott tranzakcióról. Minden munkaállomás elküldi a naplójában található utolsó bejegyzést *T*-vel kapcsolatban, ha van ilyen. A lehetséges esetek a következők:

1. Valamelyik munkaállomás naplójában szerepel a <Commit *T*> bejegyzés. Az eredeti koordinátor bizonyára a *T* véglegessé válik üzenetet akarta mindenhova elküldeni, tehát *T* véglegesítése egy biztonságos megoldás.
2. Hasonlóan, ha valamelyik munkaállomás naplójában az <Abort *T*> bejegyzés sze-

[6] Ez az állapot, *T* blokkolódása az adott csomópontban, akkor is bekövetkezik, ha minden elérhető csomópontban a *T*-vel kapcsolatos utolsó feljegyzés <*T* Készenáll>. *A szerkesztő megjegyzése.*

repel, akkor az eredeti koordinátor biztos, hogy abortáltatni akarta T-t, tehát biztonságos lépés, ha az új koordinátor is ezt teszi.

3. Most tegyük fel, hogy egyetlen munkaállomás naplójában sem szerepel a <Commit T> vagy az <Abort T> bejegyzés, és legalább egy munkaállomás naplójában a <T Készenáll> feljegyzés sem található meg. Mivel az elküldött üzeneteket még a továbbítás előtt naplózzák, világos, hogy a régi koordinátor ettől a munkaállomástól nem kapott T készenáll üzenetet, ezért nem dönthetett T véglegesítése mellett. Az új koordinátor tehát nyugodt lélekkel abortáltathatja T-t.
4. Nehéz a dolgunk abban az esetben, ha egyik helyen sem található <Commit T> vagy <Abort T>, viszont minden túlélő munkaállomás naplójában szerepel a <T Készenáll> bejegyzés. Ilyenkor nem lehetünk biztosak abban, hogy a régi koordinátornak nem volt valamilyen oka T abortáltatására; a saját munkaállomásán történt események, vagy egy másik, jelenleg hibás munkaállomás T ne váljon véglegessé üzenete például megfelelő alapot nyújthatott ehhez a döntéshez. De az is lehet, hogy véglegesíteni akarta T-t, és a saját alkotórésze már véglegessé is vált. Vagyis az új koordinátor nem tudja eldönteni, hogy mit csináljon T-vel, ezért meg kell várnia, amíg az eredeti koordinátor helyreáll. A valódi rendszerekben az adatbázis-adminisztrátornak megvan a lehetősége arra, hogy közbelépjen és manuálisan kényszerítse a várakozó tranzakciókat a továbblépésre. Lehetséges, hogy ezzel sérül a tranzakció atomossága, de a blokkolt tranzakciót végrehajtó személy ennek tudatában megfelelően ellensúlyozza majd ezt a hatást.

10.5.4. Feladatok

! **10.5.1. feladat:** Vegyünk egy olyan T tranzakciót, amelyet egy otthoni számítógépről indítanak, és a segítségével a B bank egy számlájáról 10 000 dollárt akarnak átutalni egy C bankbeli számlára.

* a) Melyek lesznek az osztott T tranzakció alkotórészei? Mi a feladata a B-ben, illetve C-ben található alkotórésznek?

b) Milyen hibát okozhat az, ha a B bankban vezetett számlán nincs 10 000 dollár?

c) Milyen problémát okoz, ha az egyik vagy mindkét bank számítógépe meghibásodik, vagy ha a hálózati kapcsolat megszűnik?

d) Ha a c)-ben felsoroltak közül valamelyik bekövetkezik, hogyan folytatódna helyesen a tranzakció, amikor a számítógépek, illetve a hálózat rendbe jön?

10.5.2. feladat: Ebben a feladatban valahogy jelölnünk kell a kétfázisú véglegesítés folyamata során váltott üzeneteket. Jelentse (i, j, M) a következőt: i munkaállomás az M üzenetet küldi a j munkaállomásnak, ahol M az alábbi értékeket veheti fel: P (jelentése felkészül), R (készenáll), D (ne váljon véglegessé), C (commit), A (abort). Nézzünk meg egy egyszerű esetet, ahol a 0-s sorszámú munkaállomás a koordinátor, amelynek a tranzakcióban máskülönben nincs része, az 1-es és a 2-es munkaállomásokon pedig fut egy-egy alkotórész. Ekkor a tranzakció sikeres véglegesítése során például a következő üzenetsorozat jöhet létre:

$(0, 1, P)$, $(0, 2, P)$, $(2, 0, R)$, $(1, 0, R)$, $(0, 2, C)$, $(0, 1, C)$

* a) Adjunk példát egy olyan üzenetsorozatra, amely akkor jön létre, ha az 1-es munkaállomás a véglegesítés, a 2-es pedig az abortálás mellett van!

*! b) Hány, a fentihez hasonló lehetséges üzenetsorozat jöhet létre, ha a tranzakció sikeresen véglegessé válik?

! c) Hány különböző üzenetsorozat jöhet létre, ha az 1-es munkaállomás a véglegesítés, a 2-es viszont az abortálás mellett van, és feltehető, hogy nem fordul elő semmiféle hibajelenség?

! d) Hány különböző üzenetsorozat jöhet létre, ha az 1-es munkaállomás a véglegesítés mellett szavaz, viszont a 2-es valamilyen meghibásodásnál fogva nem válaszol az üzenetekre?

!! **10.5.3. feladat:** Az előző feladat jelöléseit használva tegyük fel, hogy a koordinátor mellett n másik munkaállomásunk van, és ezek adják a tranzakció alkotórészeit. Adjuk meg n függvényében, hogy hány különböző üzenetsorozat jöhet létre abban az esetben, ha a tranzakció sikeresen véglegessé válik!

10.6. Osztott zárolás

Ebben a fejezetben azzal fogunk foglalkozni, hogy hogyan lehet a zárolás alapú ütemezőket kiterjeszteni egy olyan környezetben, ahol a tranzakciók meg vannak osztva, és különböző helyeken futó alkotórészekből állnak. Feltesszük, hogy a zártáblákat az egyes munkaállomások külön-külön kezelik, és hogy az alkotórész csak azokat az adatbáziselemeket zárolhatja, amelyek azon a helyen megtalálhatók.

Amikor többszörözött adatokkal van dolgunk, úgy kell rendeznünk, hogy egy X adatbáziselem minden másodpéldánya egyformán tükrözze az X-en végrehajtott változtatásokat. Ezért különbséget kell tennünk az X *logikai* adatbáziselem és az X egy vagy több másolatának a zárolása között. Ebben a részben megismerkedünk egy költségmodellel, amelyet az osztott zárolási algoritmusokhoz fejlesztettek ki, és amely akkor is alkalmazható, ha nem többszörözött adatokat használunk. Ennek tárgyalása előtt azonban bemutatjuk a központi zárolás technikáját, amely egy kézenfekvő (és néha éppen megfelelő) megoldás az osztott zárolás problémájára.

10.6.1. Központosított zárolási rendszerek

Talán a legegyszerűbb módszer az, ha a munkaállomások közül kijelölünk egyet, hogy az legyen a *zárállomás* (lock site), azaz hogy az tartsa nyilván a logikai elemek zártábláit, függetlenül attól, hogy rendelkezik-e az elemek másodpéldányával vagy sem. Ha egy tranzakció zárolni akar egy X logikai elemet, akkor ezt az igényét a zárállomásnak nyújtja be, az pedig az adott helyzettől függően vagy engedélyezi, vagy

megtagadja a zárolást. Mivel X globális zárolása megegyezik X lokális zárolásával a zárállomáson, biztosak lehetünk benne, hogy a globális zárak helyesen működnek, feltéve, hogy a lokális zárak adminisztrációja a hagyományos módon történik. A költség általában három üzenet zárolásonként (igénylés, engedélyezés, feloldás), ha a tranzakció nem a zárállomáson fut.

Hogy csak egyetlen zárállomást használunk, ez néhány esetben megfelelő lehet, de ha sok munkaállomáson egyszerre sok tranzakció fut, a zárállomásnál hamar kialakulhat a torlódás. Ráadásul, ha az állomás összeomlik, ez teljesen lebénítja a rendszert, hiszen ilyenkor egyetlen tranzakció sem juthat hozzá semelyik zárhoz, függetlenül attól, hogy éppen hol fut. Ezen problémák miatt számos más módszer is született az osztott zárolás megoldására; ezekre a költségbecslés tárgyalása után kerül sor.

10.6.2. Költségmodell az osztott zárolási algoritmusokhoz

Tegyük fel, hogy minden adatelem pontosan egy helyen fordul elő (vagyis hogy nincs adattöbbszörözés), és hogy az egyes munkaállomásokon működő zárkezelő tartja nyilván az ott elérhető adatelemek zárjait és az azokra benyújtott igényeket. Lehetnek osztott tranzakciók a rendszerben, és mindegyikük egy vagy több különböző helyen futó alkotórészből áll.

A zárkezeléshez ugyan többféle költség is kapcsolódik, de sokuk rögzített, független a hálózaton keresztül történt zárigénylés módjától. Az egyetlen költségtényező, amelyet befolyásolni tudunk, a zárak kiosztásakor és feloldásakor váltott üzenetek száma. Ezért a különböző zárolási sémáknál mindig ezt fogjuk figyelni azon feltevés mellett, hogy az igényelt zárakat mindig ki is osztják. Persze lehet, hogy a zárkezelő nem engedélyezi a zárolást, és ezzel további üzenetek küldésére kerül sor a zárolás megtagadásával, illetve a későbbi engedélyezéssel kapcsolatban. Mivel azonban nem látjuk előre, hogy ez milyen arányban fog előfordulni, és ezt az értéket egyébként sem tudjuk befolyásolni, az összehasonlításban ezeket a pluszüzeneteket nem fogjuk figyelembe venni.

10.18. példa: Ahogy ezt már a 10.6.1. részben említettük, a központi zárolás módszerével a zárkérelmekhez jellemzően három üzenet szükséges: egy az igénylés bejelentéséhez, egy a központi zárállomástól a zár kiadásához, és a harmadik a zár feloldásához. Kivételt a következő esetek jelentenek:

1. Nincs szükség az üzenetekre, ha maga a központi zárállomás igényli a zárolást.
2. További üzenetek küldésére van szükség, ha az első kérelmet nem lehet teljesíteni.

Feltesszük azonban, hogy mindkét eset viszonylag ritkán fordul elő, azaz a legtöbb zárigénylés a központi zárállomástól különböző helyről fut be, és hogy a kérelmek többsége kielégíthető. A zárankénti három üzenet tehát egy jó becslés a központi zár módszer költségére. □

Most vizsgáljunk meg a központi zárolásnál egy rugalmasabb helyzetet, ahol minden X adatbáziselemhez a saját munkaállomásán tartjuk nyilván a hozzá tartozó zárat.

Úgy tűnhet, hogy mivel az X-et zárolni kívánó tranzakció az X munkaállomásán is rendelkezik egy alkotórésszel, a munkaállomások közötti üzenetcserére nincs is szükség: az alkotórész egyszerűen az adott munkaállomás zárkezelőjével tárgyal X-t illetően. Ha azonban az osztott tranzakciónak több elemet is zárolnia kell, mondjuk X-t, Y-t és Z-t is, akkor nem fejezheti be addig a számításait, amíg meg nem szerzi mindhárom elem zárját. Ha X, Y és Z különböző munkaállomásokon található, akkor az ott futó alkotórészeknek legalább szinkronizáló üzeneteket kell cserélniük, hogy a tranzakció nehogy „előbbre járjon saját magánál".

Az összes lehetséges variáció áttekintése helyett csak egy egyszerű modellt veszünk arra, hogy a tranzakciók hogyan gyűjtik be a szükséges zárakat. Feltesszük, hogy minden tranzakciónak az egyik alkotórésze, a *zárkoordinátor* felelős az egyes alkotórészek által igényelt zárak összegyűjtéséért. A zárkoordinátor a saját munkaállomásán üzenetcsere nélkül zárolja az elemeket, de egy másik helyen található X zárolásához három üzenet szükséges:

1. Az igény benyújtása X munkaállomására.
2. A válasz üzenet, amellyel a zárolást engedélyezik (korábban feltettük, hogy az igényeket azonnal kielégítik; ha mégsem, akkor ebben a pontban az üzenet a zárolás elutasítására vonatkozik, amelyet majd később követ az engedélyező üzenet).
3. A zárról való lemondás továbbítása X munkaállomására.

Mivel az osztott zárolási protokollokat csak összehasonlítani akarjuk és nem kívánjuk megadni az üzenetek átlagos számát, ez a leegyszerűsítés megfelel a céljainknak.

Ha azt a munkaállomást választjuk zárkoordinátornak, ahonnan a legtöbb zárat kell beszereznie a tranzakciónak, akkor minimalizáljuk a szükséges üzenetek számát. Ez a többi munkaállomáson található adatbáziselemek számának háromszorosával egyenlő.

10.6.3. Többszörözött elemek zárolása

Óvatosan kell az X adatelem zárolását értelmeznünk, amikor X-ből különböző helyeken másodpéldányok is megtalálhatók.

10.19. példa: Tegyük fel, hogy az X adatbáziselem két példányban létezik, ezek X_1 és X_2. Tegyük fel továbbá, hogy a T tranzakció X_1 munkaállomásán osztott zár alá helyezte X_1-et, U viszont kizárólagos zárat birtokol X_2 munkaállomásán X_2-n. Ilyenkor U csak X_2-t módosíthatja, X_1-et nem, és ez azt eredményezi, hogy X két példánya egymástól eltérő lesz. Sőt mivel T és U más elemeket is zár alá helyezhet, és az, hogy milyen sorrendben olvassák, illetve írják X-et, független a másodpéldányokon birtokolt záraktól, T-nek és U-nak arra is lehetősége van, hogy nem sorba rendezhető módon viselkedjenek. □

A 10.19. példában bemutatott probléma lényege, hogy többszörözött adatok esetén különbséget kell tennünk az X logikai elem osztott, illetve kizárólagos zárolása, és az

X egy másodpéldányának az adott munkaállomáson történő helyi zárolása között. Vagyis annak érdekében, hogy biztosítani lehessen a sorbarendezhetőséget, a tranzakcióknak globálisan kell zárolniuk a logikai elemeket. Viszont a logikai elemek fizikailag nem léteznek – csak a másodpéldányaik –, és globális zártábla sincs. Így a tranzakció csak egyetlen módon juthat hozzá X globális zárjához: ha megszerzi X egy vagy több másodpéldányán az adott munkaállomáson a lokális zárakat. Most olyan módszerekkel fogunk foglalkozni, amelyek segítségével a lokális zárak globális zárakká alakíthatók, és rendelkeznek a szükséges tulajdonságokkal:

- Semelyik két tranzakció sem birtokolhat globális kizárólagos zárat egy X logikai elemen ugyanabban az időben.
- Ha egy tranzakció birtokában van az X logikai elem globális kizárólagos zárja, akkor semelyik másik tranzakció nem birtokolhat globális osztott zárat X-en.
- Bármennyi tranzakció rendelkezhet X globális osztott zárjával, amíg nincs olyan tranzakció, amely birtokolná a globális kizárólagos zárat.

10.6.4. Az elsődleges példány zárolása

A központi zárolás módszere továbbfejleszthető úgy, hogy megosztjuk a zárállomás feladatát, de továbbra is ragaszkodunk ahhoz az elképzeléshez, hogy minden logikai elemhez tartozik egy olyan egyedi munkaállomás, amely az elem globális zárjáért felelős. Ezt az osztott zárolási módszert az *elsődleges példány* (primary copy) módszerének nevezzük. Ezzel a változtatással sikerül a központosított módszer egyszerűségét megőrizni, ugyanakkor elkerülhető annak a lehetősége, hogy a központi zárállomás túlterheltté váljon.

Az elsődleges példány zárolási módszerben minden X logikai elem másodpéldányai közül kijelölünk egyet „elsődleges példánynak". Ha egy tranzakció zárolni szeretné az X logikai elemet, akkor arra a munkaállomásra kell elküldenie az igényét, ahol X elsődleges példánya található. Ez a munkaállomás tartja nyilván X-et a helyi zártáblázatban, és az adott helyzettől függően vagy engedélyezi, vagy megtagadja a zár kiadását. Az előző módszerhez hasonlóan a globális (logikai) zárak helyes működése itt is az elsődleges példányhoz tartozó zárak megfelelő helyi nyilvántartásán múlik.

Ahogy ezt a központi zárállomás esetében láttuk, ennél a módszernél is a legtöbb zárigénylés három üzenettel jár, kivéve, ha a tranzakció az elsődleges példány munkaállomásán fut. Ha azonban az elsődleges példányokat okosan választjuk, várhatóan ez sokkal gyakrabban fog előfordulni, mint az ellenkezője.

10.20. példa: Az üzletláncos példánkban a vásárlásokra vonatkozó adatok elsődleges példányait úgy kellene kiválasztanunk, hogy azok annak az üzletnek az adatbázisában legyenek, ahol maga a vásárlás történt. Az adat többi példánya, amely például a központi részlegben vagy az elemzők által használt adattárházban van, nem elsődleges. A jellemző tranzakciók valószínűleg az áruházakban futnak le, és csak az adott áruházra vonatkozó vásárlási adatokat módosítják. Amikor egy ilyen típusú tranzakció zárolja

Osztott holtpontok

Miközben a tranzakció többszörözött adatot próbál globális zár alá helyezni, számos alkalma van arra, hogy holtpontra jusson. Annak is számos módja van, hogy globális várakozási gráfot készítsünk, és így észlelni tudjuk a holtpontok kialakulását. Osztott környezetben azonban gyakran az a legegyszerűbb és leghatékonyabb megoldás, ha az időkorlátos módszert alkalmazzuk. Bármely tranzakcióról, amely a megfelelő időmennyiség eltelte után még mindig nem zárult le, feltesszük, hogy holtpontra jutott, és ezért visszagörgetjük.

az elemeket, akkor nincs szükség üzenetek küldésére. A zárolással kapcsolatos üzenetekre csak akkor lenne szükség, ha a tranzakció egy másik áruház adatait vizsgálná vagy módosítaná. □

10.6.5. A lokális zártól a globálisig

Egy másik megoldás, ha több, összegyűjtött lokális zárból képzünk globális zárat. Ezekben a sémákban az X adatbáziselem egyik példánya sem „elsődleges" – az elem másolatai szimmetrikusak, és bármelyikükre igényelhető lokális osztott vagy kizárólagos zár. Egy jól működő globális zároló séma nyitja, hogy a tranzakcióknak be kell gyűjteniük bizonyos számú lokális zárat X példányain ahhoz, hogy az X-en lévő globális zárat birtokolhassák.

Tegyük fel, hogy az A adatbáziselem n példányban létezik. Választunk két értéket:

1. s – A példányai közül s számút kell osztott módban zárolni ahhoz, hogy egy tranzakció globális osztott módban zárolhassa A-t.
2. x – A példányai közül x számút kell kizárólagos módban zárolni ahhoz, hogy egy tranzakció globális kizárólagos módban zárolhassa A-t.

Ha $2x > n$ és $s + x > n$, akkor megvannak a kellő tulajdonságok: csak egy globális kizárólagos zár létezhet A-n, és nem létezhet egyszerre globális osztott és globális kizárólagos zár rajta. Ezeket a következőképpen magyarázhatjuk: ha két tranzakció birtokában is lenne egy-egy globális kizárólagos zár A-n, akkor mivel $2x > n$, A-nak legalább egy másodpéldányán mindkét tranzakció lokális kizárólagos zárat birtokolna (mert több lokális kizárólagos zár van kiosztva, mint ahány másodpéldánya van A-nak). Ekkor azonban a lokális zárolási módszer nem működne helyesen. Hasonlóan, ha egy tranzakció globális osztott módban, egy másik pedig globális kizárólagos módban zárolja A-t, akkor, mivel $s + x > n$, valamelyik másodpéldányon az egyikük lokális osztott, a másikuk pedig lokális kizárólagos zárat birtokolna egyszerre.

Általában, a globális osztott, illetve kizárólagos zár megszerzéséhez szükséges üzenetek száma rendre $3s$, illetve $3x$. Ez a szám igen nagynak tűnik, összehasonlítva a

központi módszerekkel, ahol záranként átlagosan három vagy annál kevesebb üzenet szükséges. Vannak azonban ezt ellensúlyzó tulajdonságai a rendszernek speciális (s, x) választása esetén, ahogy ezt a következő két példában látjuk:

- *Egy-olvasás-zár; Minden-írás-zár.* Itt $s = 1$, $x = n$. A globális kizárólagos zár megszerzése nagyon drága, de globális osztott zár esetén legfeljebb három üzenet is elég. Ráadásul ennek a sémának van egy előnye az elsődleges példány módszerével szemben: míg az utóbbival elkerülhető az üzenetek küldése, ha az elsődleges példányt olvassuk, addig az egy-olvasás-zár sémával ugyanez megtehető, valahányszor a tranzakció egy olyan helyen fut, ahol megtalálható az olvasni kívánt adatbáziselem *akármelyik példánya*. Így ez a séma jobban megfelelhet, amikor a tranzakciók többsége csak olvas, de az X-et olvasó tranzakciók más-más helyeken futnak. Példának hozható fel az osztott digitális könyvtár, amely egy dokumentum másodpéldányait azokon a helyeken tárolja el, ahol azokat gyakran olvassák.
- *Többségi zárolás.* Itt $s = x = \lceil (n+1)/2 \rceil$. Úgy tűnik, ebben a rendszerben a tranzakció helyétől függetlenül mindenképpen sok üzenetre lesz szükség. Van azonban számos más tényező, amely figyelembe vételével már elfogadhatóvá válik a séma. Ezek közé tartozik, hogy sok hálózati rendszer támogatja a *csomagszórást*[7], amely segítségével lehetővé válik, hogy a tranzakció egyetlen általános igény elküldésével próbálja begyűjteni az X elem lokális zárjait, hiszen ez az üzenet minden munkaállomáshoz eljut. Hasonlóan, a zárak feloldásához is elég lehet egyetlen üzenet. A módszer javára írandó még az is, hogy s és x értékének fenti megválasztása az üzenetek nagy számának ellenére olyan előnnyel jár, amely más értékek esetén nem érhető el: lehetőség van a rendszer részleges működésére, még akkor is, ha a hálózat több részre szakadt. Amíg a hálózatnak van olyan része, amely az X másodpéldányait tároló munkaállomások többségét tartalmazza, addig lehetősége van a tranzakciónak arra, hogy zárolni próbálja X-et. Ha más, a hálózatról leszakadt munkaállomások aktívak is maradnak, még osztott zárat sem kaphatnak X-en, így nincs veszélye annak, hogy a hálózat különböző (egymástól elszakadt) részein futó tranzakciók nem sorba rendezhető módon fognak viselkedni.

10.6.6. Feladatok

! **10.6.1. feladat:** Megmutattuk, hogyan lehet lokális osztott, illetve kizárólagos zárból a globális változatot létrehozni. Hogyan hoznánk létre a megfelelő típusú lokális zárból a következőket:

* a) Globális osztott, kizárólagos és növelési zárakat.
b) Globális osztott, kizárólagos és módosítási zárakat.
!! c) Globális osztott, kizárólagos és mindenféle típusú szándékjelölő zárakat.

[7] Angolul *broadcast*. Olyan üzenettovábbítás, amely a hálózat minden elemének szól. *A fordító megjegyzése.*

10.6.2. feladat: Tegyük fel, hogy van öt munkaállomás, amelyek mindegyike rendelkezik az X adatbáziselem egy-egy másodpéldányával. Ezek közül P a legfontosabb X-re nézve, és az elsődleges példány osztott zárolási rendszerben ezt használjuk X elsődleges munkaállomásaként. Az X elérésére vonatkozó statisztikák a következők:

i) A hozzáférések 50%-a P-ből ered és ezek során X-et csak olvassák.
ii) A többi négy munkaállomás mindegyike egyenként 10%-ban igényel hozzáférést, és ezek csak olvasó-hozzáférések.
iii) A hozzáférések maradék 10%-a kizárólagos, és az öt munkaállomás közül akármelyik igényelheti egyenlő valószínűséggel (azaz mindegyik 2%-ban igényli).

Az alábbi zárolási módszerek mindegyikéhez adjuk meg az egyes zárak megszerzéséhez szükséges üzenetek átlagos számát! Feltehető, hogy minden kérelmet jóváhagynak, azaz hogy elutasító üzenetekre nem lesz szükség.

* a) Egy-olvasás-zár; minden-írás-zár.
b) Többségi zárolás.
c) Elsődleges példány zárolás, az elsődleges példány P-nél található.

10.7. Hosszú tranzakciók

Az adatbázis-alkalmazásoknak van egy olyan csoportja, ahol az adatok nyilvántartásához megfelel az adatbázisrendszer, de a sok, rövid tranzakciós modell, amelyre az adatbázis konkurenciavezérlő mechanizmusai alapoznak, alkalmatlan. Ebben a fejezetben ilyen alkalmazásokkal és a velük kapcsolatban felmerülő problémákkal foglalkozunk, majd bemutatjuk a „kiegyenlítő tranzakciókon" alapuló megoldást. Ezek a tranzakciók érvénytelenítik az olyan véglegesített tranzakciók hatását, amelyeknek nem kellett volna véglegessé válniuk.

10.7.1. A hosszú tranzakciók problémái

Durván fogalmazva a *hosszú tranzakció* egy olyan tranzakció, amely túl sokáig tart ahhoz, hogy megengedhető legyen számára az, hogy olyan elemeket tartson zár alatt, amelyekre más tranzakcióknak is szükségük van. A „túl hosszú" a környezettől függően jelenthet másodperceket, perceket vagy órákat; mi feltesszük, hogy egy „hosszú" tranzakció legalább percekig, talán órákig is eltart. A hosszú tranzakciókkal kapcsolatos alkalmazások három nagy osztálya a következő:

1. *Hagyományos ABKR-alkalmazások.* A közönséges adatbázis-alkalmazások főleg rövid tranzakciókat futtatnak, sok esetben azonban szükség van alkalmanként hosszú tranzakciókra is. Például egy tranzakció végigvizsgálhatja egy bank összes

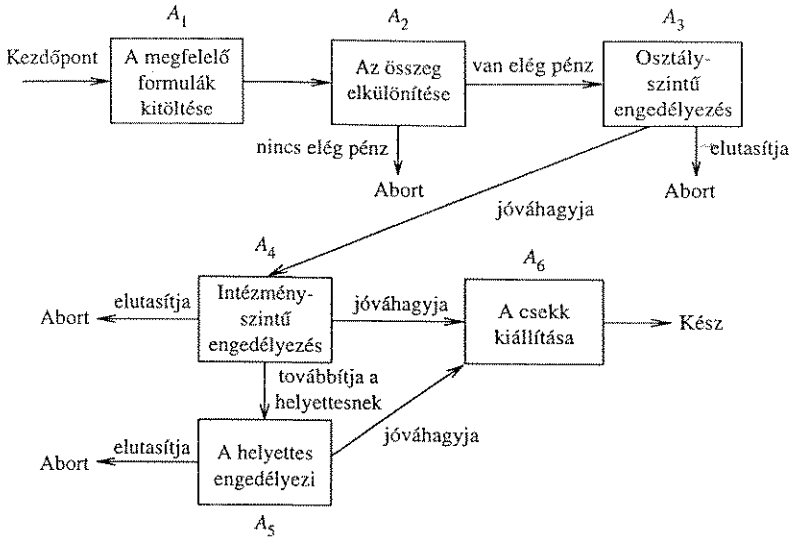
számláját, hogy megállapítsa, hogy a végösszeg helyes-e, egy másik alkalmazásban pedig alkalmanként szükség lehet egy-egy index újraszervezésére, hogy a teljesítmény továbbra is maximális legyen.

2. *Tervezőrendszerek.* Függetlenül attól, hogy a tervezni kívánt eszköz mechanikus (például gépkocsi) vagy elektronikus (például mikroprocesszor vagy programrendszer), a tervezőrendszerek egy dologban mindig megegyeznek: a terv alkotóelemeire van szétbontva (például fájlokra egy felhasználói program esetén), és a különböző részeken különböző tervezők dolgoznak egyszerre. Nem akarjuk viszont, hogy két tervező egyszerre ugyanazon a fájlon dolgozzon, hiszen ezzel csak felülírnák egymás munkáját, ezért egy *ki- és bejelentkező* rendszer segítségével lehetővé tesszük, hogy a tervező „kijelentsen" egy fájlt, majd a változtatások végeztével, talán órákkal vagy napokkal később, bejelentse. Egy tervező azonban valamilyen oknál fogva talán akkor is bele akar nézni egy fájlba, amikor azon éppen egy másik tervező dolgozik. Ha a fájl kijelentése egyenértékű lenne egy kizárólagos zárolással, akkor néhány indokolt és ésszerű műveletet talán csak napokkal később lehetne végrehajtani.
3. *Munkafolyamat-rendszerek.* Ezek a rendszerek olyan folyamatokat foglalnak magukban, amelyek közül néhányat egy szoftver hajt végre egyedül, néhánynak emberi beavatkozás is része, és néhány talán csak ember által végrehajtott műveletekből áll. Rövidesen láthatunk egy példát, amely egy számla kifizetésével járó hivatalos papírmunkát mutat be. Az ilyen alkalmazások végrehajtása napokig is eltarthat, és ez alatt az idő alatt lehet, hogy néhány adatbáziselemen változtatni kell. Ha a rendszer engedélyezné a tranzakcióhoz szükséges adatok kizárólagos zárolását, akkor a többi tranzakció napokig nem férhetne hozzájuk.

10.21. példa: Vegyük azt az esetet, amikor egy alkalmazott megpróbálja elszámoltatni az utazási költségeit. Azt szeretné, hogy a cége az A123-as számláról visszafizesse a kiadásait – a költségtérítés folyamata a 10.17. ábrán látható. Az eljárás az A_1 művelettel kezdődik, amikor az utazó titkárnője egy on-line kérdőív kitöltésével leírást ad az utazásról, megadja a terhelendő számla számát és a visszatérítendő összeget. Feltesszük, hogy ebben a példában a számlaszám A123 és az összeg 1000 dollár.

Az utazó számláit fizikailag átküldik az osztályos titkárságra, a kitöltött formula pedig a hálózaton keresztül eljut az A_2 automatikus eljáráshoz. A_2 ellenőrzi, hogy a megadott számlán (A123) van-e elég pénz a költségek (1000 dollár) fedezésére, és ha van, akkor a számlán elkülönít egy ennek megfelelő összeget; vagyis próbaképpen levon a számlán található pénzmennyiségből 1000 dollárt, de erről még nem állít ki csekket. Ha nincs elég pénz a számlán, akkor a tranzakció abortál, és feltehetően majd újraindul, vagy amikor már rendelkezésre áll a kívánt pénzösszeg, vagy ha egy másik számláról kérik a kifizetést.[8]

[8] Természetesen az utazó (aki úgysem a stanfordi egyetemen dolgozik) *soha* sem írná a költségeit – helytelenül – egy másik kormányszámla terhére, hanem megfelelő pénzügyi forrásokat keresne. Ezt azért kell hangsúlyoznunk, mert a kormány számvevői, akiknek fogalmuk sincs arról, hogy egy egyetemnek hogyan kéne működnie, még mindig itt nyüzsögnek Stanford körül.



10.17. ábra. *Munkafolyamat-ábra az utazási költségtérítésekhez*

Az A_3 műveletet, amelyre lehet, hogy csak napokkal később kerül sor, az osztály ügyintézője hajtja végre: megvizsgálja a benyújtott számlákat és az on-line formulát. Ha mindent rendben talál, a kérvényt jóváhagyja, és a számlákkal együtt továbbküldi egy magasabb adminisztrációs szintre, az intézményes ügyintézőhöz. Ellenkező esetben a tranzakció abortál, az utazónak pedig feltehetően módosított formában újra be kell adnia a jelentkezését.

Az A_4 művelet során, amelyre lehet, hogy csak napokkal később kerül sor, az intézményes ügyintéző vagy jóváhagyja, vagy elutasítja a kérelmet, vagy továbbadja a helyettesének, aki az A_5 műveletben dönt. Ha a kérvényt elutasítják, akkor a tranzakció abortál, és a jelentkezést újra be kell adni. Jóváhagyás esetén az A_6 művelet során kiállítják a csekket, és véglegesítik az 1000 dollár levonását a számláról.

Tegyük fel, hogy ez a munkafolyamat csak a hagyományos zárak használatával valósítható meg. Ekkor például, mivel a sikeresen lezárult tranzakció megváltoztathatja az A123-as számla egyenlegét, az A_2 műveletnél kizárólagos zár alá kell helyezni a számlát, és azt addig nem szabad feloldani, amíg a tranzakció nem abortál, vagy az A_6 művelet le nem zárul. A zárat talán napokig fenn kell tartani, amíg a kérvény elbírálásával megbízott emberek végre foglalkozni tudnak az üggyel. Ilyenkor viszont az A123-as számláról semmilyen más kifizetés sem történhet, még próbaképpen sem. Másrészről viszont, ha a számlához való hozzáférést egyáltalán nem szabályozzuk, akkor különböző tranzakciók egyszerre terhelhetik a számlát, illetve foglalhatnak le a számlán különféle összegeket, ez pedig a hitelkeret túllépéséhez vezethet. A két szélsőséges módszer helyett tehát egy kompromisszumos megoldást kell találni. □

10.7.2. Regék

A *rege* (saga) olyan műveletek összessége, amelyek együtt egy hosszú „tranzakciót" alkotnak (ilyen műveleteket láttunk a 10.21. példában). A rege tehát a következő részekből áll:

1. Valamilyen műveletek összessége.
2. Egy gráf, amelyben a csúcspontok vagy műveletek, vagy a speciális *Abort*, illetve *Kész* csúcsok, az élek pedig a csúcsok között futnak. A kétféle speciális csúcsból, amelyeket *végpontoknak* (terminal node) nevezünk, nem indulnak ki élek.
3. Azon csúcspont megjelölése, ahonnan az egész folyamat indul. Ez a csúcs a *kezdőpont* (start node).

A kezdőpontból akármelyik végpontba vezető út a műveletek egy lehetséges sorozatát adja. Az *Abort* csúcspontba vezető utak olyan műveletsorozatot jelentenek, amely után az egész tranzakciót vissza kell görgetni, a műveleteknek pedig változatlanul kell hagyniuk az adatbázist. A *Kész* csúcsba vezető utak sikeres műveletsorozatokat jelentenek, és az ezen műveletek által az adatbázisrendszeren véghezvitt változtatásoknak meg kell maradniuk az adatbázisban.

10.22. példa: A 10.17. ábrán látható gráfban az *Abort* csúcsba vezető utak a következők: A_1A_2, $A_1A_2A_3$, $A_1A_2A_3A_4$ és $A_1A_2A_3A_4A_5$. A *Kész* csúcsba vezetnek $A_1A_2A_3A_4A_6$ és $A_1A_2A_3A_4A_5A_6$. Vegyük észre, hogy most a gráf nem tartalmaz kört, vagyis a végpontokba vezető utak száma véges. Általános esetben azonban létrejöhetnek körök a gráfban, és ilyenkor az utak száma végtelen is lehet.[9] □

A regék konkurenciavezérlésére kétféle lehetőség kínálkozik:

1. Minden egyes műveletet tekinthetünk egy-egy önálló (rövid) tranzakciónak, amely a végrehajtáskor egy olyan hagyományos konkurenciavezérlő mechanizmust használ, mint a zárolás. Az A_2 művelet például megvalósítható úgy is, hogy arra a (rövid) időre, amíg az A123-as számla egyenlegét csökkenti az utazási elismervényen jelölt összeggel, zárolja a számlát, majd a zárat feloldja. A zárolás segítségével megelőzhető, hogy egyszerre két tranzakció próbáljon a számla egyenlegének új értéket adni, amivel az első változtatás hatása elveszne, és „csodával határos módon pénz jelenne meg a számlán".
2. Az egész tranzakciót, amelyet a végpontokba vezető utak bármelyike jelenthet, a „kiegyenlítő tranzakciók" mechanizmusán keresztül kezeljük, amelyek a rege csúcspontjaiban található tranzakciók inverzei. Feladatuk az, hogy a véglegessé vált műveletek hatását olyan módon görgessék vissza, amely nem függ attól, hogy mi történt az adatbázissal a véglegesített művelet és a kiegyenlítő tranzakció végrehajtása közötti időben. A kiegyenlítő tranzakciókról a következő részben lesz szó.

[9] A bürokrácia útvesztői... *A fordító megjegyzése.*

Mikor „ugyanaz" két adatbázis-állapot?

A kiegyenlítő tranzakciók tárgyalása közben óvatosnak kell lennünk azzal kapcsolatban, hogy mit jelent az adatbázist „ugyanabba" az állapotba visszaállítani, mint amelyben előtte volt. Már kaptunk egy kis ízelítőt ebből a problémából, amikor a 10.6. példában a B-fák logikai naplózásáról volt szó. Láthattuk, hogy amikor „visszaállítottunk" egy műveletet, a B-fa állapota nem feltétlenül volt azonos a művelet előtti állapottal, viszont ekvivalens volt vele a B-fa elérési műveleteit illetően. Általánosabban fogalmazva, megtörténhet, hogy egy műveletet és a hozzá tartozó kiegyenlítő tranzakciót végrehajtva, az adatbázis nem áll visszabitről bitre ugyanabba az állapotba, amelyben előtte volt, de a különbségeket nem érzékelheti egyetlen, az adatbázis által támogatott felhasználói program sem.

10.7.3. Kiegyenlítő tranzakciók

Egy regében minden A művelethez tartozik egy *kiegyenlítő tranzakció* (compensating transaction), amit A^{-1}-gyel jelölünk. Ha végrehajtjuk A-t, majd később A^{-1}-et, akkor eredményül ugyanazt az adatbázis-állapotot kapjuk, mintha sem A-t, sem A^{-1}-et nem hajtottuk volna végre. Formálisabban fogalmazva:

- Ha D egy tetszőleges adatbázis-állapot, $B_1B_2...B_n$ pedig műveletek és kiegyenlítő tranzakciók sorozata (akár a kérdéses regéből, akár egy másikból vagy akármilyen más tranzakcióból, amely szabályosan fut az adatbázison), akkor ugyanazt az adatbázis-állapotot kapjuk D-ből a $B_1B_2...B_n$ sorozat futtatása után, mint $AB_1B_2...B_nA^{-1}$ futtatása után.

Ha a rege végrehajtása az *Abort* csúcspontba vezet, akkor a visszagörgetés úgy történik, hogy a végrehajtott műveletek fordított sorrendjében lefuttatjuk az egyes műveletekhez tartozó kiegyenlítő tranzakciókat. A kiegyenlítő tranzakciók fent említett tulajdonsága miatt a rege hatása érvénytelenné válik, és az adatbázis-állapota ugyanaz lesz, mintha a regét soha nem is hajtottuk volna végre. Hogy miért biztos az, hogy a visszagörgetett rege hatása érvénytelenné válik, arra részletesebb magyarázatot a 10.7.4. részben találunk.

10.23. példa: Vegyük a 10.17. ábra műveleteit, és nézzük meg, hogy mik lehetnek a megfelelő kiegyenlítő tranzakciók. Elsőként A_1 létrehoz egy on-line dokumentumot. Ha ezt az adatbázisban tároljuk, akkor A_1^{-1}-nek ezt el kell onnan távolítania. Vegyük észre, hogy ez a kiegyenlítés engedelmeskedik a kiegyenlítő tranzakciók alaptulajdonságának: ha létrehozzuk a dokumentumot, és összeállítunk egy α műveletsorozatot (amely akár a dokumentum törlését is tartalmazhatja, ha akarjuk), akkor $A_1\alpha A_1^{-1}$ hatása megegyezik α hatásával.

A_2 implementálásával óvatosan kell bánnunk. A pénzt úgy „foglaljuk le", hogy a megfelelő mennyiséget levonjuk a számlaegyenlegből. Ez a pénz egészen addig törölve marad, amíg az A_2^{-1} kiegyenlítő tranzakció vissza nem helyezi a számlára. Azt állítjuk, hogy A_2^{-1} kiegyenlítő tranzakció helyesen fog működni, ha a számlavezetés általános szabályait követjük. Hogy jobban ráérezzünk a probléma lényegére, hasznos, ha megnézünk egy hasonló tranzakciót, ahol a logikusnak tűnő kiegyenlítés nem működik. A 10.24. példában látunk majd egy ilyen esetet.

Az A_3, A_4 és A_5 műveletek mindegyikében egy jóváhagyással egészül ki az on-line nyomtatvány. A kiegyenlítő tranzakciók tehát egyszerűen eltávolíthatják ezeket a jóváhagyásokat.[10]

Végül A_6-hoz, a csekket kiállító művelethez nem tudunk egyértelműen kiegyenlítő tranzakciót megadni. A gyakorlatban erre nincs is szükség, hiszen ha A_6-ot egyszer már végrehajtottuk, akkor a regét már nem görgethetjük vissza. A_6-nak viszont a szó szoros értelmében úgy sincs hatása az adatbázisra, mivel a csekket fedező összeget a számláról már levonta A_2. Ha az „adatbázist" esetleg tágabb határok között kéne értelmeznünk, ahol a csekk beváltásához hasonló eseményeknek hatásuk van az adatbázisra, akkor A_6^{-1}-et úgy kéne megterveznünk, hogy először próbálja meg a csekket visszavonni, ha ez nem megy, akkor levélben követelje vissza a pénzt attól, aki a csekket beváltotta, végül, ha ez a kísérlet is kudarcba fulladt, nyilatkozzon a behajthatatlan követelés miatti veszteségről, és állítsa vissza az egyenleg eredeti értékét. □

Most fogjunk hozzá a 10.23. példában említett eset felvázolásához, ahol is a számlán végrehajtott változtatást nem lehet kiegyenlíteni a változtatás inverzével. A problémát az okozza, hogy a számlák egyenlege általában nem lehet negatív.

10.24. példa: Tegyük fel, hogy B egy olyan tranzakció, amely 1000 dollárt rak egy olyan számlára, amelyen eredetileg 2000 dollár van, B^{-1} pedig a kiegyenlítő tranzakció, amely ugyanezt az összeget leszedi a számláról. Ésszerű továbbá azt is feltennünk, hogy az a tranzakció, amely egy számláról a rendelkezésre álló összegnél nagyobbat akar eltávolítani, nem mindig futhat le sikeresen. Legyen C egy olyan tranzakció, amely 2500 dollárt emel le ugyanarról a számláról. Ekkor BCB^{-1} nem ekvivalens C-vel. Ennek az az oka, hogy C magában nem fut le sikeresen, így a számlán ott marad 2000 dollár, viszont ha először B-t, majd C-t is végrehajtjuk, a számlán 500 dollár marad, ami után B^{-1} nem zárulhat le sikeresen.

Azt a következtetést kell levonnunk, hogy egy számlák közti tetszőleges átutalásokból álló regét és azt a szokást, hogy a számlák nem mehetnek negatívba, nem lehet egyszerűen kiegyenlítő tranzakciókkal támogatni. Valamilyen változtatást kell bevezetnünk a rendszerbe, például megengedhetjük a negatív egyenleget is. □

[10] A 10.17. ábrán bemutatott regében ezeket a műveleteket csak akkor kell kiegyenlíteni, amikor a nyomtatványt különben is megsemmisítenék. A kiegyenlítő tranzakciók definíciója azonban megköveteli, hogy a tranzakciók egymástól elkülönítve is működjenek, arra való tekintet nélkül, hogy valamelyik másik tranzakció esetleg lényegtelenné teszi az általuk eszközölt változtatásokat.

10.7.4. Miért működnek jól a kiegyenlítő tranzakciók?

Azt mondjuk, hogy két műveletsorozat *ekvivalens* ($\equiv$), ha bármely D adatbázis-állapotot azonos állapotba visznek. A kiegyenlítő tranzakciókról tett alapvető feltevésünk ekkor a következő alakban írható fel:

- Ha A egy művelet, α pedig legális műveletek és kiegyenlítő tranzakciók sorozata, akkor $A\alpha A^{-1} \equiv \alpha$.

Most azt kell megmutatnunk, hogy ha egy $A_1A_2\ldots A_n$ rege végrehajtása után a fordított sorrendben vett kiegyenlítő tranzakciók is lefutnak, akkor akármilyen műveletek is estek a rege műveletei illetve a tranzakciók közé, ennek olyan a hatása, mintha sem a rege műveletei, sem a kiegyenlítő tranzakcióik nem futottak volna le egyáltalán. Ezt az állítást n-re vonatkozó indukcióval bizonyítjuk.

Alap: Ha $n = 1$, akkor az A_1 és az ezt kiegyenlítő A_1^{-1} tranzakció közti műveletsorozat: $A_1\alpha A_1^{-1}$. A kiegyenlítő tranzakciókról való alapfeltevés szerint $A_1\alpha A_1^{-1} \equiv \alpha$; azaz a regének nincs hatása az adatbázis-állapotra.

Indukció: Tegyük fel, hogy az állítás teljesül minden legfeljebb $n-1$ műveletből álló útra. Vegyük most egy n hosszú út műveleteit, amelyeket a fordított sorrendben vett kiegyenlítő tranzakciók követnek, és a műveletek és a tranzakciók közé másféle műveletek is eshetnek. A sorozatot ilyen alakban írhatjuk fel:

$$A_1\alpha_1A_2\alpha_2\ldots\alpha_{n-1}A_n\beta A_n^{-1}\gamma_{n-1}\ldots\gamma_2A_2^{-1}\gamma_1A_1^{-1} \qquad (10.1.)$$

ahol minden görög betű nulla vagy több művelet sorozatát jelöli. A kiegyenlítő tranzakciók definíciója miatt $A_n\beta A_n^{-1} \equiv \beta$. Vagyis (10.1.) ekvivalens a következővel:

$$A_1\alpha_1A_2\alpha_2\ldots A_{n-1}\alpha_{n-1}\beta\gamma_{n-1}A_{n-1}^{-1}\gamma_{n-1}\ldots\gamma_2A_2^{-1}\gamma_1A_1^{-1} \qquad (10.2.)$$

Az indukciós feltevés miatt pedig ez ekvivalens

$$\alpha_1\alpha_2\ldots\alpha_{n-1}\beta\gamma_{n-1}\ldots\gamma_2\gamma_1$$

kifejezéssel, hiszen (10.2.)-ben csak $n-1$ művelet szerepel. Tehát ha a rege után a kiegyenlítését is lefuttatjuk, ez olyan állapotban hagyja az adatbázist, mintha a regét nem is hajtottuk volna végre.

10.7.5. Feladatok

*! **10.7.1. feladat:** Szoftverek „eltávolítását" (uninstalling), mint folyamatot felfoghatjuk az adott szoftver telepítésének (installing) a kiegyenlítő tranzakciójaként is. A telepítés és eltávolítás egy egyszerű modelljében feltehető, hogy egy művelet egy vagy több fájl *feltöltéséből* (loading) áll a forrásról (például CD-ROM) a számítógép merevle-

mezére. Egy *f* fájl feltöltésekor átmásoljuk *f*-et a CD-ROM-ról, felülírva az *f*-fel azonos nevű és útnevű fájlt, ha volt ilyen. Hogy meg tudjuk különböztetni az ilyen értelemben azonos fájlokat, feltehetjük, hogy minden fájl rendelkezik egy időbélyegzővel.

a) Mi a kiegyenlítő tranzakciója *f* fájl feltöltésének? Gondoljuk meg mindkét esetet: amikor létezett ugyanazon a helyen egy ugyanolyan nevű *f'* fájl, illetve amikor nem.
b) Magyarázzuk meg, hogy az a) feladat megoldásaként adott tranzakció miért lesz valóban kiegyenlítő tranzakció! *Segítség:* alaposan gondoljuk át azt az esetet, amikor *f*-fel felülírtuk *f'*-t, majd egy későbbi művelet egy másik, azonos útnevű fájllal felülírta *f*-et.

! **10.7.2. feladat:** Adjunk meg a repülőjegy-foglalás folyamatához egy regét! Vegyük számításba azt a lehetőséget, hogy az ügyfél csak érdeklődik egy jegy iránt, de nem foglalja le. A jegy lefoglalása után az ügyfél lemondhatja a foglalást, de az is lehetséges, hogy nem fizeti ki a jegyet időre, vagy kifizeti, de végül mégsem repül. Minden művelethez adjuk meg a megfelelő kiegyenlítő tranzakciót is!

10.8. Összefoglalás

- *Piszkos adat:* A központi memória puffereiben vagy a lemezen található adatot, amelyet egy, még folyamatban lévő tranzakció írt, „piszkos" adatnak nevezünk.
- *Továbbgyűrűző visszagörgetés:* Olyan naplózás és konkurenciavezérlés együttese esetén, amely megengedi, hogy egy tranzakció piszkos adatot olvasson, szükség lehet az olyan tranzakciók visszagörgetésére, amelyek egy később abortált tranzakció által írt (piszkos) adatokat olvastak.
- *Szigorú zárolás:* A szigorú zárolás elve azt követeli meg a tranzakcióktól, hogy a zárakat (az osztott zárak kivételével) ne csak a tranzakció lezárulásáig tartsák fenn, hanem még azt követően egészen addig, amíg a commit vagy abort naplóbejegyzés ki nem kerül a lemezre. A szigorú zárolás biztosítja, hogy egyetlen tranzakció sem olvas piszkos adatot, még visszamenőlegesen sem egy rendszerhiba és annak helyreállítása során sem.
- *Csoportos véglegesítés:* Gyengíthetünk a szigorú zárolásnak a naplóbejegyzésre vonatkozó feltételén, ha biztosítjuk, hogy a bejegyzések abban a sorrendben kerülnek ki a lemezre, amelyben eredetileg a pufferbe írtuk őket. Ekkor továbbra is garantált, hogy a tranzakciók nem olvasnak piszkos adatot, még egy esetleges hiba és helyreállítása során sem.
- *Az adatbázis állapotának helyreállítása abort után:* Ha egy tranzakció abortál, miután a pufferbe írt, az adatelemek régi értékét a napló vagy az adatbázis lemezen található példánya alapján állíthatjuk vissza. Ha az új értékek már elérték a lemezt, a napló még akkor is használható a régi értékek visszaállítására.
- *Logikai naplózás:* Olyan nagy adatbáziselemek esetén, mint a lemezblokkok, sok helyet takaríthatunk meg, ha a régi és az új értékeket a naplóba növekményesen jegyezzük be, azaz csak a változtatásokat tüntetjük fel. Néhány esetben a változtatások logikai feljegyzése – azaz a blokkok tartalmának absztrakt leírása – lehetővé teszi, hogy egy tranzakció abortálása után az adatbázis állapotát logikailag visszaállítsuk, még akkor is, ha szigorúan ugyanabba az állapotba való visszaállítás lehetetlen.
- *Nézet-sorbarendezhetőség:* Olyan ütemezésekben, ahol a tranzakciók talán olyan értékeket is írnak, amelyeket a későbbiekben egyetlen más tranzakció sem olvas, csak egy következő felülír, a konfliktus-sorbarendezhetőség túl erős feltételnek bizonyul. Egy gyengébb feltétel, amit nézet-sorbarendezhetőségnek nevezünk, csak azt követeli meg, hogy az ekvivalens soros ütemezésben minden tranzakció ugyanabból a forrásból olvassa az adatelemek értékeit, mint az eredeti ütemezésben.
- *Poligráf:* A nézet-sorbarendezhetőség ellenőrzése magában foglalja egy poligráf felépítését, amelyben az élek az értékek az írótól az olvasó irányába haladását jelölik, az élpárok pedig azt a követelményt tükrözik, amely szerint bizonyos írásműveletek nem eshetnek az adott írás- és olvasásművelet közé. Az ütemezés pontosan akkor nézet-sorbarendezhető, ha minden élpárnak elhagyhatjuk az egyik felét úgy, hogy eredményül egy körmentes gráfot kapjunk.
- *Holtpont:* Ez bármikor kialakulhat, amikor a tranzakcióknak olyan erőforrásokra, például zárakra, kell várakozniuk, amelyek az adott pillanatban egy másik tranzakció birtokában vannak. Megfelelő előkészületek nélkül fennáll a veszélye egy várakozási kör kialakulásának, amikor is az ebben részt vevő tranzakciók egyike sem képes a továbblépésre.
- *Várakozási gráf:* Rajzoljunk egy csúcsot minden várakozó tranzakcióhoz, és kössük össze azokkal a tranzakciókkal, amelyekre várakozik. A holtpont kialakulása pontosan azt jelenti, hogy létrejött egy vagy több kör a várakozási gráfban. A holtpontok kialakulása elkerülhető, ha a várakozási gráf nyilvántartásával minden olyan tranzakciót abortáltatunk, amely várakozásával kör jönne létre a gráfban.
- *Holtpontmegelőzés az erőforrások sorbarendezésével:* Ha a tranzakcióktól megkívánjuk, hogy az erőforrásokat valamilyen lexikografikus sorrendben használják fel, akkor ezzel megelőzhető a holtpontok kialakulása.
- *Időbélyegző alapú holtpontmegelőzés:* Más sémák időbélyegzőket vezetnek be, és ezek alapján döntik el, hogy az erőforrást igénylő tranzakciót abortáltassák-e vagy várakoztassák. A megvár-meghal sémában az erőforrást birtokló tranzakciónál idősebb tranzakció várakozik, az újabbakat pedig visszagörgetjük, de az időbélyegzőjük nem változik. A megsebez-megvár sémában az újabb tranzakció várakozik, az idősebb viszont kényszeríti az erőforrást birtokló tranzakciót, hogy mondjon le az erőforrásairól, és később kezdje el újra a működését.
- *Osztott adatok:* Egy osztott adatbázisban az adatok részekre bonthatók vízszintesen (a reláció sorai különböző munkaállomásokra vannak szétszórva), vagy függőlegesen (a reláció sémája több sémára van bontva [dekompozíció], és az ezekhez tartozó relációk különböző munkaállomásokon találhatók). Lehetőség van az adatok többszörözésére is, vagyis arra, hogy a relációnak egymással feltehetőleg azonos másodpéldányai legyenek különböző helyeken.
- *Osztott tranzakciók:* Az osztott adatbázisban egy logikai tranzakció több alkotórészből állhat, amelyek mindegyike különböző munkaállomáson fut. A konziszten-

cia megőrzésének érdekében az összes alkotórésznek egyet kell értenie abban, hogy a logikai tranzakció abortáljon vagy véglegessé váljon.

- *Kétfázisú véglegesítés:* Ez a módszer támogatja a tranzakció-alkotórészek döntéshozatalát a logikai tranzakció lezárását illetően, gyakran még rendszerhiba esetén is. Az első fázisban a koordinátor vezetésével minden alkotórésznek szavaznia kell arról, hogy az abortálás vagy a véglegesítés mellett van-e. A második fázisban a koordinátor pontosan akkor utasítja az alkotórészeket a véglegesítésre, ha mindegyikük azt akarta.
- *Osztott zárak:* Ha a tranzakcióknak olyan adatbáziselemeket kell zárolniuk, amelyek egyszerre több helyen is megtalálhatók, akkor megfelelő módszert kell alkalmaznunk a zárak összehangolására. A központi zárolási módszerben egy munkaállomás tartja nyilván az összes elem zárját. Az elsődleges példány módszerben az egyes elemek „szülőállomása" foglalkozik a zárak nyilvántartásával.
- *Többszörözött adat zárolása:* Amikor egy adatbáziselem többszörözve van, és több helyen is megtalálhatók a példányai, akkor az elem globális zárjához csak egy vagy több másodpéldány zárján keresztül lehet hozzájutni. A többségi zárolási módszerben a globális zár megszerzéséhez a másodpéldányok többségén kell osztott vagy kizárólagos zárat birtokolni. Az is megoldás lehet azonban, ha a globális osztott zár birtoklását már egy példány osztott zár alá helyezése után is lehetővé tesszük, globális kizárólagos zár esetén viszont megköveteljük az összes példány kizárólagos zárolását.
- *Regék:* Amikor a tranzakciók hosszabb időt igénybe vevő lépéseket is tartalmaznak, amelyek órákig vagy akár napokig is eltarthatnak, a hagyományos zárolási mechanizmusok túlságosan is korlátozhatják a konkurenciát. A rege műveletek hálózatából áll, amelyek mindegyike egy vagy több másik művelethez, az egész rege befejezéséhez vagy a rege abortálásához vezet.
- *Kiegyenlítő tranzakció:* Hogy egy regének értelme legyen, egy kiegyenlítő műveletnek kell tartoznia minden művelethez, amely érvényteleníti az első művelet hatását az adatbázis-állapoton, viszont érintetlenül hagy minden olyan más műveletet, amely egy másik, befejeződött vagy éppen működésben lévő regéhez tartozik. A rege abortáltatása esetén a kiegyenlítő műveletek egy megfelelő sorozatát hajtjuk végre.

10.9. Irodalomjegyzék

Az itt tárgyalt témákhoz néhány hasznos, általános forrásmű [2], [1] és [9]. A nézetsorbarendezhetőség és a poligráfteszt [10]-ből való. A holtpontmegelőzésről ad áttekintést [7]; a várakozási gráf is itt szerepel. A megvár-meghal és a megsebez-megvár sémák [11]-ből jönnek.

A kétfázisú véglegesítés protokollt [8]-ban ajánlották. Egy hatékonyabb (a könyvünkben nem tárgyalt) sémáról, amelyet háromfázisú véglegesítés protokollnak nevezünk, [12]-ben olvashatunk. A helyreállítást a vezetőválasztás szempontjából [4]-ben vizsgálják.

Az osztott zárolási módszereket [3]-ban (központi zárolás módszere), [13]-ban (elsődleges példány módszere) és [14]-ben (globális zárak a másodpéldányokon tartott zárakból) javasolták.

A hosszú tranzakciókat [6]-ban vezették be. A regéket [5] írja le.

1. N. S. Barghouti and G. E. Kaiser, „Concurrency control in advanced database applications," *Computing Surveys* **23**:3 (Sept., 1991), pp. 269–318.
2. S. Ceri and G. Pelagatti, *Distributed Databases: Principles and Systems*, McGraw-Hill, New York, 1984.
3. H. Garcia-Molina, „Performance comparison of update algorithms for distributed databases," TR Nos. 143 and 146, Computer Systems Laboratory, Stanford Univ., 1979.
4. H. Garcia-Molina, „Elections in a distributed computer system," *IEEE Trans. on Computers* **C-31**:1 (1982), pp. 48–59.
5. H. Garcia-Molina and K. Salem, „Sagas," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1987), pp. 249–259.
6. J. N. Gray, „The transaction concept: virtues and limitations," *Proc. Intl. Conf. on Very Large Databases* (1981), pp. 144–154.
7. R. C. Holt, „Some deadlock properties of computer systems," *Computing Surveys* **4**:3 (1972), pp. 179–196.
8. B. Lampson and H. Sturgis, „Crash recovery in a distributed data storage system," Technical report, Xerox Palo Alto Research Center, 1976.
9. M. T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems*, Prentice-Hall, Englewood Cliffs NJ, 1999.
10. C. H. Papadimitriou, „The serializability of concurrent updates," *J. ACM* **26**:4 (1979), pp. 631–653.
11. D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II, „System-level concurrency control for distributed database systems," *ACM Trans. on Database Systems* **3**:2 (1978), pp. 178–198.
12. D. Skeen, „Nonblocking commit protocols," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1981), pp. 133–142.
13. M. Stonebraker, „Retrospection on a database system," *ACM Trans. on Database Systems* **5**:2 (1980), pp. 225–240.
14. R. H. Thomas, „A majority consensus approach to concurrency control," *ACM Trans. on Database Systems* **4**:2 (1979), pp. 180–219.

11. fejezet

Információk egyesítése

Bár a modern adatbázisrendszerek sokféle irányba fejlődnek, az *információegyesítés* általános iránya az új alkalmazások népes családját tudhatja magáénak. Az ilyen alkalmazások két vagy több adatbázisban *(információforrásban)* tárolt adatokból hoznak létre egy nagy (virtuális) adatbázist, amelyben a forrásokban fellelhető összes információ megtalálható. Így a több helyen „szétszórtan" tárolt adatokat egységesen lehet lekérdezni. Az adatforrások lehetnek hagyományos vagy más típusú adatbázisok, például világhálós weboldalak gyűjteményei.

Ebben a fejezetben az információegyesítés fontos szempontjait vezetjük be. Először vázlatosan áttekintjük az információegyesítés legfontosabb módszereit: a szövetséget, a tárház létrehozását és a közvetítést. Ezután néhány alkalmazásban megtalálható integrált adatszervezési módszeren keresztül megismerkedünk egy különleges adatbáziselemmel, az úgynevezett „adatkockával". Az információegyesítéshez kapcsolódó speciális alkalmazásokat is áttekintjük. Így kerül sorra az „OLAP" (on-line analitikus feldolgozás)[1], és az „adatbányászat" témaköre.

11.1. Az információegyesítés módjai

Sokféle módszer létezik, amelynek segítségével adatbázisok vagy más (osztott) információforrások együttműködhetnek. Ebben a részben a három leggyakrabban alkalmazott eljárást mutatjuk be:

1. *Adatbázis-szövetség.* Az adatforrások egymástól függetlenek, de mindegyik kérhet a másiktól információt.
2. *Tárház létrehozása.* A különböző adatforrásokban megtalálható adatok másolatait egyetlen adatbázisban, az *adattárházban* tároljuk. Lehetséges, hogy az adatok még a tárházbavétel előtt valamilyen feldolgozó eljáráson mennek keresztül, például szűrhetjük vagy összesíthetjük (aggregáljuk) őket, vagy összekapcsolhatunk néhány relációt. Az adattárházat valamilyen időközönként, talán éjszakánként, frissítjük. Mivel az adatokat az adatforrásokból másoljuk, szükség lehet bizonyos átalakításokra, hogy az összes adat megfeleljen az adattárház sémájának.
3. *Közvetítés.* A közvetítő egy szoftverkomponens, amely egy olyan *virtuális adatbázist* támogat, amelyet a felhasználó úgy kérdezhet le, mintha az *valódi* lenne (azaz mintha fizikailag létezne, mint egy adattárház). Maga a közvetítő nem tárol semmiféle adatot – más módszert alkalmaz. A felhasználó lekérdezését lefordítja az adatforrások számára, egy vagy több lekérdezés képében, majd az adatforrások válaszait egységbe fogva adja meg a választ a felhasználónak.

[1] Az OLAP mozaikszó az angol *On-Line Analytic Processing* kifejezésnek felel meg. *A fordító megjegyzése.*

Mindegyik módszerrel részletesebben is megismerkedünk a fejezet során. Valamennyi megközelítésben kulcskérdés, hogy hogyan alakítjuk át az információforrásból nyert adatot. Az efféle információalakítók, a *borítékolók* (wrappers) vagy *adatkinyerők* (extractors) felépítését tárgyaljuk a 11.2. részben.[2] A következőkben bemutatunk néhány olyan problémát, amelyeket a borítékolóknak kell megoldaniuk.

11.1.1. Az egyesítés problémái

Akármilyen információegyesítő felépítést is választunk, kényes problémákba ütközünk, amikor a különböző adatforrások nyers adatainak próbálunk jelentést tulajdonítani. Az olyan adatforrások összességét, amelyek ugyanolyan, mégis sok apró részletben eltérő adatokkal dolgoznak, *heterogén* adatforrásnak nevezzük. Egy hosszabb példa segítségével bemutatjuk, miről van szó.

11.1. példa: A Süni Gépkocsigyártó Rt.-nek 1000 forgalmazója van, mindegyik adatbázist tart fenn a maga készletének a nyilvántartására. A cég létre akar hozni egy olyan egyesített adatbázist, amely az összes forgalmazójánál (azaz az 1000 adatforrásban) tárolt információt tartalmazza.[3]

Ha az egyik kereskedőnek éppen nincs raktáron valamilyen modellje, az egyesített adatbázis segítségével könnyen meg tudja állapítani, hogy melyik másiknál találhatja meg a hiányzó modellt. A cég elemzői pedig az előrelátható keresletre tehetnek megfigyeléseket, így a termelést a várhatóan népszerű modellekre lehet összpontosítani. Az 1000 forgalmazó azonban nem ugyanazt az adatbázissémát használja. Például az egyikük tárolhatja az autók adatait egyetlen relációban:

[2] Általában borítékolóról beszélünk közvetítő esetén, és adatkinyerőről tárház esetén, de nem mindig következetes a szóhasználat. *A fordító megjegyzése.*

[3] A legtöbb valódi gépkocsigyártó cég ma hasonló eszközökkel rendelkezik, bár ezeknek a fejlesztése eltérhet a példában leírtaktól. Lehet például, hogy a központi adatbázis jön létre először, majd később a forgalmazók, akik ebből letölthetik a saját adatbázisukba a megfelelő részeket. Az itt említett eset mégis arra szolgál például, amivel manapság sokféle szférában próbálkoznak.

```
Kocsik(sorszám, modell, szín, autoSeb, cdJátszó,...)
```

ahol minden lehetséges extra felszereléshez (automata sebességváltó, CD-lejátszó...) egy logikai érték van rendelve. Egy másik forgalmazó viszont nyilvántarthatja az extra felszereléseket egy külön relációban, például így:

```
Autók(sorsz, modell, szín)
Extrák(sorsz, extra)
```

Vegyük észre, hogy nem csak a két séma eltérő, hanem a nyilvánvalóan „azonos" nevek is megváltoztak: `Kocsikból` `Autók` lett, `sorszámból` `sorsz`.

Rosszabb esetben az azonos jelentésű adatok a különböző adatbázisokban másképpen vannak ábrázolva.

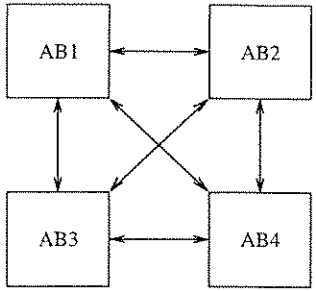
1. *Adattípus-különbségek.* A sorszámok lehetnek változó hosszúságú karakterláncok az egyik helyen vagy rögzített hosszúak a másikon. Maga a rögzített hossz is lehet más és más az egyes adatbázisokban, sőt néhány adatforrás használhat egész számokat a karakterlánc helyett.
2. *Értékkülönbségek.* Ugyanazt a fogalmat ábrázolhatják különböző konstansokkal a különböző adatforrásokban. A fekete színt például ábrázolhatják egy egész számmal, mint kóddal az egyik helyen, a `FEKETE` karakterlánccal egy másikon, és a `FE` kóddal egy harmadikon. Még rosszabb a helyzet, ha a `FE` a „fehéret" jelenti egy negyedik helyen.
3. *Szemantikus különbségek.* Sok fontos kifejezést másképpen értelmezhetnek az egyes adatforrások. Az egyik forgalmazó nyilvántarthat tehergépkocsikat is a `Kocsik` relációban, míg a másik csak személygépkocsikat tárol ugyanebben a relációban. Az egyik forgalmazó talán különbséget tesz furgon és kisteherautó között, a másik pedig nem.
4. *Hiányzó értékek.* Lehet, hogy az egyik adatforrás egyáltalán nem tárol egy olyan típusú információt, amit az összes többi (vagy a legtöbb) forrás nyilvántart. Például az egyik forgalmazó a színeket egyáltalán nem tartja nyilván. Hogy ezeket a helyzeteket kezelni tudjuk, néha használhatunk `NULL` értéket vagy valamilyen alapértelmezett értéket. A fejlődés iránya azonban a „felstrukturált" adatmodell használata felé mutat, amellyel olyan egyesített adatokat ábrázolhatunk, amelyek nem egészen egyeztethetők össze. Ezzel a témával kapcsolatban található néhány hasznos olvasmány a fejezet irodalomjegyzékében.

Az adatforrások közti minden efféle összeférhetetlenséget meg kell szüntetni valamilyen átalakítás, „fordítás" segítségével, még mielőtt az egyesített adatbázis létrejönne. □

11.1.2. Adatbázis-szövetség

Több adatbázis összefogásának talán az a legegyszerűbb módja, ha 1-1 kapcsolatot építünk azon adatbázispárok között, amelyek kommunikálni akarnak egymással. Ezeken a kapcsolatokon keresztül D_1 adatbázisrendszer lekérdezheti D_2-t, D_2 által érthető formában. A probléma ezzel a felépítéssel abból adódik, hogy ha az n adatbázis mindegyike szeretne kommunikálni a másik $(n-1)$-gyel, akkor $n(n-1)$ „rendszerközi" lekérdezéseket támogató kódot is kell írnunk. A 11.1. ábrán látható szövetségben a négy adatbázis mindegyikének három fordító komponensre van szüksége, hogy a másik hármat el tudja érni.

Mindezek ellenére bizonyos körülmények között lehet, hogy az adatbázis-szövetséget a legkönnyebb létrehozni, különösen, ha az adatbázisok egymás közti kommunikációja természeténél fogva korlátozott. Egy példán keresztül bemutatjuk, hogyan működhet a fordító komponens.



11.1. ábra. *A négy adatbázisból álló szövetségnek összesen 12 fordító komponensre van szüksége*

11.2. példa: Tegyük fel, hogy a Süni típusú gépkocsi forgalmazói meg akarják osztani egymás közt a leltárukat, de minden forgalmazónak csak arra van szüksége, hogy egy megfelelő lekérdezés segítségével megállapíthassa, hogy a nála megrendelt kocsik közül melyek találhatók meg valamelyik tőle nem messze működő forgalmazónál. Kicsit formálisabban, tekintsük Forgalmazó 1-et, aki a következő relációval rendelkezik:

```
IgényeltKocsik(modell, szín, autoSeb)
```

A reláció minden sora egy vásárló által igényelt kocsi leírása: milyen modell, milyen színű, és akarnak-e bele automata sebességváltót vagy nem. Forgalmazó 2 a leltárához a 11.1. példában bevezetett kétrelációs sémát használja:

```
Autók(sorsz, modell, szín)
Extrák(sorsz, extra)
```

Forgalmazó 1 ír egy programot, amely egy távoli lekérdezés segítségével olyan kocsikat keres Forgalmazó 2 adatbázisában, amelyek leírása megfelel valamelyik olyan

```
for(minden IgényeltKocsik-hoz tartozó (:m, :c, :a) sorra) {
   if(:a= TRUE) {/*automata sebességváltót akarnak*/
      SELECT sorsz
      FROM Autók, Extrák
      WHERE Autók.sorsz = Extrák.sorsz AND
            Extrák.extra = 'autoSeb' AND
            Autók.modell = :m AND
            Autók.szín = :c;
   }
   else { / *nem akarnak automata sebességváltót */
      SELECT sorsz
      FROM Autók
      WHERE Autók.modell = :m AND
         Autók.szín = :c AND
         NOT EXISTS (
            SELECT *
            FROM Extrák
            WHERE sorsz = Autók.sorsz AND
                  extra = 'autoSeb'
         );
   }
}
```

11.2. ábra. *Forgalmazó 1 lekérdezi Forgalmazó 2 adatbázisát*

autónak, amely Forgalmazó 1 `IgényeltKocsik` relációjában szerepel. A program vázlata a 11.2. ábrán látható.

A beágyazott SQL rész jelenti Forgalmazó 2 adatbázisának távoli lekérdezését. Ennek az eredményét kapja vissza Forgalmazó 1. A szabvány SQL-ben használatos konvenciónak megfelelően kettőspontot tettünk az olyan változók elé, amelyek az adatbázisból visszakapott konstans értékeket jelölik.

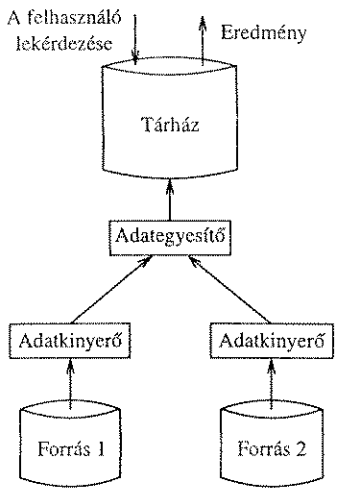
Ez a lekérdezés Forgalmazó 2 sémáját használja. Ha Forgalmazó 1 ugyanezt a lekérdezést Forgalmazó 3 adatbázisán is végre akarná hajtani, aki a 11.1. példában bemutatott

```
Kocsik(sorszám, modell, szín, autoSeb,...)
```

egyrelációs sémát használja, akkor a lekérdezés egész másképpen nézne ki. De mindegyik lekérdezés helyesen működik azon az adatbázison, amelyikre írták. □

11.1.3. Adattárházak

Az *adattárházban* mint információegyesítő architektúrában, a különböző információforrásokból nyert adatokat egy *globális* sémába illesztjük. Ezután az adatot a tárházban tároljuk, amely a felhasználó számára úgy néz ki, mint egy közönséges adatbázis.



11.3. ábra. *Az adattárház az egyesített információt egy külön adatbázisban tárolja*

A 11.3. ábrán látható egy adattárház felépítése, bár az ott szereplő két forrásnál sokkal többet is felvehettünk volna a rendszerbe.

Ha az adat egyszer már bekerült a tárházba, akkor a felhasználó ugyanolyan lekérdezéseket hajthat rajta végre, mint bármilyen más adatbázison. Azt viszont általában megtiltják, hogy a felhasználó módosítsa az itt tárolt adatokat, mivel ezek a változtatások nem jelennének meg a tárház adatforrásaiban, és így az adattárház inkonzisztenssé válna a forrásaival szemben.

A tárházban tárolt adat létrehozásának legalább három megközelítési módja van:

1. A tárházat a források aktuális adatai alapján újra és újra létrehozzuk. Ez az eljárás a leggyakoribb, általában az éjszakánként vagy akár hosszabb időközönként végrehajtott rekonstrukcióval (az éjszakai időpont azért nagyon kedvező, mert ilyenkor ki lehet kapcsolni a rendszert, így a tárház építése idején nincsenek lekérdezések). A módszer fő hátránya, hogy a rendszert le kell állítani, és hogy néha a tárház létrehozása hosszabb ideig tart, mint egy normális „éjszaka". Néhány alkalmazásban további hátrányt jelent, hogy a tárházban tárolt adat sokat veszthet az aktualitásából.
2. A tárházat időszakosan frissítjük (például minden éjjel) a forrásokon a tárház utolsó frissítése óta végrehajtott változtatások alapján. Ez az eljárás kisebb adattömeggel dolgozik, mint az előző módszer, és ez nagyon fontos szempont, amikor egy nagy tárházat (sok gigabájt, illetve terabájt nagyságúak vannak manapság használatban) rövid idő alatt kell módosítani. Hátrányt jelent azonban, hogy a tárházon végzendő változtatásokat számító eljárás, a *növekményes frissítés* (incremental update), nagyon összetett azokhoz az algoritmusokhoz képest, amelyek a semmiből építik újjá az adattárházat.

```
INSERT INTO AutóTház(sorszám, modell, szín, autoSeb, forgalmazó)
   SELECT sorszám, modell, szín, 'igen', 'forgalmazó2'
   FROM Autók, Extrák
   WHERE Autók.sorsz = Extrák.sorsz AND
         Extrák.extra = 'autoSeb';

INSERT INTO AutóTház(sorszám, modell, szín, autoSeb, forgalmazó)
   SELECT sorszám, modell, szín, 'nem', 'forgalmazó2'
   FROM Autók
   WHERE NOT EXISTS (
      SELECT *
      FROM Extrák
      WHERE sorsz = Autók.sorsz AND
            extra = 'autoSeb'
   );
```

11.4. ábra. *Az adatkinyerő Forgalmazó 2 adatait áthelyezi a tárházba*

3. A tárházat azonnal változtatjuk, mihelyst valami módosítás történik valamelyik adatforráson. Ez a megközelítés túl sok kommunikációt és feldolgozást igényel, hogy a gyakorlatban is alkalmazható legyen. Lassan változó forrásokból épített kis tárházak esetén azonban alkalmas lehet a használata. Mindezek ellenére ez a téma jó kutatási terület, és egy ilyen típusú tárház sikeres megvalósításának számos fontos alkalmazása lehetne, például az automatizált értékpapír-kereskedés területén.

11.3. példa: Az egyszerűség kedvéért tegyük fel, hogy a Süni rendszerben csak két forgalmazó szerepel, amelyek rendre a következő sémákat használják:

```
Kocsik(sorszám, modell, szín, autoSeb, cdJátszó,...),
```

illetve

```
Autók(sorsz, modell, szín)
Extrák(sorsz, extra)
```

Létre akarunk hozni egy adattárházat a következő séma szerint:

```
AutóTház(sorszám, modell, szín, autoSeb, forgalmazó)
```

A globális séma tehát hasonló ahhoz, amelyet az első forgalmazó használ, de a tárházban az extra felszerelések közül csak az automata sebességváltót tartjuk nyilván, és felveszünk egy új attribútumot is, amiből megtudhatjuk, hogy melyik forgalmazónál található meg az adott gépkocsi.

A szoftver, amely a második forgalmazó adatbázisából nyert adatokból a globális sémának megfelelően feltölti az adattárházat, SQL-lekérdezések segítségével is megírható. Az első forgalmazóhoz intézett lekérdezés egyszerű:

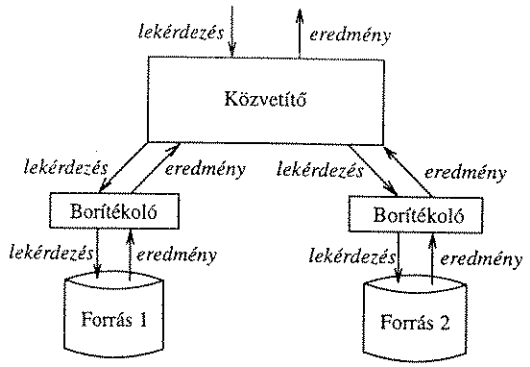

```
INSERT INTO AutóTház(sorszám, modell, szín, autoSeb, forgalmazó)
   SELECT sorszám, modell, szín, autoSeb, 'forgalmazó1'
   FROM Kocsik;
```

A második forgalmazóhoz írt adatkinyerő bonyolultabb, mert el kell döntenünk, hogy az adott autóban van-e automata sebességváltó vagy nincs. A nyilvánvaló jelentéssel bíró `'igen'`, illetve `'nem'` karakterláncokat használjuk az `autoSeb` attribútum értékeként. Az adatkinyerő SQL-kódja a 11.4. ábrán látható.

Ebben az egyszerű példában nincs adategyesítő. Mivel a tárház a forrásokból kiszűrt relációk uniója, az adatokat közvetlenül, előzetes feldolgozás nélkül töltöttük a tárházba. Sok tárház azonban a forrásokból nyert relációkon különböző műveleteket hajt végre. Például két relációt összekapcsolnak, és az eredményt teszik a tárházba, vagy több reláció unióján valamilyen összesítést hajtanak végre. Még általánosabban, minden forrásból több relációt is kiszűrhetnek, és különböző relációkat sokféle módon kombinálhatnak. □

11.1.4. Adatközvetítő

Az adatközvetítő egy vagy több virtuális nézettáblát támogat, amelyek az adattárházban testet öltött relációkhoz nagyon hasonló módon egyesítik az egyes forrásokat. Mivel azonban maga a közvetítő nem tárol adatokat, az adattárháztól elég eltérő módon működik. A 11.5. ábrán látható közvetítő két forrást koordinál. Ahogy azt az adattárház felépítésénél is említettük, kettőnél több forrás használata a tipikus. Első lépésként a felhasználó megad egy lekérdezést a közvetítőnek. Mivel maga a közvetítő nem tárol semmilyen adatot, az információforrásoktól kell megszereznie a szükséges adatokat, és ezek alapján kell választ adnia a felhasználónak.



11.5. ábra. *A közvetítő és a borítékolók átalakítják a lekérdezéseket a források számára, és összegyűjtik az eredményt*

A 11.5. ábrán látható, hogy a közvetítő mindegyik borítékolójának küld egy lekérdezést, és ezek a következő lépésben a saját adatforrásukhoz intéznek egy-egy lekérdezést. Valójában a közvetítő több lekérdezést is küldhet a borítékolónak, és lehet, hogy néhány borítékolónak egyáltalán nem küld lekérdezést. A visszakapott adatokat a közvetítő egyesíti. Nem tüntettünk fel egy kifejezett adategyesítő alkotóelemet, mint ahogy ezt a 11.3. ábrán az adattárház esetén tettük, mert az adatközvetítő esetén a forrásokból visszakapott eredmények egyesítése a közvetítő egyik feladata.

11.4. példa: Tekintsük a 11.3. példában tárgyalt helyzetet, de most adattárház helyett használjunk adatközvetítőt. A közvetítő tehát ugyanazt a két gépkocsi-adatforrást egyesíti, mint az előző példában, és egy nézettáblát hoz létre az alábbi sémával:

```
AutóKözv(sorszám, modell, szín, autoSeb, forgalmazó)
```

Tegyük fel, hogy a felhasználó a piros autók iránt érdeklődik, és a következő lekérdezést teszi fel:

```
SELECT sorszám, modell
FROM AutóKözv
WHERE szín = 'piros';
```

A közvetítő a felhasználó kérdésére reagálva továbbíthatja a lekérdezést mindkét borítékolójának. Hogy hogyan lehet ilyen és ehhez hasonló lekérdezéseket kezelő borítékolókat tervezni és megvalósítani, az a 11.2. rész témája. Bonyolultabb esetekben szükség lehet a lekérdezés alkotórészeinek az átalakítására és szétosztására is, de ebben az esetben az átalakítás műveletét egyedül a borítékolóra is lehet hagyni.

A Forgalmazó 1-hez tartozó borítékoló lefordítja a lekérdezést a forgalmazó sémájának megfelelően, amely emlékeztetőül a következő:

```
Kocsik(sorszám, modell, szín, autoSeb, cdJátszó,...)
```

Egy megfelelő fordítás:

```
SELECT sorszám, modell
FROM Kocsik
WHERE szín = 'piros';
```

A `sorszám-modell` párokból álló válaszhalmazt visszaküldi a borítékoló a közvetítőnek.

Eközben a Forgalmazó 2-höz tartozó borítékoló ugyanazt a lekérdezést a második forgalmazó sémájának megfelelően alakítja át. A séma:

```
Autók(sorsz, modell, szín)
Extrák(sorsz, extra)
```


és az ennek megfelelő lekérdezés:

```
SELECT sorsz, modell
FROM Autók
WHERE szín = 'piros';
```

A borítékoló válaszként visszaküldi a `sorsz-modell` párokból álló halmazt a közvetítőnek, esetleg előtte végrehajtva azt a nem túl megerőltető feladatot, hogy az eredménytáblázat sémájában a `sorsz` attribútumot `sorszám`-ra cserélje.

A közvetítő ezek után veheti a két reláció unióját, és ezzel megadhatja a választ a felhasználó kérdésére. Mivel a kocsisorszámtól elvárjuk, hogy „globális kulcsként" működjön, vagyis hogy ne legyen két autó, még különböző adatbázisokban sem, amely ugyanazzal a sorszámmal rendelkezik, vehetjük a két reláció zsák-unióját, feltételezve, hogy úgysem lesznek az eredményrelációban azonos sorok. □

A közvetítőnek számos olyan választási lehetősége van egy lekérdezés megválaszolása közben, amelyre a 11.4. példában nem került sor. Például megteheti, hogy először csak egy adatforrást kérdez le, majd ennek az eredménye alapján dönt a következő lekérdezés(ek) felől. Ez a módszer megfelelő lenne például, ha a felhasználót az érdekelte volna, hogy van-e Süni „Tüskés" sportkupé kékben. A közvetítő először csak Forgalmazó 1-et kérdezné le, és ha ennek az eredménye az üres halmaz lenne, csak akkor fordulna Forgalmazó 2-höz a következő lekérdezéssel.

11.1.5. Feladatok

11.1.1. feladat: Ellenőrizzük néhány on-line könyvkereskedés weboldalán, hogy milyen információk találhatók erről a könyvről. Hogyan hoznánk létre ezek alapján egy olyan globális sémát, amely megfelel egy adattárház vagy egy közvetítő számára?

! **11.1.2. feladat:** Az *A* számítógépekkel foglalkozó cég a következő sémának megfelelően tartja nyilván az árusított modelleket:

```
Szgépek(szám, proc, seb, memória, hd)
Monitorok(szám, képernyő, maxX, maxY)
```

A (123, PIII, 500, 128, 18,7) sor az `Szgépek` relációban például azt jelenti, hogy a 123-as számú modellhez egy 500 MH-es Pentium–III processzor, 128 Mb memória és 18,7 Gb merevlemez tartozik. A (456, 19, 1600, 1200) sor a `Monitorok` relációban azt jelenti, hogy a 456-os számú modell képernyője 19 inches és a maximális felbontása 1600 × 1200.

A *B* számítógépekkel foglalkozó cég csak teljes konfigurációkkal foglalkozik, számítógépet és monitort csak együtt ad el. Ez a következő sémát használja:

```
Konfig(azon, processzor, mem, lemez, képMéret)
```

A `processzor` attribútum értéke egész szám és a processzor sebességét adja meg. A processzor típusa (például Pentium III) és a monitor maximális felbontása nincs nyilvántartva. Az `azon`, `mem` és `lemez` attribútumok az *A* cég `szám`, `memória` és `hd` attribútumaihoz hasonló jelentéssel bírnak azzal a különbséggel, hogy a merevlemez mérete itt gigabájt helyett megabájtban van megadva.

a) Milyen SQL-utasítást kellene kiadnia az *A* cégnek, ha a *B* cég által árusított cikkekről szeretne információt tárolni a saját adatbázisában?

* b) Milyen SQL-utasítás lenne *B* számára a legmegfelelőbb, ha az *A* számítógépeiből és monitoraiból összeállítható konfigurációkról a lehető legtöbb információt akarja felvenni a saját `Konfig` relációjába?

*! **11.1.3. feladat:** Milyen globális séma segítségével tudnánk a lehető legtöbb információt nyilvántartani a 11.1.2. feladat *A* és *B* cégei által kínált termékekről?

11.1.4. feladat: Adjuk meg azt a lekérdezéssorozatot, amely az *A* és *B* cégek adataiból gyűjtött információt egy olyan adattárházba teszi, amely a 11.1.3. feladat megoldásaként adott globális sémát használja! Szükség esetén használható a szerzők által megadott séma is.

11.1.5. feladat: Tegyük fel, hogy egy közvetítő a 11.1.3. feladat sémáját (akár a saját megoldást, akár a szerzők megoldását) használja. Hogyan lehetne megadni az 500 MHz-es számítógépekkel együtt elérhető maximális merevlemez méretét?

! **11.1.6. feladat:** Adjunk meg két másik sémát, amelyek segítségével a számítógépes cégek a 11.1.2. feladatban leírtakhoz hasonló adatokat tarthatnak nyilván!

! **11.1.7. feladat:** A 11.3. példában szó volt a Forgalmazó 1 `Kocsik` relációjáról, amely tartalmazott egy kényelmesen használható `autoSeb` attribútumot, amelynek csak „igen"/„nem" értékei lehettek. Mivel a globális sémában ehhez az attribútumhoz ugyanezeket az értékeket használtuk, az `AutóTház` relációt nagyon könnyű volt létrehozni. Most tegyük fel, hogy a `Kocsik.autoSeb` attribútum egész értékeket vehet fel; 0 jelenti azt, hogy nincs automata sebességváltó beszerelve, $i > 0$ pedig azt, hogy i sebességes automata sebességváltó tartozik a kocsihoz. Adjuk meg azt az SQL-lekérdezést, amely segítségével a `Kocsik` reláció átültethető az `AutóTház` relációba!

11.1.8. feladat: Hogyan fordítaná a 11.4. példa közvetítője a következő lekérdezéseket:

* a) Adjuk meg az automata sebességváltós gépkocsik sorszámát!

b) Adjuk meg azon gépkocsik sorszámát, amelyek nem rendelkeznek automata sebességváltóval!

! c) Adjuk meg a Forgalmazó 1-nél elérhető kék gépkocsik sorszámát!

11.2. Borítékolók a közvetítő alapú rendszerekben

A 11.3. ábrához hasonló adattárház adatkinyerői a következő alkotóelemekből állnak:

1. Egy vagy több beépített lekérdezés, amelyek végrehajtásával az adatforrásból adatok állíthatók elő az adattárház számára.
2. Megfelelő kommunikációs mechanizmus, amely segítségével az adatkinyerő képes az alábbi feladatokra:

   a) ad hoc lekérdezéseket továbbítani a forrásnak,
   b) válaszokat fogadni a forrástól és
   c) információt átadni az adattárháznak.

A forrás számára beépített lekérdezések lehetnek SQL-lekérdezések, ha az adatforrás egy olyan SQL-adatbázis, mint amilyenekkel a 11.1. rész példáiban találkoztunk. Ha a forrás azonban nem hagyományos adatbázis, az adatkinyerőbe beépített lekérdezések akármilyen más, a forrás által értelmezhető műveletek is lehetnek. Például a borítékoló kitölthet egy weboldalon található űrlapot, lekérdezhet egy on-line bibliográfiát a rendszer saját, speciális nyelvén, vagy számtalan más eszközt is igénybe vehet, hogy a forrással meg tudja értetni a lekérdezést.

Az adatközvetítőnek azonban sokkal összetettebb borítékolókra van szüksége, mint a legtöbb adattárháznak. A közvetítő lekérdezések egész változatát intézheti a borítékolóhoz, és a borítékolónak képesnek kell lennie ezeket elfogadni és akármelyiket lefordítani a megfelelő adatforrás nyelvére. Ezek után természetesen a lekérdezés eredményét továbbítania kell a közvetítő felé, ahogy ezt a borítékoló a tárház alapú rendszerben is megteszi. Ebben a részben a közvetítők számára használható rugalmas borítékolók konstrukciójával foglalkozunk.

11.2.1. Sablonok lekérdezési formákhoz

Az adatforrás és közvetítő kapcsolatát alkotó borítékoló tervezésének létezik egy szisztematikus módja: a közvetítő által feltett lehetséges lekérdezéseket egy-egy *sablonnak* megfelelő osztályba soroljuk. A sablonok olyan paraméteres lekérdezések, ahol a paraméterek csak konstans értékeket vehetnek fel. A borítékoló a közvetítő által kínált konstansokkal végzi el a lekérdezést. A következő példán keresztül bemutatjuk a módszer lényegét. $T \Rightarrow S$ azt jelöli, hogy a borítékoló a T sablont a forrás által feldolgozható S lekérdezéssé, úgynevezett forráslekérdezéssé alakítja át.

11.5. példa: Olyan borítékolót akarunk létrehozni, amely a közvetítő és Forgalmazó 1 között teremti meg a kapcsolatot. Forgalmazó 1, illetve a közvetítő rendre a következő sémát használja:

```
Kocsik(sorszám, modell, szín, autoSeb, cdJátszó,…)
AutóKözv(sorszám, modell, szín, autoSeb, forgalmazó)
```

Gondoljuk meg, hogyan tudna a borítékoló adott színű kocsikat megadni a közvetítőnek. Ha a szín számára bevezetjük a `$c` paramétert, akkor az adott színtől függetlenül mindig használható lesz a 11.6. ábrán látható sablon. Hasonlóan, a borítékoló rendelkezhet egy másik sablonnal is, a modellnek megfelelő `$m` paraméterrel, sőt egy harmadikkal az automata sebességváltó paraméterezésével és így tovább. Ha a lekérdezés e három attribútum közül bármelyiket rögzítheti, akkor összesen nyolc sablonra van szükségünk.[4] Általában, ha n attribútum rögzítésére van lehetőség, a szükséges sablonok száma 2^n.[5] Másfajta lekérdezésekhez, mint például „bizonyos típusú autók száma" vagy „van-e raktáron egy bizonyos típusú autó?", más sablonok kellenek. A sablonok száma túlságosan is nagyra nőhet, de a borítékoló finomításával egyszerűsíthető a helyzet. Erről a 11.2.3. részben lesz szó. □

```
SELECT *
FROM AutóKözv
WHERE szín = '$c';
   =>
SELECT sorszám, modell, szín, autoSeb, 'forgalmazól'
FROM Kocsik
WHERE szín = '$c';
```

11.6. ábra. *Borítékoló sablon adott színű kocsik lekérdezésére*

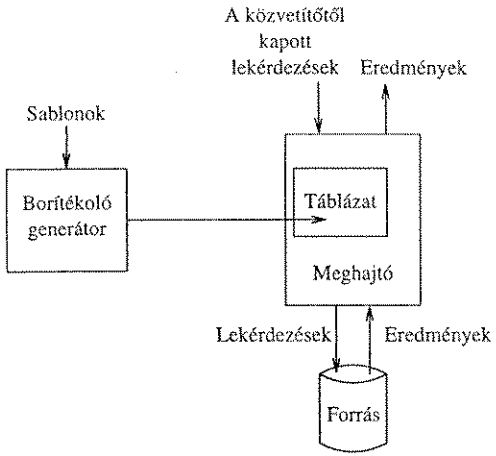
11.2.2. Borítékoló generátor

A borítékolót meghatározó sablonok alapján olyan programot kell írni, amely létrehozza magát a borítékolót. Ez a program a *borítékoló generátor*. Szellemében nagyon hasonló az elemző generátorokhoz (például YACC[6]), amelyek magas szintű specifikációk alapján hoznak létre fordítóprogram-komponenseket. A 11.7. ábrán bemutatott eljárás azzal kezdődik, hogy a specifikációt, vagyis a sablonkészletet, átadjuk a borítékoló generátornak.

[4] A három alapsablonon kívül szükségünk lesz még azokra is, amelyekben egyszerre adható meg a szín és modell, szín és automata sebességváltó és így tovább. A nyolcba beleszámoljuk az „üres" sablont is, amelyben a három paraméter közül egy sem szerepel, vagyis amely az egész egyesített adatbázist adja eredményül. *A fordító megjegyzése.*

[5] Ha az adatforrás ehhez a példához hasonlóan egy olyan adatbázis, amely lekérdezhető SQL-ben, jogosan elvárhatnánk, hogy egy sablon akárhány konstansértékű attribútumot képes legyen kezelni, egyszerűen úgy, hogy a `WHERE` záradékot paraméterré teszi. Ez a módszer működik az SQL-források és az olyan lekérdezések esetén, amelyekben az attribútumok értéke egy konstanshoz van kötve, de nem szükségszerű, hogy bármilyen adatforrás, például egy weboldal esetén is megfelel, ahol csak bizonyos formák használhatók interfészként. Általában nem tehetjük fel, hogy ahogy mi fordítunk le egy lekérdezést, az bármiben is hasonlít ahhoz, ahogy egy hasonló lekérdezés van lefordítva.

[6] *Yet Another Compiler-Compiler,* magyarul Ismét Egy Újabb Fordító Fordító. Értelmező- és fordítóprogramok fejlesztéséhez használt eszköz. *A fordító megjegyzése.*



11.7. ábra. *A borítékoló generátor létrehozza a táblázatokat a meghajtó számára; a meghajtó és a táblázatok alkotják a borítékolót*

A borítékoló generátor ez alapján táblázatot készít, amely a sablon és az annak megfelelő forráslekérdezési formapárokat tartalmazza. Minden borítékolóban használunk egy *meghajtót* (driver), amely általában lehet ugyanaz mindegyik generált borítékolóhoz. A meghajtó feladata a következő:

1. A közvetítőtől érkezett lekérdezések fogadása. Mivel a kommunikációs mechanizmus közvetítőfüggő lehet, a kommunikációs rutint mint „plug-in"-t adjuk a meghajtónak, így ugyanaz a meghajtó más plug-in-nel másféle kommunikációt használó rendszerekben is használható.
2. A lekérdezésnek megfelelő sablon keresése a táblázatban. Ha a keresés sikeres, akkor a lekérdezésből vett paraméterértékekkel kell végrehajtani a forráslekérdezést. Ha a keresés sikertelen, a borítékoló negatív választ küld a közvetítőnek.
3. A forráslekérdezés továbbítása az adatforrás felé, a „plug-in" kommunikációs rutin használatával. A lekérdezésre adott választ a borítékoló fogadja.
4. A válasz továbbítása a közvetítő felé, amelyen szükség esetén a borítékoló valamilyen feldolgozó eljárást hajt végre. Hogy ezzel a borítékolók hogyan támogathatják a lekérdezések egy nagyobb osztályát, azt a következő fejezetekben tárgyaljuk.

11.2.3. Szűrők

Tegyük fel, hogy az egyik forgalmazó adatbázisához tartozó borítékoló rendelkezik a 11.6. ábrán látható sablonnal, azonban az adatközvetítőtől modell *és* szín szerint kérnek tájékoztatást. A borítékoló talán fel van szerelve egy kicsit összetettebb sablonnal

is, amely segítségével kezelhetők a modellt és színt is rögzítő lekérdezések (11.8. ábra). De ahogy a 11.5. példa végén említettük, a gyakorlatban nem mindig valósítható meg, hogy az összes lehetséges lekérdezéshez megírjuk a megfelelő borítékoló sablont.

```
SELECT *
FROM AutóKözv
WHERE modell = '$m' AND szín = '$c';
   =>
SELECT sorszám, modell, szín, autoSeb, 'forgalmazól'
FROM Kocsik
WHERE modell = '$m' AND szín = '$c';
```

11.8. ábra. *Borítékoló sablon, amely az adott modellhez tartozó, adott színű kocsikat adja meg*

Létezik azonban egy másik megoldás is, amellyel növelhető a támogatott lekérdezések száma. Ennek az a lényege, hogy a borítékoló a forráslekérdezések eredményén egy *szűrést* (filter) hajt végre. Ha a borítékoló rendelkezik egy olyan sablonnal, amely (a paraméterek gondos behelyettesítése után) az eredeti lekérdezés eredményének egy bővebb részhalmazát adja, akkor lehetőség van arra, hogy a sablonnak megfelelő sorokból „kiszűrje" azokat, amelyek a kívánt lekérdezés eredményét adják, és csak ezeket küldje tovább a közvetítőnek. Annak eldöntése, hogy a közvetítő által igényelt lekérdezés végeredménye részhalmaza-e a borítékoló egyik sablonjának megfelelő lekérdezés végeredményének, általában nehéz probléma, bár az eddig látott példákhoz hasonló egyszerű esetekre az elmélet jól ki van dolgozva. A kérdés további tanulmányozásához jó kiindulópontok találhatók az irodalomjegyzékben.

11.6. példa: Tegyük fel, hogy egyetlen sablonnal rendelkezünk, azzal, amellyel az adott színű kocsikat kérdezhetjük le (11.6. ábra). A közvetítőnek azonban a kék „Tüskés" autókra van szüksége, azaz a következő lekérdezés végeredményét várja:

```
SELECT *
FROM AutóKözv
WHERE szín = 'kék' and modell = 'Tüskés';
```

A válaszadás egy lehetséges módja:

1. A 11.6. ábra sablonját `$c = 'kék'` helyettesítéssel használva megkeressük az összes kék autót.
2. Az eredményt az alábbi ideiglenes relációban tároljuk:

   ```
   TmpAutók(sorszám, modell, szín, autoSeb, forgalmazó)
   ```

3. Ebből a következő lekérdezés segítségével kiválasztjuk a Tüskéseket, az eredményt pedig visszaküldjük.

Hol végezzük a szűrést?

A példákban eddig mindig feltettük, hogy a szűrés műveletét a borítékoló végzi el. Az is lehetséges azonban, hogy a borítékoló továbbadja a közvetítőnek a még feldolgozatlan információt, és a szűrésre ott kerül sor. Ha viszont a sablon által visszaadott adatok többsége nem felel meg a közvetítő lekérdezésének, akkor a legjobb megoldás még a borítékoló szintjén elvégezni a szűrést, így kerülve el a szükségtelen sorok továbbításából eredő többletköltséget.

```
SELECT *
FROM TempAutók
WHERE modell = 'Tüskés';
```

A végeredmény a kívánt személygépkocsik halmaza. A gyakorlatban a `TempAutók` relációnak egyszerre csak egy sora jönne létre, amit rögtön meg is szűrnénk, csővezetékes jellegű feldolgozással. Inkább ezt a módszert használnánk, minthogy az egész relációt tároljuk és szűrjük a borítékolóban. □

11.2.4. A borítékoló más műveletei

A szűrésen kívül másféle átalakításokat is elvégezhet a borítékoló, amennyiben biztosak vagyunk benne, hogy az ehhez szükséges adatokat hiány nélkül visszakapja a sablon forráslekérdezés részétől. A közvetítőnek való továbbítás előtt elvégezhet például egy vetítést, de az is elképzelhető, hogy összekapcsolások vagy valamilyen összesítések után küldi csak tovább az eredményt.

11.7. példa: Tegyük fel, hogy a közvetítő az egyes forgalmazóknál fellelhető kék Tüskésekről akar információt szerezni, de csak a sorszámra, forgalmazóra és a sebességváltó típusára (automata-e) van szüksége, mivel a `modell` és `szín` mezők értéke a kérdésből nyilvánvaló. A borítékoló követheti ugyanazokat a lépéseket, mint a 11.6. példában, de az utolsó pontban, amikor az eredményeket kell visszaküldenie a közvetítőnek, el kell végeznie egy újabb „szűrést" is a `SELECT` záradékbeli vetítés képében a Tüskés modellek `WHERE` záradékbeli szűrése mellett:

```
SELECT sorszám, autoSeb, forgalmazó
FROM TempAutók
WHERE modell = 'Tüskés';
```

A 11.6. példához hasonlóan a `TempAutók` reláció itt sem jönne létre fizikailag a borítékolóban, a vetítést valószínűleg csővezetékes jellegű feldolgozás segítségével soronként végeznénk el. □

11.8. példa: Egy összetettebb példa kedvéért tegyük fel, hogy a közvetítőnek olyan forgalmazó-modell párokat kell találnia, ahol a forgalmazónak ebből a modellből van két piros példánya, az egyik automata sebességváltóval, a másik e nélkül. Még azt is tegyük fel, hogy Forgalmazó 1-hez csak a 11.6. ábráról már jól ismert sablon használható egyedül. A közvetítő tehát a 11.9. ábrán megadott lekérdezésre várja az eredményt a borítékolótól. Vegyük észre, hogy a forgalmazót nem kell megadnunk sem `A1` sem `A2` esetén, mert ez a borítékoló úgyis csak Forgalmazó 1 adataihoz fér hozzá. A közvetítő ugyanezt a lekérdezést küldi el az összes többi forgalmazó borítékolójához is.

```
SELECT A1.modell, A1.forgalmazó
FROM AutóKözv A1, AutóKözv A2
WHERE A1.modell = A2.modell AND
   A1.szín = 'piros' AND
   A2.szín = 'piros' AND
   A1.autoSeb = 'nem' AND
   A2.autoSeb = 'igen';
```

11.9. ábra. *Lekérdezés a közvetítőtől a borítékolóhoz*

Egy okosan tervezett borítékoló felismerné, hogy a közvetítő lekérdezését úgy is meg tudja válaszolni, ha először elkészíti a Forgalmazó 1-nél található összes piros autó adatait tartalmazó relációt:

```
PirosAutók(sorszám, modell, szín, autoSeb, forgalmazó)
```

Ehhez a 11.6. ábra sablonját használja, amelynek a segítségével az olyan lekérdezéseket lehet kezelni, amelyek csak egy színt adnak meg. Valójában a borítékoló úgy viselkedik, mintha a következő lekérdezésre válaszolna:

```
SELECT *
FROM AutóKözv
WHERE szín = 'piros';
```

A 11.6. ábra sablonját a `$c = 'piros'` behelyettesítéssel használva létre tudja hozni a `PirosAutók` relációt a Forgalmazó 1 adatbázisa alapján. Ahhoz, hogy a 11.9. ábrán látható lekérdezés végeredményét adja meg, a következő lépésben össze kell kapcsolnia a `PirosAutók` relációt saját magával, és el kell végeznie a szükséges kiválasztást. A 11.10. ábrán látható a borítékoló[7] által végrehajtott lekérdezés. □

```
SELECT DISTINCT A1.modell, A1.forgalmazó
FROM PirosAutók A1, PirosAutók A2
WHERE A1.modell = A2.modell AND
   A1.autoSeb = 'nem' AND
   A2.autoSeb = 'igen';
```

11.10. ábra. *A borítékoló (vagy a közvetítő) által végrehajtott lekérdezés a 11.9. ábra lekérdezésére ad választ*

[7] Néhány információegyesítő architektúrában ezt a feladatot inkább a közvetítő látná el.

11.2.5. Feladatok

* **11.2.1. feladat:** A 11.6. ábrán egy olyan borítékoló sablont láttunk, amely a közvetítő adott színű gépkocsikra vonatkozó lekérdezéseit a `Kocsik` relációval rendelkező forgalmazó számára értelmezhető lekérdezésekké alakította át. Most tegyük fel, hogy a közvetítő sémájában használt színkódolás más, mint amit a forgalmazó használ, de a következő konverziós táblázattal rendelkezünk: `GbólL(globSzín, LokSzín)`. Ennek megfelelően írjunk új sablont a borítékoló számára!

11.2.2. feladat: A 11.1.2. feladatban két számítógépes cégről, *A*-ról és *B*-ről beszéltünk, amelyek különböző sémákat használtak az adatbázisaikban. Tegyük fel, hogy létezik egy közvetítő a következő sémával:

```
PCKözv(gyártó, seb, mem, lemez, képernyő)
```

A reláció egy sora megadja a gyártót (*A* vagy *B*) és az annál a cégnél vásárolható összeállítás jellemzőit: a processzor sebességét, a központi memória, a merevlemez és a képernyő méretét. Írjunk a borítékolóban használható sablonokat a következő típusú lekérdezésekhez (lekérdezésenként összesen két sablont kell készíteni, egyet-egyet mindkét gyártó számára):

* a) Adjuk meg azokat a sorokat, amelyekben a sebesség nagysága egy előre meghatározott értékkel egyenlő!
b) Adjuk meg azokat a sorokat, amelyekben a képernyő mérete egy előre meghatározott értékkel egyenlő!
c) Adjuk meg azokat a sorokat, amelyekben a memória és a merevlemez mérete egy előre meghatározott értékkel egyenlő!

11.2.3. feladat: Tegyük fel, hogy mindkét információforrás (számítógépgyártó) borítékolója rendelkezik az előző feladatban leírt sablonokkal. Hogyan tudná ezeket a közvetítő felhasználni a következő lekérdezések megválaszolásában:

* a) Adjuk meg az összes olyan konfiguráció gyártóját, memóriájának és képernyőjének a méretét, amelynek a processzora 400 MHz-es és merevlemeze 12 Gb!
! b) Adjuk meg az 500 MHz-es konfigurációkban megtalálható maximális merevlemez-méretet!
c) Adjuk meg az összes olyan konfigurációt, amely 128 Mb memóriával rendelkezik, és a képernyő mérete (inchben) meghaladja a merevlemez méretét (gigabájtban)!

11.3. On-line analitikus feldolgozás

Ebben a fejezetben az egyesített információs rendszerek, különösen a tárház alapú rendszerek köré nőtt alkalmazások egy fontos osztályával ismerkedünk meg. Különböző cégek és szervezetek hoznak létre hatalmas adatbázisok felhasználásával olyan

Adattárházak és OLAP

Különböző okai vannak annak, hogy a tárházak miért játszanak olyan fontos szerepet az OLAP-alkalmazásokban. Egyrészt lehetséges, hogy az adataink kezdetben sok különböző adatbázisban vannak szétszórva, vagyis ahhoz, hogy OLAP-lekérdezéseket tudjunk végrehajtani, egy adattárházat kell létrehoznunk, amelyben megfelelően tudjuk szervezni és központosítani az egyesített adatokat. Másrészt viszont legtöbbször fontosabb annak a ténye, hogy az OLAP-lekérdezések végrehajtása – mivel ezek összetettek és az adatbázis nagy részét érintik – túl sok időt vesz igénybe az olyan tranzakciókezelő rendszerekben, amelyektől nagy teljesítményt várunk el. Az OLAP-lekérdezéseket gyakran tekinthetjük a 10.7. rész értelmében „hosszú tranzakcióknak".

Az egész adatbázist zároló hosszú tranzakciók miatt a szokásos OLTP-műveletek elvégzése lehetetlenné válna (például az eladásokból származó átlagos bevételt számító OLAP-lekérdezés futtatása közben nem vehetnénk fel az újabb vásárlások adatait az adatbázisba). Ezt a problémát rendszerint úgy oldjuk meg, hogy a még feldolgozatlan, nyers adatok másolatait összegyűjtjük egy adattárházba, itt futtatjuk az OLAP-lekérdezéseket, az adatváltoztató és az OLTP-lekérdezéseket pedig az adatforrásokon hajtjuk végre. A tárházat legtöbbször csak éjszakánként módosítjuk, napközben az elemzők a „befagyasztott" adatokon dolgoznak. Vagyis az adattárház adatai már 24 óra alatt elavulnak, ami az OLAP-lekérdezések eredményének az időszerűségét elég jól korlátozza, de ez az „időeltolódás" sok döntéstámogató alkalmazásban még a tűréshatáron belül marad.

tárházakat, amelyek alapján az arra kijelölt elemzők a szervezet számára fontos mintákat, irányvonalakat próbálnak megállapítani. Ez a tevékenység, az *on-line analitikus feldolgozás (OLAP)*, általában nagyon összetett, egy vagy több összesítő függvényt is felhasználó lekérdezéseket foglal magában. Ezeket a lekérdezéseket gyakran hívjuk *OLAP-* vagy *döntéstámogató lekérdezéseknek*. A 11.3.1. részben láthatunk ezekre néhány példát. Tipikus kérdés az olyan termékek keresése, amelyek iránt mindent egybevéve növekszik vagy éppen csökken a kereslet.

Az OLAP-alkalmazásokban használt lekérdezések jellemzően nagyon nagy mennyiségű adatot vizsgálnak át, még ha az eredmény nagyon kicsi is lesz. Ezzel szemben a mindennapos adatbázis-műveletek (például banki befizetés vagy repülőjegy-foglalás) az adatbázisnak ennél csak lényegesen kisebb hányadát érintik. Az ilyen típusú műveleteket gyakran nevezzük *on-line tranzakció-feldolgozásnak (OLTP)*.[8]

Az utóbbi időben olyan új lekérdezésfeldolgozó technikákat dolgoztak ki, amelyek különösen jól használhatók az OLAP-lekérdezések hatékony végrehajtásához. Az OLAP-lekérdezések bizonyos osztályainak különböző természete miatt pedig speciá-

[8] Az OLTP mozaikszó az angol *On-Line Transaction Processing* kifejezésnek felel meg. *A fordító megjegyzése.*

lis adatbázis-kezelő rendszereket, az *adatkocka*rendszereket fejlesztették ki és dobták piacra, hogy az OLAP-alkalmazásokat megfelelően támogassák. Ezek a rendszerek a 11.4. részben kerülnek tárgyalásra.

11.3.1. OLAP-alkalmazások

Az OLAP-alkalmazások általában fogyasztásra vonatkozó adatokból összeállított tárházat használnak. Nagyobb üzlethálózatok terabájtnyi információt halmoznak fel arról, hogy melyik üzletükben melyik cikkből mennyit adtak el. A cégre váró problémák vagy nagy lehetőségek előrejelzésében nagy szerepe lehet az olyan lekérdezéseknek, amelyek a fogyasztási adatok csoportosítása, összesítése alapján a valamilyen szempontból jelentős csoportokat azonosítani tudják.

11.9. példa: Tegyük fel, hogy a Süni cég létrehoz egy adattárházat, amely alapján elemezni tudja majd a gépkocsik iránti keresletet. A tárház sémája lehet a következő:[9]

```
Eladások(sorszám, dátum, forgalmazó, ár)
Autók(sorszám, modell, szín)
Forgalmazók(név, város, állam, tel)
```

Egy jellemző döntéstámogató lekérdezés az 1999. április 1-jén vagy azóta történt értékesítések alapján megállapíthatná, hogy az eladott járművek átlagára hogyan változik államonként. Egy ilyen lekérdezést mutatunk be a 11.11. ábrán.

```
SELECT állam, AVG(ár)
FROM Eladások, Forgalmazók
WHERE Eladások.forgalmazó = Forgalmazók.név AND
      dátum >= '1999-01-04'
GROUP BY állam;
```

11.11. ábra. *Államonkénti átlagos fogyasztói ár*

Figyeljük, meg, ahogy a lekérdezés végigveszi az adatbázis nagy részét, miközben az `Eladások` relációban tárolt vásárlási adatokat osztályozza a forgalmazó címe szerint. Ezzel szemben a „milyen áron adták el a 123-as sorszámú autót?" típusú gyakori OLTP-lekérdezések az adatbázis egyetlen sorát érintik csupán. □

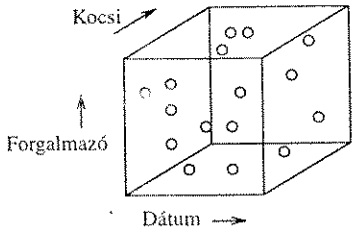
Hogy lássunk egy másik OLAP-példát is, vegyünk egy hitelkártya-kibocsátó céget, amely a kártyaigénylőiről akarja eldönteni, hogy hitelképesek lesznek-e. Készítenek egy adattárházat, amely a cég összes ügyfeléről és az eddigi befizetéseiről tartalmaz információkat. Az OLAP-lekérdezések olyan tényezőket (például életkor, jövedelem,

[9] Az Egyesült Államokban ésszerű a forgalmazó címében az államot is nyilvántartani, ezért szerepel a `Forgalmazók` relációban az `állam` attribútum. *A fordító megjegyzése.*

ingatlantulajdon, irányítószám) fognak keresni, amelyek segíthetnek annak az előrejelzésében, hogy egy adott ügyfél be fogja-e fizetni időben a számláit vagy nem. Ehhez hasonlóan kórházak is rendelkezhetnek betegek adatait (felvétel időpontja, elvégzett laborvizsgálatok, ezek eredménye, diagnózis, gyógykezelés leírása és így tovább) tartalmazó adattárházzal, hogy elemezni tudják a kezeléssel járó kockázatot, és ennek tükrében válasszák ki a legjobbnak ítélt módszert.

11.3.2. OLAP-adatok többdimenziós nézete

A tipikus OLAP-alkalmazásokban mindig létezik egy központi reláció vagy adatgyűjtemény: a *ténytáblázat* (fact table). A ténytáblázat olyan érdekes eseményeket vagy objektumokat tartalmaz, mint a 11.9. példában megismert vásárlási adatok. Gyakran segít, ha a ténytáblázatban tárolt objektumokra úgy gondolunk, mintha egy többdimenziós térben vagy „kockában" lennének elrendezve. A 11.12. ábrán háromdimenziós adatobjektumok láthatók, a kocka belső pontjaiként ábrázolva.[10] A dimenziók elnevezései: kocsi, forgalmazó, dátum, a korábbi személygépkocsi értékesítős példának megfelelően. A kocka minden egyes belső pontjára gondolhatunk úgy, mint egy-egy gépkocsi eladására, a dimenziókra pedig úgy, mintha ennek az eladásnak a körülményeit, jellemzőit írnák le.



11.12. ábra. *Az adatok többdimenziós kockába szervezése*

A 11.4. részben bevezetünk egy, az OLAP-adatok kezelésére alkalmas, speciális architektúrát, az „adatkockát". Ez a felépítés egy kicsit más szempontból közelíti meg a többdimenziós adatokat, ugyanis az adatkockában a pontok jelenthetnek összesített adatot is. Például ahelyett, hogy a „kocsi" dimenzió mentén minden egyes pont más és más kocsit jelölne, elképzelhető, hogy a dimenzió modellenként van összesítve, és a 11.12. ábra pontjai nem egyes autók, hanem egyes modellek összeladását jelentik a forgalmazó és a dátum dimenziók által meghatározott paraméterek mellett. A többdimenziós adat e kétfajta értelmezése közti különbség tükröződik a kockaszerkezetű OLAP-adatot támogató speciális rendszerek által vett két fő irányvonalban is:

[10] Háromnál több dimenzióval rendelkező objektumok esetén természetesen legalább négydimenziós kockát kellene felhasználnunk, de azt jóval nehezebb elképzelni és lerajzolni is. *A fordító megjegyzése.*

1. *ROLAP*[11] vagy *Relációs OLAP*. Ebben a megközelítésben az adatot relációkban tároljuk, de egy speciális szerkezetben, a „csillag sémában". A relációk egyike a ténytáblázat, amely a *nyers*, vagyis a „még nem összesített" adatokat tartalmazza. A rendszer lekérdezőnyelve és más képességei ezt az adatszervezési módot kihasználva alakíthatók. A csillag sémával a 11.3.3. részben foglalkozunk.
2. *MOLAP*[12] vagy *Többdimenziós OLAP*. Ebben az esetben a fent említett speciális szerkezetet, az „adatkockát" használjuk az adat tárolására. Ahogy már említettük, ez az adat gyakran már részben összesített. Az efféle adat OLAP-lekérdezéseit támogatandó, a rendszer nem relációs műveleteket is megvalósíthat.

11.3.3. A csillag séma

A *csillag séma* a ténytáblázat sémájából áll, amely több más relációhoz, a „dimenziótáblázatokhoz" kapcsolódik. A dimenziótáblázatokról kicsit később lesz részletesebben szó. A „csillag" központjában található a ténytáblázat, a csúcsaiban pedig a dimenziótáblázatok. A ténytáblázatnak általában van néhány dimenzió attribútuma – ezek az egyes *dimenziókat* jelölik –, és egy vagy több *függő* attribútuma, amelyek az adat, mint a többdimenziós tér egy pontjának, mint egésznek érdekes tulajdonságait jelölik. Például a vásárlásra vonatkozó adatok dimenziói között szerepelhet a vásárlás időpontja, helyszíne (melyik üzletben történt), a vásárolt cikk típusa, a fizetési mód (készpénz vagy hitelkártya) és így tovább. Függő attribútum(ok) lehet(nek) például a fogyasztói ár, a kereskedelmi ár vagy az adó mennyisége.

11.10. példa: Az előző példában bevezetett `Eladások` reláció

```
Eladások(sorszám, dátum, forgalmazó, ár)
```

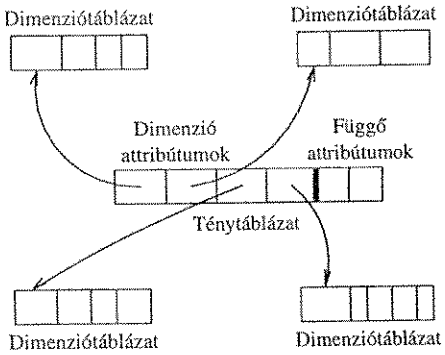
a ténytáblázat. A dimenziók a következők:

1. `sorszám`, az eladott gépkocsit jelöli, azaz az összes gépkocsi terében az ezzel a sorszámmal rendelkező autónak megfelelő pontot.
2. `dátum`, a vásárlás időpontját jelöli, azaz a vásárlás, mint esemény pozícióját az idő dimenzióban.
3. `forgalmazó`, az esemény pozícióját jelöli az összes forgalmazó terében.

Az egyetlen függő attribútum az `ár`, amire az adatbázis OLAP-lekérdezéseinek jellemzően valamilyen összesített formában lesz majd szüksége. Persze az összár vagy az átlagár megállapítása mellett a leszámlálós lekérdezéseknek is lehet értelmük. Például „adjuk meg, hogy 1999 májusában forgalmazóként összesen hány autót adtak el". □

[11] A ROLAP mozaikszó az angol *Relational OLAP* kifejezésnek felel meg. *A fordító megjegyzése.*

[12] A MOLAP mozaikszó az angol *Multidimensional OLAP* kifejezésnek felel meg. *A fordító megjegyzése.*



11.13. ábra. *A ténytáblázat dimenzió attribútumai a dimenziótáblázatok kulcsmezőire hivatkoznak*

A ténytáblázat kiegészítéseképpen használhatók a *dimenziótáblázatok*, amelyek az egyes dimenziók lehetséges értékeit írják le. A ténytáblázat mindegyik dimenzió attribútuma rendszerint idegen kulcsként működik a megfelelő dimenziótáblázatban, ahogy ezt a 11.13. ábra szemlélteti. A dimenziótáblázat attribútumai azt is leírják, hogy melyek azok a lehetséges csoportosítások, amelyeknek értelmük lenne egy SQL-lekérdezés `GROUP BY` záradékában. A következő példán keresztül bemutatjuk az új fogalmakat.

11.11. példa: A 11.9. példa gépkocsi-adatbázisában a dimenziótáblázatok közül kettő nyilvánvaló:

```
Autók(sorszám, modell, szín)
Forgalmazók(név, város, állam, tel)
```

A ténytáblázat

```
Eladások(sorszám, dátum, forgalmazó, ár)
```

`sorszám` attribútuma idegen kulcs, és az `Autók` dimenziótáblázatban[13] a sorszám attribútumra hivatkozik. Az `Autók.modell` és `Autók.szín` attribútumok pedig az adott autó tulajdonságait írják le. Szerepelhetne még sokkal több attribútum is ebben a relációban, igaz/hamis értékekkel jelölve, hogy a kocsiban van-e automata sebességváltó vagy akármilyen más extra felszerelés. Ha az `Eladások` ténytáblázatot összekapcsoljuk az `Autók` dimenziótáblázattal, akkor a `modell` és `szín` attribútumok szerint különböző érdekes módon lehet csoportosítani a vásárlásokat. Például lebonthatjuk az adatokat szín szerint, vagy a Tüskés modell értékesítéseket hónap és forgalmazó szerint.

[13] Most véletlenül a `sorszám` az `Eladások` relációban is kulcsként működik, de nem kell, hogy legyen olyan attribútum, amely a ténytáblázatnak is kulcsa és ideiglenes kulcs is egyben valamelyik dimenziótáblázatban.

Hasonlóan, az `Eladások` táblázat `forgalmazó` attribútuma idegen kulcs a `Forgalmazó` dimenziótáblázatban a `név` attribútumra hivatkozva. Ha az `Eladások` és `Forgalmazó` táblázatokat összekapcsoljuk, további lehetőségek adódnak az adatok csoportosítására. Például lebonthatjuk a vásárlásokat állam, város vagy forgalmazó szerint.

Eltűnődhetünk azon, hogy hol találjuk az időnek (az `Eladások` táblázat `dátum` attribútumának) megfelelő dimenziótáblázatot. Mivel az idő egy fizikai adottság, nincs értelme az adatbázisban időre vonatkozó tényeket tárolni, hiszen a „melyik évre esik 2000. július 5.?" típusú lekérdezések eredményét úgysem tudjuk megváltoztatni. De minthogy az elemzőknek gyakran van szükségük különböző időegységek, például hét, hónap, negyedév, év szerinti csoportosításra, segít, ha az idő fogalmát beépítjük az adatbázisba, éppen úgy, mintha a következő dimenziótáblázattal rendelkeznénk:

```
Napok(nap, hét, hónap, év)
```

A „reláció" egy jellegzetes, 2000. július 5-nek megfelelő sora lenne a következő:

```
(5, 27, 7, 2000)
```

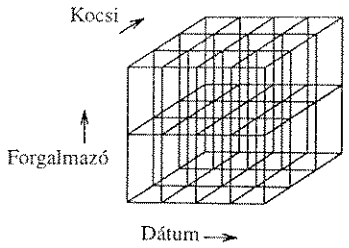
Ezt úgy értelmezzük, hogy ez a nap a 2000. év hetedik hónapjának az ötödik napjára esik, és történetesen a 2000. év 27-edik teljes hetéhez tartozik. Ez bizonyos fokig redundáns, mert a hetet ki lehet számolni a másik három attribútum alapján, viszont a hetek nem illeszthetők pontosan a hónapokhoz, azaz hetek szerinti csoportosításból nem nyerhetünk hónapok szerinti csoportosítást, és megfordítva sem. Tehát van értelme annak, ha úgy képzeljük el, hogy a hetek és a hónapok is jelölve vannak ebben a „dimenziótáblázatban". □

11.3.4. Szeletelés és kockázás

Az adatkockára úgy is gondolhatunk, mintha minden dimenzió mentén valamekkora finomsággal fel lenne osztva. Például az idő dimenziót feloszthatjuk (SQL-es szóhasználattal „csoportosíthatjuk"[14]) napok, hetek, hónapok, évek szerint, de azt is megtehetjük, hogy nem osztjuk fel egyáltalán. Az autó dimenziót feloszthatjuk modell szerint, szín szerint, modell és szín szerint, a forgalmazó dimenziót pedig a forgalmazó neve, a város vagy az állam szerint. Természetesen e két dimenzió esetén is megtehetjük, hogy egyáltalán nem csoportosítunk.

Ha minden dimenzióhoz választunk egy felosztást, ez „felkockázza" az adatkockát, ahogy ez a 11.14. ábrán látható. A kockázás eredményeképpen az adatkockában kisebb kockák jönnek létre. Az ezeket alkotó pontcsoportok jellemzőit az a lekérdezés

[14] Az angol *group by* kifejezésből. *A fordító megjegyzése.*



11.14. ábra. *A dimenziók felosztása darabolja (kockázza) a kockát*

összesíti, amely a `GROUP BY` záradékban a felosztásnak megfelelő csoportosítást hajtja végre. A `WHERE` záradékon keresztül a lekérdezésnek arra is lehetősége van, hogy csak bizonyos felosztásokra koncentráljon egy (vagy több) dimenzió mentén, azaz a kockának csak bizonyos „szeleteivel" foglalkozzon. Ennek az lesz az eredménye, hogy a lekérdezés a kockának csak bizonyos alterein végez összesítést.

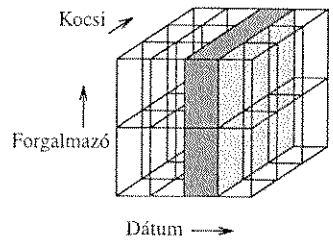
11.12. példa: A 11.15. ábra egy olyan lekérdezésre utal, amely a kockából az egyik dimenzió (dátum) mentén kivág egy szeletet, majd ezt a másik két dimenzió mentén (kocsi és forgalmazó) tovább kockázza. A dátum négy részre van felosztva, talán arra a négy évre, amelyek alatt ezek az adatok összegyűltek. Az ábra satírozása azt akarja kifejezni, hogy ebből a négy évből csak egyetlenegyre vagyunk kíváncsiak.

A kocsik három csoportra vannak osztva, például szedánokra (négyüléses, zárt autókra), kupékra (sportkocsikra) és a kabrioletekre (leereszthető tetejű kocsikra). A forgalmazók kettőre, például a keleti, illetve a nyugati országrészen működőkre. A lekérdezés eredménye egy olyan táblázat, amely a számunkra érdekes évben megadja a gépkocsi-értékesítésekből származó összbevételt mind a hat kategóriában. □

Az úgynevezett „szeletelős-kockázós" lekérdezés általános formája tehát a következő:

```
SELECT csoportosító attribútumok és összesítések
FROM ténytáblázat nulla vagy több dimenziótáblával
     összekapcsolva
WHERE bizonyos attribútumok konstans értékekkel vannak
     összehasonlítva
GROUP BY csoportosító attribútumok;
```

11.13. példa: Folytassuk a személygépkocsis példánkat, de a 11.11. példában megtárgyalt `Napok` fogalmi szintű dimenziótáblázattal együtt. Ha a Tüskés modellt nem vásárolják annyian, mint gondoltuk, megpróbálhatjuk kiszűrni azokat a színeket, amelyek nem mennek olyan jól. Ez a lekérdezés csak az `Autók` dimenziótáblázatot használja, és SQL-ben így nézhet ki:



11.15. ábra. *Egy szelet kiválasztása a darabolt (kockázott) kockából*

```
SELECT szín, SUM(ár)
FROM Eladások NATURAL JOIN Autók
WHERE modell = 'Tüskés'
GROUP BY szín;
```

Először szín szerint kockázza, majd modellenként szeleteli az adatkockát. Eközben a Tüskés modellre összpontosít, és a többi adatot figyelmen kívül hagyja.

Tegyük fel, hogy a lekérdezés eredménye nem mondott sokat, minden színből körülbelül ugyanakkora jövedelmünk származott. Mivel a lekérdezés az időt nem bontotta részekre, a színek szerinti végösszeget az egész időtartamra számította ki. Feltehetjük, hogy egy vagy több szín gyenge szereplése csak az utóbbi időre jellemző. Ekkor megpróbálkozhatunk a következő, immár bővített lekérdezéssel, amely hónapok szerint is particionál:

```
SELECT szín, hónap, SUM(ár)
FROM (Eladások NATURAL JOIN Autók) JOIN Napok ON dátum = nap
WHERE modell = 'Tüskés'
GROUP BY szín, hónap;
```

Fontos észben tartanunk, hogy a `Napok` reláció nem egy szokásos módon tárolt reláció, bár kezelhetjük úgy, mintha a következő sémával rendelkezne.

```
Napok(nap, hét, hónap, év)
```

Az adatkockarendszerek[15] többek között azért is tekinthetők speciális adatbázis-kezelő rendszereknek, mert képesek az efféle „relációk" használatára is.

Az előző lekérdezés alapján esetleg azt állapíthatjuk meg, hogy a piros Tüskések iránt az utóbbi időben nem volt túl nagy kereslet. Következő lépésként megpróbálhatnánk kideríteni, hogy ez általánosan az összes forgalmazóra igaz, vagy csak egy részüknél figyelhető meg ez a probléma. Az újabb lekérdezésben csak a piros Tüskésekre összpontosítunk, és a forgalmazó dimenzió szerint is csoportosítunk. Ily módon a lekérdezés:

[15] Adatkockarendszereknek az adatkocka adatmodellt támogató rendszereket hívjuk. Ezekről bővebben a 11.4. részben lesz szó. *A fordító megjegyzése.*

```
SELECT forgalmazó, hónap, SUM(ár)
FROM (Eladások NATURAL JOIN Autók) JOIN Napok ON dátum = nap
WHERE modell = 'Tüskés' AND szín = 'piros'
GROUP BY hónap, forgalmazó;
```

Ezen a ponton azt vesszük észre, hogy a piros Tüskésekből havonta olyan keveset adtak el, hogy ez alapján nem figyelhető meg könnyen semmilyen jellemző irányvonal. Úgy döntünk tehát, hogy hiba volt a hónapok szerinti felosztás. Jobb ötlet lenne csak évekre bontani az adatokat és csak az utolsó két évet vizsgálni (ebben a képzeletbeli példában 1999-et és 2000-et). A végleges lekérdezés a 11.16. ábrán látható. □

```
SELECT forgalmazó, év, SUM(ár)
FROM (Eladások NATURAL JOIN Autók) JOIN Napok ON dátum = nap
WHERE modell = 'Tüskés' AND
      szín = 'piros' AND
      (év = 1999 OR év = 2000)
GROUP BY év, forgalmazó;
```

11.16. ábra. *A végleges szeletelő-kockázó lekérdezés a piros Tüskés vásárlásokról*

11.3.5. Feladatok

* **11.3.1. feladat:** Egy on-line számítógép-értékesítő cég adatbázisban akarja nyilvántartani a vásárlók megrendeléseit. A vevők a PC-jükbe különböző processzorok, merevlemezek és CD-, illetve DVD-olvasók közül választhatnak, és megadhatják, hogy mekkora központi memóriára van szükségük. Az adatbázis ténytáblázata lehet a következő:

```
Megrendelések(vásárló, dátum, proc, memória, lemez, cd, menny, ár)
```

A `vásárló` attribútum a vevő azonosítójaként működik, és egyúttal idegen kulcs a vásárlókat nyilvántartó dimenziótáblázathoz. Ugyanez igaz a `proc`, `lemez` és `cd` attribútumokra is. A lemezazonosító szerepelhet például abban a dimenziótáblázatban, amely megadja a merevlemez gyártóját és más jellemzőit. A `memória` attribútum egy egész szám, a megrendelt memória méretét adja meg megabájtban. A `menny` attribútum a vásárló által megrendelt konfigurációk számát jelenti, az `ár` attribútum pedig ebből egy darab összköltségét.

a) Melyek a dimenzió attribútumok és melyek a függő attribútumok?
b) Néhány dimenzió attribútumhoz valószínűleg dimenziótáblázat szükséges. Adjunk meg ezekhez megfelelő sémákat!

! **11.3.2. feladat:** Tegyük fel, hogy az előző feladat adatbázisát vizsgáljuk annak érdekében, hogy a jellemző trendek alapján előre jelezhessük a cégnek, hogy milyen alkotórészekből kell majd többet rendelnie. Adjuk meg mélyre ásó és felgörgető lekérdezések egy olyan sorozatát, amely ahhoz a következtetéshez vezethet, hogy a vásárlók egyre inkább előnyben részesítik a DVD-meghajtót a CD-meghajtóval szemben!

Mélyre ásás és felgörgetés

A 11.13. példában az adatkockát szeletelő és kockázó lekérdezések sorozatában két gyakori mintával találkozunk.

1. A *mélyre ásás* az a folyamat, amely során a megfelelő dimenziókat egyre finomabb részekre osztjuk, és/vagy a dimenzió bizonyos értékeire koncentrálunk. A 11.13. példában az utolsó lépés kivételével mindegyik a mélyre ásás folyamatába illik.
2. A *felgörgetés* az egyre durvább particionálás folyamata. Az utolsó két lépést hozhatjuk fel erre példaként, amelyben hónapok helyett évek szerint csoportosítottunk, hogy az adatok esetlegességét kiküszöböljük.

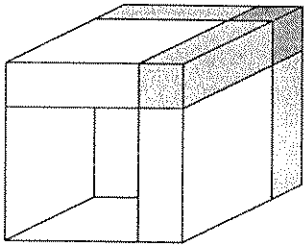
11.4. Adatkocka

A döntéstámogató lekérdezések eddig mindig ad hoc lekérdezések formájában jelentek meg. Ezzel szemben egy másik lehetőség az összes lehetséges összesítés módszeres kiszámítása, előre. Meglepő, de az ehhez szükséges tártöbblet mennyisége gyakran a még ésszerű határon belül marad, és amíg az adattárházban tárolt adat nem változik, az összesített adat frissítése sem jelent többletköltséget. Ebben a fejezetben az adatbázis-kezelő rendszerek egy családjával, az *adatkocka*rendszerekkel, más néven MOLAP- (többdimenziós OLAP) rendszerekkel foglalkozunk. Ezek közvetlenül a 11.12. ábrán bemutatott (adat)kocka adatmodellt támogatják, és a legfontosabb OLAP-műveletek elvégzését is biztosítják.

Az adatkockarendszerekben teljesen hétköznapi dolognak számít, ha még az adatkocka-tároló rendszerbe való felvétel előtt a ténytáblázat nyers adata néhány szempont szerint összesítve van. Az autós adatbázisunkban például a csillag sémában szereplő sorszám dimenziót kicserélhetjük a modell dimenzióra, így minden bejegyzés az adatkockában egy modell, egy forgalmazó, egy dátum, és ezzel együtt az adott modellből az adott napon az adott forgalmazónál történt vásárlások összegének leírásává válik. Az adatkocka pontjainak összességét továbbra is „ténytáblázatnak" nevezzük, még ha a pontok értelmezése egy kicsit el is tér a csillag séma ténytáblázatától.

11.4.1. A kockaművelet

Adott F ténytáblázat esetén definiálhatunk egy kibővített KOCKA(F) táblázatot, amely egy további, *-gal jelölt értéket ad minden dimenzióhoz. A *-nak a jelentése „bármi", és összesítést jelent annak a dimenziónak a mentén, ahol feltűnik. A 11.17. ábra szemlélteti azt az eljárást, amely során a kockához minden dimenzió mentén a * értéket, és az ennek megfelelő összesített értéket képviselő új határvonalakat adjuk hozzá. Az ábrán három dimenziót látunk, a leghalványabb részek jelölik az egy dimenzió mentén vett összesítéseket, a kicsit sötétebbek a két dimenzió mentén, a legsötétebb kis kocka a sarokban pedig a mind a három dimenzió mentén vett összesítéseket. Gondoljuk meg, hogy ha az egyes dimenziók lehetséges értékeinek száma elég nagy, de nem annyira, hogy a legtöbb pont a kocka belsejében szabad legyen (azaz, hogy ne tartozzon hozzá sor a ténytáblázatban), akkor a „határ" csak kis többletet jelent a kocka terjedelméhez (azaz a ténytáblázat sorainak számához) képest. Ebben az esetben a KOCKA(F) táblázat mérete nem sokkal nagyobb az F ténytáblázat méreténél.



11.17. ábra. *A kockaművelet a dimenziók minden lehetséges kombinációján kiszámított összesítéseknek megfelelő határvonalakkal bővíti az adatkockát*

A KOCKA(F) táblázat egy olyan sora, amelyben egy vagy több dimenzió attribútum mentén * található, a függő attribútumoknak megfelelő mezőkben egy összeget (vagy egy más összesítő függvénnyel számított mennyiséget) tartalmaz. Ez úgy számítható ki, hogy vesszük az összes olyan sort, amelyben a dimenzió attribútum értéke nem *, hanem egy valódi érték, és ezen sorok függő attribútumai mentén végezzük el az összesítést. Gyakorlatilag beépítjük az adatokba az összes lehetséges dimenzióhalmazon végzett összesítés eredményét. Vegyük azonban észre, hogy a KOCKA művelet nem támogatja a „köztes szinten" számított összesítéseket. Az adatokat például vagy hagyjuk napokra (illetve az idő dimenzió legfinomabb felbontásának megfelelő időközre) lebontva, vagy a teljes időtartamra nézve összesítünk. A KOCKA művelettel önmagában nem összesíthetünk hetek, hónapok vagy évek szerint.

11.14. példa: A 11.9. példa Süni adatbázisát vizsgáljuk meg a KOCKA művelet lehetőségeinek fényében. Emlékeztetőül, az ott használt ténytáblázat a következő:

```
Eladások(sorszám, dátum, forgalmazó, ár)
```


A `sorszám` attribútumnak megfelelő dimenzió azonban nem alkalmas arra, hogy a kockában használjuk, mert a sorszám egyértelműen azonosítja a gépkocsikat, és így a `sorszám` kulcsként működik az `Eladások` relációban. Vagyis ha a kocsik árát összegezzük a teljes időtartamra vagy az összes forgalmazóra nézve úgy, hogy közben a sorszám rögzítve marad, akkor ennek nem lesz hatása – az adott sorszámú (egyetlen) autó árának az összegét kapjuk. Használhatóbb az adatkocka, ha a sorszámot két másik attribútummal, a modellel és a színnel cseréljük fel. Ezek azok az attribútumok, amelyekhez a sorszám az `Autók` dimenziótáblázaton keresztül az `Eladások` relációt kapcsolja. Vegyük észre, hogy ha a `sorszám` attribútumot kicseréljük a `modell` és `szín` attribútumokra, akkor a kockának már egyetlen dimenziója sem működik kulcsként. Így a kocka egy-egy bejegyzése az olyan kocsik összértékét adja, amelyek adott színűek, adott modellek és az adott napon az adott forgalmazónál vásárolták meg őket.

Ha az `Eladások` ténytáblázatot az adatkockarendszerben akarjuk megvalósítani, egy további változtatás is hasznunkra válhat. Mivel a KOCKA operátor a függő attribútumokat általában összegezni szokta, ezért ha kíváncsiak vagyunk az átlagárra is, szükségünk lehet az eladott kocsik összértékére minden kategóriában (adott szín és modell az adott napon az adott forgalmazónál) és az abba a kategóriába eső vásárlások darabszámára is. Vagyis az `Eladások` reláció, amire a KOCKA műveletet alkalmazzuk, a következő:

```
Eladások(modell, szín, dátum, forgalmazó, összért, db)
```

Az `összért` attribútum az adott modellhez, színhez, dátumhoz és forgalmazóhoz tartozó eladott kocsik összértéke, a `db` pedig az ebbe a kategóriába eső autók darabszáma. Vegyük észre, hogy ebben az adatkockában az egyes autók külön-külön nem azonosíthatók, vagyis ilyen értelemben nincsenek jelen a kockában. Természetes viszont, hogy a saját kategóriájukon belül az `összért` és `db` attribútumok értékére kifejtik a hatásukat.

Vegyük szemügyre a KOCKA(`Eladások`) relációt. Ennek egy lehetséges sora, amely az `Eladások` relációban is megtalálható, a következő:

```
('Tüskés', 'piros', '1999-05-21', 'Undok Ubul', 45000, 2)
```

Ezt úgy értelmezzük, hogy 1999. május 21-én Undok Ubul eladott két piros Tüskést, összesen 45 000 dollár értékben. A következő sor:

```
('Tüskés', *, '1999-05-21', 'Undok Ubul', 152000, 7)
```

azt jelenti, hogy 1999. május 21-én Undok Ubul összesen hét Tüskést adott el, amelyek összértéke 152 000 dollár. Vegyük észre, hogy ez a sor előfordulhat a KOCKA(`Eladások`) relációban, de az `Eladások` biztosan nem tartalmazza.

A KOCKA(`Eladások`) reláció olyan sorokat is tartalmaz, amelyek egynél több attribútum mentén is tartalmaznak összesítést. Például a következő sor:

```
('Tüskés', *, '1999-05-21', *, 2348000, 100)
```

azt jelenti, hogy 1999. május 21-én a forgalmazók összesen 100 Tüskést adtak el, és ezek összértéke 2 348 000 dollár volt.

```
('Tüskés', *, *, *, 1339800000, 58000)
```

Ez a sor pedig azt jelenti, hogy a nyilvántartott egész időtartam alatt a forgalmazók összesen 58 000 Tüskést adtak el (mindenféle színből), összesen 1 339 800 000 dollár értékben. Végül, ebből a sorból:

```
(*, *, *, *, 3521727000, 198000)
```

azt tudhatjuk, meg, hogy a nyilvántartott időtartam alatt a Süni típusú személygépkocsikból összesen 198 000 példányt adtak el (színre és forgalmazóra való tekintet nélkül), és ezek értéke összesen 3 521 727 000 dollár. □

Nézzük meg, hogyan lehet válaszolni az olyan lekérdezésekre, amelyekben az `Eladások` reláció bizonyos attribútumaira feltételeket adunk meg, más attribútumok szerint csoportosítunk, eredményül pedig az összeget, darabszámot vagy az átlagos fogyasztói árat várjuk. A KOCKA(`Eladások`) relációban az olyan t sorokat keressük, amelyek rendelkeznek a következő tulajdonságokkal:

1. Ha a lekérdezés az a attribútum értékeként v-t adja meg, akkor a t sor a mezőjében v szerepel.
2. Ha a lekérdezés a attribútum szerint csoportosít, akkor a t sor a mezőjében tetszőleges nem-* érték szerepel.
3. Ha a lekérdezés nem ad meg a-nak értéket és nem is csoportosít a szerint, akkor a t sor a mezőjében * szerepel.

Minden t sor tartalmazza az összeget és a darabszámot a csoportosítások egyikére nézve. Ha az átlagárra van szükségünk, akkor minden t sorban az összeg és darabszám hányadosát kell venni.

11.15. példa: Ezt a lekérdezést

```
SELECT szín, AVG(ár)
FROM Eladások
WHERE modell = 'Tüskés'
GROUP BY szín;
```

úgy válaszoljuk meg, hogy kikeressük a KOCKA(`Eladások`) reláció minden olyan sorát, amely a következő formában írható:

```
('Tüskés', c, *, *, v, n)
```


A „kocka" különféle fogalmai

A 11.3. részben kezdtünk el „kockákkal" foglalkozni, de ott még ez a fogalom sokkal filozofikusabb jellegű volt. A relációs adatok bizonyos fajtáira hasznos úgy gondolnunk, mintha egy ténytáblázatból és dimenziótáblázatokból állnának. Ebben az esetben a „kocka" a ténytáblázat egy megjelenési formája.

A 11.4. részben bevezettünk egy formális kocka műveletet, amely összesített értékeket számol ki egy vagy több dimenzió mentén. Ez a művelet a 11.3. rész ténytáblázataira is alkalmazható, hogyha csak a mindent-vagy-semmit összegzéseknek van értelmük. A 11.14. példában azonban azt is láttuk, hogyan lehet egy dimenziót, mondjuk az „autók" dimenziót több, a dimenziótáblázatból nyert dimenzióval helyettesíteni, amelyek az autókat más nézőpontból közelítik meg (ebben a példában a kocsikat a sorszámuk helyett a színükkel és a modellel írtuk le). E változtatás után sokkal jobban ki tudtuk használni a KOCKA művelet lehetőségeit. Addig csak kétféle módon csoportosíthattunk: vagy minden autót összevontunk, vagy nem vontuk össze őket egyáltalán. A régi dimenzió lecserélése után azonban már bármely új dimenzió mentén tudtunk összesíteni (modell, szín vagy mindkettő szerint).

A 11.4.2. részben aztán olyan esetekkel ismerkedtünk meg, ahol a dimenzió több, független dimenzióra osztása (mint a kocsik színre és modellre osztása) nem volt elegendő. A dimenziók rendelkezhetnek ennél összetettebb struktúrával, ahol az egyedeket (például a forgalmazókat) különböző finomságú szinteken vagy még egy ennél is bonyolultabb rendszer szerint lehet csoportosítani, ahogy azt az időegységek példáján láttuk. Az olyan dimenziók esetén, amelyek nem oszthatók további dimenziókra, a KOCKA művelet lehetőségeit kevésbé lehet jól kihasználni, de a bizonyos felosztások szerinti előösszesítés a KOCKA művelet egy olyan általánosítása, amely hatékony lehet és számos forgalomban lévő rendszer is alkalmazza.

ahol c bármilyen jellemző szín lehet. Ebben a sorban v a megfelelő színű eladott Tüskések összértékét, n pedig az ilyen színű eladott Tüskések számát adja meg. A lekérdezés (c, v/n) formában írható sorokat vár eredményül. Bár a átlagár nem egy közvetlen attribútum az `Eladások` vagy a KOCKA(`Eladások`) relációban, az értéke kiszámítható az összérték és a darabszám hányadosaként. A lekérdezésre adott válasz tehát a ('Tüskés', c, *, *, v, n) sorokból nyert (c, v/n) párok halmaza. □

11.4.2. Kockaimplementáció megvalósított nézettáblákkal

A 11.17. ábra kapcsán azt állítottuk, hogy a kocka kibővítése az összesített értékekkel nem jár jelentős tártöbblettel, sőt időt takarítunk meg vele a leggyakrabban előforduló döntéstámogató lekérdezések tekintetében. Ez az elemzésünk azonban azon a feltevé-

sen alapult, hogy a lekérdezés vagy mindent átfogóan összesít egy dimenzióra nézve, vagy nem összesít egyáltalán. Néhány dimenzió esetén viszont a csoportosítás többféle finomsági fokon is elvégezhető.

Már említettük az idő dimenziót, ahol számos lehetőségünk van a csoportosításra. Az alapvető „mindent vagy semmit", azaz az egész időtartamra, illetve a napokra való lebontás mellett összesíthetünk például hetek, hónapok, negyedévek vagy évek szerint is. Egy másik példa kedvéért gondoljunk a személygépkocsis adatbázisra. Lehetőségünk volt a forgalmazók teljes vagy „egyáltalán nem" összevonására. Dönthetünk azonban a város, az állam vagy más kisebb-nagyobb területegység szerinti összegzés mellett is. Vagyis az idő szerinti csoportosítás esetén legalább hat, forgalmazók szerint pedig legalább négy választási lehetőségünk van.

Ha a választható csoportosítási szintek száma minden dimenzió mentén növekszik, egyre drágább az összes lehetséges csoportosítási kombináció összesített eredményeit tárolni. Nem csak az jelent gondot, hogy túl sok adatot kell tárolni, hanem az is, hogy nem olyan könnyen szervezhetők, mint a 11.17. ábrán a „mindent vagy semmit" esetén. Ezért a forgalomban lévő adatkockarendszerek segítenek a felhasználónak az adatkocka valamilyen *megvalósított nézettábláit* kiválasztani. A megvalósított nézettábla egy lekérdezés végeredménye, amelyet inkább az adatbázisban tárolunk, mint hogy újra és újra előállítsuk az egészet vagy bizonyos részeit egy lekérdezés megválaszolása közben. A megvalósított nézettáblák rendszerint az egész adatkocka különböző szempontok szerint számított összesítéseit tartalmazzák.

Minél durvább a csoportosításnak megfelelő felosztás, annál kevesebb helyet foglal a megvalósított nézettábla. Másrészről azonban, ha a nézettáblát fel akarjuk használni egy bizonyos lekérdezés megválaszolásához, akkor nem szabad egyetlen egy dimenziót sem durvábban osztani, mint ahogy azt a lekérdezés teszi. Vagyis, hogy maximálisan kihasználhassuk a megvalósított nézettáblákban rejlő lehetőségeket, nagy nézettáblákat veszünk, amelyek elég finoman osztják fel a dimenziókat csoportokra. Hogy milyen nézettáblákat valósítsunk meg, azt még az elemzőktől várt lekérdezések várható típusa is erősen befolyásolja. A következő példán keresztül bemutatjuk, hogy milyen problémákra kell odafigyelni.

11.16. példa: Térjünk vissza a 11.14. példa adatkockájához:

```
Eladások(modell, szín, dátum, forgalmazó, összért, db)
```

Egy lehetséges megvalósított nézettábla a dátumokat hónaponként, a forgalmazókat városonként csoportosítja. Ezt a nézettáblát, amelyet `EladásokN1`-nek nevezünk, a 11.18. ábrán látható lekérdezés hozza létre. Ez nem szigorú SQL-lekérdezés, mivel úgy képzeljük, hogy a dátumokat és a hozzá tartozó csoportosítási egységeket, mint például a hónap, az adatkockarendszer anélkül is értelmezni tudja, hogy az `Eladások` és a 11.11. példa képzetes `Napok` relációját összekapcsolná.

Egy másik lehetséges megvalósított nézettábla a színeket teljesen összevonja, a dátumokat hetenként, a forgalmazókat pedig államonként csoportosítja. Ezt a nézettáblát, az `EladásokN2`-t a 11.19. ábra lekérdezése adja meg. Mindkét nézettábla


```
INSERT INTO EladásokN1
   SELECT modell, szín, hónap, város, SUM(összért) AS összért,
      SUM(db) AS db
   FROM Eladások JOIN Forgalmazók ON forgalmazó = név
   GROUP BY modell, szín, hónap, város;
```

11.18. ábra. *Az* `EladásokN1` *megvalósított nézettábla*

```
INSERT INTO EladásokN2
   SELECT modell, hét, állam,
      SUM(összért) AS összért, SUM(db) AS db
   FROM Eladások JOIN Forgalmazók ON forgalmazó = név
   GROUP BY modell, hét, állam;
```

11.19. ábra. *Az* `EladásokN2` *egy másik megvalósított nézettábla*

használható az olyan lekérdezések megválaszolására, amelyek egyiknél sem bontják finomabb részekre a dimenziókat. Ez a lekérdezés tehát:

```
L1:  SELECT modell, SUM(összért)
FROM Eladások
GROUP BY modell;
```

megválaszolható így:

```
SELECT modell, SUM(összért)
FROM EladásokN1
GROUP BY modell;
```

de akár így is:

```
SELECT modell, SUM(összért)
FROM EladásokN2
GROUP BY modell;
```

Az L_2 lekérdezést

```
L2:  SELECT modell, év, állam, SUM(összért)
     FROM Eladások JOIN Forgalmazók ON forgalmazó = név
     GROUP BY modell, év, állam;
```

azonban csak az `EladásokN1` nézettábla felhasználásával válaszolhatjuk meg, a következő formában:

```
SELECT modell, év, állam, SUM(összért)
FROM EladásokN1
```

```
GROUP BY modell, év, állam;
```

Mellékesen jegyezzük meg, hogy ez a lekérdezés, hasonlóan azokhoz, amelyek valamilyen időegység szerint csoportosítanak, nem szigorú SQL-lekérdezés. Ez azt jelenti, hogy az `EladásokN1` nézettáblának az `állam` nem attribútuma, csak a `város`. Fel kell tennünk, hogy az adatkockarendszer tudja, hogyan lehet városokat államonként összevonni, valószínűleg a forgalmazókat leíró dimenziótáblázat segítségével.

Az L_2 lekérdezéshez nem használható az `EladásokN2` nézettábla. Ugyan a városokat össze tudtuk vonni az államok szerint úgy, hogy az `EladásokN1`-t használni tudjuk, ugyanezt a hetekkel nem tudjuk megtenni, hogy az `EladásokN2`-t is használhassuk, mert az évek nem egyenletesen vannak hetekre osztva. Az 1999. december 29-ével kezdődő hét adatainak egy része például még az 1999-es évhez tartozik, a másik része viszont már 2000-hez. Hogy pontosan hol lehetne a kettőt elválasztani egymástól, azt a hetenként összesített adatokból nem tudjuk megállapítani.

Végül egy olyan lekérdezés, mint:

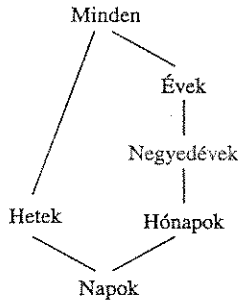
```
L3:  SELECT modell, szín, dátum, SUM(összért)
     FROM Eladások
     GROUP BY modell, szín, dátum;
```

nem válaszolható meg sem az `EladásokN1`, sem az `EladásokN2` segítségével. Nem használhatjuk az `EladásokN1`-t, mert a napok havonkénti összevonása túl durva ahhoz, hogy ez alapján a vásárlásokat napok szerint listázzuk. Nem használható az `EladásokN2` sem, mert ez a nézettábla nem csoportosít színek szerint. Az L_3 lekérdezést közvetlenül az egész adatkocka felhasználásával kellene megválaszolnunk. □

11.4.3. Nézetháló

Hogy a 11.16. példában tett megfigyeléseinket formalizálni tudjuk, hasznunkra válhat egy olyan háló, amely a kocka egyes dimenziói mentén az összes lehetséges csoportosítást tartalmazza. A háló pontjai az adott dimenzióhoz tartozó dimenziótáblázat egy vagy több attribútuma szerinti csoportosítási módot jelölik. Azt mondjuk, hogy a P_1 partíció a P_2 partíció alá esik, ha a P_1 partíció minden egyes csoportját tartalmazza P_2 valamely csoportja. Jelölés: $P_1 \leq P_2$.

11.17. példa: Az idő dimenzió felosztásához választhatnánk a 11.20. ábrán látható hálót. Ha létezik valamilyen P_2 csúcsból P_1-be vezető út, akkor ez azt jelenti, hogy $P_1 \leq P_2$. Az ábrán nem szerepel az összes lehetséges időegység, de mindenesetre jó példáját láthatjuk annak, hogy melyek azok az egységek, amelyeket egy rendszer támogathat. Vegyük észre, hogy a napok a hetek, illetve a hónapok alá esnek, de a hetek nem esnek a hónapok alá. Ennek az az oka, hogy egy adott napon bekövetkezett események biztosan egy megfelelő héten, illetve hónapon belül történtek, de az már nem igaz, hogy egy adott hét folyamán bekövetkezett események összessége szükségszerűen



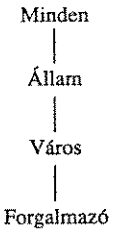
11.20. ábra. *Partíciók hálója az idő intervallumok felosztásához*

egy megfelelő hónapon belül történt volna. Hasonlóan, a hetek eseménycsoportjait nem lehet negyedévek vagy évek eseménycsoportjai alá sorolni. A háló legmagasabb pontjában a „minden" partíció áll. Ez azt jelenti, hogy az események egyetlenegy csoportba vannak beosztva, azaz a különböző időpillanatok között nem teszünk különbséget.

A 11.21. ábrán egy másik háló látható, ezúttal a gépkocsis példánk forgalmazó dimenziójához. Ez egy egyszerűbb háló, amely alapján megtudhatjuk, hogy az autóvásárlások forgalmazó szerinti csoportosítása finomabb, mintha a forgalmazóhoz tartozó város szerint csoportosítottunk volna, a város szerinti felosztás viszont finomabb az állam szerinti felosztásnál. A háló legtetején az a partíció áll, amely az összes forgalmazót egy csoportba osztja be. □

Most, hogy már minden dimenzióhoz meg tudunk adni egy hálót, definiálhatunk egy hálót az adatkocka azon megvalósított nézettábláihoz is, amelyek a dimenziók valamilyen felosztásának megfelelő csoportosítással jönnek létre. Ha N_1 és N_2 a dimenziók valamely partíciója (csoportosítása) alapján megvalósított nézettáblák, akkor $N_1 \leq N_2$ azt jelenti, hogy minden egyes dimenzió N_1-hez használt P_1 partíciója legalább olyan finom, mint ugyanennek a dimenziónak a N_2-höz használt P_2 partíciója, azaz $P_1 \leq P_2$.

A nézettáblák hálójába sok OLAP-lekérdezés is elhelyezhető. Tulajdonképpen az OLAP-lekérdezések gyakran ugyanolyan formájúak, mint az előbb leírt nézettáblák, azaz minden dimenzióhoz megadnak egy-egy felosztást (lehet, hogy a „minden" vagy a „semmi" partíciót). Más OLAP-lekérdezések egy ugyanilyen típusú csoportosítás

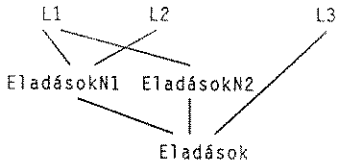


11.21. ábra. *Háló a személygépkocsi-forgalmazók felosztásához*

után még „szeletelik" is a kockát (ahogy a 11.15. ábrán láttuk), hogy az adatoknak csak egy részhalmazára fordítsák a figyelmüket. Az általános szabály a következő:

- Egy L lekérdezés pontosan akkor válaszolható meg a N nézettábla segítségével, ha $N \leq L$.

11.18. példa: A 11.22. ábrán a 11.16. példa nézettábláihoz és lekérdezéseihez adtunk meg egy hálót. Vegyük észre, hogy gyakorlatilag maga az `Eladások` adatkocka is egy nézettábla, ahol minden dimenzió mentén a lehető legfinomabb a felosztás. Ahogy azt az eredeti példában is megfigyeltük, L_1 lekérdezés az `EladásokN1` és `EladásokN2` nézettáblák bármelyikével megválaszolható. Természetesen az egész `Eladások` adatkockát is segítségül hívhattuk volna, de ha a többi nézettáblázat közül legalább az egyik megvalósított (azaz fizikailag jelen van az adatbázisban), akkor nincs okunk az adatkockát választani. L_2 megválaszolható akár az `EladásokN1`, akár az `Eladások` felhasználásával, míg L_3 lekérdezéshez csak az `Eladások` használható. Ezeket a kapcsolatokat a 11.22. ábrán a lekérdezéstől az őket támogató nézettáblákhoz vezető utak fejezik ki. □



11.22. ábra. *Nézettáblák és lekérdezések hálója a 11.16. példához*

Ha a nézettáblák hálójába a lekérdezéseket is beillesztjük, ez segít az adatkocka-adatbázisok tervezésében. Néhány újonnan fejlesztett adatkockarendszer-tervező eszköz az alkalmazás szempontjából „tipikus" lekérdezések halmazából indul ki. Ezután a megvalósítandó nézettáblák halmazát úgy választja ki, hogy minden lekérdezést legalább egy nézettábla támogasson (azaz a hálóban a lekérdezés legalább egy nézettábla fölé essen), de még jobb, ha a lekérdezés valamelyik nézettáblával azonos vagy majdnem azonos (vagyis a legtöbb dimenzió mentén a lekérdezés és a nézettábla ugyanazt a csoportosítást használja).

11.4.4. Feladatok

11.4.1. feladat: Adjuk meg a KOCKA(F) és az F ténytáblázat méretarányát, ha F a következő tulajdonságokkal rendelkezik:

* a) F-nek tíz dimenzió attribútuma van, mindegyik tíz különböző értékkel.
b) F-nek tíz dimenzió attribútuma van, mindegyik két különböző értékkel.

11.4.2. feladat: Ebben a feladatban a 11.14. példa KOCKA(`Eladások`) adatkockát használjuk, amelyet a következő reláció alapján hoztunk létre:


```
Eladások(modell, szín, dátum, forgalmazó, összért, db)
```

Határozzuk meg a kocka azon sorait, amelyeket a következő lekérdezések megválaszolásához használnánk:

* a) Adjuk meg, hogy az egyes forgalmazóknak mennyi bevételük származott a kék autók eladásából!
b) Adjuk meg, hogy „Kedves Kunigunda" a zöld Tüskésekből összesen hányat adott el!
c) Adjuk meg, hogy 2000 márciusában naponta az egyes forgalmazók a különböző színű Tüskésekből átlagosan hány darabot adtak el!

*! **11.4.3. feladat:** A 11.3.1. feladatban egy kockába rendezett adatbázisról volt szó, amelyet számítógép-rendelések tárolására használtunk. Ha alkalmazni akarnánk a KOCKA műveletet, kényelmesebb lenne, ha néhány dimenziót tovább osztanánk. Például az egyetlen processzor dimenzió helyett lehetne egy dimenziónk a processzor típusára (AMD K–6 vagy Pentium–III), egy másik pedig a sebességére. Adjunk meg olyan dimenzió és függő attribútumokat, amelyek sokféle hasznos összesítő lekérdezés végrehajtását támogatják! Mi a szerepe a vásárlónak? A 11.3.1. feladatban az `ár` attribútum egyetlen egy számítógép árát adta meg, holott egyszerre több, ugyanarra a konfigurációra vonatkozó megrendelést is nyilvántarthatnánk ugyanabban a sorban. Melyek lennének a függő attribútumok?

11.4.4. feladat: Határozzuk meg a 11.4.3. feladatban szereplő kocka azon sorait, amelyeket a következő lekérdezések megválaszolásához használnánk:

a) Adjuk meg, hogy 2000-ben havonként összesen mennyit rendeltek a különböző sebességű gépekből!
b) Adjuk meg, hogy összesen hány számítógépet rendeltek, az egyes processzor- és merevlemeztípusok (például SCSI vagy IDE) kombinációira lebontva!
c) Adjuk meg 1999 januárjától kezdve havonként a 400 MHz-es számítógépek átlagárát!

! **11.4.5. feladat:** A 11.4.3. feladatban leírt adatkocka nem tartalmazza a monitorok adatait. Milyen dimenziók segítségével írhatnánk le őket? Feltehetjük, hogy a monitor ára benne van a számítógép árában.

11.4.6. feladat: Tegyük fel, hogy egy kockának 10 dimenziója van, és ezek mindegyikét 5-féle finomsági szinten oszthatjuk fel (beleszámítva a két triviális „minden", illetve „semmi" partíciót is). Hány egymástól eltérő nézettábla nyerhető a dimenziók különböző finomságú felosztásával?

11.4.7. feladat: Mutassuk meg, hogyan illeszthetők be a 11.20. ábrán látható hálóba a következő időegységek: óra, perc, másodperc, két hét, évtized, évszázad!

11.4.8. feladat: Hogyan illesztenénk be a 11.21. ábra forgalmazó hálójába a következő „régiókat", ha:

a) Egy régió az államok egy halmaza.
* b) A régiókat nem lehet államok szerint megadni, de minden várost csak egy régió tartalmaz.
c) A régiók az irányítószámokhoz hasonlóak[16]. Minden régió egy-egy államon belül helyezkedik el, néhány város két vagy több régiónak is része, és néhány régiónak sok város is része lehet.

! **11.4.9. feladat:** A 11.4.3. feladatban egy olyan kockát terveztünk, amelyre alkalmazható a KOCKA művelet. Azonban néhány dimenzióhoz megadható egy nemtriviális hálóstruktúra is. A processzortípust szervezhetnénk például gyártó (SUN, Intel, AMD, Motorola), sorozat (SUN UltraSparc, Intel Pentium vagy Celeron, AMD K-sorozat, Motorola G-sorozat) és modell (Pentium–III, AMD K–6) szerint.

a) A megadott példák alapján tervezzünk hálót a processzortípus dimenzióhoz!
b) Adjuk meg azt a nézettáblát, amely a processzorokat sorozat szerint, a merevlemezeket típus szerint, a CD-meghajtókat sebesség szerint csoportosítja, ezeken kívül pedig minden mást összevon!
c) Adjuk meg azt a nézettáblát, amely a processzorokat gyártó szerint, a merevlemezeket sebesség szerint csoportosítja, ezeken kívül pedig minden mást összevon, kivéve a memóriaméretet!
d) Adjunk példákat olyan lekérdezésekre, amelyekhez

i) csak a b) nézettábla
ii) csak a c) nézettábla
iii) mindkét nézettábla
iv) egyik nézettábla sem

használható!

*!! **11.4.10. feladat:** Amennyiben az F ténytáblázat ritka (azaz sokkal kevesebb sora van, mint a dimenziók lehetséges értékei számának a szorzata), akkor a KOCKA(F) és F táblázatok méretaránya nagyon nagy lehet. Mennyire?

11.5. Adatbányászat

Az adatbázis-alkalmazások egy családja, az *adatbányászat* (data mining) vagy *tudásbányászat* (knowledge discovery in databases) iránt jelentős érdeklődés mutatkozott az utóbbi időben, mert segítségükkel létező adatbázisok alapján meglepő megfigyelések birtokába juthatunk. Az adatbányászó lekérdezésre úgy is gondolhatunk, mint a

[16] Természetesen az Egyesült Államokban használt irányítószámokról van szó. *A fordító megjegyzése.*

döntéstámogató lekérdezések egy kiterjesztett formájára, bár a kettő között formálisan nincs különbség (lásd „Adatbányászati lekérdezések és döntéstámogató lekérdezések" című doboz). Az adatbányászatban a hagyományos adatbázisrendszereknek mind a lekérdezésoptimalizáló, mind az adatkezelő komponense hangsúlyos szerephez jut. Emellett nagyon fontos téma az adatbázisnyelvek kibővítése is. Itt elsősorban olyan nyelvi építőelemekről (nyelvprimitívekről) van szó, amelyek támogatják a hatékony mintavételt. Ebben a részben bemutatjuk, hogy milyen fő irányokat vett az adatbányász alkalmazások fejlődése. Kicsit részletesebben is foglalkozunk a „társítási szabályok" problémakörrel, amely adatbázisos szempontból az utóbbi időben a legtöbb figyelmet kapta.

11.5.1. Adatbányászati alkalmazások

Nagy vonalakban körülírva, az adatbányászó lekérdezések az adatok egy „hasznos" összefoglalását várják eredményül, gyakran a paraméterként a célnak legjobban megfelelő értékre vonatkozó mindenféle útmutató nélkül. Emiatt újra át kell gondolnunk, hogy hogyan nyerhetünk efféle bepillantást az adatok mélyére az adatbázis-kezelő rendszerek segítségével. Ebben a részben néhány olyan alkalmazást és problémát tekintünk át, amelyek nagyon nagy mennyiségű adattal kapcsolatban kerülnek elő. Mivel sok esetben még nyitott kérdés, hogy hogyan használhatók legjobban az ABKR-ek az adott problémával kapcsolatban, a megoldásokat itt nem tárgyaljuk, csupán néhány szóban utalunk arra, hogy miből áll a feladat nehézsége. A 11.5.2. részben viszont egy olyan probléma tárgyalásába kezdünk, amellyel kapcsolatban figyelemre méltó haladás következett be. Az utolsó részben bemutatunk egy nem triviális, adatbázis-orientált megoldást.

Döntési fa építése

Egy adatbázis felhasználói az adatok alapján egy fontos kérdést akarnak eldönteni. Az 5.7. példában egy olyan kérdést tettünk fel, amely akár egy érdekes adatbányászós probléma alapja is lehetne: „Ki vásárol arany ékszereket?". Abban a példában a vásárlóknak csak két tulajdonságával foglalkoztunk: az életkorukkal és a jövedelmükkel. A mai adatbázisok azonban sokkal több információt is felvehetnek a vásárlókról, közvetlenül vagy legitim források adatait integrálva egy tárházba. Ilyen lehet például a vásárló irányítószáma, családi állapota, lakáshelyzete vagy az általa beszerzett árucikkek különböző jellemzői.

Az 5.7. példával ellentétben, ahol az adatbázis csak olyan embereket tart nyilván, akik már vásároltak arany ékszereket, a *döntési fa* (decision tree) egy olyan eszköz, amely segítségével az adatok két részre bonthatók, az „igen-halmazra" illetve a „nem-halmazra". Az arany ékszerek esetében ezek az adatok emberekről nyilvántartott információkat jelentenének. Az igen-halmazba sorolnánk azokat, akikről valószínűnek tartjuk, hogy vásárolnának arany ékszereket, a nem-halmazba pedig azokat,

Adatbányászati lekérdezések és döntéstámogató lekérdezések

A döntéstámogató lekérdezéseknek lehet, hogy az adatbázis nagy részét kell megvizsgálniuk és összesíteniük, cserébe viszont a kérdést feltevő elemző pontosan megadja a végrehajtandó lekérdezést, azaz tudtára adja a rendszernek, hogy az adat mely részeivel foglalkozzon. Az adatbányászati lekérdezés még egy lépést tesz előre. A rendszerre hagyja annak az eldöntését, hogy a lekérdezés szempontjából az adatbázis melyik része lehet fontos. Egy döntéstámogató lekérdezés például „összesítsük a Süni gépkocsik eladását szín és év szerint" az adatbányászat nyelvén így hangzana: „mely tényezők befolyásolták legjobban a Süni gépkocsik eladását?". Az adatbányászati lekérdezések naiv megvalósításai nagyszámú döntéstámogató lekérdezés végrehajtását jelentik, és ezért olyan sok időbe telhet a válaszadás, hogy ez a fajta megközelítés teljesen használhatatlanná válik.

akikről nem tartjuk valószínűnek, hogy vásárolnának arany ékszereket. Ha az előrejelzésünk megbízható, máris rendelkezésünkre áll egy megfelelő célcsoport, amely tagjainak például közvetlen levél útján küldhetünk reklám-összeállítást az aranyékszerkínálatunkról.

Maga a döntési fa valahogy úgy nézne ki, mint az 5.13. ábrán, azzal a különbséggel, hogy a levelek nem tartalmaznának számszerű értékeket. Minden belső csúcshoz tartozik egy attribútum és egy küszöbértékként szolgáló attribútumérték. Egy belső pont közvetlen leszármazottja vagy szintén belső pont vagy levél. A leveleket döntési csúcsnak nevezzük, és „igen" vagy „nem" érték tartozhat hozzájuk. Egy reláció adott sorában található adat kiértékelése úgy történik, hogy a döntési fa legtetejéről indulva minden lépésben a belső csúcsokhoz tartozó attribútumérték alapján hol a bal, hol a jobb oldali él mentén haladunk tovább, egészen addig, amíg egy döntési csúcshoz nem érkezünk.

A döntési fát az adatok egy olyan *mintahalmaza* (training set) alapján építjük, amelyekről tudjuk, hogy az igen-halmazba vagy a nem-halmazba tartoznak. Az arany ékszerek esetén vesszük a vásárlók adatbázisát, amely arról is tartalmaz információkat, hogy az egyes vevők vásároltak-e már arany ékszereket vagy sem. Adatbányászati probléma ezen adatok alapján egy olyan döntési fát létrehozni, amely a legnagyobb biztonsággal dönt egy adott tulajdonságokkal (életkor, jövedelem és így tovább) rendelkező új vásárló esetén annak a valószínűségéről, hogy fog-e arany ékszert venni vagy nem. Azaz meg kell határozni a legjobb A attribútumot és a hozzá tartozó legjobb v küszöbértéket, amit a gyökérre írhatunk, majd hasonlóan a gyökér bal- illetve jobboldali leszármazottjára írandó optimális attribútumot, és a küszöbértéket arra az esetre, ha a besorolandó új vásárlóhoz tartozó A attribútum értéke kisebb, illetve legalább akkora, mint v. Ugyanezt a problémát kell megoldanunk a döntési fa minden szintjén, egészen addig, amíg már nem érdemes újabb pontokat felvennünk a gráfba (mert a mintahalmazból túl kevés adat ér el egy adott csúcspontot, hogy ez alapján

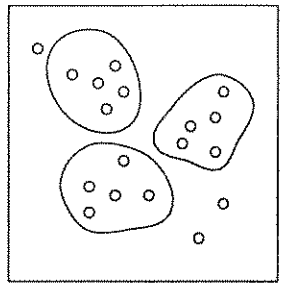
hasznos döntést hozhatnánk). A csúcspontok tervezése során használatos lekérdezések az adatok legnagyobb részét összesítik, így ezek alapján meg tudunk adni egy olyan attribútum-küszöbérték párosítást, amely a mintaadatok „igen" felének nagy részét elválasztja a mintaadatok „nem" felének nagy részétől.

Csoportosítás

Az adatbányászati problémák egy másik osztályába tartozik a „csoportosítás" kérdéskör. A feladat az, hogy az adatokat beosszuk néhány csoportba úgy, hogy az egyes csoportok elemeiben legyen valami lényeges vonás, ami közös. A 11.23. ábrán kétdimenziós adatok csoportosítása látható, a gyakorlatban természetesen a dimenziók száma sokkal nagyobb is lehet. Az ábrán a három legjobb csoport hozzávetőleges körvonalait rajzoltuk be, bár néhány pont mindegyik csoport központjától távol esik. Ezeket tekinthetjük „kívülállóknak", de az egyes pontokat hozzá is csaphatjuk a legközelebb eső csoporthoz.

Egy mintaalkalmazás kedvéért nézzük meg, hogyan működnek az internetes keresőprogramok. Ezek a programok gyakran százezrével találnak olyan weboldalakat, amelyek az adott keresési feltételnek megfelelnek. Hogy ezeket megfelelően lehessen rendszerezni, a keresőprogramok talán a használt szavak alapján csoportosítják a dokumentumokat. Vehetjük például azt a teret, amelyben minden szónak megfelel egy-egy dimenzió, talán a leggyakoribb szavakat, mint „és", „az", „a" *(leállító szavak)* kivéve, amelyek lényegében minden dokumentumban előfordulnak a tartalomtól függetlenül. Minden dokumentum elhelyezhető ebben a térben aszerint, hogy az egyes szavak milyen arányban fordulnak elő a szöveges részében. Ha például egy weboldalon 1000 szó található összesen és ebből kettő az „adatbázis", akkor ezt az „adatbázis" szónak megfelelő dimenzió mentén a 0,002 koordinátához igazítjuk. Ha ebben a térben csoportosítjuk az adott dokumentumokat, akkor a csoportok a dokumentumok témái szerint fognak szerveződni. Például az adatbázisokról szóló szövegekben előfordulhatnak olyan szavak, mint „adat", „lekérdezés", „zárolás" és így tovább, viszont a foci témájú szövegek aligha tartalmaznak ezekhez hasonlókat.

Itt az jelenti az adatbányászati problémát, hogy hogyan határozzuk meg a csoportok



11.23. ábra. *Kétdimenziós adatok három csoportja*

„középértékét" vagy középpontját az adott adatok alapján. Gyakran a csoportok száma előre rögzített, bár ezt az értéket az adatbányászati eljárás is megválaszthatja. Mindkét esetben azonban az a naiv algoritmus, amely úgy választja meg a csoportok középpontját, hogy egy tetszőleges pont átlagos távolsága a hozzá legközelebbi középponttól minimális legyen, számos bonyolult összesítést végrehajtó lekérdezést foglal magában.

11.5.2. Társítási szabály bányászat

Ebben a fejezetben egy olyan adatbányászati problémával ismerkedünk meg, amelyhez eredményesen fejlesztettek ki olyan algoritmusokat, amelyek a háttértárakat használják. A feladat, az úgynevezett „társítási szabályok" keresése, legkönnyebben a legfontosabb alkalmazásán, a *bevásárlókosár-adatok* elemzésén keresztül írható le. A mai áruházak gyakran tárolnak arról feljegyzéseket az adattárházban, hogy egyes vásárlók milyen árucikkeket vásároltak egyszerre. A vevő a teli „bevásárlókosarával" beáll a pénztárhoz, a pénztáros pedig az összes cikket, amit a kosárban talál, egyetlen tranzakció formájában rögzíti. Így aztán, ha magáról a vevőről nem is tudunk semmit, és azt sem tudjuk megmondani, hogy vásárol-e még valaha ebben az üzletben, bizonyos cikkekről *határozottan* tudjuk, hogy ezeket valaki egyszerre vásárolta.

Ha néhány cikk gyakrabban fordul elő együtt a kosarakban, mint az egyébként várható lenne, akkor az áruház ez alapján következtetéseket vonhat le arra vonatkozólag, hogy a vásárlók milyen útvonalon járják be az üzletet. Az árucikkek elrendezhetők úgy, hogy a vevők kénytelenek legyenek bizonyos útvonalakon végigmenni, és ezek mentén vonzó cikkeket lehet elhelyezni.

11.19. példa: Egy híres példa, sokak által megfigyelt jelenség, hogy azok az emberek, akik eldobható pelenkát vásárolnak, nagy valószínűséggel sört is vesznek. Hogy miért áll fenn ez a kapcsolat a két árucikk között, arra több elmélet is született. Többek között azzal is magyarázzák, hogy a pelenkavásárló emberek, mivel kicsi gyerekük van, ritkán járnak kocsmába, ezért otthon isszák a sört. Az áruházak aztán így vagy úgy kihasználhatják azt a tényt, hogy sok vásárló a pelenkáktól a sör felé veszi az útját, vagy fordítva. Az ötletes kereskedő burgonyaszirmot tesz a sör és a pelenka közé. Az állítás az, hogy ilyenkor mindhárom cikk iránt növekszik a kereslet. □

A bevásárlókosár-adatokat tárolhatjuk a következő relációban:

```
Kosarak(kosár, árucikk)
```

Az első attribútum egy „kosárazonosító", amely egyértelműen azonosít minden bevásárlókosarat, a második attribútum pedig a kosárban található cikk azonosítója. Vegyük észre, hogy a reláció szempontjából lényegtelen, hogy az adatok valódi bevásárlókosár-adatok legyenek. Bármilyen olyan adat megfelel, ahol a cikkek közötti kapcsolatokat akarjuk vizsgálni. A „kosarak" lehetnének például dokumentumok, a „cikkek" pedig szavak, ami azt jelenti, hogy valójában olyan szavakat keresünk, amelyek sok dokumentumban együtt fordulnak elő.

A *társítási szabály* legegyszerűbb formája bevásárlókosár-adatok esetén a cikkek egy halmaza. Az $\{i_1, i_2, \ldots, i_n\}$ cikkhalmazok jelentősége változó lehet. Egy halmaz legelemibb tulajdonsága, amelyet vizsgálhatunk, hogy azon kosarak száma, amelyekben a halmaz *minden* eleme szerepel, nagy. Egy cikkhalmaz *tartója* azon kosarak száma, amelyekben a halmaz minden eleme megtalálható. *Erős tartójú cikkhalmazok* felkutatása azt jelenti, hogy adott s küszöbértékhez meg kell találnunk az összes olyan cikkhalmazt, amely tartója legalább s.

Ha az adatbázisba felvett cikkek száma nagy, akkor még abban az esetben is, ha csak kis elemszámú cikkhalmazokkal, mondjuk cikkpárokkal foglalkozunk, az összes kérdéses cikkhalmaz tartóját kiszámítani nagyon sok időt vehet igénybe. Így a kézenfekvő megoldás az erős tartójú cikkhalmazok felkutatására még cikkpárok esetén sem működik. Ilyenkor, mint ahogy erre a 11.24. ábrán látható SQL-lekérdezés utal, minden $\{i, j\}$ árucikkpár tartóját ki kell számítanunk. A lekérdezés összekapcsolja a `Kosarak` relációt saját magával, az így kapott sorokat a bennük található két árucikk szerint csoportosítja, majd a csoportok közül kihagyja azokat, amelyekben a kosarak száma (azaz a tartó) nem éri el az s küszöbértéket. Érdemes végiggondolni, hogy a `WHERE` záradékban rögzített `I.árucikk < J.árucikk` feltétel mire használható. Ennek segítségével előzhető meg, hogy ugyanaz az $\{i, j\}$ pár fordított sorrendben, azaz $\{j, i\}$ alakban is előforduljon, továbbá, hogy az $\{i, i\}$ típusú, vagyis ugyanabból az egy elemből alkotott „párok" egyáltalán létrejöjjenek.

```
SELECT I.árucikk, J.árucikk, COUNT(I.kosár)
FROM Kosarak I, Kosarak J
WHERE I.kosár = J.kosár AND
      I.árucikk < J.árucikk
GROUP BY I.árucikk, J.árucikk
HAVING COUNT(I.kosár) >= s;
```

11.24. ábra. *Naiv megoldás az erős tartójú cikkpárok felkutatására*

11.5.3. Az előzetes algoritmus

A 11.24. ábrán látható lekérdezés futási ideje optimalizálható abban az esetben, ha az s küszöbérték elég magas ahhoz, hogy csak néhány pár feleljen meg a feltételnek. Ésszerű magas s értékkel dolgoznunk, hiszen cikkpárok százaival, ezreivel úgysem mennénk semmire. Az adatbányászati lekérdezéstől azt várjuk, hogy egy kis létszámú, de a lehető legjobb cikkpárokat tartalmazó halmazra hívja fel a figyelmünket. Az *előzetes* algoritmus a következő megfigyelésen alapul:

- Ha egy X cikkhalmaz tartója s, akkor minden $Y \subseteq X$ cikkhalmaz tartója is legalább s.

Például ha az $\{i, j\}$ cikkpár 1000 kosárban szerepel, akkor világos, hogy i is és j is feltűnik ugyanezekben a kosarakban, vagyis létezik legalább 1000 kosár az i cikkhez és 1000 kosár a j cikkhez is.

A fenti szabály megfordításából adódik, hogy ha legalább s tartójú cikkpárokat ke-

A társítási szabály más formái

A társítási szabály egy általánosabb formája a cikkhalmazt egy másik cikkel hozza kapcsolatba. A szabály $\{i_1, i_2, \ldots, i_n\} \Rightarrow j$ alakban írható. Ezzel az alakkal két tulajdonság adható meg:

1. *Bizonyosság:* Annak a valószínűsége, hogy egy $\{i_1, i_2, \ldots, i_n\}$ cikkeket tartalmazó kosárban j árucikk is megtalálható, egy bizonyos küszöbérték, például 50% felett van. Például „a pelenkavásárlók legalább 50%-a sört is vásárol".
2. *Érdekesség:* Annak a valószínűsége, hogy egy $\{i_1, i_2, \ldots, i_n\}$ cikkeket tartalmazó kosárban j árucikk is megtalálható, lényegesen magasabb vagy alacsonyabb annak a valószínűségénél, hogy j egy véletlenül választott kosárban előfordul. Statisztikai szóhasználattal j pozitívan vagy negatívan korrelál az $\{i_1, i_2, \ldots, i_n\}$ halmazzal. A 11.19. példa megfigyelése valójában az volt, hogy a {pelenka} ⇒ sör társítási szabály nagyon érdekes.

Vegyük észre, hogy egy nagy bizonyosságú vagy érdekességű szabály legtöbb esetben csak akkor lesz igazán használható, ha a kérdéses árucikkeknek is erős a tartójuk. Ez azért van így, mert ha a tartó nem erős, akkor a szabály előfordulásainak a száma sem magas, ami korlátozza az adott szabályt hasznosító stratégiával járó előnyöket.

resünk, akkor eleve kizárhatók azok a cikkek, amelyek – más cikkektől függetlenül – nem fordulnak elő legalább s kosárban. Ennek megfelelően az *előzetes algoritmus* a következő lépéseket hajtja végre:

1. Keressük meg a „jó" cikkeket – vagyis azokat, amelyek megtalálhatók elegendő számú kosárban, majd
2. Futtassuk csak a jó cikkeken a 11.24. ábra lekérdezését.

Az előzetes algoritmus az egymás után végrehajtott két SQL-lekérdezés formájában a 11.25. ábrán látható. Először feltölti a `JóKosarak` relációt az elég erős tartójú cikkekkel, azaz a `Kosarak` reláció egy megfelelő részhalmazával, majd a 11.24. ábra naiv algoritmusának mintájára összekapcsolja a `JóKosarak` relációt saját magával.

11.20. példa: Hogy érezzük, mennyit segít az előzetes algoritmus, nézzük meg, hogyan működik egy 10 000 árucikkes élelmiszer-áruház esetén. Tegyük fel, hogy átlagosan 20 cikk van egy-egy kosárban, és hogy az adatbázis 1 000 000 kosár adatait tárolja (a valósághoz képest még ez az érték is kicsiny). Ekkor a `Kosarak` reláció 20 000 000 sorból áll és a naiv algoritmus összekapcsolása után 190 000 000 pár jön létre, ugyanis az 1 000 000 kosárban $\binom{20}{2} = 190$-féle cikkpár szerepelhet összesen. A csoportosítást és a számlálást tehát 190 000 000 soron kell végrehajtani.


```
INSERT INTO JóKosarak
   SELECT *
   FROM Kosarak
   WHERE árucikk IN (
      SELECT árucikk
      FROM Kosarak
      GROUP BY árucikk
      HAVING COUNT(*) >= s
   );

SELECT I.árucikk, J.árucikk, COUNT(I.kosár)
FROM JóKosarak I, JóKosarak J
WHERE I.kosár = J.kosár AND
      I.árucikk < J.árucikk
GROUP BY I.árucikk, J.árucikk
HAVING COUNT(*) >= s;
```

11.25. ábra. *Az előzetes algoritmus az erős tartójú cikkpárok felkutatását az erős tartójú cikkek keresésével kezdi*

Tegyük fel azonban, hogy s = 10 000, azaz a kosarak számának 1%-a. Lehetetlen, hogy több, mint 20 000 000/10 000 = 2000 cikk legalább 10 000 kosárban szerepeljen, hiszen a `Kosarak` relációban csak 20 000 000 sor van, és minden cikk, amely 10 000 kosárban megjelenik, a reláció 10 000 sorában is feltűnik. Vagyis a 11.25. ábra előzetes algoritmusában az erős tartójú cikkeket kereső lekérdezés nem eredményezhet 2000-nél több cikket, de valószínűleg ennél sokkal kevesebbet fog találni.

Nem tudhatjuk előre, hogy a `JóKosarak` reláció mekkora lesz, a legrosszabb esetben ugyanis a `Kosarak` relációban előforduló *összes* árucikk megjelenik a kosarak 1%-ában is. Ha azonban s elég nagy, akkor a `JóKosarak` reláció a gyakorlatban lényegesen kisebb lesz, mint a `Kosarak` reláció. Tegyük fel például, hogy a `JóKosarak` relációban egy-egy kosár átlagosan 10 árucikket tartalmaz, azaz a reláció fele akkora, mint a `Kosarak` reláció. Ekkor a második lépésben az összekapcsolás után $1\,000\,000 \times \binom{10}{2} = 45\,000\,000$ sort kapunk, azaz kevesebb, mint a negyedét annak a sormennyiségnek, amit a `Kosarak` reláció önmagához kapcsolása eredményez. Ezek alapján az várható, hogy az előzetes algoritmus körülbelül negyedannyi időt vesz igénybe, mint a naiv algoritmus. Legtöbb esetben, ahol a `JóKosarak` reláció a `Kosarak` relációnak jóval kevesebb, mint a felét tartalmazza, a futási idő csökkenése még nagyobb mértékű. Általában, ha az összekapcsolásban részt vevő sorok számát n-ed részére csökkentjük, akkor a futási idő n^2-ed részére csökken. □

11.6. Összefoglalás

- *Információk egyesítése:* Gyakran előfordul, hogy többféle adatbázis vagy más információforrás egymással összefüggő adatokat tartalmaz. Megvan a lehetőség arra, hogy ezeket a forrásokat egyesítsük. Gyakran azonban heterogén forrásokkal van dolgunk. Az inkompatibilitás sokféle formában jelentkezhet: ugyanazokhoz az értékekhez különböző típus, kód, illetve konvenció tartozhat, azonos fogalmakhoz egymástól eltérő értelmezések adhatók, az egyes sémákban pedig megvannak a fogalmi különbségek.
- *Az információegyesítés megközelítési módjai:* A korai módszerek közé sorolható a „szövetség" létrehozása, ahol az egyes adatbázisok egymás nyelvén intézhetik egymáshoz a lekérdezéseket. Ennél újabb módszer az adattárház használata, ahol az adatokat egy globális sémának megfelelően alakítjuk át, és a tárházban tároljuk a másolatokat. Ennek egy alternatívája a közvetítő rendszer, ahol egy virtuális adattárháznak tehetők fel a globális sémának megfelelő lekérdezések, amelyeket az egyes adatforrások sémáihoz igazítva alakítunk át.
- *Adatkinyerő és borítékoló:* A tárház, illetve közvetítő rendszerek mindegyike tartalmaz egy, az adatforrásokhoz rendelt alkotóelemet, az adatkinyerőt illetve a borítékolót. Fő feladatuk a lekérdezések és az eredmények átalakítása a globális, illetve a lokális séma szerint.
- *Borítékoló generátor:* A borítékoló tervezésének egy módja a borítékoló sablonok használata, amelyek leírják, hogy egy speciális formájú globális sémának megfelelő lekérdezés hogyan alakítható át a lokális séma nyelvére. A sablonok táblázatba foglalása és értelmezése egy meghajtó feladata, amely majd az adott lekérdezéshez kiválasztja a neki megfelelő sablont, ha van ilyen. A meghajtó képes lehet arra, hogy a sablonokat különféle módon variálja, és/vagy összetettebb lekérdezések esetén további feladatokat is (például az adatok szűrését) elvégezzen.
- *OLAP:* Az adattárházak egy fontos alkalmazását jelenti az a lehetőség, hogy miközben az adatforrásokon a szokásos tranzakciófeldolgozó eljárások működnek, feltehetők a tárház egészét vagy nagy részét érintő bonyolult, általában összesítő jellegű lekérdezések. Ezek a lekérdezések az on-line analitikus adatfeldolgozó vagy OLAP-lekérdezések.
- *ROLAP és MOLAP:* Ha OLAP-alkalmazás céljából építünk adattárházat, gyakran hasznos, ha az adatra egy kocka képében gondolunk, ahol a kocka dimenziói mentén az adatot más és más megvilágításban látjuk. Az olyan rendszer, amely ezt az adatmodellt támogatja, vagy relációs szempontból tekint a kockára (ROLAP- vagy relációs OLAP-rendszerek), vagy az adatkockára specializálódik (MOLAP- vagy többdimenziós OLAP-rendszerek).
- *Csillag séma:* ROLAP-megközelítés esetén minden adatelemet (például egy árucikk eladását) egy relációban, a ténytáblázatban tárolunk, a dimenziók különböző értékeinek az értelmezéséhez (például az 1234 azonosítójú cikk milyen típusú termék?) szükséges információkat pedig az egyes dimenziókhoz tartozó dimenziótáblázatokban. Ezt a típusú adatbázissémát csillag sémának nevezzük. A ténytáblázat a csillag központja, a dimenziótáblázatok pedig a csillag csúcsai.

- *A KOCKA művelet:* MOLAP-megközelítés esetén hasznosnak bizonyul, ha a ténytáblázat dimenzióinak lehetséges részhalmazai mentén végrehajtunk egy előösszesítést. Az így kibővített táblázat kicsit több helyet foglal, mint az eredeti ténytáblázat, de segítségével nagymértékben csökkenthető az OLAP-lekérdezések futási ideje.
- *Dimenzióhálók és megvalósított nézettáblák:* Néhány adatkocka-megvalósítás a KOCKA műveletnél is hatékonyabb eszközt használ: minden dimenzióhoz létrehoz egy-egy hálót, amely az adott dimenzió – összesítésekhez használható – különböző finomsági fokait tartalmazza (például különböző időegységeket, mint nap, hónap, év). Az adattárházba aztán bizonyos megvalósított nézettáblák is bekerülnek, amelyek különféle módokon más és más dimenziók mentén összesítik az adatokat. Egy lekérdezés megválaszolására aztán a vele leginkább megegyező nézettábla használható.
- *Adatbányászat:* A tárházakon olyan tág értelmű lekérdezések is végrehajthatók, amelyek nem csak az előre meghatározott összesítéseket végzik el (miként az OLAP-lekérdezések), hanem keresik a „megfelelő" összesítést. Az adatbányászat gyakran előforduló típusai: az adatok hasonló csoportokba osztása; döntési fa tervezése, amely egy adott attribútum értékét jósolja meg a többi attribútum értéke alapján; sokféle érték között gyakran előforduló párokra vonatkozó társítási szabályok keresése.
- *Előzetes algoritmus:* Az előzetes algoritmus használatával hatékonyan tudunk társítási szabályok után kutatni. Ez a technika azt a tényt használja ki, hogy ha egy halmaz gyakran előfordul, akkor annak minden részhalmaza is gyakori.

11.7. Irodalomjegyzék

Az adattárház-rendszerekről és a kapcsolódó technológiákról ad áttekintést [10], [4] és [8]. Az adatbázis-szövetségekről [12]-ben van szó. A közvetítő fogalma [14]-ből származik. A közvetítő és a borítékoló megvalósításával, különös tekintettel a borítékoló generátor alkalmazására, [6] foglalkozik.

Manapság a legtöbb információegyesítő módszer a „felstrukturált" adatmodellen alapszik, amelynek segítségével megoldható a hiányzó értékek problémája, és a sémák közti egyéb különbségek is áthidalhatók. Az adatmodell ötlete [11]-ből származik; [1] és [13] áttekintést nyújt a témával kapcsolatban.

A KOCKA műveletet [7] vetette fel. A megvalósított nézettáblák segítségével megvalósított adatkockákról [9]-ben esik szó.

Adatbányászati technikák áttekintését adja [5]. Az előzetes algoritmussal [2] és [3] foglalkozik.

1. S. Abiteboul, „Querying semi-structured data," *Proc. Intl. Conf. on Database Theory* (1997), Lecture Notes in Computer Science 1187 (F. Afrati and P. Kolaitis, eds.), Springer-Verlag, Berlin, pp. 1–18.

2. R. Agrawal, T. Imielinski, and A. Swami, „Mining association rules between sets of items in large databases," *Proc. ACM SIGMOD Intl. Conf on Management of Data* (1993), pp. 207–216.
3. R. Agrawal, and R. Srikant, „Fast algorithms for mining association rule," *Proc. Intl. Conf. on Very Large Databases* (1994), pp. 487–499.
4. S. Chaudhuri and U. Dayal, „An overview of data warehousing and OLAP technology," *SIGMOD Record* **26**:1 (1997), pp. 65–74.
5. U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, *Advances in Knowledge Discovery and Data Mining*, AAAI Press, Menlo Park CA, 1996.
6. H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, V. Vassalos, J. D. Ullman, and J. Widom, The TSIMMIS approach to mediation: data models and languages, *J. Intelligent Information Systems* **8**:2 (1997), pp. 117–132.
7. J. N. Gray, A. Bosworth, A. Layman, and H. Pirahesh, „Data cube: a relational aggregation operator generalizing group-by, cross-tab, and subtotals," *Proc. Intl. Conf. on Data Engineering* (1996), pp. 152–159.
8. A. Gupta and I. S. Mumick, *Materialized Views: Techniques, Implementations, and Applications*, MIT Press, Cambridge MA (1999).
9. V. Harinarayan, A. Rajaraman, and J. D. Ullman, „Implementing data cubes efficiently," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1996), pp. 205–216.
10. D. Lomet and J. Widom (eds.), Special issue on materialized views and data warehouses, *IEEE Data Engineering Bulletin* **18**:2 (1995).
11. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom, „Object exchange across heterogeneous information sources." *Proc. Intl. Conf. on Data Engineering* (1995), pp. 251–260.
12. A. P. Sheth and J. A. Larson, „Federated databases for managing distributed, heterogeneous, and autonomous databases," *Computing Surveys* **22**:3 (1990), pp. 183-236.
13. D. Suciu (ed.), Special issue on management of semistructured data, *SIGMOD Record* **26**:4 (1997).
14. G. Wiederhold, „Mediators in the architecture of future information systems," *IEEE Computer* **C-25**:1 (1992), pp. 38–49.

Index

adatbáziscím 151
Abiteboul, S. 671
abort 603, 616
ABORT naplóbejegyzés 465
abortálás, vagy leállítás a befejezés előtt, vagyis sikertelen befejezés 498, 551
Achilles, Alf-Christian 45
ACID-tulajdonságok 34
adat 24, 33
adatbányászat 624, 662
adatbázis 26
adatbázis-adminisztrátor 30
adatbázis-állapot 459, 617
adatbáziscím 129
adatbázis-címterület 126
adatbáziselem 458, 538
adatbázis-kezelő 24
adatbázis-programozás 36
adatbázisséma 39
adatbázis-szövetség 624, 627
adatbázis-tervezés 36
adatdefiníciós nyelv 30
adategyesítő 631
adatelem 111
adatfájl 32, 154
adatforrás 624
adatkocka 218, 624, 644, 651
adatközvetítő 625, 631, 635
adatlemezek 95
adatraktár 38
adatszolgáltató 631
adattárház 38, 624, 628, 642
Agrawal, R. 672
Aho, A.V. 361, 429
algebrai szabályok 367, 391
alkérdés 385, 386
alulról felfelé tervválasztás 419
archívmentés 490, 494
archiválás 454, 456, 490, 492
archiválás működés közben 492
Astrahan, M. M. 45, 453
asszociatív szabály 368, 370, 393
asszociatív törvény 98
át nem nyúló rekord 143
átlag 298
átlapolás 505
átnyúló rekord 142
atom 360
atomos tranzakció 460
atomosság 25, 34, 601
attribútum 38, 122, 366
átviteli idő 61
automatikus helyreigazítás 131

Baeza-Yates R. 215
bal-mély összekapcsolási fa 425, 429
Barghouti, N. S. 623
Bayer R. 215
blokkcím 125
beágyazott ciklusú összekapcsolás 303, 424
Beekmann, N. 263
belső csúcs 186
bemeneti attribútum 375
Bentley, J. L. 263
Bernstein, P. A. 45, 497, 568
beszúrás 146, 182, 191, 202, 205, 209, 532, 542
bevásárlókosár-adat *lásd* társítási szabály
B-fa 184, 221, 224, 334, 415, 544, 578
bináris, nagy objektum 143, 151
bit 114
bitsorozat 118
bittérképindex 217, 253, 255, 256, 258, 259, 260, 261
bizonyosság 668
Blasgen, M. W. 358, 497
BLOB 144, 151
blokk 49, 151, 184, 397
blokkfejléc 123

borítékoló 632, 635
borítékoló generátor 636
Bosworth, A. 672
bozótszerű fa 425
Burkhard, W. A. 263

cache 47
Cattel, R. G. G. 152
Ceri, S. 264, 623
Chamberlin, D.D. 453
Chang, P.Y. 453
Chaudhuri, S. 358, 672
Chen, P. M. 109
Chou, H.-T. 358
cikkcakk-összekapcsolás 334
cilinder 58, 77, 107, 125
cilinderes szervezés 77
címkézett mezők 141
címterület 49
Codd, E. F. 358
Comer D. 215
COMMIT 467, 486
COMMIT naplóbejegyzés 465
csak olvasási tranzakció 538
csatolt blokk 573
csíkokra szedés 144
csillag séma 645
csomópont szétvágása 249, 250
csoportos mód 535, 542
csoportos véglegesítés 574
csoportosítás 268, 380, 408, 665
csoportosító attribútum 277

Date, C. J 45
dátumtípus 120
Dayal, U. 672
deadlock *lásd* holtpont
DeWitt, D. J. 358
dimenzió attribútum 645
dimenziótáblázat 645, 646, 647
dinamikus programozás 421, 429
disztributív szabály 370
döntési fa 663
döntéstámogató lekérdezés 642, 664
lásd még OLAP
dupla pufferezés 84

egész szám 115
egyenlő magasság hisztogram 411
egyenlő szélesség hisztogram 411
egyenlőség alapú összekapcsolás 275, 394, 401
lásd még equijoin
egyesítés 268, 269, 368, 372, 377, 407
egymenetes algoritmus 293
egymenetes összekapcsolás 426, 440
egy-olvasás-zár, minden-írás-zár 612
egyszerű vetítés 376
ekvivalens kifejezés 281
elágazás-és-korlát 420
elemző 35, 281, 367
elemzőfa 359, 360, 386
elkülönítés 25, 34
elkülönítési szint 572
ellenőrző összegek 91
ellenőrzőpont-képzés 454, 471, 480, 487, 492, 493
ellenőrzőpont képzés működés közben 473
előfeldolgozó 35, 366
előzetes algoritmus 667
előzetesen véglegesített tranzakció 602
első érkezés, első kiszolgálás 82
elsődleges kulcs 122, 155
elsődleges példány zárolása 610
eltolási érték 120
eltolásiérték-tábla 127
END CKPT naplóbejegyzés 474
END DUMP naplóbejegyzés 493
építő reláció 424, 426
equijoin 392, 401 *lásd még* egyenlőség alapú összekapcsolás
érdekesség 668
értékhalmazok megőrzése 402
értekhalmazok tartalmazása 402
értékszámláló 397
érvényesítés 550, 560, 564, 565
Eswaran, K. P. 358, 568
eszközhiba 89, 455, 490, 494, 575

fa *lásd* döntési fa, B-fa
Fagin, R. 215
fájl 24, 111
Faloutsos, C., 215, 264
fantom 542, 543
faprotokoll 545
Fayyad, U. M. 672
fej 53
fejgyűjtemény 56
felejtő tároló 54
felgörgetés 651
félig anti összekapcsolás 283
félig összekapcsolás 283
felminősítés kizárólagos zárrá 524, 537
felsorolási típus 119
feltétel 363
feltűzött blokk 134, 152
felülről lefelé tervválasztás 418
figyelmeztető protokoll 539

figyelmeztető zár 539
Filter 438, 446
Finkel, R. A. 263
fizikai cím 130, 152
fizikai lekérdezésterv 285, 359, 395, 418, 437, 449
fizikailag nem megvalósítható viselkedés 552
FLUSH LOG 467
forrás *lásd* adatforrás
Friedman, J. H. 263
from-lista 362
futószalagosítás 437, 441
független lemezek redundáns tömbje 94
függő attribútum 645
függőleges irányú dekompozíció 597

Gaede, V. 263
Garcia-Molina, H. 110, 623, 672
Gibson, G. A. 109
Glaser, T. 263
globális séma 628
Goodman, N. 497, 568
Gotlieb, L. R. 358
Graefe, G. 358, 453
gráf *lásd* poligráf, várakozási gráf
Gray, J. N. 109, 497, 568, 623, 672
Gunther, O. 263
Gupta, A. 358, 672
Guttman, A. 263
gyökér 184

Haderle, D. J. 497
Hadzilacos, V. 497, 568
Haerder, T. 497
hajlékonylemez 54
Hall, P. A. V. 453
halmaz 370, 377
Hamming-kód 101
Hamming-távolság 106
Harinarayan, V. 358, 672
harmadlagos tároló 52
hegymászás 420
Held, G. 45
helyességi elv 499, 501
helyreállítás 33, 454, 468, 471, 479, 482, 485, 494, 604
helyreállítás-kezelő 458, 468
helyreállíthatóság 25, 454
helyrehozó naplózás 466, 477, 478, 479, 480, 482
helyreigazítás 152
heterogén adatforrás 625
heurisztikus tervválasztás 419
hézag 108
hibajavítás 99
hivatkozás 123
hibrid tördeléses összekapcsolás 325
Hinterberger, H. 263
hisztogram 411
Holt, R. C. 623
holtpont 34, 464, 519, 525, 586, 611
hol-vagyok-én lekérdezés 218, 249
Hopcroft, J.E. 429
hosszú tranzakció 613, 642
Hsu, M. 497
HTML 180

ideiglenes meghibásodás 89
idempotencia 577
idempotens 284, 471
időtípus 118
időbélyegző 123, 550, 551, 558, 564, 565, 591
időkorlát *lásd* időtúllépés
időtúllépés 586
igény szerinti helyreigazítás 132
ismétlődő mező 137
Imielinski, T. 672
index 32, 35, 153, 415, 433
index alapú összekapcsolás 333
index alapú átvizsgálás 286
indexes összekapcsolás 420, 424, 441
indexfájl 32, 154
indexolvasás 439
IndexScan 446
információforrás *lásd* adatforrás
információintegráció 38
információk egyesítése 624 *lásd még* adatbázis-szövetség, adatközvetítő, adattárház
INGRES 45
inkrementális mentés *lásd* növekményes mentés
INPUT akció 460
input művelet 499
invertált index 178
IP-cím 136
írási halmaz 560
írási hiba 89
írási idő 551
írási művelet 499, 560
írási zár *lásd* kizárólagos zár 521
írj korábban naplózási szabály 478
ismétlődések elhagyása 379, 392, 408
ismétlődések kiküszöbölése 268, 276
ismétlődésérzéketlen csoportosítás 381
iterátor 290, 426

jellemzők 43
jobb-mély összekapcsolási fa 425

Kaiser, G. E. 623
Kannellakis, P. C. 568
kapcsolatok 43
katasztrofális hibák 456
Katz, R. H. 110, 358
kazetta 46
kd-fa 217, 238, 241, 243, 245
Kedem, Z. 568
keresés 189
keresési idő 60
keresési kulcs 154, 155, 161
késés 60
késleltetett tranzakció 532
kétargumentumú kiválasztás 386
kétmenetes algoritmus 356
kétfázisú véglegesítés 601
kétfázisú zárolás 517, 522
kétfázisú, többutas, összefésülő rendezés 70
ki-be jelentkezés 614
kiegyenlítő tranzakció 617
kiéheztetés 593
kifejezésfa 280
kimeneti attribútum 375
kiterjeszthető tördelés 204
Kitsuregawa, M. 358
kiválasztás 268, 270, 371, 379, 392, 398, 419, 437, 442, 446
kiválasztás tologatása 371, 374, 392
kizárólagos zár 521
kliens-szerver rendszerek 125
Knuth, D. E. 152, 215
Ko, H.-P. 568
KOCKA művelet 652
kockázás 647
kommunikációs költség 597
kommutatív szabály 368, 393
kommutatív törvény 98
kompatibilitási mátrix 523, 526, 528, 540
konfliktus 505, 507
konfliktus-sorbarendezhető ütemezés 507, 508
konfliktus-sorbarendezhetőség 499, 509
konkurencia 33, 37, 460, 498, 527
konkurenciavezérlés 498
konzisztencia 34, 456, 459, 514, 515, 522
koordinátor 602, 609
korai beolvasás 84
korrektség alapelve 459
korrelatív alkérdés 385
Korth, H. F. 568
kosár 201, 204, 208, 227, 230, 236
költség alapú felsorolás 395
költség alapú tervválasztás 410
kör 508, 547
központi memória 48
központi zárolás 607
közvetett kosár 175
közvetítő *lásd* adatközvetítő
Kreps, P. 45
Kriegel, H.-P. 263
Kumar, V. 497
Kung, H.-T. 568
kupac szerkezet 173
különbség 268, 269, 372, 377, 408
külső összekapcsolás 283

Lampson, B. 110, 623
lap 49
Larson, J. A. 672
Layman, A. 672
legközelebbi szomszéd-lekérdezés 218, 222, 224, 232, 234, 238, 241, 244
legrégebben használt 339
leggyakoribb értékek hisztogram 412
leképezési tábla 126
lekérdezés 44
lekérdezésátírás 266
lekérdezésfeldolgozás 30
lekérdezésfordítás 266
lekérdezés részleges egyezéssel 218, 232, 235, 240, 244
lekérdezésfeldolgozó 265
lekérdezésfordító 31, 359
lekérdezési terv *lásd* tervválasztás
lekérdezőterv 31
lemez 51, 54
lemezblokk 458
lemez I/O-művelet 51, 287, 415, 433
lemezblokk 32, 152, 288
lemezgyűjtemény 56
lemezütemező algoritmus 76
lemezvezérlő 57
levél 185
Lewis, P. M. II. 623
lexikografikus 117
Ley, Michael 45
lift algoritmus 81
Lindsay, B. G. 497
lineáris hasítás 207
Litwin W. 215
logikai cím 126, 152
logikai lekérdezésterv 266, 359, 360, 391, 416
logikai naplózás 576
Lomet, D. 152, 672
Lorie, R. A. 453, 568
Lozano, T. 264
LRU 339

mágneses szalag 46
maradandó tárolás 24
másodlagos index 172
másodlagos tároló 51
materializáció 437, 441
McCreight E. M. 215
McJones, P. R. 497
Megatron 2000 adatbázisrendszer 25
Megatron 737 78
Megatron 747 59
Megatron 777 65
megelőzési gráf 508, 510, 547
meghibásodás várható ideje 94
megismételhető olvasás 572
megosztás nélküli gép 349
megosztott lemez 348
megosztott memória 348
megsebez-megvár 591
megszorítás 122
megvalósított nézettábla 655
mélyre ásás 651
memóriacím 129
memóriaméret 266
memóriamutató 130
memóriahierarchia 47
mentés 456
  növekményes 491
  teljes 491
méretbecslés 396, 411
metaadat 25
metódus 43, 113
metszet 268, 269, 368, 372, 377, 408
mező 151
mintahalmaz 664
módosítás 146, 629
módosítási zár 526
módosítást leíró naplóbejegyzés 466, 478, 484
modulo-2 96, 102
Mohan, C. 497
mohó algoritmus 434
MOLAP 645. *lásd még* adatkocka
Moore törvénye 50
Moto-oka, T. 358
multihalmaz 268, 300, 370, 377
Mumick, I. S. 672
munkafolyamat 614
mutatók helyreigazítása 130
működés közbeni archiválás 493
működés közbeni ellenőrzőpont-képzés 487
művelet, tranzakció művelete 504

naplóbejegyzés 463, 465, 473
  ABORT 465
  COMMIT 465
  END CKPT 474
  END DUMP 493
  START 465
  START CKPT 473
  START DUMP 493
naplókezelő 457, 465
naplózás 33, 454, 494, 574 *lásd még* logikai naplózás
nem felejtő tároló 54
nem komplett tranzakció 469, 479
nézet-sorbarendezhetőség 580
nézetháló 658
nézettábla 42, 631 *lásd még* megvalósított nézettábla
Nievergelt, J. 215, 263
növekményes frissítés 629 *lásd még* módosítás
növekményes mentés 491
növelési művelet 527
növelési zár 528
NULL 140
NULL érték 626
nullérték 140
nullkarakter 116
nullmutató 140, 149
nyalábolt fájl 174, 175, 331
nyalábolt index 329, 331, 439
nyalábolt reláció 289, 331
nyelvtan 361

objektum 43, 111
objektum alapú adatbázis-kezelő 338
objektumazonosító 113
objektumbróker 125
objektumorientált adatbázis 43
ODL 113
OLAP 624, 641 *lásd még* MOLAP, ROLAP
Olken, F. 358
OLTP 642
olvasásbiztos 572
olvasási halmaz 560
olvasási idő 551
olvasási művelet 499, 560
olvasási zár *lásd* osztott zár 521
O'Neil, P. 263
on-line analitikus feldolgozás *lásd* OLAP
on-line másolat 456
on-line tranzakciófeldolgozás *lásd* OLTP
optikai lemez 46
optikai lemeztár 53
optimalizálás 36
OQL 113
osztály 43

osztály-előfordulás 43
osztott adatbázisok 595
osztott zár 521, 536
OUTPUT akció 461
output művelet 499
Ozsu, M. T. 623
összefésülő rendezés 68
összekapcsolás 28, 268, 379, 393, 433, 440
összekapcsolási fa 424
összekapcsolási sorrend 391, 423
összesítés 380, 408
összesítő operátor 277

Palermo, F. P. 453
Papadimitriou, C. H. 568, 623
Papakonstantinou, Y. 672
párhuzamos algoritmus 347
párhuzamos számolás 564
paritás 91
paritásbit 91
particionált tördelőfüggvény 217, 233, 235, 236
Patterson, D. A. 110
Pelagatti, G. 623
példányváltozó 43, 113
Peterson W. W. 215
Piatetsky-Shapiro, G. 672
Pirahesh, H. 497, 672
piszkos adat 553, 570
piszkos puffer 480
poligráf 582
paritásblokk 96
Price, T. G. 453
puffer 51, 295, 461, 570
pufferkezelő 32, 338, 428, 457
pufferterület 338
Putzolo, F. 109, 568

Quad-fa 217, 238, 246, 247
Quass, D. 263, 358, 672

Rácsos állomány 217, 227, 234
RAID 94, 455, 456
Rajaraman, A. 672
READ akció 460
recovery manager 458 *lásd* helyreállítás-kezelő
redo logging *lásd* helyrehozó naplózás
redundáns lemez 95
rege 616
rekord 31, 119, 151
rekordbeszúrás 229, 231, 243, 245, 260
rekordkeresés 222, 223, 225, 226, 227, 229, 243, 259, 260
rekordtörlés 260
rekordcím 125
rekordséma 122
rekordtöredék 143
reláció 38, 366
reláció mérete 397, 415
relációtöredék 597
relációs algebra 266, 367, 384
relációs OLAP *lásd* ROLAP
relációsor 42
rendezés 268, 421
rendezéses átvizsgálás 356
rendezéses összekapcsolás 317, 420, 441
rendezési kulcs 75, 155
rendezett részlista 71, 309
rendszerhibák 456
Reuter, A. 497, 568
R-fa 217, 248, 249
Robinson, J. T. 568
ROLAP 645
Rosenkrantz, D. J. 623
rotációs késés 61
Rothnie, J. B. Jr. 264, 568
Roussopoulos, N. 264
rekordfejlécek 121
rögzített hosszú karakterláncok 114
rögzített hosszú rekordok 119

sablon 635
Sagiv, Y. 672
Salem, K. 110, 623
Salton G. 215
sáv 56, 152
Schneider, R. 263
Schwarz, P. 497
Seeger, B. 263
select-from-where kifejezés 361
select-lista 362
Selinger, P.G. 453
Selinger-féle optimalizálás 421, 434
séma 25
semmisségi naplózás 463, 466, 468
  szabályai 466
semmisségi/helyrehozó naplózás 466, 484, 485, 487
  szabályai 484
Sethi, R. 361
Sevcik, K. 263
Shapiro, L. D. 358
Shaw, D. E. 358
Sheth, A. P. 672
Silberschatz, A. 568
sírkő 129
Skeen, D. 623
Smith, J.M. 453

Smyth, P. 672
Snodgrass, R. T. 264
sorba rendezhető ütemezés 501, 503
sorbarendezhetőség 498, 509, 572 *lásd még* nézet-sorbarendezhetőség
soros ütemezés 500
SortScan 446
SQL 361, 572
Srikant, R. 672
stabil tárolás 92
START CKPT naplóbejegyzés 473
START DUMP naplóbejegyzés 493
START naplóbejegyzés 465
statisztika 414
statisztikák 33
státusbit 90
Stearns, R. E. 623
Stonebraker, M. 45, 623
Strong H. R. 215
struct 42, 122
strukturált cím 127
Sturgis, H. 110, 623
Subrahmanian, V. S. 264
Suciu, D. 672
sűrű index 155, 187
Swami, A. 672
System R 45
szabadon lógó mutató 134
szakaszhosszkódolás 256, 258
szalagsilók 53
szektor 56
szekvenciális fájl 155
szelektivitás 435
szeletelés 647
szemcsésség (vagy granulátum) 539
szétvágási szabály 371
szigorú zárolás 573
szimultán kezelés 469
szintaktikus kategória 361
szorzat 268, 273, 368, 372, 376, 379, 392, 406
szótőképzés 180
szövetség *lásd* adatbázis-szövetség
szűrés 420
szűrő, borítékolóhoz 637

táblaátvizsgálás 356
táblaolvasás 438
TableScan 446
Tanaka, H. 358
tárház *lásd* adattárház
tárkezelő 32, 44
társítási szabály 666
tartó 667
tartománylekérdezés 218, 221, 222, 223, 232, 234, 240, 244, 259
tartományt eredményező lekérdezés 190
tartósság 32, 33
teljes mentés 491
ténytáblázat 644, 645, 651
térinformatikai rendszer 217, 218
természetes összekapcsolás 274, 368, 372, 376, 379, 393, 401
tervválasztás 599
théta-összekapcsolás 275, 276, 370, 372, 376, 379, 393, 401
Thomas, R. H. 623
Thomasian, A. 568
Thuraisingham, B. 568
típus 366
topologikus sorrend 508
továbbgyűrűző visszagörgetés 572
többdimenziós index 216, 224
többdimenziós OLAP *lásd* MOLAP
többkulcsos index 217, 238, 239, 240
többmenetes algoritmus 343
többségi zárolás 612
többszörözött adat 598, 609
többváltozatú időbélyegző 556
többverziós időbélyeg 574
töltelékszó 180
töltelékkarakter 114
tömörített bittérkép 256
tömörített lemez 53
tördeléses összekapcsolás 324, 420
tördelőfüggvény 200
tördelőkulcs 200
tördelőtábla 200, 297
töredék *lásd* relációtöredék
törlés 146, 182, 198, 202
Traiger, I. L. 568
tranzakció 456, 457, 458, 498, 504, 505, 597 *lásd még* hosszú tranzakció
  atomos 460
tranzakciófeldolgozó 37
tranzakciókezelő 33, 457
trigger 455, 457
tudásbányászat *lásd* adatbányászat
túlcsordulási blokk 147
tükrözés 79

Ullman, J. D. 45, 361, 429, 672
undo logging *lásd* semmisségi naplózás
undo/redo logging *lásd* semmisségi/helyrehozó naplózás
Uthurusamy, R. 672

ütemezés 499, 500, 504, 505
ütemezések jogszerűsége 514, 522
ütemező 498, 514, 516, 532, 534, 536, 551, 554, 557, 560, 561

Valduriez, P. 623
valós szám 120
változó formátumú rekord 138
változó hosszú karakterláncok 115
változó hosszú mező 577
változó méretű adattétel 137
várakozási bit 536
várakozási gráf 587
Vassalos, V. 672
véglegesítés 575, 600 *lásd még* csoportos véglegesítés, kétfázisú véglegesítés
véglegesítési bit 551, 553, 571
végrehajtás 35
végrehajtómotor 30
véletlen hozzáférés 48
vetítés 268, 272, 375, 380, 397, 406, 442
vetítés tologatása 375, 377, 392
virtuális memória 125
virtuális memória címterület 49
visszagörgetés 554 *lásd* abort, továbbgyűrűző visszagörgetés
Vitter, J. S. 110
vizsgáló reláció 424, 427
vízszintes irányú dekompozíció 597

Widom, J. 45, 672
Wiederhold, G. 152, 672
Wong, E. 45, 453
Wood, D. 358
`WRITE` akció 460

Y2K 117
Youssefi, K. 453

Zaniolo, C. 264
zárolás 516, 607. *lásd még* szigorú zárolás
zárolás feloldása 515
zárolás kérése 515
zártábla 514, 516, 534
zárolások 558, 564, 565
Zicari, R. 264
Zipfian-eloszlás 183, 400