6.12. Irodalomjegyzék

A [7] és a [2] a lekérdezésoptimalizálás egy-egy áttekintése. Az összekapcsolás módszereinek egy korai tanulmányát a [6]-ban találhatjuk. A pufferkezelés áttekintését, elemzését és a javítási lehetőségeket a [3] tartalmazza.

A relációs algebra a [4]-ben szerepel először, amely Coddnak a relációs modellről szóló cikke. A csoportosító operátor kiterjesztését és Codd vetítésének általánosítását a [8]-ból vettük.

A rendezés alapú technikákat az [1] vezette be. A tördelés alapú algoritmusok összekapcsolásban történő alkalmazásának előnyét a [9] és [5] fejtette ki. Ez utóbbiból ered a hibrid tördeléses összekapcsolás. A tördelés használatát párhuzamos összekapcsolásban és egyéb műveletekben többször is javasolták. Az általunk ismert legelső forrás a [10].

1. M. W. Blasgen and K. P. Eswaran, „Storage access in relational databases," *IBM Systems J.* **16**:4 (1977), pp. 363–378.
2. S. Chaudhuri, „An overview of query optimization in relational systems," *Proc. Seventeenth Annual ACM Symposium on Principles of Database Systems*, pp. 34–43, June, 1998.
3. H.-T. Chou and D. J. DeWitt, „An evaluation of buffer managament strategies for relational database systems," *Proc. Intl. Conf. On Very Large Databases* (1985), pp. 127–141.
4. E. F. Codd, „A relational model for shared data banks," *Comm. ACM* **13**:6 (1970), pp. 377–387.
5. D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. Wood, „Implementation techniques for main-memory database systems," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1984), pp. 1–8.
6. L. R. Gotlieb, „Computing joins of relations," *Proc. ACM SIGMOD Intl. Conf. in Management of Data* (1975), pp. 55–63.
7. G. Graefe, „Query evaluation techniques for large databases," *Computing Surveys* **25**:2 (June, 1993), pp. 73–170.
8. A. Gupta, V. Harinarayan, and D. Quass, „Aggregate-query proccessing in data warehousing environments," *Proc. Intl. Conf. on Very Large Databases* (1995), pp. 358-369.
9. M. Kitsuregawa, H. Tanaka, and T. Moto-oka, „Application of hash to data base machine and its architecture," *New Generation Computing* **1**:1 (1983), pp. 66–74.
10. D. E. Shaw, „Knowledge-based retrieval on a relational database machine," Ph. D. thesis, Dept. of CS, Stanford Univ. (1980)

7. fejezet

A lekérdezésfordító

Miután a 6. fejezetben láttuk a fizikai lekérdezésterv operátorainak végrehajtásához használt alapvető algoritmusokat, most a lekérdezésfordító és az ahhoz tartozó optimalizáló felépítését vesszük sorra. Amint a 6.2. ábránál megjegyeztük, a lekérdezés-feldolgozó feladata három fő lépésből áll:

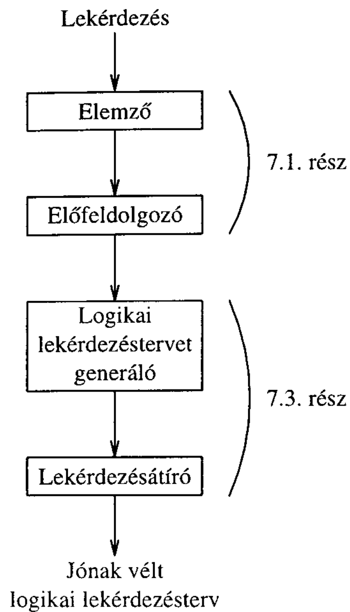
1. Az SQL-szerű nyelven megfogalmazott lekérdezés *elemzése,* azaz átalakítása egy elemzőfává, amely a lekérdezés szerkezetének egy jól használható reprezentációját adja.
2. Az elemzőfa átalakítása egy, a relációs algebrával (vagy más hasonló jelölésrendszerrel) megfogalmazott kifejezésfává, amit *logikai lekérdezéstervnek* nevezünk.
3. A logikai lekérdezéstervet egy olyan *fizikai lekérdezéstervvé* kell alakítani, amely már nem csak a végrehajtásra kerülő műveleteket mutatja, hanem ezek végrehajtási sorrendjét, az egyes lépések végrehajtásához használt algoritmusokat is, továbbá azt is, hogy a tárolt adatokat hogyan kapjuk meg, és hogy az adatokat hogyan adják át egymásnak a műveletek.

Az első lépés, az elemzés, a 7.1. rész tárgya. Ennek a lépésnek az eredménye a lekérdezés egy elemzőfája. A másik két lépés számos választást tartalmaz. Ha kezünkben van egy logikai lekérdezésterv, lehetőségünkben áll különféle algebrai műveletek alkalmazása, azzal a céllal, hogy a legjobb logikai lekérdezéstervet állítsuk elő. A 7.2. rész a relációs algebrához tartozó algebrai szabályokat tárgyalja. A 7.3. rész foglalkozik az elemzőfák kiindulási logikai lekérdezéstervvé történő átalakításával, valamint azzal, hogy a 7.2. részben bemutatott algebrai szabályokat hogyan lehet a kiindulási terv javítására használni.

Amikor egy logikai tervből egy fizikai lekérdezéstervet állítunk elő, fel kell mérnünk az egyes választási lehetőségek várható költségét. A költségbecslés maga is egy önálló tudomány, amit a 7.4. részben mutatunk be. A 7.5. részben azt mutatjuk meg, hogy a költségbecslés hogyan használható a tervek kiértékelésére. A 7.6. részben azokat a speciális problémákat tárgyaljuk, amelyek a sorrend meghatározásával kapcsolatban merülnek fel sok reláció összekapcsolásakor. Végül a 7.7. részben kitérünk a fizikai lekérdezésterv kiválasztásával kapcsolatos további témákra és stratégiákra: az algoritmus megválasztására, valamint a futószalag-technika és a materializáció kérdésére.

7.1. Elemzés

A lekérdezések fordításának első fázisait a 7.1. ábra illusztrálja. Az ábrán látható négy doboz a 6.2. ábra első két lépésének felel meg. Elkülönítettünk egy „előfeldolgozás" lépést az elemzés és a kiindulási logikai tervvé történő átalakítás között, amit a 7.1.3. részben fogunk tárgyalni.



7.1. ábra. *Lekérdezés átalakítása logikai lekérdezéstervvé*

Ebben a részben az SQL elemzését tárgyaljuk, és megadunk egy leegyszerűsített nyelvtant, amely ehhez a nyelvhez használható. A 7.2. részben letérünk a lekérdezésfordítás vonaláról, és alaposan megvizsgáljuk a relációs algebrai kifejezésekre vonatkozó különféle szabályokat. A 7.3. részben visszatérünk a lekérdezésfordításhoz. Először azt nézzük meg, hogyan alakítható át egy elemzőfa egy relációs algebrai kifejezéssé, amely kiindulási logikai lekérdezéstervként szolgál majd. Ezután olyan módszereket tárgyalunk, amelyekben a 7.2. részben ismertetett transzformációk használhatók úgy, hogy jobb lekérdezéstervet kapunk ahelyett, hogy a tervet egyszerűen csak ekvivalens tervvé alakítanánk, aminek hasznossága kétséges.

7.1.1. Szintaktikus elemzés és elemzőfák

Az elemző feladata az, hogy egy SQL-ben vagy ahhoz hasonló nyelvben megírt szöveget egy elemzőfává konvertáljon. Az *elemzőfa* (parse tree) csomópontjai az alábbiak lehetnek:

1. *Atomok*, melyek lexikai elemek, mint például kulcsszavak (pl. `SELECT`), attribútumok vagy relációk nevei, konstansok, zárójelek, operátorok (pl. + vagy <) és egyéb sémaelemek, vagy
2. *Szintaktikus kategóriák*. Ezek nevek, amelyek a lekérdezések olyan részegységeit képviselik, amelyek hasonló szerepet töltenek be a lekérdezésekben. A szintaktikus kategóriákat kisebb-nagyobb jelek közé tett informatív nevekkel jelöljük. Az <SFW> például a megszokott „select-from-where" alakú lekérdezéseket reprezentálja, míg a <Feltétel> olyan tetszőleges kifejezést jelöl, ami egy feltétel, vagyis az SQL-ben a `WHERE` után állhat.

Ha egy csomópont atom, akkor annak nincsenek gyerekei. Ha azonban a csomópont egy szintaktikus kategória, akkor annak gyerekeit a nyelvet megadó nyelvtan valamely *szabálya* írja le. Ezeket az elveket példákon keresztül mutatjuk be. Annak részletei, hogy egy nyelvet definiáló nyelvtant hogy lehet megtervezni, illetve hogy az „elemzés" – azaz egy program vagy lekérdezés elemzőfává történő átalakítása – hogyan történik, valójában egy olyan kurzus témája, amely a fordításról szól.[1]

7.1.2. Egy leegyszerűsített SQL-részletet leíró nyelvtan

Az elemzési folyamat bemutatásához megadunk néhány szabályt, amelyekkel az SQL egy részhalmazát adó lekérdezőnyelvet definiálunk. Kitérünk majd arra is, hogy milyen további szabályok kellenének ahhoz, hogy az SQL-t teljesen leíró nyelvtant alkossunk.

```
<Lekérdezés> ::= <SFW>
<Lekérdezés> ::= ( <Lekérdezés> )
```

Vegyük észre, hogy a `::=` szimbólum a szokásos módon azt jelenti: „úgy fejezhető ki, hogy". Az első szabály azt mondja, hogy egy lekérdezés lehet egy „select-from-where" kifejezés; az <SFW>-t leíró szabályokat látni fogjuk a későbbiekben. A második szabály azt mondja, hogy az is egy lekérdezést ad, ha egy lekérdezést zárójelek közé teszünk. Egy teljes SQL-nyelvtanban olyan szabályokra is szükség lehet, amelyek lehetővé teszik, hogy egy lekérdezés egyetlen reláció legyen, vagy egy olyan kifejezés, amely relációkat és különböző típusú műveleteket – mint például egyesítést (`UNION`) és összekapcsolást (`JOIN`) – tartalmaz.

[1] Akik számára ez a téma ismeretlen, a következő irodalmat ajánljuk tanulmányozásra: A.V. Aho, R. Sethi és J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading MA, 1986, jóllehet a 7.1.2. részben adott példák elégségesek kell hogy legyenek arra, hogy az elemzést a lekérdezésfeldolgozó környezetébe elhelyezzük.

„Select-From-Where" kifejezések

Az <SFW> szintaktikus kategóriát egyetlen szabállyal definiáljuk:

```
<SFW> ::= SELECT <SelLista> FROM <FromLista> WHERE <Feltétel>
```

Ez a szabály a megfelelő SQL-lekérdezés egy korlátozott formáját engedi meg. Nem gondoskodik sem a különböző opcionális záradékokról, mint amilyen a GROUP BY, HAVING vagy ORDER BY, sem egyéb opciókról, mint például a SELECT utáni DISTINCT. Ne felejtsük el, hogy egy igazi SQL-nyelvtan sokkal bonyolultabb struktúrákat tartalmaz a lekérdezések leírására, beleértve a „select-from-where" alakok fenti variációit, operátorok (pl. UNION, NATURAL JOIN) segítségével felépített lekérdezéseket és sok egyebet.

Vegyük észre az alkalmazott konvenciónkat, miszerint a kulcsszavak nagybetűsen szerepelnek. A <SelLista> és <FromLista> szintaktikus kategóriák listákat reprezentálnak, amelyek a SELECT, illetve FROM után állhatnak. A <Feltétel> szintaktikus kategória SQL-feltételeket (igaz vagy hamis értéket adó kifejezéseket) jelöl; ehhez a kategóriához adunk később néhány egyszerűsített szabályt.

Select-listák

```
<SelLista> ::= <Attribútum> , <SelLista>
<SelLista> ::= <Attribútum>
```

Ezek a szabályok azt fogalmazzák meg, hogy egy select-lista attribútumoknak vesszővel elválasztott tetszőleges listája lehet: vagy egyetlen attribútum, vagy egy attribútum, egy vessző és egy attribútumokból álló lista. Megjegyezzük, hogy egy teljes SQL-nyelvtanban biztosítani célszerű kifejezések és összesítő függvények használatát a select-listában, és az attribútumok és kifejezések átnevezési lehetőségét.

From-listák

```
<FromLista> ::= <Reláció> , <FromLista>
<FromLista> ::= <Reláció>
```

Ezek alapján egy from-lista relációknak vesszővel elválasztott tetszőleges listájaként definiált. Az egyszerűség kedvéért elhagyjuk azt a lehetőséget, hogy egy from-lista elemei tartalmazhatnak olyan kifejezéseket, mint például *R* JOIN *S* vagy „select-from-where" kifejezéseket. Ezen túlmenően, egy teljes SQL-nyelvtannak a from-listában szereplő relációk átnevezési lehetőségét is biztosítani kell; mi most nem engedjük meg, hogy egy relációhoz egy sorváltozót rendeljünk a reláció reprezentálása céljából.

Feltételek

Az alábbi szabályokat használjuk:

```
<Feltétel> ::= <Feltétel> AND <Feltétel>
<Feltétel> ::= <Sor> IN <Lekérdezés>
<Feltétel> ::= <Attribútum> = <Attribútum>
<Feltétel> ::= <Attribútum> LIKE <Minta>
```

Habár a feltételekhez több szabályt soroltunk fel, mint a többi kategóriához, ezek a szabályok a feltételek lehetséges alakjainak csak felszínét érintik. Elhagytuk az OR, NOT és EXISTS operátorok bevezetését jelentő szabályokat, az egyenlőség és LIKE vizsgálatán túlmutató összehasonlításokat, a konstans operandusokat és számos egyéb struktúrát, amelyek egy teljes SQL-nyelvtanban nélkülözhetetlenek. Ráadásul, annak ellenére, hogy egy sor többféle formában megjelenhet, a <Sor> szintaktikus kategóriához csak egy szabályt vezetünk be, amely azt mondja, hogy egy sor egyetlen attribútumból állhat:

```
<Sor> ::= <Attribútum>
```

Alap szintaktikus kategóriák

Az <Attribútum>, <Reláció> és <Minta> speciális szintaktikus kategóriák, amennyiben azokat nem nyelvtani szabályok definiálják, hanem az olyan atomokra vonatkozó szabályok, amelyek helyén azok szerepelnek. Egy elemzőfában például az <Attribútum> egyetlen gyereke egy olyan karakterlánc lehet, amely egy attribútum neveként értelmezhető abban az adatbázissémában, amelyre a lekérdezés vonatkozik. Hasonlóképpen, a <Reláció> egy olyan karakterlánccal helyettesíthető, amely értelmes relációnevet jelent az adott sémában, és a <Minta> egy olyan (egyszeres) idézőjelek között szereplő karaktersorozat, ami egy érvényes SQL-minta.

7.1. példa: Az elemzési és a lekérdezés átírási fázisok tanulmányozásához a szokásos „filmes" példa relációit, illetve egy azokra vonatkozó lekérdezés két változatát fogjuk használni:

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

A lekérdezés mindkét változata azoknak a filmeknek a címét kéri, amelyekben legalább egy 1960-ban született színész szerepel. A színészek 1960-as születését úgy állapítjuk meg, hogy a LIKE operátor segítségével megvizsgáljuk, hogy a születési idő (egy SQL-karakterlánc) '1960'-ra végződik-e.

A lekérdezés megfogalmazásának egyik módja, hogy egy alkérdéssel felépítjük azon színészek neveinek halmazát, akik 1960-ban születtek, majd minden egyes

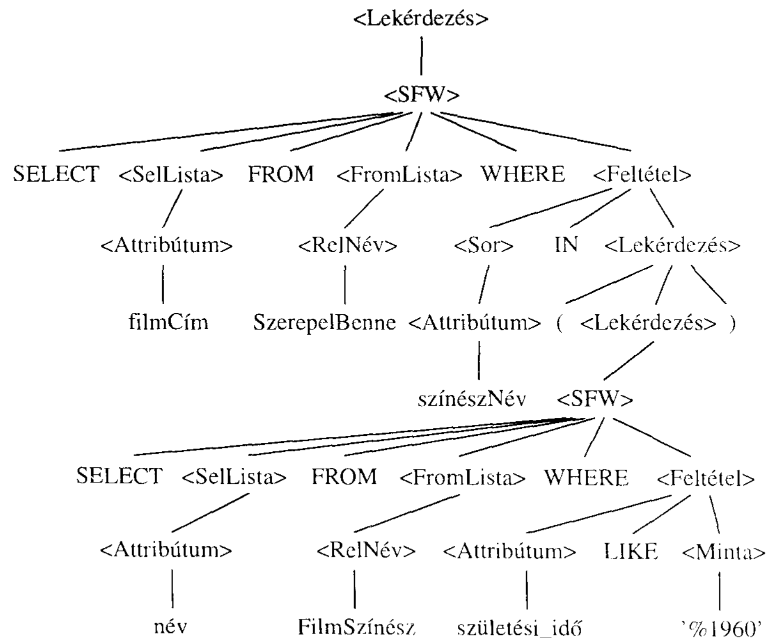
```
SELECT filmCím
FROM SzerepelBenne
WHERE színészNév IN (
   SELECT név
   FROM FilmSzínész
   WHERE születési_idő LIKE '%1960'
);
```

7.2. ábra. *Keressük meg azokat a filmeket, amelyekben játszik 1960-as születésű színész*

`SzerepelBenne` sorról megkérdezzük, hogy a `színészNév` sora-e az alkérdés által visszaadott halmaznak. A 7.2. ábrán látható ennek a változatnak az SQL-megfelelője.

A 7.3. ábra mutatja a 7.2. ábrán szereplő lekérdezéshez tartozó elemzőfát, az általunk felvázolt nyelvtannak megfelelően. A gyökérben a <Lekérdezés> szintaktikus kategória áll, aminek minden elemzőfa esetében így kell lennie. Lefelé haladva a fában azt látjuk, hogy ez egy „select-from-where" alakú lekérdezés, amelynek select-listája csak a `filmCím` attribútumból áll, és a from-lista csak a `SzerepelBenne` relációt tartalmazza.

A külső `WHERE` záradék feltétele már összetettebb. Sor-`IN`-lekérdezés alakú, és maga a lekérdezés egy zárójelezett alkérdés, mivel az SQL-ben az alkérdéseket zárójelek közé kell tenni. Maga az alkérdés egy további „select-from-where" kifejezés, a saját egyelemű select- és from-listájával és egy egyszerű feltétellel, amely a `LIKE` operátort használja. □



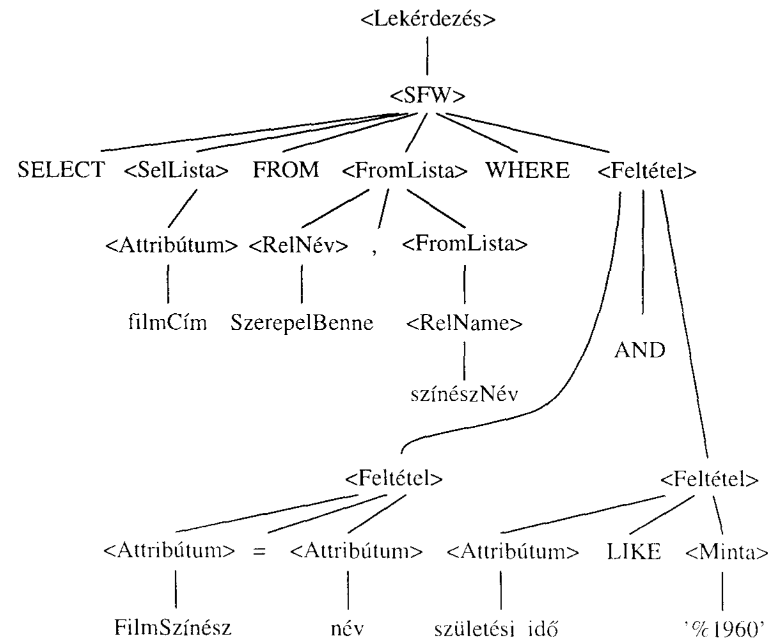
7.3. ábra. *A 7.2. ábrához tartozó elemzőfa*


```
SELECT filmCím
FROM SzerepelBenne, FilmSzínész
WHERE színészNév = név AND
   születési_idő LIKE '%1960';
```

7.4. ábra. *Egy másik módja azon filmek megkeresésének, amelyekben játszik 1960-as születésű színész*

7.2. példa: Vegyük most a 7.2. ábrán bemutatott lekérdezés egy másik változatát, amikor is nem használunk alkérdést. Helyette összekapcsolhatjuk a `SzerepelBenne` és `FilmSzínész` relációkat a `színészNév = név` feltételt használva, amivel azt biztosítjuk, hogy a két reláció azon sorai kapcsolódnak össze, ahol a színész ugyanaz. Vegyük észre, hogy a `színészNév` a `SzerepelBenne` reláció egy attribútuma, míg a `név` a `FilmSzínész` reláció egy attribútuma. A 7.2. ábra lekérdezésének ez a változata a 7.4. ábrán látható.[2]

A 7.4. ábrához tartozó elemzőfát a 7.5. ábrán látjuk. Az ebben az elemzőfában



7.5. ábra. *A 7.4. ábrához tartozó elemzőfa*

[2] A két lekérdezés között abban van egy kis különbség, hogy a 7.4. ábrán szereplő ismétlődéseket állít elő akkor, ha valamely filmnek több olyan szereplője is van, akik 1960-ban születtek. Ha szigorúak akarnánk lenni, akkor a 7.4. ábra lekérdezésénél a `DISTINCT` kulcsszót is meg kellene adni, de a példa nyelvtanunkat oly mértékben leegyszerűsítettük, hogy kihagytuk ezt az opciót.

használt szabályok közül sok ugyanaz, mint a 7.3. ábránál használtak. Figyeljük meg azonban, hogy a több relációt tartalmazó from-listát hogyan fejezi ki az elemzőfa, valamint azt is megfigyelhetjük ebben az esetben, hogy egy feltétel hogyan áll össze több kisebb feltételből az AND operátor segítségével. □

7.1.3. Az előfeldolgozó

A 7.1. ábrán *előfeldolgozónak* (preprocessor) nevezettnek több fontos funkciója van. Ha a lekérdezésben használt valamely reláció egy nézettábla, akkor a relációt a nézettáblának megfelelő elemzőfával kell helyettesíteni, valahányszor a from-listában szerepel. Ezt az elemzőfát a nézettábla definíciója alapján kapjuk meg, ami valójában egy lekérdezés.

Az előfeldolgozó *szemantikus ellenőrzések* (semantic checking) elvégzéséért is felelős. Még ha érvényes is a lekérdezés szintaktikai szempontból, esetleg megsérthet bizonyos szabályokat a nevek használatára vonatkozóan. Az előfeldolgozónak például az alábbiakat kell elvégezni:

1. *Relációk használatának ellenőrzése.* A FROM záradékban szerepeltetett relációk mindegyike annak a sémának egy relációja vagy nézettáblája kell, hogy legyen, amelyre a lekérdezés vonatkozik. Példának okáért, a 7.3. ábrán található elemzőfa esetében az előfeldolgozó ellenőrizni fogja, hogy a két from-listában megadott SzerepelBenne és FilmSzínész relációk valóban a séma relációi-e.
2. *Attribútumnevek ellenőrzése és feloldása.* A SELECT vagy WHERE záradékokban előforduló attribútumok mindegyike az aktuális érvényességi kör valamely relációjának egy attribútuma kell hogy legyen; ha ez nem teljesül, akkor az elemzőnek hibát kell jeleznie. A 7.3. ábra első select-listájában például a filmCím attribútum csak a SzerepelBenne reláció hatáskörébe esik. Szerencsére a SzerepelBenne relációnak attribútuma a filmCím, így az előfeldolgozó itt jóváhagyja a filmCím használatát. Ezen a ponton a tipikus lekérdezésfeldolgozó minden egyes attribútum feloldását elvégezné, összekapcsolva a megfelelő relációval, feltéve, hogy ez nem történt meg explicit módon a lekérdezésben (pl. SzerepelBenne.filmCím). Ellenőrizné továbbá az egyértelműséget is, és hibát jelezne, ha a vizsgált attribútum több relációban szerepelne attribútumként az érvényességi körön belül.
3. *Típusellenőrzés.* Minden attribútum csak a használatának megfelelő típusú lehet. A 7.3. ábrán például a születési_idő-t egy LIKE összehasonlításban használjuk, ami megköveteli, hogy a születési_idő típusa karakterlánc legyen, vagy egy olyan típus, amely karakterlánccá konvertálható. Mivel a születési_idő egy dátum, és az SQL-ben a dátumokat kezelhetjük karakterláncokként, az attribútumnak ez a használata elfogadható. Ehhez hasonlóan azt is ellenőrizni kell, hogy operátorokat csak megfelelő típusú értékekre alkalmazunk.

Sikeresen túljutva ezeken az ellenőrzéseken, az elemzőfáról azt mondjuk, hogy *érvényes,* és miután megtörtént az esetleges nézettáblák kifejtése és az attribútumnevek használatának feloldása, a fát továbbadjuk a logikai lekérdezéstervet generálónak. Ha az elemzőfa nem érvényes, akkor megfelelő diagnosztika készül, és a feldolgozás leáll.

7.1.4. Feladatok

7.1.1. feladat: Bővítsük vagy módosítsuk az <SFW>-t definiáló szabályokat úgy, hogy az SQL „select-from-where" kifejezéseinek alábbi egyszerű lehetőségeit magukban foglalják:

* a) A DISTINCT kulcsszó használata, hogy egy halmazt tudjunk előállítani.
b) GROUP BY záradék és HAVING záradék megadása.
c) Az eredmény rendezése az ORDER BY záradék használatával.
d) Where záradék nélküli lekérdezések.

7.1.2. feladat: Adjunk a <Feltétel> szabályaihoz további szabályokat, amelyek az SQL-feltételek következő jellemzőit is biztosítják:

* a) Az OR és a NOT logikai operátorok.
b) Az egyenlőség vizsgálatán túlmutató további összehasonlítások.
c) Zárójelezett feltételek.
d) EXISTS kifejezések.

7.1.3. feladat: Legyenek $R(a, b)$ és $S(b, c)$ relációk. Készítsük el a következő lekérdezésekhez tartozó elemzőfákat, felhasználva az ebben a részben bemutatott egyszerű SQL-nyelvtant:

```
a) SELECT a, c
   FROM R, S
   WHERE R.b = S.b;
b) SELECT a FROM R WHERE b IN (
      SELECT a FROM R, S WHERE R.b = S.b
   );
```

7.2. Algebrai szabályok lekérdezéstervek javítására

A lekérdezésfordító tárgyalását a 7.3. részben fogjuk folytatni, ahol az elemzőfát először egy kifejezéssé transzformáljuk, amely teljesen vagy többnyire a 6.1. részben bevezetett kiterjesztett relációs algebra operátoraiból áll. Ugyanabban a részben azt is megnézzük, hogy a relációs algebrára érvényes algebrai szabályok felhasználásával

milyen heurisztikákat alkalmazhatunk, a lekérdezés algebrai kifejezésének javulását remélve. Ezek előkészítéseként, ebben a részben összegyűjtjük azokat az algebrai szabályokat, amelyek egy kifejezésfát olyan ekvivalens kifejezésfává alakítanak, amelyhez esetleg hatékonyabb fizikai lekérdezésterv tarozik.

Az ilyen algebrai transzformációk alkalmazásának eredménye a logikai lekérdezésterv, ami egyben a lekérdezés átírási fázis kimenete. Ezután történik a logikai lekérdezésterv átfordítása fizikai lekérdezéstervvé, amikor is az optimalizáló számos döntést hoz az operátorok megvalósításával kapcsolatban. A fizikai lekérdezésterv generálását a 7.4. résztől kezdődően tárgyaljuk. Egy másik – a gyakorlatban nem nagyon használt – lehetőség az, hogy több jó logikai tervet generálunk a lekérdezés átírási fázisban, és az ezekből generált fizikai terveket vizsgáljuk meg, és végül a legjobb fizikai tervet kiválasztjuk.

7.2.1. Kommutatív és asszociatív szabályok

A különféle kifejezések egyszerűsítésére használt legáltalánosabb szabályok a kommutatív és asszociatív szabályok. Egy operátorra vonatkozó *kommutatív szabály* azt mondja ki, hogy nem számít, hogy milyen sorrendben adjuk meg az operátor argumentumait, az eredmény ugyanaz lesz. A + és × például az aritmetika kommutatív operátorai. Pontosabban, $x + y = y + x$ és $x \times y = y \times x$ tetszőleges x és y számok esetén. Másrészről azonban a – nem egy kommutatív aritmetikai operátor: $x - y \neq y - x$.

Egy operátorra vonatkozó *asszociatív szabály* azt mondja ki, hogyha az operátort kétszer használjuk, akkor egyaránt csoportosíthatunk balról vagy jobbról. A + és × például asszociatív aritmetikai operátorok, ami azt jelenti, hogy $(x + y) + z = x + (y + z)$ és $(x \times y) \times z = x \times (y \times z)$. Ugyanakkor a – nem asszociatív: $(x - y) - z \neq x - (y - z)$. Amikor egy operátor egyszerre kommutatív és asszociatív is, akkor ha bármennyi operandust kötünk is össze az operátorral, az operandusokat tetszés szerint csoportosíthatjuk és rendezhetjük anélkül, hogy az eredmény megváltozna. Például: $((w + x) + y) + z = (y + x) + (z + w)$.

A relációs algebra néhány operátora egyszerre kommutatív és asszociatív:

- $R \times S = S \times R$; $(R \times S) \times T = R \times (S \times T)$,
- $R \bowtie S = S \bowtie R$; $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$,
- $R \cup S = S \cup R$; $(R \cup S) \cup T = R \cup (S \cup T)$,
- $R \cap S = S \cap R$; $(R \cap S) \cap T = R \cap (S \cap T)$.

Megjegyezzük, hogy az egyesítésre és a metszetre vonatkozó szabályok egyaránt érvényesek halmazokra és multihalmazokra.

Nem fogjuk mindegyik szabályt bizonyítani, de adunk egy példát a bizonyításra. Egy relációkkal kapcsolatos szabály igazolásának általános menete az, hogy be kell látni, hogy a bal oldali kifejezés által előállított minden sort a jobb oldali kifejezés is előállítja, valamint hogy a jobb oldali kifejezés által előállított minden sort a bal oldali kifejezés is előállítja.

7.3. példa: Bizonyítsuk be a $\bowtie$-ra vonatkozó kommutatív szabályt: $R \bowtie S = S \bowtie R$.

Először is tegyük fel, hogy egy t sor benne van az $R \bowtie S$, azaz a bal oldali kifejezés eredményében. Ekkor léteznie kell egy r sornak az R-ben és egy s sornak az S-ben, amelyek a t-vel megegyeznek a t-vel közös összes attribútum vonatkozásában. Így amikor kiértékeljük a jobb oldali $R \bowtie S$ kifejezést, az s és r sorok ismét összekapcsolódnak, létrehozva a t sort.

Azt gondolhatjuk, hogy a t komponenseinek sorrendje különbözni fog a bal és jobb oldal esetén, formálisan azonban a relációs algebrában a sorok attribútumainak nincs rögzített sorrendje. Sőt a komponenseket szabadon átrendezhetjük mindaddig, amíg az oszlopfejlécekben az attribútumokat is megfelelőképpen átvezetjük. Például a

a	*b*	*c*
0	1	2

sor ugyanaz a sor, mint a

b	*c*	*a*
1	2	0

vagy mint az oszlopok további négy permutációjával kapott sorok.

Még nem fejeztük be a bizonyítást. Mivel a mi relációs algebránk egy multihalmazokkal, és nem halmazokkal operáló algebra, azt is be kell látni, hogy ha t a bal oldalon n-szer jelenik meg, akkor a jobb oldalon legalább n-szer megjelenik, és fordítva, ha a jobb oldalon n-szer jelenik meg, akkor a bal oldalon legalább n-szer megjelenik. Tegyük fel, hogy t a bal oldalon n-szer jelenik meg. Ekkor az R-ből származó, t-vel egyező r sor n_R-szer jelenik meg, és az S-ből származó, t-vel egyező s sor n_S-szer jelenik meg, ahol $n_R n_S = n$. Így, amikor kiértékeljük a jobb oldali $R \bowtie S$ kifejezést, az s sor n_S-szer fog megjelenni, az r pedig n_R-szer, tehát t-nek $n_R n_S$, azaz n darab példányát fogjuk megkapni.

Még mindig nem vagyunk kész. A bizonyításnak azt a felét fejeztük be, amelyik azt mondja ki, hogy minden, amit megkapunk a bal oldalon, megjelenik a jobb oldalon is, de azt is meg kell mutatnunk, hogy minden, amit megkapunk a jobb oldalon, megjelenik a bal oldalon is. A nyilvánvaló szimmetria miatt a fentivel megegyező gondolatmenet követhető, így ennek részleteibe most nem is megyünk bele. □

7.4. példa: Az asszociatív szabályok bizonyítása valamivel bonyolultabb. Példaként vegyük a $\bowtie$-ra vonatkozó asszociatív szabályt:

$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

Ennek a szabálynak az igazolásához azt bizonyítjuk, hogy a bal oldalon megjelenő sorok pontosan azok a sorok, amelyeket azokból az R-beli r, S-beli s és T-beli t sorokból kapunk, amelyek kölcsönösen megegyeznek az összes közös attribútumon. Majd

azt látjuk be, hogy pont ezek a sorok a jobb oldalon is előállnak, és éppen annyiszor, mint a bal oldalon. □

Az asszociatív-kommutatív operátorok közé nem vettük fel a théta-összekapcsolást. Ez az operátor kommutatív:

- $R \underset{C}{\bowtie} S = S \underset{C}{\bowtie} R$

Ezen túlmenően, ha az érintett feltételeknek van értelme ott, ahová kerülnek, akkor a théta-összekapcsolás asszociatív. Vannak azonban példák, mint amilyen a következő is, amikor az asszociatív szabály nem alkalmazható, mert a feltételek nem az éppen összekapcsolni kívánt relációk attribútumaira vonatkoznak.

7.5. példa: Tegyük fel, hogy van három relációnk: $R(a, b)$, $S(b, c)$ és $T(c, d)$. Az

$$(R \underset{R.b>S.b}{\bowtie} S) \underset{a<d}{\bowtie} T$$

kifejezést egy feltételezett asszociatív szabállyal a következővé alakíthatnánk:

$$R \underset{R.b>S.b}{\bowtie} (S \underset{a<d}{\bowtie} T)$$

Az S-et és a T-t azonban nem kapcsolhatjuk össze az $a < b$ feltétel alapján, hiszen az a sem az S-nek, sem a T-nek nem attribútuma. A théta-összekapcsolásra vonatkozó asszociatív szabály tehát nem alkalmazható tetszőlegesen. □

A multihalmazokra és a halmazokra vonatkozó szabályok különbözhetnek

Óvatosnak kell lennünk, amikor a halmazokkal kapcsolatos ismerős szabályokat multihalmazokra próbáljuk alkalmazni. A halmazelméletből ismerjük például a következő szabályt: $A \cap_H (B \cup_H C) = (A \cap_H B) \cup_H (A \cap_H C)$, ami a metszet halmaz feletti disztributív szabálya. Ez igaz a halmazokra, de nem igaz a multihalmazokra.

Tegyük fel például, hogy A, B és C mindegyike a $\{x\}$ multihalmaz. Ekkor $B \cup_M C = \{x, x\}$ és $A \cap_M (B \cup_M C) = \{x\}$, hiszen a multihalmazok metszete az előfordulások számának minimumát veszi. Ugyanakkor az $A \cap B$ és $A \cap C$ mindegyike $\{x\}$, vagyis a jobb oldali kifejezés: $(A \cap_M B) \cup_M (A \cap_M C) = \{x, x\}$, ami különbözik a bal oldalon kapott $\{x\}$-től.

7.2.2. Kiválasztással kapcsolatos szabályok

A kiválasztás művelete döntő jelentőségű a lekérdezésoptimalizálás szempontjából. Mivel a kiválasztások lényegesen csökkenthetik a relációk méretét, a hatékony lekérdezésfeldolgozás egyik legfontosabb szabálya, hogy a kiválasztásokat vigyük lefelé a fában mindaddig, amíg ez nem változtatja meg a kifejezés eredményét. A korai lekérdezésoptimalizálók valóban ennek a transzformációnak a változatait használták a jó logikai tervhez vezető elsődleges stratégiaként. Amint röviden rá fogunk mutatni, a „kiválasztások tologatása lefelé a fában" transzformáció már nem teljesen általános, de a „kiválasztások tologatása" elv a lekérdezésoptimalizálóknak még mindig egy jelentős eszköze.

Ebben a részben a σ operátorra vonatkozó szabályokat fogjuk tanulmányozni. Elsőként, ha egy kiválasztás feltétele összetett (azaz AND vagy OR által összekapcsolt feltételekből áll), akkor azzal segíthetünk, hogy a feltételt szétvágjuk az alkotóelemeire. Ezt az indokolja, hogy egy olyan rész, amely kevesebb attribútumot tartalmaz, mint az egész feltétel, esetleg elmozdítható egy megfelelő helyre, ahová a teljes feltétel nem. Ennek megfelelően, a kiválasztásra vonatkozó első két szabályt *szétvágási szabálynak* nevezzük:

- $\sigma_{C_1 \text{ AND } C_2}(R) = \sigma_{C_1}(\sigma_{C_2}(R))$
- $\sigma_{C_1 \text{ OR } C_2}(R) = (\sigma_{C_1}(R)) \cup_H (\sigma_{C_2}(R))$

A második – OR-ra vonatkozó – szabály azonban csak akkor működik, ha az R reláció halmaz. Láthatjuk ugyanis, hogy ha az R multihalmaz lenne, akkor a halmazegyesítés eltüntetné az ismétlődéseket, helytelenül.

Vegyük észre, hogy a C_1 és C_2 sorrendje nem kötött. Úgy is írhattuk volna a fenti első szabályt, hogy a C_2-t a C_1 után alkalmazzuk, vagyis: $\sigma_{C_2}(\sigma_{C_1}(R))$. Általánosabban azt mondhatjuk, hogy a σ operátor tetszőleges sorozata esetén a sorrend felcserélhető:

- $\sigma_{C_1}(\sigma_{C_2}(R)) = \sigma_{C_2}(\sigma_{C_1}(R))$

7.6. példa: Legyen $R(a, b, c)$ egy reláció. Ekkor a $\sigma_{(a=1 \text{ OR } a=3) \text{ AND } b<c}(R)$ kifejezés szétvágható az alábbivá: $\sigma_{a=1 \text{ OR } a=3}(\sigma_{b<c}(R))$. Majd ezt a kifejezést az OR-nál tovább vághatjuk a következővé: $\sigma_{a=1}(\sigma_{b<c}(R)) \cup \sigma_{a=3}(\sigma_{b<c}(R))$. Mivel esetünkben lehetetlen, hogy egy sorra mind az $a = 1$, mind az $a = 3$ teljesüljön, még ha az egyesítésre az $\cup_M$-et használjuk is, ez az átalakítás érvényes, függetlenül attól, hogy az R halmaz-e vagy sem. Általában azonban az kell a VAGY szétvághatóságához, hogy az argumentum halmaz legyen, és hogy az $\cup_H$-t használjuk.

A szétvágást úgy is elkezdhettük volna, hogy a $\sigma_{b<c}$-ből készítünk külső műveletet, azaz: $\sigma_{b<c}(\sigma_{a=1 \text{ OR } a=3}(R))$. Amikor azután szétvágva az OR-t, azt kapjuk, hogy $\sigma_{b<c}(\sigma_{a=1}(R) \cup \sigma_{a=3}(R))$, ami az elsőként kapott kifejezéssel ekvivalens, de attól valamelyest különböző. □

A kiválasztásra vonatkozó szabályok következő csoportja azt teszi lehetővé, hogy a kiválasztásokat a szorzat, egyesítés, metszet, különbség és összekapcsolás bináris operátorokon áttoljuk. Háromféle szabály van, attól függően, hogy opcionális vagy kötelező a kiválasztást az egyes argumentumokhoz odavinni:

1. Egyesítés esetén a kiválasztást mindkét argumentumra alkalmazni *kell.*
2. Különbség esetén a kiválasztást az első argumentumra alkalmazni kell, a másodikra pedig lehet.
3. A többi operátor esetében csak azt követeljük meg, hogy a kiválasztást egy argumentumra alkalmazzuk. Az összekapcsolásnál és a szorzatnál lehet, hogy nincs annak értelme, hogy a kiválasztást mindkét argumentumhoz bevigyük, mivel egy argumentum vagy rendelkezik, vagy nem azokkal az attribútumokkal, amelyeket a kiválasztás megkíván. Ha lehetséges is a mindkettőre történő alkalmazás, ez vagy javít a terven, vagy nem; lásd a 7.2.1. feladatot.

Az egyesítésre vonatkozó szabály:

- $\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$

Itt kötelezően le kell vinni a kiválasztást a fa mindkét ágán.
A különbségre vonatkozó szabály egyik változata:

- $\sigma_C(R - S) = \sigma_C(R) - S$

Az is megengedett azonban, hogy mindkét argumentumhoz odavisszük a kiválasztást:

- $\sigma_C(R - S) = \sigma_C(R) - \sigma_C(S)$

A soron következő szabályok megengedik, hogy a kiválasztást az egyik vagy mindkét argumentumhoz elvigyük. Ha a kiválasztás σ_C, akkor ezt a kiválasztást csak egy olyan relációhoz tolhatjuk, amely rendelkezik a C-ben említett összes attribútummal, ha van ilyen. Az alábbi szabályokat azzal a feltételezéssel élve fogalmazzuk meg, hogy az R relációban megvan az összes C-ben szereplő attribútum.

- $\sigma_C(R \times S) = \sigma_C(R) \times S$
- $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$
- $\sigma_C(R \underset{D}{\bowtie} S) = \sigma_C(R) \underset{D}{\bowtie} S$
- $\sigma_C(R \cap S) = \sigma_C(R) \cap S$

Ha a C-ben csak S-beli attribútumok szerepelnek, akkor azt írhatjuk, hogy:

- $\sigma_C(R \times S) = R \times \sigma_C(S)$

és hasonlóan írhatók át a $\bowtie$, $\underset{D}{\bowtie}$ és $\cap$ operátorok szabályai. Ha az R és S relációk mindegyikében szerepel az összes C-beli attribútum, akkor használható az alábbi szabály:

- $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie \sigma_C(S)$

Vegyük észre, hogy nem alkalmazható ilyen szabály, ha az operátor $\times$ vagy $\underset{D}{\bowtie}$, ezekben az esetekben ugyanis R-nek és S-nek nincsenek közös attribútumai. A $\cap$ esetében viszont a szabály mindig érvényes lesz, hiszen ekkor az R és S sémája ugyanaz kell hogy legyen.

7.7. példa: Tekintsük az $R(a, b)$ és $S(b, c)$ relációkat és a következő kifejezést:

$$\sigma_{(a = 1 \text{ OR } a = 3) \text{ AND } b < c}(R \bowtie S)$$

A $b < c$ feltétel egyedül az S-re alkalmazható, az $a = 1$ OR $a = 3$ feltétel pedig csak az R-re alkalmazható. Ezért a két feltétel összekötő AND szétvágásával kezdjük, csakúgy mint a 7.6. példa első változatánál:

$$\sigma_{a = 1 \text{ OR } a = 3}(\sigma_{b<c}(R \bowtie S))$$

Ezt követően bevihetjük a $\sigma_{b<c}$ kiválasztást az S-hez, ami az alábbi kifejezést adja:

$$\sigma_{a = 1 \text{ OR } a = 3}(R \bowtie \sigma_{b<c}(S))$$

Végül az első feltételt bevisszük az R-hez: $\sigma_{a = 1 \text{ OR } a = 3}(R) \bowtie \sigma_{b < c}(S)$. Ha akarjuk, szétvághatjuk az OR-ral kapcsolt két feltételt, mint ahogy a 7.6. példában tettük. Ez azonban vagy előnyös, vagy nem. □

Néhány triviális szabály

Nem áll szándékunkban a relációs algebrára érvényes összes szabályt megfogalmazni. Az olvasó legyen különösen óvatos a szabályokkal kapcsolatban, ha speciális esetekről van szó: például amikor egy reláció üres, egy kiválasztás vagy théta-összekapcsolás feltétele mindig igaz vagy hamis, vagy a teljes attribútumlistára történik vetítés. Lássunk néhányat a sok lehetséges speciális esetre vonatkozó szabály közül:

- Üres relációra vonatkozó bármilyen kiválasztás üres relációt ad.
- Ha egy C feltétel mindig igaz (pl. $x > 10$ OR $x \leq 10$ olyan relációra vonatkozóan, amely kizárja, hogy x = NULL), akkor $\sigma_C(R) = R$.
- Ha R üres, akkor $R \cup S = S$.

7.2.3. Kiválasztások tologatása

Amint már említettük, egy kiválasztás tologatása lefelé a fában – azaz a 7.2.2. részben szereplő valamely szabály bal oldalának helyettesítése annak jobb oldalával – a lekérdezésoptimalizáló egyik leghatékonyabb eszköze. Sokáig azt feltételezték, hogy úgy optimalizálhatunk, ha a kiválasztásra vonatkozó szabályokat ebbe az irányba alkalmazzuk. Amikor viszont általánossá vált a nézettáblák támogatása, úgy találták, hogy bizonyos esetekben lényeges volt, hogy egy kiválasztást először olyan fentre felvigyünk a fában, amennyire lehet, és utána tologassuk lefelé a kiválasztásokat a lehetséges ágakon. A kiválasztások tologatásának jó megközelítését egy példával szemléltetjük.

7.8. példa: Tegyük fel, hogy adottak a következő relációk:

```
SzerepelBenne(filmCím, év, színészNév )
Film(cím, év, hossz, stúdióNév)
```

és az alábbi SQL-nézettábla:

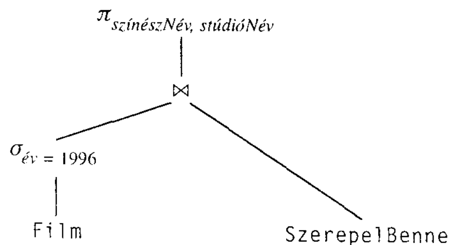
```
CREATE VIEW Filmek1996 AS
   SELECT *
   FROM Film
   WHERE év = 1996;
```

A „mely színészek mely stúdiónak dolgoztak 1996-ban?" kérdést megfogalmazó SQL-lekérdezés:

```
SELECT színészNév, stúdióNév
FROM Filmek1996 NATURAL JOIN SzerepelBenne;
```

A `Filmek1996` nézettáblát egy relációs algebrai kifejezés definiálja:

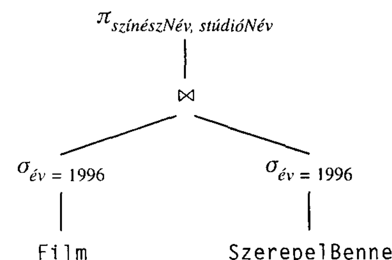
$\sigma_{év\,=\,1996}$(`Film`)



7.6. ábra. *Egy lekérdezés és nézettábla alapján készített logikai lekérdezésterv*

A lekérdezéshez, ami ennek a kifejezésnek a természetes összekapcsolása a `SzerepelBenne` relációval, majd egy vetítés a `színészNév` és `stúdióNév` attribútumokra, a 7.6. ábrán látható kifejezés vagy „logikai lekérdezésterv" tartozik.

A kifejezésben szereplő egyetlen kiválasztás már olyan lent van a fában, amennyire lehet, így nincs lehetőség a „kiválasztás tologatására lefelé a fában". A $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$ szabályt viszont alkalmazhatjuk „visszafelé", hogy a $\sigma_{év\,=\,1996}$ kiválasztást az összekapcsolás fölé vigyük. Ezután, mivel az *év* a `Film` és a `SzerepelBenne` relációknak egyaránt attribútuma, a kiválasztást az összekapcsolás csomópont *mindkét* gyereke felé levihetjük. Az eredményül kapott logikai lekérdezéstervet a 7.7. ábra mutatja. Az új terv valószínűleg javulást jelent, hiszen a `SzerepelBenne` reláció méretét csökkentjük, mielőtt összekapcsoljuk az 1996-os filmekkel. □



7.7. ábra. *A lekérdezésterv javítása a kiválasztás felfelé és lefelé tologatásával*

7.2.4. Vetítéssel kapcsolatos szabályok

A kiválasztáshoz hasonlóan a vetítéseket is „tolhatjuk lefelé", át más operátorokon. A vetítések tologatása abban különbözik a kiválasztások tologatásától, hogy amikor vetítést tolunk, akkor a vetítés általában ott is megmarad, ahol van. Másképpen szólva, vetítés „tolása" valójában egy új vetítés bevezetését jelenti valahol a létező vetítés alatt.

A vetítések eltolása hasznos ugyan, de általában nem annyira, mint a kiválasztás tologatása. Ennek az az oka, hogy míg a kiválasztás gyakran nagymértékben csökkenti egy reláció méretét, a vetítés során a sorok száma ugyanaz marad, csak a sorok hossza csökken. Sőt, amint a 6.1.3. részben megfigyeltük, a vetítés néha növeli a sorok hosszát.

Hogy a transzformációkat a vetítés általános alakjának segítségével írhassuk le, be kell vezetnünk néhány terminológiát. Nézzünk egy $E \rightarrow x$ kifejezést egy vetítési listából, ahol E vagy egy attribútum, vagy egy attribútumokat és konstansokat tartalmazó kifejezés. Azt mondjuk, hogy az E-ben előforduló összes attribútum a vetítés *bemeneti* kifejezés. Azt mondjuk, hogy az E-ben előforduló összes attribútum a vetítés *bemeneti* attribútuma, az x pedig egy *kimeneti* attribútum. Ha a kifejezés egyetlen attribútum, akkor az egyben bemeneti és kimeneti attribútum is. Láthatjuk, hogy a kifejezés nem lehet más, mint egyetlen attribútum nyíl nélkül vagy átnevezés, így az összes esetet lefedtük.

Ha a vetítési lista csak attribútumokból áll, tehát nincs átnevezés vagy olyan kifejezés, ami más, mint egyetlen attribútum, akkor *egyszerű* vetítésről beszélünk. A klasszikus relációs algebrában minden vetítés egyszerű.

7.9. példa: A $\pi_{a,\,b,\,c}(R)$ vetítés egyszerű; a, b és c egyszerre bemeneti attribútumok és kimeneti attribútumok. A $\pi_{a\,+\,b\,\to\,x,\,c}(R)$ vetítés viszont nem egyszerű. Ennek bemeneti attribútumai az a, b és a c, kimeneti attribútumai pedig az x és a c. □

A vetítésre vonatkozó szabályok mögött az alábbi alapelv húzódik meg:

- A kifejezésfában bárhol bevezethetünk egy vetítést mindaddig, amíg az csakis olyan attribútumokat tüntet el, amelyeket egyetlen fentebb elhelyezkedő operátor sem használ, valamint a teljes kifejezés eredményében sem szerepelnek.

A szabályok legegyszerűbb alakjaiban a bevezetett vetítések mind egyszerűek:

- $\pi_L(R \bowtie S) = \pi_L(\pi_M(R) \bowtie \pi_N(S))$, ahol M az R azon attribútumainak listája, amelyek vagy összekapcsolási attribútumok (az R és S sémájában egyaránt szerepelnek), vagy bemeneti attribútumai az L-nek, és N az S azon attribútumainak listája, amelyek vagy összekapcsolási attribútumok, vagy bemeneti attribútumai az L-nek.
- $\pi_L(R \underset{C}{\bowtie} S) = \pi_L(\pi_M(R) \underset{C}{\bowtie} \pi_N(S))$, ahol M az R azon attribútumainak listája, amelyek vagy összekapcsolási attribútumok (vagyis a C feltételben előfordulnak), vagy bemeneti attribútumai az L-nek, és N az S azon attribútumainak listája, amelyek vagy összekapcsolási attribútumok, vagy bemeneti attribútumai az L-nek.
- $\pi_L(R \times S) = \pi_L(\pi_M(R) \times \pi_N(S))$, ahol M az R, illetve N az S azon attribútumainak listája, amelyek bemeneti attribútumai az L-nek.

7.10. példa: Legyen $R(a, b, c)$ és $S(c, d, e)$ két reláció. Vegyük a következő kifejezést: $\pi_{a\,+\,e\,\to\,x,\,b\,\to\,y}(R \bowtie S)$. A vetítés bemeneti attribútumai a, b és e, valamint c az egyetlen összekapcsolási attribútum. A vetítések összekapcsolás alá történő eltolásának szabályát alkalmazva ekvivalens kifejezést kapunk:

$$\pi_{a\,+\,e\,\to\,x,\,b\,\to\,y}(\pi_{a,\,b,\,c}(R) \bowtie \pi_{c,\,e}(S))$$

Vegyük észre, hogy a $\pi_{a,\,b,\,c}(R)$ egy triviális vetítés, ami az R összes attribútumára vetít. Ez a vetítés így kiküszöbölhető, ami egy harmadik ekvivalens kifejezést eredményez:

$$\pi_{a\,+\,e\,\to\,x,\,b\,\to\,y}(R \bowtie \pi_{c,\,e}(S))$$

Az egyetlen változás tehát az eredetihez képest az, hogy az összekapcsolás előtt az S-ből elhagyjuk a d attribútumot. □

Egy vetítést teljesen végrehajthatunk egy multihalmaz-egyesítés előtt:

- $\pi_L(R \cup_M S) = \pi_L(R) \cup_M \pi_L(S)$

Nem vihetők azonban a vetítések sem a halmazegyesítések, sem a metszet és különbség halmaz vagy multihalmaz változatai elé.

7.11. példa: Legyenek $R(a, b)$ a $\{(1, 2)\}$, $S(a, b)$ pedig a $\{(1, 3)\}$ relációk. Ekkor $\pi_a(R \cap S) = \pi_a(\emptyset) = \emptyset$. Ugyanakkor $\pi_a(R) \cap \pi_a(S) = \{(1)\} \cap \{(1)\} = \{(1)\}$. □

Ha a vetítés számításokat tartalmaz, és a vetítési lista valamely kifejezésének bemeneti attribútumai teljes egészében egy, a vetítés alatt elhelyezkedő összekapcsolás vagy szorzat egyik argumentumához tartoznak, akkor megvan az a lehetőségünk, bár nem kötelező, hogy az adott számítást közvetlenül azon az argumentumon végezzük el. Egy példával szemléltetjük ezt az esetet.

7.12. példa: Legyen ismét az $R(a, b, c)$ és $S(c, d, e)$ két reláció, és tekintsük a következő összekapcsolást és vetítést: $\pi_{a\,+\,b\,\to\,x,\,d\,+\,e\,\to\,y}(R \bowtie S)$. Az $a + b$ összeadást és annak x-re történő átnevezését közvetlenül az R relációhoz vihetjük, és ugyanezt tehetjük a $d + e$ összeggel az S vonatkozásában. Az így kapott ekvivalens kifejezés: $\pi_{x,\,y}(\pi_{a\,+\,b\,\to\,x,\,c}(R) \bowtie \pi_{d\,+\,e\,\to\,y,\,c}(S))$.

Speciálisan kell kezelni azt az esetet, ha x vagy y megegyezik c-vel. Ekkor nem nevezhetnénk át az összeget c-re, mert egy relációnak nem lehet két attribútuma ugyanazzal a c névvel. Be kellene vezetni egy ideiglenes nevet, és az összekapcsolás fölött végre kellene hajtani egy további átnevezést. A $\pi_{a\,+\,b\,\to\,c,\,d\,+\,e\,\to\,y}(R \bowtie S)$ kifejezés például a következő kifejezéssé alakítható át: $\pi_{z\,\to\,c,\,y}(\pi_{a\,+\,b\,\to\,z,\,c}(R) \bowtie \pi_{d\,+\,e\,\to\,y,\,c}(S))$. □

Egy vetítést be lehet iktatni egy kiválasztás alá is:

- $\pi_L(\sigma_C(R)) = \pi_L(\sigma_C(\pi_M(R)))$, ahol M azoknak az attribútumoknak a listája, amelyek vagy bementi attribútumai az L-nek, vagy szerepelnek a C feltételben.

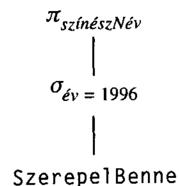
Csakúgy mint a 7.12. példában, lehetséges, hogy az L lista számításait inkább az M-ben végezzük el, feltéve, hogy a C feltétel nem igényli az L azon attribútumait, amelyek valamely számításban érintettek.

Gyakran akkor is lejjebb akarjuk vinni a vetítéseket a kifejezésfában, ha fent ott kell hagyni egy másik vetítést, mert a vetítések általában csökkentik a sorok méretet, és így egy köztes reláció által elfoglalt blokkok számát. Vigyáznunk kell azonban, amikor ezt tesszük, mert vannak tipikus példák, amikor egy vetítés levitele időbe kerül.

7.13. példa: Vegyük azt a `SzerepelBenne(filmCím, év, színészNév)` relációra vonatkozó lekérdezést, amely az 1996-ban dolgozó színészeket keresi:

```
SELECT színészNév
FROM SzerepelBenne
WHERE év = 1996;
```

A 7.8. ábra mutatja ennek a lekérdezésnek a közvetlen átalakítását logikai lekérdezéstervvé.

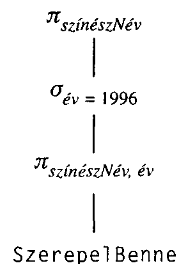


7.8. ábra. *Logikai lekérdezésterv a 7.13. példában szereplő lekérdezéshez*

A kiválasztás alá beilleszthetünk egy vetítést két attribútummal:

1. `színészNév`, ugyanis ez az attribútum kell az eredményhez, és
2. `év`, mert ez az attribútum szükséges a kiválasztási feltételhez.

Az eredmény a 7.9. ábrán látható.



7.9. ábra. *Vetítés bevezetésének az eredménye*

Ha a `SzerepelBenne` nem tárolt reláció lenne, hanem valamilyen művelet – mint például összekapcsolás – által létrehozott reláció, akkor lenne értelme a 7.9. ábra szerinti tervnek. „Futószalagosíthatjuk" a vetítést (lásd a 7.7.3. részt), amint az összekapcsolás sorait előállítjuk, egyszerűen elhagyva a nem használt `filmCím` attribútumot.

A mi esetünkben azonban a `SzerepelBenne` egy tárolt reláció. Az alsó vetítés a 7.9. ábrán valójában nagy időpocsékolást jelent, különösen akkor, ha létezik index az `év` attribútumra. Ekkor a 7.8. ábra logikai lekérdezéstervén alapuló fizikai lekérdezésterv először az indexet használná az olyan `SzerepelBenne` sorok megtalálására, ahol az `év` 1996, ami feltehetően a soroknak csak egy kis hányadát jelenti. Ha a vetítést végezzük el először a 7.9. ábrának megfelelően, akkor a `SzerepelBenne` reláció minden sorát be kell olvasni és vetíteni kell.

Hogy a dolgok még rosszabbul nézzenek ki, az `év`-hez tartozó index valószínűleg nem használható a vetített $\pi_{színészNév,\ év}$(`SzerepelBenne`) relációhoz, így a kiválasztásnak a vetítés eredményeként megkapott összes sort végig kell olvasnia. □

7.2.5. Összekapcsolásra és szorzatra vonatkozó szabályok

A 7.2.1. részben már láttunk több szabályt az összekapcsolással és a szorzattal kapcsolatban: azok kommutatív és asszociatív szabályait. Van azonban néhány további szabály, amelyek közvetlenül az összekapcsolás definíciójából következnek.

- $R \underset{C}{\bowtie} S = \sigma_C(R \times S)$
- $R \bowtie S = \pi_L(\sigma_C(R \times S))$, ahol C az a feltétel, amely az R-ből és S-ből származó azonos nevű attribútumpárok egyenlőségét vizsgálja, az L pedig olyan lista, amely tartalmazza az összes egyenlővé tett attribútumpár egyikét, valamint az R és S minden maradék attribútumát.

Ezeknek a szabályoknak a használatát a 6.1.5. részben adott 6.5. és 6.6. példák szemléltették. A gyakorlatban rendszerint jobbról balra alkalmazzuk ezeket a szabályokat, vagyis egy szorzatot követő kiválasztást azonosítunk összekapcsolásként. Ennek az az oka, hogy az összekapcsolások kiszámításához használt algoritmusok általában sokkal gyorsabbak, mint az olyan algoritmusok, amelyek egy szorzatot és a szorzat (nagyon nagy méretű) eredményére alkalmazott kiválasztást számítanak ki.

7.2.6. Ismétlődések elhagyására vonatkozó szabályok

Az ismétlődéseket eltávolító δ operátort sok operátoron keresztül lehet tolni, de nem mindegyiken. A δ lefelé történő mozgatása a fában csökkenti a köztes relációk méretét, így kifizetődő lehet. Sőt a δ néha olyan helyre vihető, ahol egyszerűen elhagyható, mert olyan relációra vonatkozik, amelyről tudni lehet, hogy nem tartalmaz ismétlődéseket:

- $\delta(R) = R$, ha R-ben nincsenek ismétlődések. Ilyen fontos esetekről van szó például, ha R a következő:

  a) Egy tárolt reláció, amelyhez elsődleges kulcsot deklaráltunk.
  b) Egy γ művelet eredményeként kapott reláció, mivel egy csoportosítás eredménye egy ismétlődések nélküli reláció.

Az alábbi néhány szabály a δ operátort más operátorokon „tolja" keresztül:

- $\delta(R \times S) = \delta(R) \times \delta(S)$
- $\delta(R \bowtie S) = \delta(R) \bowtie \delta(S)$
- $\delta(R \underset{C}{\bowtie} S) = \delta(R) \underset{C}{\bowtie} \delta(S)$
- $\delta(\sigma_C(R)) = \sigma_C(\delta(R))$

A δ odavihető egy metszet egyik vagy mindkét argumentumához is:

- $\delta(R \cap_M S) = \delta(R) \cap_M S = R \cap_M \delta(S) = \delta(R) \cap_M \delta(S)$

Ugyanakkor viszont a δ általában nem vihető át a $\cup_M$, $-_M$ vagy π operátorokon.

7.14. példa: Legyen az R reláció olyan, amelyben a t sor két példányban szerepel, az S pedig olyan, amelyben a t sor egy példányban szerepel. Ekkor a $\delta(R \cup_M S)$ egy példányát, míg a $\delta(R) \cap_M \delta(S)$ két példányát tartalmazza a t sornak. Továbbá, a $\delta(R -_M S)$ tartalmazza a t egy példányát, míg a $\delta(R) -_M \delta(S)$ nem tartalmazza a t sort.

Vegyük most azt a $T(a, b)$ relációt, amely az (1, 2) és (1, 3) sorok egy-egy példányát tartalmazza, és mást nem. Ekkor a $\delta(\pi_a(R))$ eredményében az (1) sor egyszer szerepel, míg a $\pi_a(\delta(R))$ eredményében az (1) sor kétszer fordul elő. □

Végül megjegyezzük, hogy a δ felcserélésének az $\cup_H$, $\cap_H$ és $-_H$ operátorokkal nincs értelme. Ehelyett a δ elhagyható, mivel halmazok előállításakor garantáltan nem kapunk ismétlődéseket. Például:

- $\delta(R \cup_H S) = R \cup_H S$

Vegyük észre azonban, hogy az $\cup_H$ vagy a többi halmazművelet megvalósítása magában foglalja az ismétlődések eltüntetésének folyamatát, ami egyenértékű a δ alkalmazásával; lásd például a 6.3.3. részt.

7.2.7. Csoportosításra és összesítésre vonatkozó szabályok

Ha megnézzük a γ operátort, akkor azt találjuk, hogy sok transzformáció alkalmazhatósága a használt összesítő operátor részleteitől függ. Emiatt nem állíthatunk fel szabályokat olyan általánosságban, mint ahogyan a többi operátor esetében tettük. Kivételt képez az a 7.2.6. részben már említett eset, amikor egy γ elnyel egy δ-t. Egész pontosan:

- $\delta(\gamma_L(R)) = \gamma_L(R)$

Egy másik általános szabály az, hogy ha úgy kívánjuk, akkor a γ operátor alkalmazása előtt az argumentumban nem használt attribútumokat elhagyhatjuk egy vetítés segítségével. Ez a szabály így fogalmazható meg:

- $\gamma_L(R) = \gamma_L(\pi_M(R))$, ahol M az R azon attribútumainak listája, amelyek L-ben előfordulnak.

Annak oka, hogy más transzformációk a γ-ban szereplő összesítésektől függnek, a következőkben áll. Bizonyos összesítéseket – ilyen a `MIN` és a `MAX` – nem befolyásol az ismétlődések jelenléte vagy hiánya. A többi összesítés – `SUM`, `COUNT` és `AVG` – viszont általában más értéket produkál, ha az összesítés alkalmazása előtt megszüntetjük az ismétlődéseket.

Egy γ_L operátort *ismétlődésérzéketlennek* nevezünk, ha L-ben csak `MIN` és/vagy `MAX` összesítések szerepelnek. Ezek után:

- $\gamma_L(R) = \gamma_L(\delta(R))$, feltéve hogy γ_L ismétlődésérzéketlen.

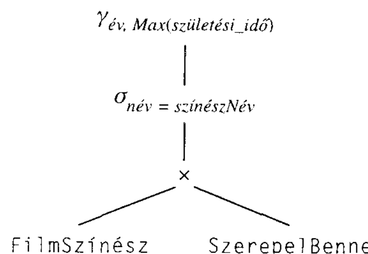
7.15. példa: Tegyük fel, hogy adottak a

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

relációk, és hogy minden évhez meg akarjuk keresni az adott évben valamilyen filmben szereplő legfiatalabb színész születési idejét. Ez az alábbi lekérdezéssel fejezhető ki:

```
SELECT év, MAX(születési_idő)
FROM FilmSzínész, SzerepelBenne
WHERE név = színészNév
GROUP BY év;
```

A közvetlenül a lekérdezésből kapott kiindulási logikai lekérdezéstervet a 7.10. árán láthatjuk. A `FROM` listát egy szorzat, a `WHERE` záradékot pedig egy fölötte lévő kiválasztás fejezi ki. A csoportosítást és összesítést az ezek fölött elhelyezkedő γ operátor fejezi ki.

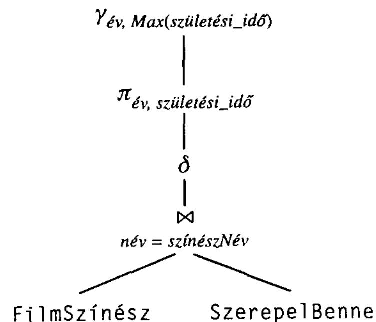


7.10. ábra. *Kiindulási logikai lekérdezésterv a 7.15. példa lekérdezéséhez*

A 7.10. ábrán elvégezhető több átalakítás is:

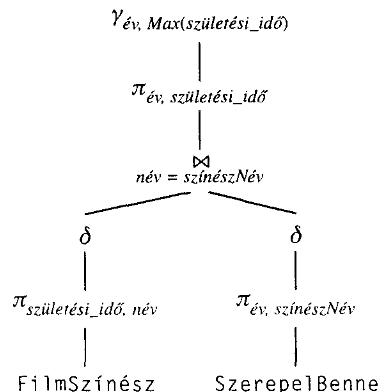
1. A kiválasztás és a szorzat összevonása egy egyenlőségen alapuló összekapcsolássá.
2. Egy δ beillesztése a γ alá, mivel γ ismétlődésérzéketlen.
3. Egy olyan π vetítés beillesztése a γ és az újonnan bevezetett δ közé, ami az `év`-re és a `születési_idő`-re, vagyis a γ szempontjából lényeges attribútumokra terjed ki.

Az eredményül kapott tervet a 7.11. ábra mutatja.



7.11. ábra. *Egy másik lekérdezésterv a 7.15. példa lekérdezéséhez*

Most levihetjük a δ-t az $\bowtie$ alá, és ez alá π-ket vezethetünk be, ha úgy kívánjuk. Az új lekérdezésterv a 7.12. ábrán található. Ha a `név` kulcsa a `FilmSzínész` relációnak, akkor az ehhez a relációhoz vezető ágon a δ elhagyható. □



7.12. ábra. *Egy harmadik lekérdezésterv a 7.15. példa lekérdezéséhez*

7.2.8. Feladatok

* **7.2.1. feladat:** Amikor egy kiválasztást egy bináris operátor mindkét argumentumához be lehet vinni, el kell döntenünk, hogy ezt megtegyük-e. Hogyan befolyásolná a döntésünket az, hogy az egyik argumentumhoz léteznek indexek? Tekintsük például a $\sigma_C(R \cap S)$ kifejezést, ahol az S-hez tartozik egy index.

7.2.2. feladat: Adjunk példákat az alábbiak bizonyítására:

* a) A vetítés nem vihető le a halmazegyesítés alá.
b) A vetítés nem vihető le a halmaz- vagy multihalmaz-különbség alá.
c) Az ismétlődések megszüntetése (δ) nem vihető le a vetítés alá.

d) Az ismétlődések megszüntetése (δ) nem vihető le a multihalmaz-egyesítés vagy különbség alá.

! **7.2.3. feladat:** Bizonyítsuk be, hogy egy vetítés minden esetben levihető egy multihalmaz-egyesítés mindkét ágán.

! **7.2.4. feladat:** A halmazokra vonatkozó szabályok némelyike érvényes multihalmazokra is, míg mások nem. Az alább felsorolt szabályok igazak halmazok esetén. Döntsük el, hogy multihalmazokra is igazak lesznek-e, avagy sem. Vagy bizonyítsuk be, hogy a szabály igaz multihalmazokra, vagy adjunk ellenpéldát.

* a) $R \cup R = R$ (az egyesítés idempotens)
b) $R \cap R = R$ (a metszet idempotens)
c) $R - R = \emptyset$
d) $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$ (az egyesítés metszet feletti disztributivitása)

! **7.2.5. feladat:** Multihalmazokra úgy definiálhatjuk a $\subseteq$ műveletet, hogy $R \subseteq S$ akkor és csak akkor, ha minden x esetén az x R-beli előfordulásainak száma kisebb vagy egyenlő az x S-beli előfordulásainak számával. Döntsük el, hogy a következő állítások (melyek igazak halmazokra) igazak-e multihalmazokra; bizonyítsuk be, vagy adjunk ellenpéldát.

a) Ha $R \subseteq S$, akkor $R \cup S = S$.
b) Ha $R \subseteq S$, akkor $R \cap S = R$.
c) Ha $R \subseteq S$ és $S \subseteq R$, akkor $R = S$.

7.2.6. feladat: Induljunk ki a $\pi_L(R(a, b, c) \bowtie S(b, c, d, e))$ kifejezésből, és tologassuk a vetítést lefelé a fában, amíg csak lehet. L az alábbi:

* a) $b + c \to x,\ c + d \to y$
b) $a, b, a + d \to z$

! **7.2.7. feladat:** A 7.15. példában említettük, hogy a bemutatott tervek egyike sem feltétlenül a legjobb terv. Tudna esetleg egy jobb tervet adni?

! **7.2.8. feladat:** Vegyük az $R(a, b)$ relációt érintő következő feltételezett egyenlőségeket. Döntsük el, hogy igazak-e; adjunk bizonyítást vagy ellenpéldát.

a) $\gamma_{MIN(a) \to y,\, x}(\gamma_{a,\, SUM(b) \to x}(R)) = \gamma_{y,\, SUM(b) \to x}(\gamma_{MIN(a) \to y,\, b}(R))$
b) $\gamma_{MIN(a) \to y,\, x}(\gamma_{a,\, MAX(b) \to x}(R)) = \gamma_{y,\, MAX(b) \to x}(\gamma_{MIN(a) \to y,\, b}(R))$

!! **7.2.9. feladat:** A 6.1.3. feladatban szereplő összekapcsolás jellegű operátorok az ismert szabályok némelyikének engedelmeskednek, másoknak viszont nem. Döntsük el, hogy a következő szabályok igazak-e, avagy sem. Bizonyítsuk be a szabályt, vagy adjunk ellenpéldát.

* a) $\sigma_C(R \ltimes S) = \sigma_C(R) \ltimes S$

* b) $\sigma_C(R \overset{\circ}{\bowtie} S) = \sigma_C(R) \overset{\circ}{\bowtie} S$

c) $\sigma_C(R \overset{\circ}{\bowtie}_L S) = \sigma_C(R) \overset{\circ}{\bowtie}_L S$ (ahol C-ben az R minden attribútuma szerepel)

d) $\sigma_C(R \overset{\circ}{\bowtie}_L S) = R \overset{\circ}{\bowtie}_L \sigma_C(S)$ (ahol C-ben az R minden attribútuma szerepel)

e) $\pi_L(R \overline{\ltimes} S) = \pi_L(R) \overline{\ltimes} S$

* f) $(R \overset{\circ}{\bowtie} S) \overset{\circ}{\bowtie} T = R \overset{\circ}{\bowtie} (S \overset{\circ}{\bowtie} T)$

g) $R \overset{\circ}{\bowtie} S = S \overset{\circ}{\bowtie} R$

h) $R \overset{\circ}{\bowtie}_L S = S \overset{\circ}{\bowtie}_L R$

i) $R \ltimes S = S \ltimes R$

7.3. Elemzőfák átalakítása logikai lekérdezéstervekké

Most pedig visszatérünk a lekérdezésfordító tárgyalásához. Miután a 7.1. részben létrehoztunk egy elemzőfát, a következő feladat az elemzőfa átalakítása a jónak vélt logikai lekérdezéstervvé. A 7.1. ábrának megfelelően ez két lépésből áll.

Az első lépés az elemzőfa csomópontjait és struktúráit helyettesíti (megfelelő csoportosításban) a relációs algebra egy vagy több operátora segítségével. Néhány ilyen szabályra javaslatot fogunk tenni, néhány továbbit pedig meghagyunk feladatnak. A második lépés veszi az első lépés által előállított relációs algebrai kifejezést, és azt egy olyan kifejezéssé alakítja, amely várhatóan a leghatékonyabb fizikai lekérdezéstervvé konvertálható.

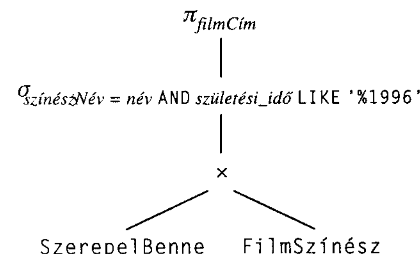
7.3.1. Átfordítás relációs algebrába

Most formalizmusok nélkül bevezetünk néhány olyan szabályt, amely az SQL-elemzőfáknak algebrai logikai lekérdezéstervvé történő transzformálásával kapcsolatos. Az első, talán legfontosabb szabály lehetővé teszi számunkra, hogy minden „egyszerű” „select-from-where” szerkezetet közvetlenül konvertáljunk a relációs algebrába. Informálisan ez a szabály így hangzik:

- Ha adott egy <Lekérdezés>, ami egy <SFW> struktúra, és a <Feltétel> ebben a struktúrában nem tartalmaz alkérdést, akkor a teljes struktúra – a select-lista, a from-lista és a feltétel – egy olyan relációs algebrai kifejezéssel helyettesíthető, amely alulról felfelé az alábbiakból áll:

  1. A <FromLista>-ban szereplő összes reláció szorzata, ami az alábbinak válik az argumentumává:
  2. Egy σ_C kiválasztás, ahol C a helyettesítés alatt álló struktúra <Feltétel> kifejezése, ami viszont az alábbinak lesz az argumentuma:
  3. Egy π_L vetítés, ahol L a <SelLista> attribútumlistája.

7.16. példa: Tekintsük a 7.5. ábrán látható elemzőfát. A fenti „select-from-where” transzformáció a 7.5. ábra teljes elemzőfájára alkalmazható. Vesszük a from-lista két relációjának – `SzerepelBenne` és `FilmSzínész` – szorzatát, kiválasztunk a <Feltétel>-ben gyökerező részfának megfelelő feltétel alapján, és vetítünk a `filmCím`-ből álló select-listára. Az eredményül kapott relációs algebrai kifejezés a 7.13. ábrán látható.



7.13. ábra. *Egy elemzőfa átalakítása algebrai kifejezésfává*

Ugyanez a transzformáció nem alkalmazható a 7.3. ábra külső szintű lekérdezésénél. Ez azért van így, mert a feltétel alkérdést tartalmaz. A 7.3.2. részben fogjuk megtárgyalni, hogy az alkérdéseket tartalmazó feltételeket hogyan lehet kezelni. Tanulmányozzuk a „Kiválasztási feltételek korlátozása” címet viselő bekeretezett részt is,

Kiválasztási feltételek korlátozása

Elcsodálkozhatunk azon, hogy miért nem engedjük meg, hogy egy σ_C kiválasztás C feltétele alkérdést tartalmazzon. A relációs algebrában az a szokás, hogy egy művelet *argumentumai* – a nem indexként szereplő elemek – olyan kifejezések, amelyek relációkat eredményeznek. Másrészt viszont, a paraméterek – az alsó indexként szereplő elemek – nem relációtípusúak. A σ_C-ben például a C paraméter egy logikai típusú feltétel, a π_L-ben pedig az L paraméter egy attribútumokból vagy formulákból álló lista.

Ha követjük ezt a hagyományt, akkor egy paraméter alkalmazható a reláció argumentum(ok) minden egyes sorára, bármilyen számítást jelent is ez. A paraméterek használatára vonatkozó korlátozás egyszerűbbé teszi a lekérdezésoptimalizálást. Tegyük fel most ellenben, hogy egy $\sigma_C(R)$ operátor C feltétele tartalmazhat egy alkérdést. Ekkor a C alkalmazása az R egyes soraira igényli az alkérdés kiszámítását. Kiszámítjuk ezt újra az R minden egyes soránál? Ez szükségtelenül drága volna, kivéve ha az alkérdés *korrelatív*, azaz annak értékei függnek valamitől, ami azon kívül definiált, mint ahogy például a 7.3. ábra alkérdése függ a `színészNév` értékétől. A legtöbb esetben még a korrelatív alkérdéseket is ki lehet értékelni anélkül, hogy azt minden sornál újra ki kellene számítani, feltéve, hogy a kiszámítást helyesen szervezzük meg.

ami arra ad magyarázatot, hogy miért teszünk különbséget az alkérdéseket tartalmazó, illetve nem tartalmazó feltételek között.

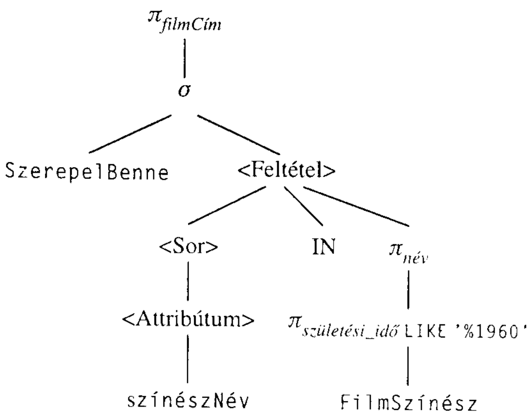
Alkalmazhatjuk azonban a „select-from-where" szabályt a 7.3. ábrán lévő alkérdésre. Az alkérdésből nyert relációs algebrai kifejezés: $\pi_{név}(\sigma_{születési_idő}$ LIKE '%1960'(FilmSzínész)). □

7.3.2. Alkérdések eltávolítása feltételekből

Azokhoz az elemzőfákhoz, amelyekben van alkérdést tartalmazó <Feltétel>, bevezetünk egy közbeeső operátort, amely az elemzőfa szintaktikus kategóriái és a relációs algebrai operátorok között helyezkedik el, és relációkra vonatkozik. Ezt az operátort *kétargumentumú kiválasztásnak* nevezzük. A kétargumentumú kiválasztást a transzformált elemzőfában egy csomópont képviseli, amelynek címkéje σ, mégpedig paraméterek nélkül. E csomópont alatt elhelyezkedik egy bal oldali gyerek, amely azt az R relációt képviseli, amelyre a kiválasztás vonatkozik, valamint van egy jobb oldali gyerek, ami az R soraira vonatkozó feltételt megtestesítő kifejezés. Mindkét argumentum ábrázolható mint elemzőfa, mint kifejezésfa és mint a kettő keveréke.

7.17. példa: A 7.14. ábrán láthatjuk a 7.3. ábra elemzőfájának egy olyan átírását, amely használ egy kétargumentumú kiválasztást. Több transzformációt hajtottunk végre, mire a 7.3. ábrától eljutottunk a 7.14. ábrához:

1. A 7.3. ábrán szereplő alkérdést egy relációs algebrai kifejezéssel helyettesítettük a 7.16. példa végén mondottak alapján.
2. A „select-from-where" kifejezésekhez a 7.3.1. részben bevezetett szabálynak megfelelően helyettesítettük a külső szintű lekérdezést. A szükséges kiválasztást azonban egy kétargumentumú kiválasztás segítségével fejeztük ki, és nem a relációs algebra hagyományos σ operátorával. Következésképpen, az elemzőfa felső



7.14. ábra. *Egy σ kétargumentumú kifejezés, középúton az elemzőfa és a relációs algebra között*

<Feltétel> csomópontját nem helyettesítettük, az megmaradt mint a kiválasztás egyik argumentuma, de a hozzá tartozó kifejezés egy részét az (1) szerint helyettesítettük relációs algebrával.

Ez a fa további transzformációt igényel, ezt tárgyaljuk következőként. □

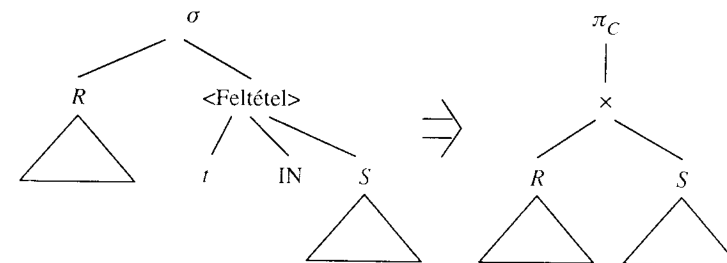
Szükségünk van szabályokra, amelyek lehetővé teszik, hogy egy kétargumentumú kiválasztást egy relációs algebrai egy argumentumú kiválasztással és egy másik relációs algebrai operátorral helyettesítsünk. A feltételek különböző formái külön szabályokat igényelhetnek. A legtöbb helyzetben a kétargumentumú kiválasztás eltávolítható, és tiszta relációs algebrai kifejezéshez juthatunk. Vannak azonban különleges esetek, amikor a kétargumentumú kiválasztást a helyén hagyjuk, és a logikai lekérdezésterv részének tekintjük.

Példaként megadjuk azt a szabályt, amelynek segítségével a 7.14. ábrán szereplő, IN operátort tartalmazó feltételt kezelhetjük. Vegyük észre, hogy az alkérdés a feltételben független, azaz a neki megfelelő reláció nem függ az éppen vizsgált sortól (elég egyszer kiszámítani). Az ilyen feltételeket elimináló szabály informálisan így fogalmazható meg:

- Tegyük fel, hogy van egy kétargumentumú kiválasztás, amelynek első argumentuma egy R relációt képvisel, a második argumentuma pedig egy t IN S, ahol az S kifejezés egy független alkérdés, t pedig az R bizonyos attribútumaiból összeállított sor. A fa az alábbi módon transzformálható:

  a) Helyettesítsük a <Feltételt>-t azzal a fával, ami nem más, mint az S kifejezés. Ha S-ben lehetnek ismétlődések, akkor egy δ műveletet is be kell iktatni az S-nek megfelelő kifejezés gyökerénél, hogy a kialakuló kifejezés ne állítson elő több sort, mint az eredeti lekérdezés.
  b) A kétargumentumú kiválasztást helyettesítsük egy σ_C egyargumentumú kiválasztással, ahol C az a feltétel, amelyet úgy kapunk, hogy a t sor minden egyes komponensét egyenlővé tesszük az S reláció neki megfelelő attribútumával.
  c) A σ_C argumentumaként az R és S szorzatát adjuk meg.

A 7.15. ábra szemlélteti ezt a transzformációt.



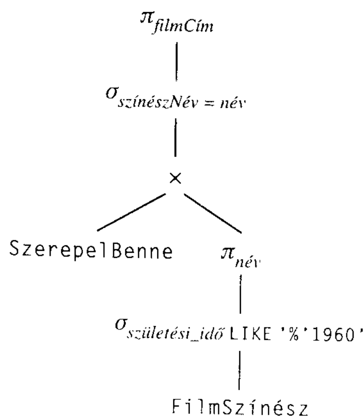
7.15. ábra. IN*-t tartalmazó feltétellel rendelkező kétargumentumú kiválasztást kezelő szabály*

7.18. példa: Vegyük a 7.14. ábrán található fát, és alkalmazzuk rá az `IN` feltételekhez megadott fenti szabályt. Ezen az ábrán az R a `SzerepelBenne` reláció, az S reláció pedig annak a relációs algebrai kifejezésnek az eredménye, amely a $\pi_{név}$ gyökerű részfából áll. A t sornak egy komponense van, nevezetesen a `színészNév` attribútum.

A kétargumentumú kiválasztás helyettesítője $\sigma_{színészNév\ =\ név}$, amelynek a C feltétele a t egyetlen tagját egyenlővé teszi az S lekérdezés eredményének attribútumával. A σ csomópont gyereke egy $\times$ csomópont, és az $\times$ csomópont argumentumai a `SzerepelBenne` címkéjű csomópont és az S-hez tartozó kifejezés gyökere. Mivel a `név` kulcsa a `FilmSzínész` relációnak, láthatjuk, hogy nincs szükség arra, hogy az S-hez tartozó kifejezésben egy ismétlődéseket megszüntető δ operátort bevezessünk. A 7.16. ábra mutatja az új kifejezést, amely teljesen a relációs algebrában van kifejezve, és ekvivalens a 7.13. ábra kifejezésével, habár a szerkezete igencsak különböző. □

Összetettebb az alkérdések relációs algebrába történő átfordítása, ha az alkérdés korrelatív. Mivel a korrelatív alkérdések magukban foglalnak rajtuk kívül definiált ismeretlen értékeket is, nem lehet őket külön átfordítani. Ehelyett az alkérdést úgy transzformáljuk, hogy az egy olyan relációt állít elő, amelyben bizonyos extra attribútumok is megjelennek – attribútumok, amelyeket később a kívül definiált attribútumokkal kell majd összehasonlítani. Az alkérdés attribútumait a külső attribútumokkal összevető feltételt erre a relációra alkalmazzuk, és az ezután már feleslegessé vált extra attribútumokat vetítés segítségével elhagyhatjuk. E folyamat során oda kell figyelnünk az ismétlődő sorok esetleges bevezetésére, amennyiben a lekérdezés a végén nem távolítja el az ismétlődéseket. A következő példa szemlélteti ezt a technikát.

7.19. példa: Használjuk a 7.1. példában bevezetett relációkat, és vegyük az alábbi lekérdezést: „Keressük meg az olyan filmeket, ahol a színészek átlagéletkora legfeljebb 40 év volt, amikor a film készült." A lekérdezés SQL-megfelelőjét a 7.17. ábra mutatja. Az egyszerűség kedvéért a `születési_idő`-t születési évnek vesszük, így vehetjük azok átlagát, amit aztán a `SzerepelBenne` reláció `év` attribútumával össze tudunk hasonlítani. A lekérdezést úgy fogalmaztuk meg, hogy mindhárom hivatkozott reláció rendelkezik a maga saját sorváltozójával, mutatva, hogy a különböző attribútumok honnan származnak.



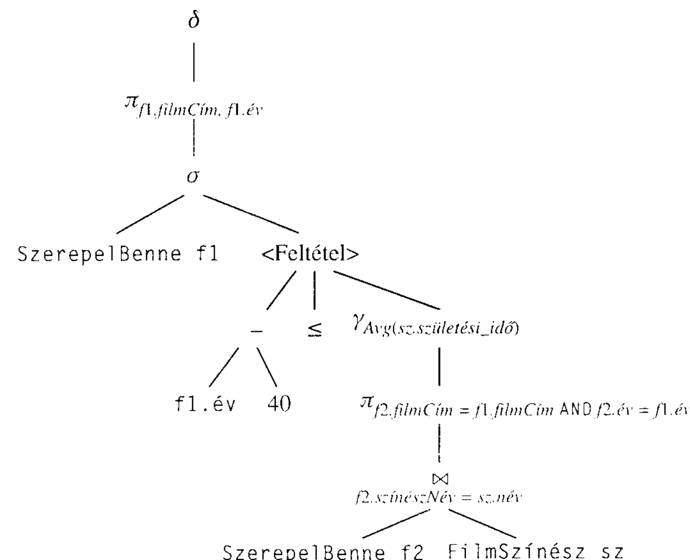
7.16. ábra. *Az* `IN` *feltételre vonatkozó szabály alkalmazása*


```
SELECT DISTINCT f1.filmCím, f1.év
FROM SzerepelBenne f1
WHERE f1.év - 40 <= (
   SELECT AVG(születési_idő)
   FROM SzerepelBenne f2, FilmSzínész sz
   WHERE f2.színészNév = sz.név AND
      f1.filmCím = f2.filmCím AND
      f1.év = f2.év
   );
```

7.17. ábra. *Bizonyos átlagéletkorú színészekkel készült filmek megkeresése*

A 7.18. ábrán a lekérdezés elemzésének és a relációs algebrába való részleges átfordítás végrehajtásának az eredménye látható. Ebben a kezdeti transzformációban ketté választottuk az alkérdés `WHERE` záradékát, és az egyik részt úgy használtuk, hogy relációk szorzatából összekapcsolást készítettünk. Az `f1`, `f2` és `sz` sorváltozó neveket megtartottuk a fában is, hogy világos legyen az egyes attribútumok eredete. Megtehettük volna azt is, hogy vetítések segítségével átnevezzük az attribútumokat, de így az eredmény nehezebben lenne követhető.

A <Feltétel> csomópont és a kétargumentumú kiválasztás eltávolításához szükség

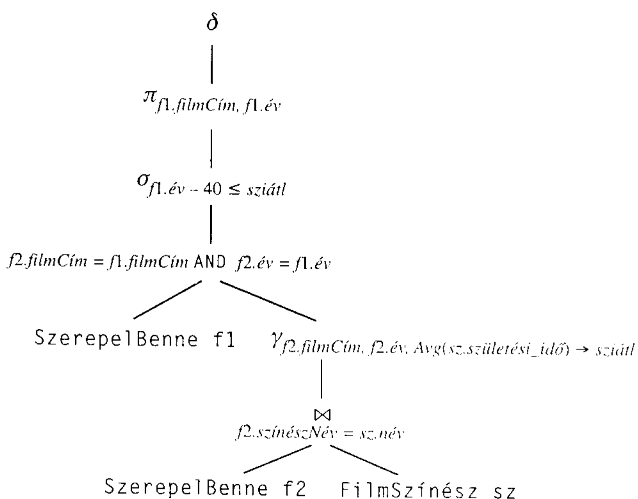


7.18. ábra. *Részlegesen transzformált elemzőfa a 7.17. ábra lekérdezéséhez*

van egy olyan kifejezésre, amely a <Feltétel> jobb oldali ágához tartozó relációt definiálja. Az alkérdés azonban korrelatív, és az `f1.filmCím` és `f1.év` attribútumok nem szerezhetők meg az alkérdésben említett relációkból, amelyek az `f2` sorváltozójú `SzerepelBenne` és a `FilmSzínész`. Emiatt a $\sigma_{f2.filmCím\ =\ f1.filCím\ \mathrm{AND}\ f2.év\ =\ f1.év}$ kiválasztást akkorra kell elhalasztani, amikor az alkérdés relációját már kombináltuk a `SzerepelBenne` relációnak a lekérdezés külső szintjén megjelenő példányával (az `f1` sorváltozójú példánnyal). A logikai lekérdezésterv ilyen átalakításához a γ operátort módosítani kell, mégpedig úgy, hogy a csoportosítás az `f2.filmCím` és `f2.év` attribútumok szerint történjen, így lesznek ugyanis elérhetők ezek az attribútumok a kiválasztáskor. Ennek hatásaként az alkérdéshez egy filmekből álló relációt számolunk ki, ahol minden egyes filmet annak címe és éve, valamint a filmben szereplő színészek születési évének átlaga képvisel.

A módosított γ operátor a 7.19. ábrán látható. A két csoportosítási attribútum bevezetésén túl az átlagot is átneveztük `sziátl`-ra (születési idők átlaga), hogy később hivatkozhassunk rá. A 7.19. ábra mutatja a relációs algebrába történő teljes átfordítást is. A γ fölött a külső lekérdezésből származó `SzerepelBenne` relációnak és az alkérdés eredményének összekapcsolása szerepel. Az alkérdésben lévő kiválasztást a `SzerepelBenne` relációnak és alkérdés relációjának szorzatára lehet alkalmazni, amit mi már egy théta-összekapcsolásként jelenítettünk meg, amivé ténylegesen azzá válna az algebrai szabályok alkalmazása után. A théta-összekapcsolás fölött egy további kiválasztás szerepel, ami a külső lekérdezés kiválasztásának felel meg, ahol a filmek gyártási évét hasonlítjuk össze a színészek születési évének átlagával. Az algebrai kifejezés a fa tetején úgy végződik, mint a 7.18. ábra kifejezése, vagyis a kívánt attribútumokra történő vetítéssel és az ismétlődések eltávolításával.

A 7.3.3. részben látni fogjuk, hogy egy lekérdezésoptimalizáló sokkal többet is tehet a lekérdezésterv javítása érdekében. A jelenlegi konkrét példánkban teljesül három

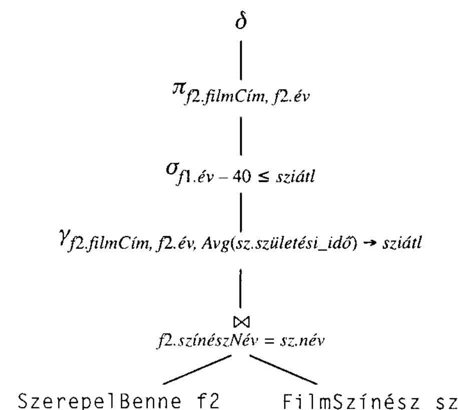


7.19. ábra. *A 7.18. ábra átalakítása egy logikai lekérdezéstervvé*

feltétel, amelyek lehetővé teszik, hogy a terven jelentősen javítsunk. Ezek a feltételek a következők:

1. Az ismétlődések megszüntetése a végén történik.
2. A vetítés a `SzerepelBenne f1` relációból kihagyja a színészek neveit.
3. A `SzerepelBenne f1` és a maradék kifejezés közti összekapcsolás egyenlővé teszi a `SzerepelBenne f1` és `SzerepelBenne f2` relációk `filmCím` és `év` attribútumait.

Mivel ezek a feltételek teljesülnek, az `f1.filmCím` és `f1.év` összes előfordulását helyettesíthetjük `f2.filmCím`-mel, illetve `f2.év`-vel. A 7.19. ábra felső összekapcsolása ezáltal feleslegessé válik, csakúgy mint a `SzerepelBenne f1` argumentum. Az így előálló logikai lekérdezésterv a 7.20. ábrán látható. □



7.20. ábra. *A 7.19. ábra egyszerűsítése*

7.3.3. Logikai lekérdezéstervek javítása

Amikor egy lekérdezést relációs algebrába átfordítunk, egy lehetséges logikai lekérdezéstervet kapunk. Ezután az következik, hogy a 7.2. részben felvázolt algebrai szabályok segítségével átírjuk a tervet. Egy másik megközelítés az lehetne, hogy több lekérdezéstervet generálunk, amelyek az operátorok különböző sorrendjének vagy kombinációjának felelnek meg. Ebben a könyvben abból a feltételezésből indulunk ki, hogy a lekérdezésátíró egyetlen logikai lekérdezéstervet választ ki, amelyet a „legjobbnak" vél, ami azt jelenti, hogy végül a legolcsóbb fizikai tervet eredményezi.

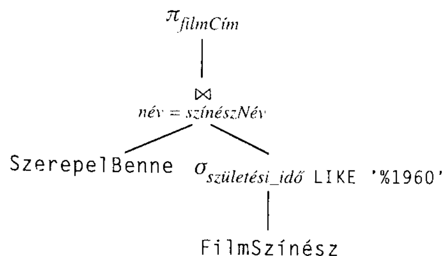
Nyitva hagyjuk viszont az „összekapcsolási sorrend" kérdését, így egy relációk összekapcsolását tartalmazó logikai lekérdezésterv úgy tekinthető, mint tervek egy családja, ami megfelel azoknak a különböző lehetőségeknek, ahogyan egy összekapcsolás sorba rendezhető és csoportosítható. Az összekapcsolási sorrend megválasztását a 7.6. részben tárgyaljuk. Ehhez hasonlóan, ha egy lekérdezésterv három vagy több

relációt tartalmaz a többi asszociatív és kommutatív operátor – mint például az egyesítés – argumentumaiként, akkor feltételezésünk szerint a logikai terv fizikai tervvé történő konvertálásakor átrendezés és átcsoportosítás megengedett. A sorrendet és a fizikai terv kiválasztását taglaló kérdéseket a 7.4. részben kezdjük tárgyalni.

A 7.2. részben számos olyan algebrai szabály szerepelt, amelyek feltehetően javítják a logikai tervet. Az optimalizálókban leggyakrabban használtak a következők:

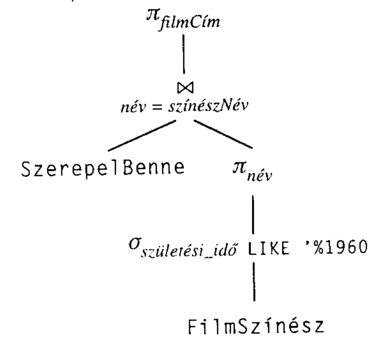
- A kiválasztásokat mindaddig tologatjuk lefelé a fában, ameddig csak mehetnek. Ha egy kiválasztási feltétel több feltétel ÉS-elése, akkor a feltételt szétvághatjuk, és az egyes darabokat külön-külön vihetjük le a fában. Ez a stratégia valószínűleg a leghatékonyabb javítási technika, de nem árt felidézni a 7.2.3. részben mondottakat, ahol azt láttuk, hogy bizonyos körülmények között a kiválasztást először a fa tetejére kellett felvinni.
- Hasonlóképpen, a vetítéseket is tologathatjuk lefelé a fában, vagy új vetítéseket adhatunk hozzá. Csakúgy mint a kiválasztások esetében, a vetítések tologatásával is óvatosan kell bánni, ahogy ezt a 7.2.4. részben elmondtuk.
- Az ismétlődések megszüntetése néha eltávolítható vagy áthelyezhető alkalmasabb helyre a fában, a 7.2.6. részben mondottaknak megfelelően.
- Bizonyos kiválasztások kombinálhatók egy alatta elhelyezkedő szorzattal úgy, hogy a műveletpár egy egyenlőségen alapuló összekapcsolássá (equijoin) alakul, amit általában sokkal hatékonyabban lehet kiértékelni, mint a két műveletet külön-külön. Ezeket a szabályokat a 7.2.5. részben tárgyaltuk.

7.20. példa: Vegyük a 7.13. ábra lekérdezését. Először a kiválasztást vágjuk ketté a $\sigma_{színészNév = név}$ és a $\sigma_{születési_idő}$ `LIKE '1960'` operátorokra. Az utóbbi levihető a fában, mivel az egyetlen érintett attribútum (`születési_idő`) a `FilmSzínész` relációból származik. Az első feltétel a szorzat mindkét tagjából tartalmaz egy-egy attribútumot, de azok egyenlővé vannak téve, ezért a szorzat és a kiválasztás együtt valójában egy összekapcsolásnak felel meg. Az átalakítások eredményét a 7.21. ábra mutatja. □



7.21. ábra. *Egy lekérdezés átírásának eredménye*

7.21. példa: A 7.16. ábrán szereplő kifejezésfán szintén lehet javítani. Hasznos transzformációt azonban csak a 7.20. példában is említett szabályok egyike jelent: egy kiválasztás és az alatta elhelyezkedő szorzat helyettesítése egy egyenlőségen alapuló összekapcsolással. A kapott lekérdezésterv a 7.22. ábrán látható, és ez majdnem ugyanaz, mint a 7.21. ábra, de van benne egy további vetítés a `név` attribútumra vonatkozóan. Amikor az 1960-ban született színészek megkeresésére végrehajtunk egy kiválasztást a `FilmSzínész` reláción, elég, ha csak a `név` komponenst állítjuk elő, mert ez az, amit a későbbi műveletekben használunk. Vegyük észre, hogy a 7.22. ábra tervét a 7.21. ábra tervéből is megkaphatjuk úgy, hogy a vetítést bevisszük a fa jobb ágába (mialatt a $\pi_{filmCím}$ vetítést meghagyjuk a gyökérben). Ugyanakkor viszont a `SzerepelBenne` tárolt reláció vetítése költséges lehet, ha emiatt nem tudunk használni egy indexet a `SzerepelBenne` azon sorainak elérésekor, amelyekre az összekapcsolásnál szükség van. □



7.22. ábra. *A 7.16. ábra egy javítása*

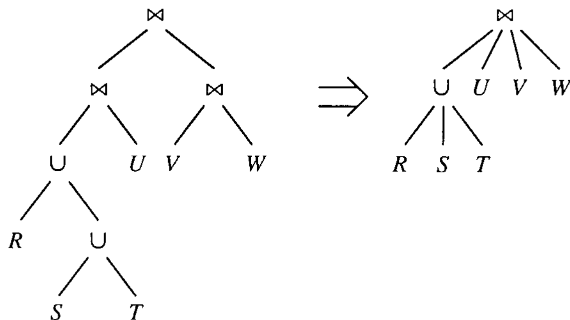
7.3.4. Asszociatív/kommutatív operátorok csoportosítása

A hagyományos elemzők nem állítanak elő olyan fákat, amelyek csomópontjai korlátlan számú gyerekkel rendelkezhetnek. Így az a normális, hogy az operátorok csak unáris vagy bináris formájukban jelennek meg. Asszociatív és kommutatív operátorok azonban felfoghatók úgy, mint amelyeknek tetszőleges számú operandusa van. Sőt, ha egy operátort, mint például az összekapcsolást, úgy tekintjük, mint egy sokoperandusú operátort, akkor lehetőséget kapunk az operandusok sorrendjének átrendezésére. Ez azt eredményezheti, hogy az új sorrendnek megfelelő bináris összekapcsolások sorozata kevesebb idő alatt hajtható végre, mint ha az összekapcsolásokat az elemzőfa által meghatározott sorrendben végeznénk el. A sokoperandusú összekapcsolások rendezését a 7.6. részben tárgyaljuk.

A végső logikai lekérdezésterv előállítása előtt tehát végrehajtunk egy utolsó lépést: ha van egy részfa, amelynek csomópontjaiban ugyanaz az asszociatív és kommutatív operátor szerepel, akkor az azonos operátort tartalmazó csomópontokat egyetlen sok gyerekkel rendelkező csomópontba csoportosítjuk. Emlékezzünk vissza, hogy a szokásos asszociatív/kommutatív operátorok a természetes összekapcsolás, egyesítés és metszet. Természetes összekapcsolások és théta-összekapcsolások is egyesíthetők egymással bizonyos körülmények között:

1. A természetes összekapcsolásokat olyan théta-összekapcsolásokkal kell helyettesíteni, amelyek egyenlővé teszik az azonos nevű attribútumokat.
2. Be kell iktatni egy vetítést az olyan attribútumok ismételt példányainak eltávolítására, amelyek a théta-összekapcsolássá vált természetes összekapcsolásban érintettek.
3. A théta-összekapcsolás feltételeinek asszociatívnak kell lenni. Emlékezzünk a 7.2.1. részben tárgyalt esetekre, ahol a théta-összekapcsolások nem asszociatívak.

Továbbá, a szorzatokat a természetes összekapcsolás speciális eseteként is felfoghatjuk, és egyesíthetjük azokat összekapcsolásokkal, ha a fában egymás szomszédjaiként helyezkednek el. A 7.23. ábra szemlélteti ezt a transzformációt, egy olyan helyzetben, ahol a logikai lekérdezéstervben egy két egyesítésből álló nyaláb, valamint egy három összekapcsolásból álló nyaláb szerepel. A betűk R-től W-ig kifejezéseket jelölnek, nem feltétlenül tárolt relációkat.



7.23. ábra. *A logikai lekérdezésterv előállításának utolsó lépése: asszociatív és kommutatív operátorok csoportosítása*

7.3.5. Feladatok

7.3.1. feladat: A következő kifejezésekben helyettesítsük a természetes összekapcsolásokat ekvivalens théta-összekapcsolásokkal és vetítésekkel. Döntsük el, hogy az eredményül kapott théta-összekapcsolások egy kommutatív és asszociatív csoportot alkotnak-e.

* a) $(R(a,b) \bowtie S(b,c)) \underset{S.c>T.c}{\bowtie} T(c,d)$

b) $(R(a,b) \bowtie S(b,c)) \bowtie (T(c,d) \bowtie U(d,e))$

c) $(R(a,b) \bowtie S(b,c)) \bowtie (T(c,d) \bowtie U(a,d))$

7.3.2. feladat: Konvertáljuk a 7.1.3.a) és b) feladatok elemzőfáit relációs algebrába. A b)-nél adjuk meg a kétargumentumú kiválasztást használó alakot, valamint annak egyargumentumú (szokásosan σ_C) kiválasztássá átalakított változatát.

! **7.3.3. feladat:** Adjunk egy-egy szabályt a következő alakú <Feltétel>-ek relációs algebrába történő átfordítására. Mindegyik feltételről feltesszük, hogy egy R relációra alkalmazzuk (egy kétargumentumú kiválasztás által). Feltételezhető továbbá, hogy az alkérdés nem korrelatív az R viszonylatában. Figyeljünk arra, hogy ne vezessünk be és ne szüntessünk meg ismétlődéseket az SQL hivatalos definíciójával szembenálló módon.

* a) `EXISTS`(<Lekérdezés>) alakú feltétel.

b) a = `ANY` <Lekérdezés> alakú feltétel, ahol a az R egy attribútuma.

c) a = `ALL` <Lekérdezés> alakú feltétel, ahol a az R egy attribútuma.

!! **7.3.4. feladat:** Tekintsük újra a 7.3.3. feladatot, megengedve ezúttal, hogy az alkérdés korrelatív legyen R-rel. Az egyszerűség kedvéért feltételezhetjük, hogy az alkérdés egy egyszerű „select-from-where" kifejezés, amely nem tartalmaz további alkérdéseket.

!! **7.3.5. feladat:** Hány különböző kifejezésfából adódhat a 7.23. ábra jobb oldali csoportosított fája? Emlékezzünk, hogy a csoportosítás után a gyerekek sorrendje nem feltétlenül tükrözi az eredeti kifejezésfában adott sorrendiséget.

7.4. Műveletek költségének becslése

Tegyük fel, hogy elemeztünk egy lekérdezést, és átalakítottuk egy logikai lekérdezéstervvé. Tegyük fel továbbá, hogy elvégeztük a kiválasztott transzformációkat, és megkaptuk a legjobbnak vélt logikai lekérdezéstervet. Következő lépésként a logikai tervet kell fizikai tervvé alakítani. Ez általában úgy történik, hogy sok különböző fizikai tervet tekintünk, amelyek a logikai tervből származnak, és kiértékeljük vagy becsüljük az ezekhez tartozó költségeket. E kiértékelés után, amit *költségalapú felsorolásnak* nevezünk, kiemeljük a legkisebb költségű fizikai tervet, és azt adjuk tovább a lekérdezés-végrehajtó motornak. Amikor az egy adott logikai lekérdezéstervből levezethető lehetséges fizikai terveket felsoroljuk, az egyes fizikai tervekhez az alábbiakat is kiválasztjuk:

1. Sorrendiség és csoportosítás az asszociatív-kommutatív operátorokra vonatkozóan, mint az összekapcsolás, egyesítés és metszet.
2. Algoritmus a logikai tervben szereplő minden egyes operátorhoz. Például, hogy beágyazott ciklusú összekapcsolást vagy tördelő összekapcsolást használjunk-e.
3. További műveletek – beolvasás, rendezés stb. –, amelyek a fizikai tervhez szükségesek, de amelyek a logikai tervben explicit módon nem voltak jelen.
4. Annak módja, ahogy egy operátor továbbadja az argumentumokat egy másiknak. Például, a közbülső eredmények lemezen történő tárolásával vagy iterátorokat használva, központi memóriapufferként továbbadva az argumentumot.

A továbbiakban megvizsgáljuk mindezeket a kérdéseket. Annak érdekében azonban, hogy az ezekkel a választásokkal kapcsolatban felmerülő kérdéseket megválaszolhassuk, meg kell értenünk, hogy a különböző fizikai tervek költsége mit is jelent. Egy terv pontos költségét nem tudhatjuk meg a terv végrehajtása nélkül, és egy lekérdezéshez nyilván nem akarunk egynél több tervet végrehajtani. Így a tervek költségének becslésére kényszerülünk, anélkül hogy azokat végrehajtanánk.

Mielőtt elkezdjük a fizikai tervek felsorolásának tárgyalását, az ilyen tervek költségbecslésének mikéntjére kell kitérni. Ezek a becslések az adatokkal kapcsolatos paraméterekre épülnek (lásd a „Jelölések áttekintése" c. részt), amelyeket vagy pontosan kiszámítunk az adatokból, vagy a „statisztikai gyűjtés" eljárással becsülünk, amit a 7.5.1. részben ismertetünk. Ha adottak a paramétereknek az értékei, akkor számos elfogadható becslés adható a relációméretekkel kapcsolatban, amelyekkel aztán egy teljes fizikai terv költsége becsülhető.

7.4.1. Közbülső relációk méretének becslése

A fizikai tervet úgy választjuk ki, hogy a lekérdezés kiértékelésének költsége minimális legyen. A legfőbb költségtényező rendszerint a lemez I/O-művelet (input/output = olvasás/írás), de néha fontos a processzoridő és – ha a lekérdezést párhuzamos gépen vagy több egymással összekötött gépen értékeljük ki – a kommunikációs idő is.

Amikor a logikai terv kifejezése több operátort tartalmaz, bizonyos dolgokat tudunk arról, hogy a közbülső relációk hogyan lesznek ábrázolva. Amíg ugyanis a kifejezés argumentumaiként szolgáló tárolt relációk többféleképpen lehetnek tárolva – nyalábolten vagy nem, indexelve vagy anélkül –, a lekérdezés végrehajtása közben kiszámított valamely reláció, amelyet lemezen tárolunk, tárolható nyalábolten úgy, hogy minél kevesebb blokkot foglaljon el. Továbbá, egy ilyen relációnak nem lesznek indexei, hacsak nem definiáljuk azokat explicit módon a fizikai lekérdezésterv részeként.

Ezek után azt mondhatjuk, hogy a köztes relációk kezeléséhez szükséges lemez I/O-műveletek száma nem függ mástól, mint a relációk méretétől. Ezt pedig úgy kapjuk meg, hogy a közbülső reláció sorainak számát megszorozzuk a sor tárolásához szükséges bájtok számával. Egy sor által elfoglalt bájtok száma levezethető a közbülső reláció attribútumaiból és azok típusaiból, így csak az marad rejtély, hogy a köztes reláció hány sort tartalmaz. Mivel általában nem tudjuk pontosan megmondani, hogy egy köztes relációnak hány sora lesz, néhány ésszerű szabályt fogunk bevezetni ezeknek a méreteknek a becslésére.

Ideális esetben egy közbülső relációban szereplő sorok számát becslő szabályokra igazak az alábbiak:

1. Elég pontos becslést adnak.
2. Könnyű kiszámolni.
3. Logikailag konzisztensek, azaz egy közbülső reláció méretére vonatkozó becslés nem függ a reláció kiszámításának módjától. Például több reláció összekapcsolására vonatkozó becslés nem függ a relációk összekapcsolásának sorrendjétől.

Jelölések áttekintése

Elevenítsük fel a 6.2.3. részben a relációk méretének jelölésére használt konvenciókat:

- $B(R)$ jelöli az R reláció összes sorának tárolásához szükséges blokkok számát.
- $T(R)$ az R reláció sorainak számát jelöli.
- $V(R, a)$ az R reláció a attribútumához tartozó *értékszámlálót* jelenti, vagyis azoknak a különböző értékeknek a számát, amelyek az R relációban az a attribútum értékeként előfordulnak. Valamint $V(R,[a_1, a_2, \ldots, a_n])$ jelöli azoknak a különböző értékeknek (értékkombinációknak) a számát, amelyek előfordulnak R-ben, amikor az $a_1, a_2, \ldots, a_n$ attribútumokat együtt tekintjük, azaz a $\pi_{a_1, a_2, \ldots, a_n}(R)$-ben szereplő különböző sorok számát jelenti.

Nincs általános egyetértés e feltételek teljesítésére vonatkozóan. Mi bemutatunk néhány egyszerű szabályt, amelyek a legtöbb helyzetben megfelelők. Szerencsére a méret becslésének nem az a célja, hogy a pontos méretet előre kiszámítsuk, hanem az, hogy hozzájáruljon egy fizikai terv kiválasztásához. Még egy pontatlan méretbecslési módszer is szolgálhat erre a célra, ha konzisztens módon hibázik, azaz, ha a méretbecslő a legjobb fizikai tervhez rendeli a legkisebb költséget, még ha annak a tervnek a tényleges költségéről az derül is ki, hogy különbözik az előre kiszámítottól.

7.4.2. Vetítés méretének becslése

A vetítés abban különbözik a többi művelettől, hogy az eredményének a mérete kiszámítható. Mivel egy vetítés minden argumentumsorhoz előállít egy eredménysort, a kimenet méretének változása csak a sorok hosszának megváltozásában jelentkezik. Emlékezzünk, hogy az itt használt vetítés operátor egy multihalmaz operátor, és nem távolítja el az ismétlődéseket. Ha egy vetítés során előálló ismétlődéseket meg akarjuk szüntetni, akkor a δ operátort kell utána alkalmazni.

Normális esetben vetítéskor a sorok összezsugorodnak, hiszen bizonyos komponenseket elhagyunk. A vetítésnek a 6.1.3. részben bevezetett általános formája azonban megengedi új komponensek létrehozását, mint attribútumok kombinációit. Vannak tehát esetek, amikor egy π operátor növeli a reláció méretét.

7.22. példa: Tegyük fel, hogy $R(a, b, c)$ egy reláció, ahol a és b négybájtos egészek, c pedig 100 bájtos karakterlánc. Mondjuk, hogy a sor fejlécek 12 bájtot igényelnek. Ekkor az R minden egyes sorának 120 bájtra van szüksége. Legyenek a blokkok 1024 bájt hosszúak, 24 bájt blokkfejlécekkel. Egyetlen blokkban így 8 sor fér el. Tegyük fel, hogy $T(R) = 10\,000$, vagyis hogy R 10 000 sort tartalmaz. Ekkor $B(R) = 1250$.

Legyen $S = \pi_{a+b,\,c}(R)$, azaz a-t és b-t az összegükkel helyettesítjük. Az S sorai 116

bájtot igényelnek: 12-t a fejlécnek, 4-et az összegnek és 100-at a karaktersorozatnak. Habár az S sorai valamivel kisebbek, mint az R sorai, még mindig csak 8 sort helyezhetünk be egy blokkba. Tehát: $T(S) = 10\,000$ és $B(S) = 1250$.

Legyen most $U = \pi_{a,\,b}(R)$, amikor is a karakterlánc komponenst elhagyjuk. Az U sorai csak 20 bájt hosszúak. $T(U)$ még mindig 10 000. Most azonban az U-nak 50 sorát pakolhatjuk egy blokkba, vagyis $B(U) = 200$. Ez a vetítés tehát a relációt mintegy 6-od részére zsugorítja. □

7.4.3. Kiválasztás méretének becslése

Amikor egy kiválasztást hajtunk végre, általában csökkentjük a sorok számát, de a sorok mérete ugyanaz marad. A kiválasztás legegyszerűbb esetében, amikor egy attribútumnak egy konstanssal való egyenlőségét vizsgáljuk, létezik egy könnyű módszer az eredmény méretének becslésére, feltéve, hogy tudjuk (vagy becsülni tudjuk) az attribútum által felvett különböző értékek számát. Legyen $S = \sigma_{A\,=\,c}(R)$, ahol A az R egy attribútuma és c egy konstans. Ekkor a következő becslést javasoljuk:

- $T(S) = T(R)/V(R, A)$

Ez a szabály biztosan igaz akkor, ha az A attribútum minden értéke egyenlő gyakorisággal fordul elő az adatbázisban. A fenti szabály azonban – az „A Zipfian-eloszlás" c. bekeretezett részben mondottaknak megfelelően – még akkor is a legjobb becslése az átlagnak, ha az A értékei nem mutatnak egyenletes eloszlást az adatbázisban. Elvárjuk viszont, hogy az A minden értéke egyforma valószínűséggel szerepeljen az A értékét meghatározó lekérdezésekben.

Problematikusabb a méret becslése, amikor a kiválasztás egyenlőtlenség-összehasonlítást tartalmaz, például ha $S = \sigma_{a\,<\,10}(R)$. Azt gondolhatnánk, hogy az átlag tekintetében a sorok fele megfelelne az összehasonlításnak, a sorok fele nem, így $T(R)/2$ jó becslése lenne az S méretének. Egy érzés azonban azt súgja, hogy egy ilyen lekérdezés a lehetséges soroknak inkább csak egy kisebb hányadát adná vissza.[3] Egy olyan szabályt javasolunk, amely figyelembe veszi ezt a tendenciát, és azzal a feltételezéssel él, hogy egy tipikus vizsgálat, amely az egyenlőtlenséget vizsgálja, körülbelül a sorok egyharmadát adja vissza, nem a felét. Ha $S = \sigma_{a\,<\,c}(R)$, akkor $T(S)$-re a becslésünk:

- $T(S) = T(R)/3$

A „nem egyenlő" összehasonlítások ritkák. Ha azonban egy olyan kiválasztással találkozunk, mint például az $S = \sigma_{a\,\neq\,10}(R)$, akkor javasoljuk annak feltételezését, hogy lényegében minden sor kielégíti majd ezt a feltétel. Vehetjük tehát becslésként a következőt: $T(S) = T(R)$. Egy másik becslés lehet a $T(S) = T(R)(V(R, a) - 1)/V(R, a)$,

[3] Ha például fizetésekről lennének adataink, azt kérdeznénk-e meg nagyobb valószínűséggel, hogy a fizetés *kisebb*, mint 500 000 Ft, vagy azt, hogy *nagyobb*, mint 500 000 Ft?

ami valamivel kevesebbet ad. Ez a megközelítés elismeri, hogy az R sorainak körülbelül $1/V(R, a)$ része elbukik a feltételen, mert azok a értéke egyenlő a konstanssal.

Amikor egy C kiválasztási feltétel több ÉS-sel összekötött egyenlőségvizsgálat vagy más összehasonlítás, akkor a $\sigma_C(R)$ kiválasztást úgy tekinthetjük, mint azoknak az egyszerű kiválasztásoknak egymás utáni alkalmazását, amelyek mindegyike a feltétel egy-egy részét ellenőrzi. Vegyük észre, hogy ezeknek a kiválasztásoknak a sorrendje nem számít. Ennek hatásaként az eredmény méretére vonatkozó becslés az lesz, hogy az eredeti reláció méretét megszorozzuk az egyes feltételekhez tartozó *szelektivitási* tényezőkkel. Ez a tényező 1/3 egyenlőtlenség esetén, 1 ≠ esetén, illetve $1/V(R, A)$ amikor a C feltételben egy A attribútumot hasonlítunk egy konstanshoz.

7.23. példa: Legyen $R(a, b, c)$ egy reláció és $S = \sigma_{a\,=\,10 \text{ AND } b<20}(R)$. Legyen továbbá $T(R) = 10\,000$ és $V(R, a) = 50$. Ekkor a legjobb becslés a $T(S)$-re: $T(R)/(50 \times 3)$, azaz 67. Vagyis az R sorainak az 1/50 része éli túl az $a = 10$ szűrőt, és az 1/3 része éli túl a $b < 20$ szűrőt.

Egy érdekes speciális eset, ami romba dönti az analízisünket, amikor a feltétel ellentmondásos. Nézzük például az $S = \sigma_{a\,=\,10 \text{ AND } a\,>\,20}(R)$ kiválasztást. Ekkor a szabályunk alapján $T(S) = T(R)/3V(R, a)$, azaz 67 sor. Ugyanakkor viszont világos, hogy egyetlen sorra sem teljesülhet az $a = 10$ és az $a > 20$ feltételek mindegyike, tehát a helyes válasz: $T(S) = 0$. A logikai lekérdezésterv átírásakor a lekérdezésoptimalizáló sok speciális esetre vonatkozó szabályt figyelembe tud venni. A fenti esetben az optimalizáló alkalmazhat egy olyan szabályt, amely a kiválasztási feltételt HAMIS-nak találja, és az S-nek megfelelő kifejezést az üres halmazzal helyettesíti. □

Amikor egy kiválasztás VAGY-gyal kapcsolt feltételeket tartalmaz, mondjuk $S = \sigma_{C_1 \text{ OR } C_2}(R)$, kevesebb bizonyosságunk van az eredmény méretét illetően. Egy egyszerű feltételezés az, hogy egyetlen sorra sem teljesül mindkét feltétel, vagyis az eredmény mérete egyenlő az egyes feltételeket kielégítő sorok számának összegével. Ez a becslés általában túlbecslést jelent, és néha valóban ahhoz az abszurd következtetéshez vezethet el minket, hogy az S-ben több sor van, mint az eredeti R relációban. Egy másik egyszerű megközelítés lehet, hogy vesszük a minimumát az R méretének, és annak, amit a C_1-et, illetve a C_2-t kielégítő sorok számának összegeként kapunk.

Egy kevésbé egyszerű, de feltehetően pontosabb becslést kapunk az

$$S = \sigma_{C_1 \text{ OR } C_2}(R)$$

méretére, ha feltesszük, hogy C_1 és C_2 függetlenek. Ekkor, ha R-nek n sora van, amelyek közül m_1-re teljesül a C_1, és m_2-re teljesül a C_2, akkor az S-ben megjelenő sorok számára a következő becslést adhatjuk:

$$n(1 - (1 - m_1/n)(1 - m_2/n))$$

Itt az $1 - m_1/n$ egyenlő a soroknak a C_1-et nem teljesítő a hányadával, $1 - m_2/n$ pedig a soroknak a C_2-et nem teljesítő a hányadát jelenti. Ezek szorzata a R sorainak azon hányadát adja, amelyek *nincsenek* benne az S-ben, és ezt a szorzatot 1-ből kivonva az S-ben szereplő hányadot kapjuk.

A Zipfian-eloszlás

Amikor feltételezzük, hogy az R reláció $V(R, a)$ sora közül egy fog kielégíteni egy $a = 10$ típusú feltételt, akkor azzal a hallgatólagos feltételezéssel élünk, hogy az a attribútum minden értéke egyforma valószínűséggel szerepel az R egy adott sorában. Azt is feltételezzük, hogy a 10 ezen értékek egyike, de ez egy ésszerű feltételezés, hiszen a legtöbbször olyan dolgokat keresünk egy adatbázisban, amelyek tényleg léteznek. Az a feltételezés azonban, hogy az értékek egyenletesen oszlanak el, többnyire nem tartható fenn, még megközelítőleg sem.

Sok attribútum olyan értékeket tartalmaz, amelyek *Zipfian-eloszlást* mutatnak, ahol az i-edik leggyakoribb érték gyakorisága az $1/\sqrt{i}$ -vel arányos. Ha például a leggyakoribb érték 1000-szer fordul elő, akkor a második leggyakoribb értéktől azt várjuk, hogy körülbelül $1000/\sqrt{2}$ -ször, azaz 707-szer szerepeljen, a harmadik leggyakoribb érték pedig körülbelül $1000/\sqrt{3}$ -szor, azaz 577-szer fordulna elő. Erről az eloszlásról az derült ki, hogy sokféle típusú adatnál fellelhető, jóllehet eredetileg az angol mondatokban előforduló szavak relatív gyakoriságának leírására használták. Az USA-ban például az államok népességei megközelítőleg a Zipfian-eloszlást követik, miszerint a második legnépesebb New York állam népessége körülbelül a 70%-a a legnépesebb Kalifornia állam népességének. Következésképpen, ha az `állam` egy amerikai embereket – mondjuk újság-előfizetőket – leíró reláció egy attribútuma lenne, akkor azt várnánk, hogy az `állam` értékei a Zipfian-eloszlásnak megfelelően oszlanak el, és nem egyenletesen.

Mindaddig, amíg a kiválasztási feltételben a konstanst véletlenszerűen választjuk meg, nem számít, hogy az érintett attribútum egyenletes, Zipfian- vagy más eloszlású-e, az eredmény halmaz *átlagos* mérete még mindig $T(R)/V(R, a)$ lesz. Ha azonban a konstansokat is Zipfian-eloszlásnak megfelelően választjuk, akkor azt várnánk, hogy a kiválasztott halmaz átlagos mérete valamivel nagyobb lesz, mint $T(R)/V(R, a)$.

7.24. példa: Tegyük fel, hogy az $R(a, b)$ relációnak $T(R) = 10\,000$ sora van, és legyen

$S = \sigma_{a\,=\,10\ \mathrm{OR}\ b\,<\,20}(R)$

Legyen $V(R, a) = 50$. Ekkor az $a = 10$ feltételt kielégítő sorok számát, ami $T(R)/V(R, a)$, 200-ra becsüljük. A $b < 20$ feltételt kielégítő sorok számát $T(R)/3$-ra, vagyis 3333-ra becsüljük.

Az S méretére vonatkozó legegyszerűbb becslés ezek összege, azaz 3533. Az $a = 10$ és $b < 20$ feltételek függetlenségére építő bonyolultabb becslés a

$10\,000(1 - (1 - 200/10\,000)(1 - 3333/10\,000))$

értéket adja, azaz 3466-ot. A két becslés között kicsi az eltérés, így nagyon valószínűtlen, hogy az egyik választása a másikkal szemben változást jelentene a legjobb fizikai terv kiválasztásában. □

Az utolsó operátor, amely egy kiválasztási feltételben szerepelhet: a `NOT`. Ha egy R relációnak n számú sora van, akkor a `NOT` C feltételt kielégítő sorok becsült számát úgy kapjuk meg, hogy n-ből kivonjuk a C-t kielégítő sorok becsült számát.

7.4.4. Összekapcsolás méretének becslése

Csak a természetes összekapcsolást fogjuk vizsgálni. A többi összekapcsolás az alábbi elveknek megfelelően kezelhető:

1. Egy egyenlőség alapú összekapcsolás (equijoin) eredményében megjelenő sorok száma, miután a változó nevekben bekövetkező változásokkal elszámoltunk, pontosan úgy számítható ki, mint természetes összekapcsolás esetén. Ezt a pontot a 7.26. példa fogja szemléltetni.
2. Más théta-összekapcsolások úgy becsülhetők, mintha szorzatot követő kiválasztások volnának, figyelembe véve a következő további megjegyzéseket:

   a) Egy szorzat sorainak számát úgy kapjuk, hogy a szorzatban részt vevő relációk sorainak számait összeszorozzuk.
   b) Egy egyenlőséget vizsgáló összehasonlítást a természetes összekapcsoláshoz kidolgozott technika segítségével becsülhetünk.
   c) Egy két attribútum egyenlőtlenségét vizsgáló $R.a < S.b$ típusú összehasonlítás úgy kezelhető, mint egy $R.a < 10$ alakú összehasonlítás, amit a 7.4.3. részben tárgyaltunk. Vagyis feltehetjük, hogy ennek a feltételnek a szelektivitási tényezője 1/3 (ha úgy gondoljuk, hogy a feltétel inkább ritkán teljesül), vagy lehet 1/2 (ha nem élünk a feltételezéssel).

Első körben tételezzük fel, hogy két reláció természetes összekapcsolása csak két attribútum egyenlőségét tartalmazza. Ez azt jelenti, hogy az $R(X, Y) \bowtie S(Y, Z)$ összekapcsolást vizsgáljuk, de kezdetben feltesszük, hogy Y egyetlen attribútum, az X és Z viszont tetszőleges attribútum halmazokat jelölhetnek.

Az a probléma, hogy nem tudjuk, hogy az R és S Y értékei milyen viszonyban állnak egymással. Például:

1. A két relációban az Y értékek lehetnek diszjunkt halmazok, amikor is az összekapcsolás üres és $T(R \bowtie S) = 0$.
2. Az Y lehet az S kulcsa és egy idegen kulcs az R-ben. Ilyenkor az R minden egyes sora pontosan egy S-beli sorral kapcsolódik, így tehát $T(R \bowtie S) = T(R)$.
3. Lehet, hogy az R és S majdnem minden sorának ugyanaz az Y értéke, ekkor $T(R \bowtie S)$ körülbelül $T(R)T(S)$ lesz.

A következő két egyszerűsítő feltételezéssel fogunk élni, hogy a leggyakoribb esetekre koncentrálhassunk:

1. *Értékhalmazok tartalmazása.* Ha Y egy több relációban is szereplő attribútum, akkor ez az attribútum mindegyik relációban egy $y_1, y_2, y_3, \ldots$ rögzített értéklistának az elejéről kap értéket, és az összes érték ebből a prefixből származik. Következésképpen, ha R és S két reláció, amelyek tartalmazzák az Y attribútumot, és $V(R, Y) \leq V(S, Y)$, akkor az R minden Y értéke az S-nek Y értéke lesz.
2. *Értékhalmazok megőrzése.* Ha egy R relációt összekapcsolunk egy másik relációval, akkor egy A attribútum, amely nem összekapcsolási attribútum (azaz nem szerepel mindkét relációban), nem veszít el értékeket az értékeinek a lehetséges halmazából. Pontosabban szólva, ha A az R-nek attribútuma, de S-nek nem, akkor $V(R \bowtie S, A) = V(R, A)$. Megjegyezzük, hogy az R és az S összekapcsolásának sorrendje nem lényeges, tehát azt is mondhattuk volna, hogy $V(S \bowtie R, A) = V(R, A)$.

Nyilván előfordulhat, hogy az 1. előfeltevés, értékhalmazok tartalmazása, nem érvényes, de teljesül akkor, ha Y kulcs az S-ben, és idegen kulcs az R-ben. Sok más esetben is megközelítőleg igaz, hiszen intuitíve azt várjuk, hogy ha S-nek sok Y értéke van, akkor egy adott R-ben előforduló Y érték jó eséllyel szerepel S-ben.

A 2. feltételezés, értékhalmazok megőrzése, szintén sérülhet, de igaz a feltevés akkor, ha az $R \bowtie S$ összekapcsolási attribútuma kulcs az S-ben, és idegen kulcs az R-ben. Valójában csak akkor fordulhat elő, hogy a 2. előfeltevés nem teljesül, ha az R-ben „lógó sorok” vannak, vagyis olyan sorok, amelyek az S egyetlen sorával sem kapcsolódnak, de még az ilyen esetekben is érvényes lehet az előfeltétel.

E feltételezések mellett az $R(X, Y) \bowtie S(Y, Z)$ mérete a következőképpen becsülhető. Legyen $V(R, Y) \leq V(S, Y)$. Ekkor $1/V(S, Y)$ az esélye annak, hogy az R egy t sora az S egy adott sorával kapcsolódik. Mivel az S-nek $T(S)$ sora van, azoknak a soroknak a várható száma, amelyekkel t kapcsolódik: $T(S)/V(S, Y)$. Minthogy az R-nek $T(R)$ sora van, az $R \bowtie S$ becsült mérete $T(R)T(S)/V(S, Y)$. Ha $V(R, Y) \geq V(S, Y)$, akkor a szimmetria alapján kapott becslés: $T(R \bowtie S) = T(R)T(S)/V(R, Y)$. Általában a $V(R, Y)$ és a $V(S, Y)$ közül a nagyobbal osztunk, tehát:

- $T(R \bowtie S) = T(R)T(S)/\max(V(R, Y), V(S, Y))$

7.25. példa: Tekintsük a következő három relációt és azok lényeges statisztikáit:

$R(a, b)$	$S(b, c)$	$U(c, d)$
$T(R) = 1000$	$T(S) = 2000$	$T(U) = 5000$
$V(R, b) = 20$	$V(S, b) = 50$	
	$V(S, c) = 100$	$V(U, c) = 500$

Tegyük fel, hogy az $R \bowtie S \bowtie U$ természetes összekapcsolást akarjuk kiszámítani. Ennek egy lehetséges csoportosítási módja $(R \bowtie S) \bowtie U$. Az általunk adott becslés a $T(R \bowtie S)$-re: $T(R)T(S)/\max(V(R, b), V(S, b))$, ami 1000 × 2000/50, azaz 40 000.

Ezután jön az $R \bowtie S$ összekapcsolása az U-val. Az eredmény méretére vonatkozó becslésünk a következő: $T(R \bowtie S)T(U)/\max(V(R \bowtie S, c), V(U, c))$. Az előfeltevésünk alapján, miszerint az értékhalmazok megőrződnek, $V(R \bowtie S, c)$ ugyanaz, mint $V(S, c)$-vel, azaz 100, vagyis az összekapcsolás során a c attribútum egyetlen értéke sem tűnik el. Ez esetben az $R \bowtie S \bowtie U$ eredményében lévő sorok számára vonatkozó becslésként a 40 000 × 5000/max(100, 500) értéket kapjuk, ami 400 000.

Az S és U összekapcsolásával is kezdhetnénk. Ekkor azt a becslést kapjuk, hogy $T(S \bowtie U) = T(S)T(U)/\max(V(S, c), V(U, c)) = 2000 \times 5000/500 = 20\,000$. Az előfeltevésünk alapján, miszerint az értékhalmazok megmaradnak, $V(S \bowtie U, b) = V(S, b) = 50$, így az eredmény méretének becslése $T(R)T(S \bowtie U)/\max(V(R, b), V(S \bowtie U, b))$, ami 1000 × 20 000/50, azaz 400 000. □

Nem véletlen, hogy a 7.25. példában a $R \bowtie S \bowtie U$ méretére ugyanazt a becslést kapjuk, függetlenül attól, hogy az $R \bowtie S$ vagy az $S \bowtie U$ összekapcsolással kezdjük. Idézzük fel, hogy a 7.4.1. rész egyik kívánalma az, hogy egy kifejezés eredményére vonatkozó becslés ne függjön a kiértékelés sorrendjétől. Belátható, hogy a fenti két előfeltevésünk – értékhalmazok tartalmazása és megőrzése – garantálja, hogy egy természetes összekapcsolásra vonatkozó becslés ugyanaz lesz, függetlenül az összekapcsolások végrehajtási sorrendjétől.

7.4.5. Természetes összekapcsolás több összekapcsolási attribútummal

Most nézzük meg, hogy mi történik akkor, amikor az $R(X, Y) \bowtie S(Y, Z)$ összekapcsolásban az Y több attribútumot jelöl. Tegyük fel, hogy az $R(x, y_1, y_2) \bowtie S(y_1, y_2, z)$ összekapcsolást akarjuk végrehajtani. Vegyük az R egy r sorát. Annak valószínűsége, hogy r az S egy adott s sorával kapcsolódik, a következőképpen számolható ki.

Először is, mi a valószínűsége annak, hogy r és s megegyeznek az y_1 attribútumon? Tegyük fel, hogy $V(R, y_1) \geq V(S, y_1)$. Ekkor, az értékhalmazok tartalmazásáról szóló előfeltevés alapján, az s sor y_1 értéke biztosan az R-ben előforduló y_1 értékek valamelyike. Így annak esélye, hogy r-nek ugyanaz az y_1 értéke, mint s-nek: $1/V(R, y_1)$. Hasonlóképpen, ha $V(R, y_1) < V(S, y_1)$, akkor az r sor y_1 értéke szerepelni fog S-ben, és $1/V(S, y_1)$ a valószínűsége annak, hogy az r és az s y_1-értéke ugyanaz lesz. Általánosan mondhatjuk, hogy $1/\max(V(R, y_1), V(S, y_1))$ az y_1 érték egyezésének valószínűsége.

Hasonló gondolatmenet alapján állíthatjuk, hogy $1/\max(V(R, y_2), V(S, y_2))$ annak a valószínűsége, hogy r és s megegyeznek az y_2 vonatkozásában. Mivel az y_1 és az y_2 értékei függetlenek, annak a valószínűsége, hogy a sorok mind az y_1, mind az y_2 attribútumon megegyeznek, e két tört szorzata lesz. Az R és S soraiból képzett $T(R)T(S)$ darab sorpár közül az y_1 és y_2 attribútumokon egyező párok száma tehát:

$$\frac{T(R)T(S)}{\max(V(R,y_1),V(S,y_1))\max(V(R,y_2),V(S,y_2))}$$

A sorok száma nem elég

Habár a relációk méreteire vonatkozó vizsgálódásaink során az eredményben szereplő sorok számára összpontosítottunk, az egyes sorok méretét is számításba kell venni. Relációk összekapcsolása például az eredeti relációkban előforduló soroknál nagyobb sorokat állít elő. Példának okáért, az $R \bowtie S$ összekapcsolás, ahol mindkét reláció 1000 sort tartalmaz, adhat olyan eredményt, amely szintén 1000 sorból áll. Az eredmény azonban több blokkot foglalna el, mint az R vagy az S.

A 7.26. példa egy érdekes eset ezzel kapcsolatban. Egy théta-összekapcsoláskor kapott sorok számának becslésére használhatunk ugyan természetes összekapcsolásra vonatkozó technikákat, ahogy ott tettük is, de egy théta-összekapcsolás több komponensből álló sorokat állít elő, mint a neki megfelelő természetes összekapcsolás. A konkrét példa esetében, az $R(a, b, c) \underset{R.b=S.d \text{ AND } R.c=S.e}{\bowtie} S(d, e, f)$ théta-összekapcsolás hat komponensből álló sorokat állít elő, egyet minden attribútumhoz a-tól f-ig, míg az $R(a, b, c) \bowtie S(b, c, d)$ természetes összekapcsolás ugyanannyi számú sort állít elő, de minden sornak csak négy komponense van.

Általánosan a következő szabály használható egy természetes összekapcsolás méretének becslésére, ha a két relációnak tetszőleges számú közös attribútuma van:

- Az $R \bowtie S$ méretének becsült értékét úgy számítjuk ki, hogy a $T(R)$-t megszorozzuk $T(S)$-vel, majd elosztjuk a $V(R, y)$ és $V(S, y)$ közül a nagyobbikkal minden közös y attribútum esetén.

7.26. példa: A következő példa a fenti szabályt alkalmazza. Egyben azt is szemlélteti, hogy a természetes összekapcsolásra vonatkozó eddigi eredményeink az egyenlőséget használó összekapcsolásokra is érvényesek. Vegyük az alábbi összekapcsolást:

$$R(a, b, c) \underset{R.b=S.d \text{ AND } R.c=S.e}{\bowtie} S(d, e, f)$$

A méretekkel kapcsolatban a paraméterek a következők legyenek:

$R(a, b, c)$	$S(d, e, f)$
$T(R) = 1000$	$T(S) = 2000$
$V(R, b) = 20$	$V(S, d) = 50$
$V(R, c) = 100$	$V(S, e) = 50$

Ezt az összekapcsolást egy természetes összekapcsolásként is felfoghatjuk, ha az $R.b$ és $S.d$ attribútumokat, illetve az $R.c$ és $S.e$ attribútumokat ugyanazoknak tekintjük. Ekkor a fenti szabály alapján az $R \bowtie S$ méretének becsült értéke az 1000×2000 szorzat, osztva a 20 és az 50 közül a nagyobbal, és tovább osztva a 100 és az 50 közül a nagyobbal. Az összekapcsolás becsült mérete tehát $1000 \times 2000/(50 \times 100) = 400$ sor. □

7.27. példa: Nézzük meg újra a 7.25. példát, de tekintsük most a harmadik lehetséges összekapcsolási sorrendet, amikor is először az $R(a, b) \bowtie U(c, d)$ összekapcsolást vesszük. Ez az összekapcsolás valójában egy szorzat, és az eredményben előálló sorok száma $T(R)T(U) = 1000 \times 5000 = 5\,000\,000$. Vegyük észre, hogy a szorzatban a különböző b-k száma $V(R, b) = 20$, a különböző c-k száma pedig $V(U, c) = 500$.

Amikor ezt a szorzatot az $S(b, c)$-vel összekapcsoljuk, akkor összeszorozzuk a sorok számait, majd ezt elosztjuk $\max(V(R, b), V(S, b))$-vel és $\max(V(U, c), V(S, c))$-vel. Az így kapott mennyiség $2000 \times 5\,000\,000/(50 \times 500) = 400\,000$. Vegyük észre, hogy az összekapcsolásnak ez a harmadik módja az eredmény méretére ugyanazt a becslést adja, mint amit a 7.25. példában kaptunk. □

7.4.6. Sok reláció összekapcsolása

Vizsgáljuk meg végül a természetes összekapcsolás általános esetét:

$$S = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$$

Tegyük fel, hogy az A attribútum az R_i-k közül k-ban fordul elő, és hogy ebben a k relációban az A értékek halmazainak méretei (elemszámai) – azaz a $V(R_i, A)$ különböző értékei $i = 1, 2, \ldots, k$ esetén $v_1 \le v_2 \le \ldots \le v_k$, a legkisebbtől a legnagyobb felé haladó sorrendben. Tegyük fel, hogy mindegyik relációból veszünk egy sort. Mennyi a valószínűsége annak, hogy a kiválasztott sorok mindegyike megegyezik az A attribútumon?

Tekintsük azt a t_1 sort, amelyiket abból a relációból választottunk, amelyben az A értékek száma, v_1, a legkisebb. Az értékhalmazok tartalmazására vonatkozó előfeltevés alapján a v_1 számú érték mindegyike az A attribútummal rendelkező összes többi reláció A értékei között megtalálható. Nézzük azt a relációt, amelynek az A attribútuma v_i értékeket vesz fel. Az ebből a relációból vett t_i sor $1/v_i$ valószínűséggel egyezik t_1-gyel az A-n. Mivel ez a kijelentés minden $i = 2, 3, \ldots, k$ esetén igaz, annak valószínűsége, hogy mind a k sor megegyezik az A-n, ezek szorzata lesz, vagyis $1/v_2 v_3 \ldots v_k$. Ez az eredmény vezet el a tetszőleges összekapcsolás méretének becslésére vonatkozó szabályhoz:

- Vegyük először az egyes relációkban szereplő sorok számainak szorzatát. Ezután minden olyan A attribútumra, amely legalább kétszer előfordul, osszuk el az összes $V(R, A)$-val a legkisebb kivételével.

Az összekapcsolás után az A attribútum értékeként megmaradó értékek számát szintén becsülhetjük. Az értékhalmazok megőrzésére vonatkozó előfeltevés alapján ez ezen $V(R, A)$-k legkisebbike lesz.

7.28. példa: Vegyük az $R(a, b, c) \bowtie S(b, c, d) \bowtie U(b, e)$ összekapcsolást, azokat a lényeges statisztikákat feltételezve, amelyeket a 7.24. ábra mutat. Az eredmény becsléséhez először képezzük a relációk méreteinek szorzatát: 1000 × 2000 × 5000. Ezután megnézzük, hogy mely attribútumok szerepelnek egynél többször, ezek a b, amely háromszor fordul elő, és a c, amely kétszer szerepel. Osztunk a $V(R, b)$, $V(S, b)$ és $V(U, b)$ közül a két legnagyobbal, ami 50 és 200. Végül osztunk a $V(R, c)$ és $V(S, c)$ közül a nagyobbal, ami 200. A kapott becslés tehát 1000 × 2000 × 5000/(50 × 200 × 200), azaz 5000.

Becsülhetjük az összekapcsolás eredményében az egyes attribútumokhoz tartozó értékhalmazok méretét is. Egy-egy ilyen becsült értéket úgy kapunk, hogy vesszük a különböző relációkban – ahol az attribútum megtalálható – az attribútumhoz tartozó értékszámlálók legkisebbikét. Az a, b, c, d és e attribútumok esetében ezek a számok sorra a következők: 100, 20, 100, 400 és 500. □

$R(a, b, c)$	$S(b, c, d)$	$U(b, e)$
$T(R) = 1000$	$T(S) = 2000$	$T(U) = 5000$
$V(R, a) = 100$		
$V(R, b) = 20$	$V(S, b) = 50$	$V(U, b) = 200$
$V(R, c) = 200$	$V(S, c) = 100$	
	$V(S, d) = 400$	
		$V(U, e) = 500$

7.24. ábra. *Paraméterek a 7.28. példához*

A két előfeltevésünkből – értékhalmazok tartalmazása és megőrzése – adódóan a becslés fent megadott szabálya rendelkezik egy meglepő és kellemes jellemzővel.

- Nem számít, hogy egy n relációt magában foglaló természetes összekapcsolást hogyan csoportosítunk és rendezünk, a becslésre vonatkozó szabályokat az egyes összekapcsolásokra egyenként alkalmazva az eredmény méretének ugyanazt a becslését kapjuk. Ez a becslés továbbá megegyezik azzal, amit akkor kapunk, ha az n reláció összekapcsolására, mint az egészre vonatkozó szabályt alkalmazzuk.

A 7.25. és 7.27 példák szemléltetik ezt a szabályt, amennyiben három reláció összekapcsolása történik háromféle csoportosításnak megfelelően, beleértve azt a csoportosítást is, ahol az „összekapcsolások" egyike ténylegesen egy szorzat.

7.4.7. Egyéb műveletek méretének becslése

Láttunk két műveletet, ahol az eredményül kapott sorok száma egy pontos formulával leírható:

1. A vetítés nem változtatja meg egy relációban szereplő sorok számát.
2. A szorzat olyan eredményt állít elő, amelyben a sorok száma egyenlő az argumentum relációkban lévő sorok számának szorzatával.

Két további műveletre – a kiválasztásra és az összekapcsolásra – elég jó becslési technikákat dolgoztunk ki. A fennmaradó műveletek esetében azonban nem könnyű az eredmény méretének meghatározása. Sorra vesszük a többi relációs algebrai operátort is, és javaslatokat fogunk tenni arra, hogy ez a becslés hogyan végezhető el.

Egyesítés

Ha a multihalmaz-egyesítést vesszük, akkor a méret pontosan az argumentumok méretének összegével egyenlő. Egy halmazegyesítésnél a méret lehet olyan nagy, mint a méretek összege, vagy olyan kicsi, mint a két argumentum mérete közül a nagyobb. Azt ajánljuk, hogy válasszunk valamit a kettő között félúton, például az összeg és a nagyobb átlagát (ami ugyanaz, mint a nagyobb plusz a kisebb fele).

Miért független a sorrendtől az összekapcsolás méretének becslése?

Ezt az állítást az összekapcsolásban szereplő relációk számára vonatkozó indukcióval lehet formálisan bebizonyítani. Ezt a bizonyítást nem közöljük, de a vezérfonalat megadjuk ebben a bekeretezett részben. Tegyük fel, hogy összekapcsolunk néhány relációt, és az utolsó lépés a következő:

$$(R_1 \bowtie \ldots \bowtie R_n) \bowtie (S_1 \bowtie \ldots \bowtie S_m)$$

Feltehető, hogy nem számít, hogy az R-ek összekapcsolását hogyan vesszük. A méret becslése ennek az összekapcsolásnak az esetében az R-ek méreteinek szorzata, osztva az olyan attribútumokhoz tartozó értékszámlálókkal, a legkisebbet kivéve, amely attribútumok az R-ekben többször is előfordulnak. Továbbá, minden egyes attribútumhoz tartozó becsült értékszámláló (az eredményben) az R-ekben az attribútumhoz tartozó értékszámlálók közül a legkisebb. Hasonló kijelentéseket fogalmazhatunk meg az S-ekre vonatkozóan.

Amikor a két reláció összekapcsolásakor kapott eredmény méretének becslésére vonatkozó szabályt (lásd 7.4.4. részt) alkalmazzuk az R-ek összekapcsolásából, illetve az S-ek összekapcsolásából származó két relációra, a becslés a két becsült érték szorzata lesz, osztva minden olyan attribútumhoz tartozó értékszámlálók közül a nagyobbal, amelyek az R-ekben és S-ekben egyaránt szerepelnek. Ez a becslés biztosan tartalmaz egy tényezőt, ami az összes $R_1, \ldots, R_n$, $S_1, \ldots, S_m$ reláció mérete. Ráadásul a becsült értéknek egy olyan osztója lesz minden egyes attribútum esetén, ami *nem* a legkisebb értékszámláló az adott attribútumhoz. Ez az osztó vagy jelen van már az R-ekre vagy S-ekre adott becslésben, vagy az utolsó lépésben kerül be, mert annak A attribútuma mind az R-ben, mind az S-ben szerepel, és ő a nagyobb a két értékszámláló közül, amelyek egyike a $V(R_i, A)$-k legkisebbike, a másik pedig a $V(S_j, A)$-k legkisebbike.

Metszet

Az eredménynek lehet olyan kevés sora, mint például 0, vagy olyan sok sora, mint a két argumentum közül a kisebbnek, függetlenül attól, hogy halmaz- vagy multihalmazmetszetről van szó. Egy lehetséges megközelítés, hogy a szélsőségek közti átlagot vesszük, ami a kisebb felét jelenti.

Egy másik lehetőség, hogy felismerjük azt, hogy a metszet a természetes összekapcsolás egy speciális esete, és a 7.4.4. részben bevezetett formulát használjuk. Halmazmetszet esetén ez a formula garantáltan olyan eredményt ad, ami nem nagyobb, mint a két reláció közül a kisebb. Egy multihalmazmetszet esetében azonban előfordulhatnak rendellenességek, amikor a becslés nagyobb, mint bármelyik argumentum. Nézzük például az $R(a, b) \cap_M S(a, b)$ metszetet, ahol az R a (0, 1) sor két példányából áll, és az S ugyanennek a sornak három példányából áll. Ekkor $V(R, a) = V(S, a) = V(R, b) = V(S, b) = 1$, $T(R) = 2$ és $T(S) = 3$. Az összekapcsolásra vonatkozó szabály alapján a becslés $2 \times 3/(\max(1, 1) \times \max(1, 1)) = 6$, de az eredményben nyilvánvalóan nem lehet több, mint $\min(T(R), T(S)) = 2$ sor.

Különbség

Amikor az $R - S$ különbséget vesszük, akkor az eredményben megkapott sorok száma $T(R)$ és $T(R) - T(S)$ között lehet. Becslésként az átlagot javasoljuk: $T(R) - T(S)/2$.

Ismétlődések megszüntetése

Ha $R(a_1, a_2, \ldots, a_n)$ egy reláció, akkor a $\delta(R)$ mérete $V(R,[a_1, a_2, \ldots, a_n])$. Sokszor azonban nem rendelkezünk ezzel a statisztikai értékkel, ezért közelíteni kell. Mint szélsőségek, a $\delta(R)$ mérete megegyezhet az R méretével (nincsenek ismétlődések, vagy lehet 1 (az R minden sora ugyanaz).[4] Egy másik felső korlát a $\delta(R)$-ben levő sorok számára az elképzelhető különböző sorok száma: a $V(R, a_i)$-k szorzata, ahol $i = 1, 2, \ldots, n$. Ez a szám lehet kisebb, mint a $T(\delta(R))$ más becslései. Több olyan szabály is van, amit használhatnánk a $T(\delta(R))$ becslésére. Az egyik elfogadható az, hogy a $T(R)/2$ és az összes $V(R, a_i)$ szorzata közül vesszük a kisebbiket.

Csoportosítás és összesítés

Tegyük fel, hogy van egy $\gamma_L(R)$ kifejezésünk, és e kifejezés eredményének méretére kell becslést adnunk. Ha rendelkezünk a $V(R,[g_1, g_2, \ldots, g_k])$ statisztikával, ahol a g_i-k az L-ben szereplő csoportosítási attribútumok, akkor az lesz a válaszunk. A statisztika

[4] Szigorúan véve, ha R üres, akkor sem az R-ben, sem a $\delta(R)$-ben nincs sor, vagyis az alsó korlát 0. Csakhogy ritkán érdekel bennünket ez a speciális eset.

azonban esetleg nem elérhető, így szükségünk van egy másik módszerre, amivel a $\gamma_L(R)$ méretét becsülhetjük. A $\gamma_L(R)$ sorainak száma megegyezik a csoportok számával. Az eredményben lehet egy csoport, vagy lehet olyan sok csoport, mint ahány sor van az R-ben. A δ-hoz hasonlóan, a csoportok számára a $V(R, A)$-k szorzatával is adhatunk felső korlátot, de itt az A attribútum csak az L csoportosítási attribútumain fut végig. Újfent azt a becslést javasoljuk, ami veszi a $T(R)/2$ és e szorzat közül a kisebbiket.

7.4.8. Feladatok

7.4.1. feladat: A W, X, Y és Z relációkhoz az alábbi statisztikák tartoznak:

$W(a, b)$	$X(b, c)$	$Y(c, d)$	$Z(d, e)$
$T(W) = 100$	$T(X) = 200$	$T(Y) = 300$	$T(Z) = 400$
$V(W, a) = 20$	$V(X, b) = 50$	$V(Y, c) = 50$	$V(Z, d) = 40$
$V(W, b) = 60$	$V(X, c) = 100$	$V(Y, d) = 50$	$V(Z, e) = 100$

Adjunk becslést a következő kifejezések eredményeként kapott relációk méreteire:

* a) $W \bowtie X \bowtie Y \bowtie Z$
* b) $\sigma_{a=10}(W)$
c) $\sigma_{c=20}(Y)$
d) $\sigma_{c=20}(Y) \bowtie Z$
e) $W \times Y$
f) $\sigma_{d>10}(Z)$
* g) $\sigma_{a=1 \text{ AND } b=2}(W)$
h) $\sigma_{a=1 \text{ AND } b>2}(W)$
i) $X \underset{X.c < Y.c}{\bowtie} Y$

* **7.4.2. feladat:** Az E, F, G és H relációkhoz az alábbi statisztikák tartoznak:

$E(a, b, c)$	$F(a, b, d)$	$G(a, c, d)$	$H(b, c, d)$
$T(E) = 1000$	$T(F) = 2000$	$T(G) = 3000$	$T(H) = 4000$
$V(E, a) = 1000$	$V(F, a) = 50$	$V(G, a) = 50$	$V(H, b) = 40$
$V(E, b) = 50$	$V(F, b) = 100$	$V(G, c) = 300$	$V(H, c) = 100$
$V(E, c) = 20$	$V(F, d) = 200$	$V(G, d) = 500$	$V(H, d) = 400$

Hány sort tartalmaz ezeknek a relációknak az összekapcsolása, ha az ebben a részben bemutatott becslési technikákat alkalmazzuk?

! **7.4.3. feladat:** Hogyan becsülné egy egyoldali összekapcsolás (semijoin) méretét?

!! **7.4.4. feladat:** Vegyük az $R(a, b) \bowtie S(a, c)$ összekapcsolást, ahol R-nek és S-nek egyaránt 1000 sora van. Az a attribútumnak mindkét relációban 100 különböző értéke van, és ez *ugyanaz* a 100 érték. Ha az értékek eloszlása egyenletes lenne, vagyis minden egyes a érték pontosan 10-szer fordulna elő mindkét relációban, akkor az összekapcsolásban 10 000 sor lenne. Tegyük fel ehelyett, hogy a 100 a érték mindkét relációban ugyanazt a Zipfian-eloszlást mutatja. Egész pontosan, legyenek az értékek a_1, a_2, ..., a_{100}. Ekkor az R és az S azon sorainak száma, melyek a értéke a_i, az $1/\sqrt{i}$ -vel arányos. Hány sort tartalmaz az összekapcsolás ezen feltételek mellett? Hagyjuk figyelmen kívül azt a tényt, hogy egy adott a értékkel rendelkező sorok száma nem lehet nem egész szám.

7.5. Bevezetés a költség alapú tervválasztásba

Akár egy logikai terv kiválasztásáról van szó, akár egy fizikai terv logikai tervből történő létrehozásáról, a lekérdezésoptimalizálónak becslést kell végeznie bizonyos kifejezések kiértékelésének a költségére vonatkozóan. A költség alapú tervválasztásban felmerülő kérdéseket itt tárgyaljuk, a 7.6. részben pedig részletesen megvizsgáljuk a költség alapú tervválasztás egyik legfontosabb és legnehezebb problémáját: több reláció összekapcsolási sorrendjének megválasztását.

Akárcsak korábban, most is azzal a feltételezéssel élünk, hogy egy kifejezés kiértékelésének „költségét" a végrehajtott lemez I/O-műveletek száma jól közelíti. A lemez I/O-műveletek számát pedig a következők befolyásolják:

1. A lekérdezés megvalósítására kiválasztott konkrét logikai operátorok, ami akkor dől el, amikor a logikai lekérdezéstervet megválasztjuk.
2. A közbülső relációk méretei, amik becsléseit a 7.4. részben tárgyaltuk.
3. A logikai operátorok megvalósítására használt fizikai operátorok, például hogy egymenetes vagy kétmenetes összekapcsolást választunk, vagy hogy rendezünk vagy nem rendezünk egy adott relációt. Ezt a kérdést a 7.7. részben tárgyaljuk.
4. A hasonló műveletek sorrendje, különös tekintettel az összekapcsolásra, amit a 7.6. rész tárgyal.
5. Az argumentumok átadásának módszere, vagyis ahogy az argumentumok egy fizikai operátortól a következő számára átadódnak. Ennek tárgyalása szintén a 7.7. részben kerül sorra.

Sok kérdést kell megoldanunk ahhoz, hogy hatékony költség alapú tervválasztást valósítsunk meg. Ebben a részben először azt nézzük meg, hogy az adatbázisból hogyan nyerhetjük ki leghatékonyabban a méretre vonatkozó paramétereket, amelyek olyan lényeges voltak, amikor a relációk méretét becsültük a 7.4. részben. Ezután újra elővesszük a jó logikai terv megtalálása érdekében bevezetett algebrai szabályokat. A költségalapú elemzéskor a logikai lekérdezésterv transzformálására való szokásos heurisztikák használata indokolt lehet, mint amilyen például a kiválasztások tologatása lefelé a fában. Végül a kiválasztott logikai tervből származtatható összes fizikai terv felsorolására vonatkozóan nézünk meg különböző megközelítéseket. Különösen fontosak a kiértékelendő tervek számának csökkentésére irányuló módszerek, melyek ugyanakkor valószínűsítik azt is, hogy a legkisebb költségű terv is köztük van.

7.5.1. Méretre vonatkozó paraméterek becslése

A 7.4. rész formuláit arra alapozva fogalmaztuk meg, hogy ismerünk bizonyos fontos paramétereket, különösen a $T(R)$-t, ami egy R reláció sorainak számát jelenti, és a $V(R, a)$-t, ami az R reláció a attribútumában előforduló különböző értékek számát jelöli. Egy modern adatbázis-kezelő rendszer általában lehetővé teszi, hogy a felhasználó vagy a rendszergazda közvetlenül kérje ezeknek a statisztikáknak az összegyűjtését. Ezek a statisztikák ezután használhatók a további lekérdezésoptimalizálások során a műveletek költségének becslésére. Ha ezt követő adatbázis-módosítások hatására a statisztikai értékek megváltoznak, a változásokat a rendszer csak egy újabb, statisztikát gyűjtő parancs után veszi figyelembe.

Ha végigolvasunk egy teljes R relációt, akkor nyilván megszámolható a benne lévő sorok $T(R)$ száma, és minden A attribútumra az általa felvett különböző értékek $V(R, A)$ száma is megkapható. Azoknak a blokkoknak a számát, amelyekben az R elfér – azaz $B(R)$-t – úgy becsülhetjük, hogy vagy megszámoljuk a blokkok tényleges számát (ha R nyaláboltan tárolt), vagy a $T(R)$-t elosztjuk azon sorok számával, amelyek egy blokkban elférnek (vagy az egy blokkban elhelyezhető sorok átlagos számával, ha a sorok változó hosszúságúak). Vegyük észre, hogy a $B(R)$ e két becslése nem feltétlenül ugyanaz, de azok rendszerint „elég közeliek" a költségek összehasonlítása szempontjából, amíg konzisztens módon az egyik vagy a másik megközelítést választjuk.

Egy adatbázis-kezelő rendszer egy adott attribútum értékeinek *hisztogramját* is ki tudja számítani. Ha $V(R, A)$ nem túl nagy, akkor a hisztogram az A attribútum minden értékéhez tartalmazhatja azok előfordulásának számát (vagy arányát). Ha ennek az attribútumnak nagyon sok értéke van, akkor az is egy lehetőség, hogy csak a leggyakoribb értékeket rögzítjük külön-külön, a többi értéket pedig csoportokba soroljuk. A hisztogramok legjellemzőbb típusai a következők:

1. *Egyenlő szélesség.* Választunk egy w szélességet és egy v_0 konstanst. Meghatározzuk azoknak a soroknak a számát, amely sorokban lévő – v-vel jelölt – értékre: $v_0 \leq v < v_0 + w$. Ugyanezt tesszük akkor is, amikor $v_0 + w \leq v < v_0 + 2w$, és így tovább. A v_0 érték lehet a legkisebb lehetséges érték vagy az aktuális minimum érték. Az utóbbi esetben, ha egy alacsonyabb értékkel találkozunk, csökkentjük a v_0 értékét w-vel és egy új (sor)számlálót adunk a hisztogramhoz.
2. *Egyenlő magasság.* Ezek a szokásos „százalékos arányok". Veszünk egy p törtet, és felsoroljuk a legkisebb értéket, azt az értéket, amelyik p törtnyire van a legkisebbtől, azt amelyik $2p$ törtnyire van a legkisebbtől és így tovább, a legnagyobb értékig.

3. *Leggyakoribb értékek.* Felsorolhatjuk a leggyakoribb értékeket és a hozzájuk tartozó előfordulási számokat. Ezt az információt megadhatjuk úgy, hogy az összes többi értekre úgy számolunk előfordulási gyakoriságot, hogy azokat egyetlen csoportnak tekintjük, de a leggyakoribb értékeket a többi értékre vonatkozó egyenlő szélességű vagy egyenlő magasságú hisztogrammal együtt is rögzíthetjük.

Egy hisztogram használatának az az előnye, hogy az összekapcsolások méreteire a 7.4. részben leírt egyszerűsített módszereknél pontosabb becslést adhatunk. Konkrétabban szólva, ha az összekapcsolási attribútum valamely értéke mindkét összekapcsolandó reláció hisztogramjában közvetlenül megjelenik, akkor pontosan tudjuk, hogy az eredménynek hány sorában fog szerepelni ez az érték. Az összekapcsolási attribútum azon értékei esetén, amelyek valamelyik reláció hisztogramjában nem jelennek meg közvetlenül, az összekapcsolásra gyakorolt hatás a 7.4. résznek megfelelően becsülhető. Ha egyenlő szélességű hisztogramot használunk úgy, hogy a két reláció összekapcsolási attribútumaira ugyanazokat a sávokat alkalmazzuk, akkor becsülhetjük az egymásnak megfelelő sávok összekapcsolásainak méreteit, majd ezeket összeadhatjuk. Az eredmény helyes lesz, mert csak az egymásnak megfelelő sávokba eső sorok kapcsolódhatnak. A következő példákban hisztogram alapú becslésre mutatunk példákat. A későbbiek során nem fogunk hisztogramokat használni a becslésekben.

7.29. példa: Vegyünk olyan hisztogramokat, amelyek a három leggyakoribb értéket és a hozzájuk tartozó számlálókat tartalmazzák, és a maradék értékeket egy csoportba sorolják. Tegyük fel, hogy az $R(a, b) \bowtie S(b, c)$ összekapcsolást akarjuk kiszámítani. Az $R.b$-re vonatkozó hisztogram legyen az alábbi:

1: 200, 0: 150, 5: 100, egyéb: 550

Az R reláció 1000 sorából tehát 200-nak a b értéke 1, 150-nek a b értéke 0 és 100-nak a b értéke 5. 550 sornak van továbbá olyan b értéke, ami nem 0, 1 vagy 5, és ezen egyéb értékek közül egyik sem fordul elő 100-nál többször.

Az $S.b$-re vonatkozó hisztogram a következő legyen:

0: 100, 1: 80, 2: 70, egyéb: 250

Tegyük fel továbbá, hogy $V(R, b) = 14$ és $V(S, b) = 13$. Ez azt jelenti, hogy az R ismeretlen b értéket tartalmazó 550 sora tizenegy érték között van elosztva, vagyis átlagosan 50 sor jut mindegyikre, és az S ismeretlen b értéket tartalmazó 250 sora tíz érték között van elosztva, azaz átlagosan 25 sor jut minden ilyen értékre.

A 0 és 1 értékek explicit módon szerepelnek mindkét hisztogramban. Így kiszámolhatjuk, hogy ha az R-nek azt a 150 sorát, amelyekre $b = 0$, összekapcsoljuk az S-nek azzal a 100 sorával, amelyeknek b értéke ugyanez az érték, akkor ez 15 000 sort eredményez. Hasonlóképpen, ha az R-nek azt a 200 sorát, amelyekre $b = 1$, összekapcsoljuk az S-nek azzal a 80 sorával, amelyekre szintén $b = 1$, akkor ez 16 000 további sort eredményez.

A maradék sorok összekapcsolásának becslése összetettebb. Továbbra is fenntartjuk azt a előfeltevést, hogy a kisebb értékhalmazzal rendelkező relációban (jelen esetben S) előforduló minden érték a másik reláció értékhalmazában is szerepel. Az S tizenegy fennmaradó b értéke közül az egyikről tudjuk, hogy az a 2, egy másikról viszont feltesszük, hogy az az 5, hiszen ez az egyik leggyakoribb érték az R-ben. Becslésként azt mondjuk, hogy a 2 az R-ben 50-szer fordul elő, az 5 az S-ben pedig 25-ször. Ezeket a becsléseket úgy kapjuk, hogy azt feltételezzük, az adott érték a megfelelő reláció hisztogramjában említett „egyéb" értékek egyike. A 2 b értékből adódó további sorok száma így $70 \times 50 = 3500$, az 5 b értékből származó további sorok száma pedig $100 \times 25 = 2500$.

Végezetül van még kilenc olyan b érték, ami mindkét relációban szerepel, és az ezekre vonatkozó becslésünk az, hogy mindegyik 50-szer fordul elő R-ben, és 25-ször S-ben. Így mind a kilenc érték $50 \times 25 = 1250$ további sorral járul hozzá az eredményhez. A végső eredmény méretének becsült értéke tehát:

$$15\,000 + 16\,000 + 3500 + 2500 + 9 \times 1250$$

azaz 48 250 sor. Megjegyezzük, hogy a 7.4. részből vett egyszerűbb becslés, ami azon a feltevésen alapul, hogy mindegyik érték ugyanannyiszor fordul elő mindegyik relációban, $1000 \times 500/14$, azaz 35 714 lenne. □

7.30. példa: Ebben a példában egy egyenlő szélesség típusú hisztogramot feltételezünk, és azt demonstráljuk, hogy hogyan befolyásolja egy összekapcsolás méretének becslését az, ha tudjuk, hogy a két reláció értékhalmazai majdnem diszjunktak. A relációk a következők:

```
Jan(nap, hőmérséklet)
Júl(nap, hőmérséklet)
```

A lekérdezés pedig az alábbi:

```
SELECT Jan.nap, Júl.nap
FROM Jan, Júl
WHERE Jan.hőmérséklet = Júl.hőmérséklet;
```

A január és július hónapoknak azokat a napjait keressük tehát, amikor ugyanaz volt a hőmérséklet. A lekérdezésterv az, hogy a `Jan` és `Júl` relációkat a `hőmérséklet` attribútumok egyenlősége alapján összekapcsoljuk, majd vetítünk a két `nap` attribútumra.

A `Jan` és `Júl` relációkhoz tartozó, a hőmérséklet attribútumokra vonatkozó feltételezett hisztogramokat a 7.25. ábra mutatja.[5] Általában, ha mindkét összekapcsolási attribútumhoz egyenlő szélesség típusú hisztogram tartozik ugyanazokkal a sávokkal (amelyik közül némelyik esetleg üres valamelyik reláció esetében), akkor az össze-

[5] Az Egyenlítőtől délre lévő barátaink felcserélhetik a január és július oszlopait.

kapcsolás méretét úgy becsülhetjük, hogy az egyes sávokra leszűkített összekapcsolások méreteit becsüljük külön-kölön, majd az így kapott becsléseket összegezzük.

Sáv	*Jan*	*Júl*
0–9	40	0
10–19	60	0
20–29	80	0
30–39	50	0
40–49	10	5
50–59	5	20
60–69	0	50
70–79	0	100
80–89	0	60
90–99	0	10

7.25. ábra. *Hőmérséklet hisztogramjai*

Ha két megfelelő sávnak T_1, illetve T_2 sora van, és a sávba tartozó értékek száma V, akkor – a 7.4.4. részben lefektetett elveket követve – az ezekre a sávokra vonatkoztatott összekapcsolás eredményének a méretére kapott becslés: T_1T_2/V. A 7.25. ábrán látható hisztogramok esetében ezeknek a szorzatoknak a nagy része 0, mert a T_1 és T_2 közül az egyik vagy mindkettő 0. Csak a 40–49 és 50–59 sávok azok, amelyekre sem a T_1, sem a T_2 nem 0. Mivel egy sáv szélessége $V = 10$, a 40–49 sáv $10 \times 5/10 = 5$ sort, az 50–59 sáv pedig $5 \times 20/10 = 10$ sor jelent.

Következésképpen, az összekapcsolás méretének becslése $5 + 10 = 15$ sor. Ha nem lennének hisztogramok, és csak annyit tudnánk, hogy mindegyik relációnak 245 sora van, amelyek 100 darab 0 és 99 közé eső érték mentén oszlanak el, akkor az összekapcsolás méretének becslése $245 \times 245/100 = 600$ sor lenne. □

7.5.2. Statisztikák növekményes kiszámítása

A lekérdezésoptimalizálókban a statisztikák bizonyos időnkénti kiszámolását részesítik előnyben, mert ezek a statisztikák nem szoktak rövid időn belül radikálisan megváltozni. Továbbá, amint már említettük, még a pontatlan statisztikák is hasznosak, ha azokat mindegyik tervre alkalmazzuk, amelyek a „legjobb" címért versengenek. Teljes relációk időnkénti megvizsgálása azonban drága dolog. A statisztikák időnkénti újra kiszámításának egy alternatívája a *növekményes kiértékelés* (incremental evaluation). Ez a módszer karbantartja és aktualizálja a paraméterekre vonatkozó becsléseket minden alkalommal, amikor az adatbázist módosítják. Íme néhány módja annak, ahogy ezt a rendszer megteheti:

- A $T(R)$ karbantartásához a rendszer 1-et mindig hozzáad, amikor egy sort beszúrnak R-be, illetve 1-et kivon belőle, valahányszor onnan törölnek egy sort. Megjegyezzük, hogy ehhez szükség van a beszúrást és a törlést végző függvények módosítására. Ez növeli minden ilyen művelet költségét, de a többletköltség általában jelentéktelen.
- Ha az R valamely attribútumához létezik B-fa-index, akkor $T(R)$-t úgy becsülhetjük, hogy csak a B-fában lévő blokkok számát számoljuk meg. Feltehetjük például, hogy minden blokk 3/4 részben van tele, és egy blokkban elférő kulcsok és mutatók számát használhatjuk a B-fa levelei által mutatott sorok számának becslésére. Ez a módszer kevésbé pontos, mint a $T(R)$ direkt megszámolása, de csak akkor igényel teendőt, amikor a B-fa szerkezete változik, ami aránylag ritka a beszúrásokhoz és törlésekhez viszonyítva.
- Ha az R reláció a attribútumához létezik index, akkor a $V(R, a)$-t pontosan karbantarthatjuk. Egy R-be történő beszúráskor mindenképpen meg kell találnunk az új sor a értékét az indexben, és ekkor megállapítjuk, hogy létezett-e már sor ugyanezzel az a értékkel. Ha nem, akkor a $V(R, a)$ számlálóhoz egyet hozzáadunk. Hasonlóképpen, amikor törlünk egy sort az R-ből, akkor ellenőrizzük, hogy az utolsó sort töröljük-e R-ből az adott a értékkel, és ha igen, akkor csökkentjük eggyel a $V(R, a)$-t.
- Ha tudjuk, hogy az a kulcsa az R-nek, akkor azt is tudjuk, hogy $V(R, a) = T(R)$, anélkül hogy az indexet vizsgálnánk vagy a $V(R, a)$-t közvetlenül karbantartanánk.
- Ha az $R.a$-ra nincs index, akkor a rendszer létrehozhat egy kezdetleges indexet azáltal, hogy fenntart egy adatstruktúrát (például egy tördelőtáblát vagy B-fát) az a értékeinek tárolására.

Végül az is egy lehetőség a $V(R, A)$ statisztika kiszámítására, hogy akkor adunk erre a mennyiségre statisztikai becslést, amikor szükség van rá, mégpedig az adatok egy

Miért becslünk méretet lemez I/O-műveletek helyett?

Amikor a logikai lekérdezésterveket vizsgáljuk, még nem döntöttük el azt, hogy mely fizikai operátorokat használjuk a relációs algebra operátorainak megvalósításához. Így nem lehetünk biztosak egy adott terv végrehajtásához szükséges lemez I/O-műveletek számában. Követhetjük azonban azt a heurisztikát, miszerint az a terv lesz valószínűleg a legjobb terv, amelyikre a közbülső relációk méreteinek összege a legkisebb. Ezt a heurisztikát az támasztja alá, hogy minél kisebbek a relációk, annál kevesebb lemez I/O-műveletet igényel azok olvasása és írása, és annál hatékonyabbak a relációkra vonatkozó műveleteket megvalósító algoritmusok.

Egy lekérdezés igazi költségének becslését tényleg a lemez I/O-műveletek száma jelenti. Egy részletesebb elemzés figyelembe venné a CPU időt is, és egy még részletesebb elemzés még a lemezfej mozgására is kitérne, számításba véve az elért blokkok helyét a lemezen. A gyakorlatban még a legegyszerűbb becslés – közbülső relációk méretei – is elég jó, hiszen a lekérdezésoptimalizálónak csak lekérdezésterveket kell összehasonlítania, és nem kell pontos végrehajtási időt számolnia.

kis részének mintája alapján. Ez egy bonyolult számítás, és számos feltételtől függ, például attól, hogy egy attribútumban előforduló értékek egyenletesen eloszlást, Zipfian-eloszlást vagy valamilyen más eloszlást mutatnak-e. A vezérfonal azonban a következő. Ha megnézzük az R egy kis mintáját, mondjuk a sorainak 1%-át, és azt találjuk, hogy a legtöbb a érték különböző, akkor $V(R, a)$ valószínűleg közel van $T(R)$-hez. Ha azt tapasztaljuk, hogy az a értékeiben nagyon kevés különböző érték szerepel, akkor az a valószínű, hogy az aktuális relációban létező legtöbb a értékkel találkoztunk.

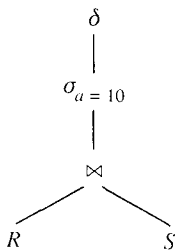
7.5.3. Logikai lekérdezéstervek költségének csökkentésére irányuló heurisztikák

A lekérdezésekre és alkérdésekre vonatkozó költségbecslések jól használhatók a lekérdezések heurisztikus átalakításai során. A 7.3.3. részben már megvizsgáltuk, hogy miként várható el, hogy bizonyos heurisztikák költségbecsléstől független alkalmazása szinte biztosan javítson egy logikai lekérdezésterv költségén. Jó példa erre a kiválasztások tologatása lefelé a fában. Vannak azonban a lekérdezésoptimalizálásnak más pontjai, ahol a költség becslése egy transzformáció előtt és után lehetővé teszi, hogy alkalmazzuk a transzformációt, ha várhatóan csökkenti a költséget, egyébként pedig elkerüljük. A végső logikai terv előállításakor például számba vehetünk számos opcionális transzformációt, és megnézhetjük az azok végrehajtása előtt és után előálló költségeket. Ezeket a kérdéseket szemlélteti a következő példa.

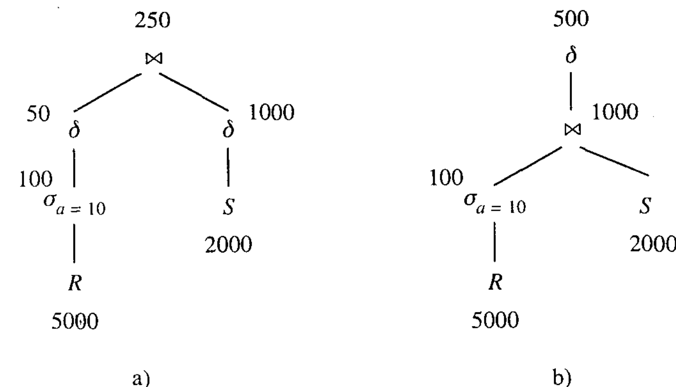
7.31. példa: Vegyük a 7.26. ábrán található kiindulási logikai tervet, valamint legyenek az R és S relációkhoz tartozó statisztikák a következők:

$R(a, b)$	$S(b, c)$
$T(R) = 5000$	$T(S) = 2000$
$V(R, a) = 50$	
$V(R, b) = 100$	$V(S, b) = 200$
	$V(S, c) = 100$

Kiindulva a 7.26. ábrából, a végső logikai lekérdezésterv előállításakor kitartunk amellett, hogy a kiválasztást levisszük a fában ameddig csak lehet. Nem vagyunk vi-



7.26. ábra. *Logikai lekérdezésterv a 7.31. példához*



7.27. ábra. *Két jelölt a legjobb logikai lekérdezéstervre*

szont biztosak abban, hogy a δ operátor összekapcsolás alá történő levitelének van-e értelme vagy sem. A 7.26. ábrából így két lekérdezéstervet generálunk, ezek a 7.27. ábrán találhatók, és abban különböznek, hogy az ismétlődések megszüntetése az összekapcsolás előtt vagy után történik-e. Észrevehetjük, hogy az a) tervben a δ-t levittük a fa mindkét ágán. Ha R és/vagy S nem tartalmaz ismétlődéseket, akkor a megfelelő ágon a δ elhagyható.

A 7.4.3. részből tudjuk, hogy a kiválasztás eredményének méretét hogyan kell becsülni: a $T(R)$-t elosztjuk a $V(R, a)$-val, ahol $V(R, a) = 50$. Azt is tudjuk, hogy az összekapcsolás méretét hogyan becsüljük: az argumentumok méreteinek szorzatát osztjuk $\max(V(R, b), V(S, b))$-vel, ahol ez a maximum érték 200. Azt még nem tudjuk viszont, hogy hogyan becsüljük az ismétlődésektől megszabadított relációk méretét.

Nézzük először a $\delta(\sigma_{a=10}(R))$ méretének becslését. Mivel a $\sigma_{a=10}(R)$-ben csak egyetlen értéke lehet az a-nak és mintegy 100 értéke a b-nek, és e reláció becsült mérete 100 sor, a 7.4.7. részben megfogalmazott szabály azt mondja számunkra, hogy az egyes attribútumokhoz tartozó értékszámlálók szorzata nem korlátozó tényező. Ezért a δ eredményének méretét a $\sigma_{a=10}(R)$-ben lévő sorok felével becsüljük. Ennek megfelelően a 7.27.a) ábrán 50 sor szerepel a $\delta(\sigma_{a=10}(R))$ méretének becsléseként.

Nézzük most a 7.27.b) ábrán lévő δ eredményére vonatkozó becslést. Az összekapcsolás eredményében egyetlen érték fordul elő az a-ban, becslés alapján a b-ben $\min(V(R, b), V(S, b)) = 100$ különböző érték fordul elő illetve becslés alapján $V(S, c) =$ $= 100$ érték szerepel a c-ben. Vagyis ismét arra a következtetésre jutunk, hogy az értékszámlálók szorzata nem korlátozza a δ eredményének méretét. Az eredmény méretére adott becslés az összekapcsolás eredményében lévő sorok fele, azaz 500 sor lesz.

Hogy össze tudjuk hasonlítani a 7.27. ábra két tervét, összeadjuk a csomópontokhoz tartozó becsült méreteket, kivéve a gyökeret és a leveleket. A gyökeret és a leveleket azért hagyjuk ki, mert ezek a méretek nem függnek a választott tervtől. Ez a költség, vagyis a közbülső csomópontokhoz tartozó becsült méretek összege, az a) terv esetében: $100 + 50 + 1000 = 1150$, míg a b) terv esetében az összeg: $100 + 1000 = 1100$. Arra a következtetésre jutunk tehát, hogy kis eltéréssel ugyan, de az ismétlődések

Az eredmény méretének becslései nem kell hogy egyformák legyenek

Vegyük észre, hogy a 7.27. ábrán a két fa gyökereihez tartozó becslések különbözőek: 250 az egyik esetben és 500 a másikban. Mivel a becslés egy pontatlan tudomány, ilyen anomáliák előfordulnak. Valójában az a kivétel, amikor a konzisztenciára vonatkozóan garanciát tudunk nyújtani, amint ezt a 7.4.6. részben tettük.

Intuitíve azt mondhatnánk, hogy a b) terv becsült költsége magasabb, mert ha mind az R-ben, mind az S-ben vannak ismétlődések, akkor ezek az ismétlődések sokszorozódnak az összekapcsolás során, azok a sorok például, amelyek háromszor szerepelnek az R-ben és kétszer az S-ben, hatszor fognak megjelenni az $R \bowtie S$ összekapcsolás eredményében. A mi egyszerű formulánk, amellyel egy δ eredményének méretét becsüljük, nem számol azzal a lehetőséggel, hogy korábbi műveletek felerősíthetik az ismétlődések hatását.

megszüntetését a végére halasztva egy jobb tervet kapunk. Az ellenkező következtetésre jutnánk akkor, ha R-nek vagy S-nek kevesebb b értéke lenne. Ekkor nagyobb lenne az összekapcsolás mérete, ami megnövelné a b) terv költségét. □

7.5.4. Fizikai tervek felsorolásának lehetőségei

Most azt vizsgáljuk meg, hogy egy logikai lekérdezésterv fizikai lekérdezéstervvé történő konvertálása során hogyan használjuk a költségbecslést. A – *kimerítőnek* nevezett – legalapvetőbb megközelítésben vesszük a 7.4. rész elején felvázolt pontokra (összekapcsolások sorrendje, operátorok fizikai implementációja stb.) adott lehetséges válaszok összes kombinációját. Mindegyik fizikai tervhez egy becsült költséget rendelünk, és a legkisebb költségű tervet választjuk.

Sok egyéb megközelítés is létezik azonban egy fizikai terv kiválasztására. Ebben a részben különböző használatban lévő megközelítéseket mutatunk be erre vonatkozóan, míg a 7.6. részben az összekapcsolási sorrend megválasztásának problémájával kapcsolatos fő elgondolásokat szemléltetjük. Mielőtt folytatnánk, hadd jegyezzük meg, hogy a lehetséges fizikai tervek tartományának feltárására alapvetően kétféle megközelítés létezik:

- *Felülről lefelé.* Itt a gyökértől indulunk el, és haladunk lefelé a logikai lekérdezésterv fájában. A gyökérben található művelet minden lehetséges megvalósításához megnézzük az argumentum(ok) lehetséges kiértékeléseit, kiszámoljuk az egyes kombinációk költségét, és a legjobbat választjuk.[6]

[6] Emlékezzünk a 7.3.4. részből arra, hogy a logikai lekérdezésterv fájának egy csomópontja egyetlen kommutatív és asszociatív operátor (például összekapcsolás) sokféle használatát képviselheti. Tehát egy adott csomópontra vonatkozó összes lehetséges terv megvizsgálása maga is nagyon sok választás felsorolását foglalhatja magában.

- *Alulról felfelé.* A logikai lekérdezésterv fájának minden részkifejezéséhez kiszámoljuk a részkifejezés lehetséges kiszámítási módjaihoz tartozó költségeket. Egy E részkifejezés kiértékelési lehetőségeit és költségeit úgy számítjuk ki, hogy vesszük az E részkifejezéseire vonatkozó lehetséges választásokat, és az összes lehetséges módon kombináljuk azokat az E gyökér operátorának lehetséges megvalósításaival.

Valójában nincs sok különbség a két megközelítés között, azok legtágabb értelmezését tekintve, hiszen mindegyik esetben tekintetbe vesszük a lekérdezésben szereplő összes operátor megvalósítási módjainak minden lehetséges kombinációját. A keresés korlátozásával kapcsolatban elmondhatjuk, hogy egy felülről lefelé megközelítés esetleg lehetővé teszi, hogy elhagyjunk bizonyos választásokat, amelyeket alulról felfelé megközelítésben nem hagyhatnánk el. Kidolgoztak azonban olyan alulról felfelé stratégiákat is, amelyek jelentősen korlátozzák a választásokat, ezért az elkövetkezőkben az alulról felfelé módszerekre helyezzük a hangsúlyt.

Észrevehettük, hogy az alulról felfélé módszernek létezik egy nyilvánvaló egyszerűsítése, nevezetesen, amikor egy nagyobb részkifejezés kiszámítása során csak a *legjobb* tervet vesszük figyelembe annak minden részkifejezése esetén. Ez a megközelítés, amely a módszerek alábbi listájában *dinamikus programozás* néven szerepel, nem biztos, hogy a legjobb tervet eredményezi, habár ez gyakran megtörténik. A *Selinger-módszer* (vagy *System-R-módszer*) elnevezésű optimalizálás, amely szintén szerepel a listában, egy részkifejezéshez tartozó tervek további tulajdonságait is kihasználja annak érdekében, hogy optimális végső terveket állítson elő olyan tervekből, amelyek bizonyos részkifejezések esetén nem optimálisak.

Heurisztikus választás

Az is egy megoldás, hogy egy fizikai terv kiválasztására ugyanazt a megközelítést alkalmazzuk, mint amit általában egy logikai terv kiválasztására használunk: válasszunk heurisztikák alapján. A 7.6.6. részben egy „mohó" heurisztikát fogunk tárgyalni az összekapcsolási sorrendre vonatkozóan. Eszerint először azt a két relációt kapcsoljuk össze, amelyek eredményének becsült mérete a legkisebb, majd ugyanezt az elvet alkalmazzuk ismételten az így kapott reláció és a többi összekapcsolásra váró reláció összekapcsolása során. Sok egyéb alkalmazható heurisztika is létezik, íme néhány a leggyakrabban használtak közül:

1. Ha a logikai tervben egy $\sigma_{A=c}(R)$ kiválasztás szerepel, és az R tárolt relációnak van egy indexe az A attribútumra vonatkozóan, akkor csak az indexet nézzük végig azoknak az R-beli soroknak a megtalálására, amelyeknél az A értéke egyenlő c-vel.
2. Általánosabban szólva, ha a kiválasztás tartalmaz egy fenti $A = c$ feltételt és más feltételeket is, akkor a kiválasztás megvalósítható egy indexes kereséssel, valamint

egy azt követő, a kapott sorokra vonatkozó további kiválasztással, amit a *szűrés* fizikai szintű operátor (`Filter`) fog képviselni.

3. Ha egy összekapcsolás valamely argumentumának van indexe az összekapcsolási attribútum(ok)ra vonatkozóan, akkor használjunk indexes összekapcsolást azzal a relációval a belső ciklusban.
4. Ha egy összekapcsolás egyik argumentuma rendezett az összekapcsolási attribútum(ok) szerint, akkor egy rendezéses összekapcsolást részesítsünk előnyben egy tördelő összekapcsolással szemben, de nem feltétlenül egy index-összekapcsolással szemben, ha olyan is lehetséges.
5. Három vagy több reláció egyesítése vagy metszete során először a legkisebb relációkat csoportosítsuk.

Elágazás-és-korlát

Ebben a – gyakorlatban gyakran használt – megközelítésben azzal kezdjük, hogy valamilyen heurisztikát alkalmazva egy jó fizikai tervet keresünk a teljes logikai lekérdezéstervhez. Legyen ennek a tervnek a költsége *C*. Ezután, miközben alkérdésekhez tartozó további terveket vizsgálunk, elvethetünk minden olyan tervet egy alkérdés esetén, amelynek költsége nagyobb *C*-nél, hiszen egy ilyen – alkérdéshez tartozó – terv nem lehet része egy olyan – a teljes lekérdezéshez tartozó – tervnek, amelytől azt várjuk, hogy jobb, mint amit már ismerünk. Ha egy olyan tervet állítunk elő a teljes lekérdezésre vonatkozóan, amelynek költsége kisebb, mint *C*, akkor a fizikai lekérdezéstervek tartományának további feltárása során *C*-t ennek a tervnek a költségével helyettesítjük.

E megközelítésnek egy nagy előnye, hogy megválaszthatjuk, mikor vetünk véget a keresésnek, és vesszük az addig talált legjobb tervet. Ha például a *C* költség kicsi, akkor még ha esetleg léteznek is sokkal jobb tervek, a megtalálásukra fordított idő meghaladhatja *C*-t, ezért nincs értelme a keresést tovább folytatni. Ha azonban a *C* nagy, akkor bölcs dolog még arra időt fordítani, hogy egy jobb tervet keressünk.

Hegymászás

Ebben a módszerben, ahol valójában egy „völgyet" keresünk a fizikai tervek és költségeik tartományában, egy heurisztikusan kiválasztott fizikai tervvel kezdünk. A terven ezután kis változtatásokat hajtunk végre, például egy operátorhoz adott módszert egy másikkal helyettesítünk, vagy a kommutatív és/vagy asszociatív szabályok segítségével átrendezzük az összekapcsolásokat, hogy kisebb költségű „közeli" terveket találjunk. Amikor elérünk egy olyan tervhez, amelynek semmilyen kis változtatása nem eredményez alacsonyabb költségű tervet, a tervet kikiáltjuk a választott fizikai lekérdezéstervnek.

Dinamikus programozás

Az általános alulról felfelé stratégia e változatában minden egyes részkifejezés esetében csak a legkisebb költségű tervet tartjuk meg. Amint haladunk felfelé a fában, megvizsgáljuk az egyes csomópontok lehetséges megvalósításait, mindegyik részkifejezéshez a legjobb tervet feltételezve. A 7.6. részben kimerítően tárgyaljuk ezt a megközelítést.

Selinger-féle optimalizálás

Ez a megközelítés a dinamikus programozás módszeren javít annyiban, hogy az egyes részkifejezésekhez nemcsak a legalacsonyabb költségű tervet tartja meg, hanem bizonyos egyéb terveket is, amelyek magasabb költségűek ugyan, de amelyek a kifejezésfa feljebb eső részeinél jól kihasználható rendezettséggel rendelkező eredményt állítanak elő. Ilyen számunkra érdekes rendezettségre példa, amikor a részkifejezés eredménye az alábbiak valamelyike szerint van rendezve:

1. Egy gyökérben elhelyezkedő rendezés (τ) operátorban megadott attribútum(ok).
2. Egy későbbi csoportosítás (γ) operátor csoportosítási attribútuma(i).
3. Egy későbbi összekapcsolás összekapcsolási attribútuma(i) szerint.

Ha egy terv költségét a közbülső relációk méretei összegének vesszük, akkor egy argumentum rendezettsége nem látszik előnyösnek. Ha azonban az ennél pontosabb becslést – a lemez I/O-műveletek számát – használjuk költségként, akkor egy argumentum rendezettségének előnye világossá válik, amennyiben használhatjuk a 6.5. rész rendezettségen alapuló algoritmusainak valamelyikét, és a már rendezett argumentumra megtakaríthatjuk az első menetet.

7.5.5. Feladatok

7.5.1. feladat: Becsüljük meg az $R(a, b) \bowtie S(b, c)$ összekapcsolás méretét, használva az $R.b$-hez és az $S.b$-hez tartozó hisztogramokat. Tegyük fel, hogy $V(R, b) = V(S, b) = 20$, és a két attribútumhoz tartozó mindkét hisztogram megadja a négy leggyakoribb elem előfordulási gyakoriságát az alábbi táblázatnak megfelelően:

	0	*1*	*2*	*3*	*4*	*egyéb*
R.b	5	6	4	5		32
S.b	10	8	5		7	48

Hogyan viszonyul ez a becslés ahhoz az egyszerűbb becsléshez, amikor azt feltételezzük, hogy mind a 20 érték egyenlő valószínűséggel fordul elő? Legyen $T(R) = 52$ és $T(S) = 78$.

* **7.5.2. feladat:** Becsüljük meg az $R(a, b) \bowtie S(b, c)$ összekapcsolás méretét, ha a következő hisztograminformáció áll rendelkezésünkre:

	$b < 0$	$b = 0$	$b > 0$
R	500	100	400
S	300	200	500

! **7.5.3. feladat:** A 7.31. példában azt sugalltuk, hogy az egyik b attribútumhoz tartozó értékek számának csökkentése a 7.27. ábra a) tervét az ábra b) tervénél jobbá tenné. A

* a) $V(R, b)$
b) $V(S, b)$

mely értékei esetén lesz az a) tervnek alacsonyabb becsült költsége, mint a b) tervnek?

! **7.5.4. feladat:** Vegyünk négy relációt: R, S, T és V. Ezek rendre 200, 300, 400 és 500 sort tartalmaznak, amely sorokat véletlenszerűen és függetlenül választjuk ugyanabból az 1000 sorból álló készletből. Egy adott sor R-beli előfordulásának valószínűsége például 1/5, S-beli valószínűsége pedig 3/10. Annak valószínűsége, hogy egy sor az R-ben és az S-ben is szerepel: 3/50.

* a) Mi az $R \cup S \cup T \cup V$ várható mérete?
b) Mi az $R \cap S \cap T \cap V$ várható mérete?
c) Az egyesítések melyik sorrendjéhez tartozik a legkisebb költség (a közbülső relációk méreteinek összegére adott becslés)?
d) A metszetek melyik sorrendjéhez tartozik a legkisebb költség (a közbülső relációk méreteinek összegére adott becslés)?

! **7.5.5. feladat:** Ismételjük meg a 7.5.4. feladatot azzal a változtatással, hogy mind a négy relációnak 500 sora legyen, véletlenszerűen választva az 1000 sorból.[7]

!! **7.5.6. feladat:** Tegyük fel, hogy ki akarjuk számolni a következő kifejezést:

$$\tau_b(R(a, b) \bowtie S(b, c) \bowtie T(c, d))$$

Összekapcsolunk tehát három relációt, és az eredményt a b attribútum szerint rendezve állítjuk elő. Éljünk az alábbi egyszerűsítő feltételezésekkel:

i) Nem az R-et és T-t „kapcsoljuk össze" először, mert az egy szorzat.
ii) Bármelyik másik összekapcsolás elvégezhető egy kétmenetes rendezéses összekapcsolással vagy tördelő összekapcsolással, de máshogy nem.
iii) Bármely reláció vagy bármely kifejezés eredménye rendezhető egy kétfázisú, sokágú fésüléses rendezéssel, de máshogy nem.

[7] E feladat megfelelő részeinek megoldása *nincs* a weben közzétéve.

iv) Az első két reláció összekapcsolásának eredményét nem tároljuk ideiglenesen a lemezen, hanem blokkonként adódik át az utolsó összekapcsolásnak, mint annak argumentuma.
v) Mindegyik reláció 1000 blokkot foglal el, és bármelyik két reláció összekapcsolásának eredménye 5000 blokkot foglal el.

Ezen előfeltevések mellett, válaszoljuk meg a következőket:

* a) Melyek azok a részkifejezések és sorrendek, amelyeket egy Selinger-módszerrel történő optimalizálás tekintetbe venne?
b) Lemez I/O-műveletek számát tekintve költségbecslésnek[8], melyik lekérdezésterv jár a legkisebb költséggel.

!! **7.5.7. feladat:** Adjunk egy példát egy $E \bowtie F$ alakú logikai lekérdezéstervre, ahol E és F kifejezések (amiket megválaszthatunk), amikor az E és F kiértékelésére használt legjobb tervek nem engedik meg a végső összekapcsolásra vonatkozóan olyan algoritmus választását, amely a teljes kifejezés kiértékelésének összköltségét minimalizálná. Bármilyen feltevéssel élhetünk a rendelkezésre álló memóriapufferekkel, valamint az E-ben és F-ben említett relációk méreteivel kapcsolatban.

7.6. Összekapcsolások sorrendjének megválasztása

Ebben a részben a költség alapú optimalizálás egyik kritikus problémájára összpontosítunk: az összekapcsolási sorrend megválasztására három vagy több reláció (természetes) összekapcsolása esetén. Hasonló kérdések megfogalmazhatók más bináris műveletek kapcsán is, mint amilyen az egyesítés és metszet, de ezek a műveletek nem annyira jelentősek a gyakorlatban, mivel végrehajtásuk általában kevesebb időt igényel, mint az összekapcsolásé, és ritkábban is szerepelnek három vagy több argumentummal.

7.6.1. Összekapcsolások bal és jobb oldali argumentumainak jelentősége

Egy összekapcsolás sorrendbe állításakor tudnunk kell, hogy a 6. fejezetben tárgyalt összekapcsolási módszerek közül sok aszimmetrikus abban az értelemben, hogy a két argumentum reláció szerepe különböző, és az összekapcsolás költsége függ attól, hogy melyik reláció játssza melyik szerepet. A 6.3.3. rész talán legfontosabb egyme-

[8] Vegyük észre, hogy mivel tettünk néhány nagyon specifikus kitételt a használandó összekapcsolási módszerrel kapcsolatban, becsülhetjük a lemez I/O-műveleteket ahelyett, hogy az egyszerűbb, de pontatlanabb, a sorok számát figyelembe vevő költségbecsléssel dolgoznánk.

netes összekapcsolása beolvassa az egyik relációt – lehetőleg a kisebbet – a központi memóriába, létrehozva mondjuk egy tördelőtábla struktúrát, hogy elősegítse a másik relációból származó sorok illesztését. Ezután beolvassa a másik relációt, egyszerre egy blokkot, és annak sorait összekapcsolja a memóriában tárolt sorokkal.

Tegyük fel, hogy egy fizikai terv kiválasztásakor egy egymenetes összekapcsolás használata mellett döntünk. Ekkor az összekapcsolás bal argumentumát tekintjük annak a (kisebb) relációnak, amelyiket a központi memória adatszerkezetében tárolni fogunk (ezt a relációt nevezzük az *építő relációnak*), míg az összekapcsolás jobb argumentumát (a *vizsgáló relációt*) blokkonként olvassuk be, és illesztjük annak sorait a tárolt reláció soraival. Az argumentumokat a következő összekapcsolási algoritmusok is megkülönböztetik:

1. Beágyazott ciklusú összekapcsolás, ahol azt feltételezzük, hogy a bal argumentum a külső ciklus relációja.
2. Indexes összekapcsolás, ahol azt feltételezzük, hogy a jobb argumentum rendelkezik az indexszel.

7.6.2. Összekapcsolási fák

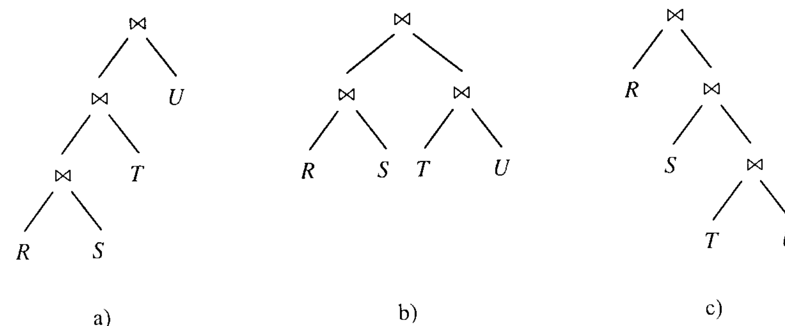
Ha vesszük két reláció összekapcsolását, sorba kell rendezni az argumentumokat. Megegyezés alapján a kisebb becsült mérettel rendelkezőt tekintjük a bal argumentumnak. Megjegyezzük, hogy a fent említett – egymenetes, beágyazott ciklusú, illetve indexes – algoritmusok mindegyike akkor működik a legjobban, ha a bal argumentum a kisebb. Pontosabban szólva, az egymenetes és a beágyazott ciklusú összekapcsolásban egy speciális szerep jut a kisebb relációnak (építő reláció, illetve külső ciklus), az indexes összekapcsolás pedig csak akkor egy ésszerű választás, ha az egyik reláció kicsi, a másiknak pedig van egy indexe. Eléggé általános ezeknél, hogy jelentős és látható különbség van az argumentumok méreteiben, ugyanis egy összekapcsolásokat tartalmazó lekérdezés nagyon gyakran tartalmaz legalább egy attribútumra vonatkozó kiválasztást is, és az a kiválasztás nagyban csökkenti az egyik reláció becsült méretét.

7.32. példa: Vegyük elő újra a 7.4. ábra következő lekérdezését:

```
SELECT filmCím
FROM SzerepelBenne, FilmSzínész
WHERE színészNév = név AND
   születési_idő LIKE '%1960';
```

Az ehhez tartozó, előnyben részesített logikai lekérdezéstervet a 7.21. ábrán látjuk, ahol a `SzerepelBenne` relációnak és a `FilmSzínész` relációra vonatkozó kiválasztás eredményének összekapcsolását vesszük. A `SzerepelBenne` és `FilmSzínész` relációk méreteire nem adtunk becslést, de feltételezhetjük, hogy egy adott évben született színészek kiválasztása a `FilmSzínész` reláció sorainak körülbelül 1/50 részét állítja elő. Mivel általában több színész szerepel egy-egy filmben, a `SzerepelBenne` reláció várhatóan nagyobb, mint a `FilmSzínész` reláció, elmondható tehát, hogy $\sigma_{születési_idő}$ `LIKE '%1960'`(`FilmSzínész`), azaz az összekapcsolás második argumentuma sokkal kisebb, mint a `SzerepelBenne` első argumentum. Ebből azt a következtetést vonhatjuk le, hogy a 7.21. ábrán meg kellene fordítani az argumentumok sorrendjét, hogy a `FilmSzínész` reláción végrehajtott kiválasztás legyen a bal argumentum. □

Két reláció esetén az összekapcsolási fára mindössze két lehetséges választás létezik – a két reláció valamelyikét tesszük bal argumentummá. Gyorsan nő a lehetséges összekapcsolási fák száma, ha az összekapcsolás kettőnél több relációt foglal magában. A 7.28. ábra például három lehetséges szerkezetű fát mutat arra az esetre, amikor az R, S, T és U négy relációt kapcsoljuk össze. Mind a három bemutatott fában azonban ábécérendben szerepel ez a négy reláció balról jobbra nézve. Mivel számít az argumentumok sorrendje, és n dolgot $n!$ módon rendezhetünk, a levelek címkézési lehetőségeit tekintve mindegyik fa 4! = 24 különböző fát képvisel.



7.28. ábra. *Változatok négy reláció összekapcsolására*

7.6.3. Bal-mély összekapcsolási fák

A 7.28.a) ábra egy példája annak, amit *bal-mély* fának nevezünk. Általánosan mondva, egy bináris fa bal-mély, ha minden jobb gyerek levél. Hasonlóképpen, egy olyan fát, mint a 7.2.8.c) ábrán látható, amelynek minden bal gyereke levél, *jobb-mély* fának nevezünk. Egy olyan fát, mint a 7.2.8.b) ábrán látható, amely se nem bal-mély, se nem jobb-mély, *bozótszerűnek* (bushy) nevezünk. Kettős előnye is van annak, ha lehetséges összekapcsolási sorrendként csak bal-mély fákat veszünk figyelembe, amint ezt az alábbi érveink alátámasztják:

1. Egy adott számú levéllel rendelkező lehetséges bal-mély fák száma nagy, de közel sem olyan nagy, mint a lehetséges fák teljes száma. Nagyobb lekérdezésekhez tartozó lekérdezésterveket kereshetünk, ha a keresést a bal-mély fákra korlátozzuk.

2. Az összekapcsolásokhoz használt bal-mély fák jól együttműködnek a szokásos összekapcsolási algoritmusokkal – különösen a beágyazott ciklusú összekapcsolásokkal és az egymenetes összekapcsolásokkal. A bal-mély fákon alapuló keréstervek plusz ezek az algoritmusok várhatóan hatékonyabbak lesznek, mint amikor ugyanezeket az algoritmusokat nem bal-mély fákkal használjuk.

Egy bal- vagy jobb-mély összekapcsolási fa „levelei" valójában lehetnek olyan közbülső csomópontok is, ahol az ott szereplő operátor nem összekapcsolás. Technikai szempontból a 7.21. ábra például egy bal-mély összekapcsolási fa egyetlen összekapcsolás operátorral. Az, hogy az összekapcsolás jobb argumentumára egy kiválasztást alkalmazunk, nem változtat azon a tényen, hogy a fa a bal-mély fák osztályába tartozik.

A bal-mély fák száma közel sem olyan mértékben nő, mint egy adott számú relációt magában foglaló sokargumentumos összekapcsoláshoz tartozó összes fa száma. n relációhoz egyetlen bal-mély fa forma létezik, amelyhez $n!$ módon rendelhetjük hozzá a relációkat. Ugyanennyi jobb-mély fa létezik n relációhoz. Az n relációhoz tartozó fa formák teljes számát viszont, amit $T(n)$-nel jelölünk, az alábbi módon kapjuk meg:

$$T(1) = 1$$

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i)$$

A második egyenlőséget az magyarázza, hogy egy tetszőleges 1 és $n-1$ közé eső i számnak vehetjük a gyökér bal oldali részfájában szereplő levelek számát, és azok a levelek $T(i)$ különböző módon rendezhetők el. Hasonlóképpen, a jobb oldali részfában szereplő $n-i$ számú levél $T(n-i)$ módon rendezhető el.

Íme a $T(n)$ első néhány értéke: $T(1) = 1$, $T(2) = 1$, $T(3) = 2$, $T(4) = 5$, $T(5) = 14$ és $T(6) = 42$. A fák teljes száma, amikor már a relációkat is hozzárendeljük a levelekhez, a $T(n)$ és az $n!$ szorzataként adódik. A 6 levelet tartalmazó megcímkézett levelű fák száma $42 \times 6! = 30\,240$, amelyek között van 720 bal-mély fa, és másik 720 jobb-mély fa.

Nézzük most a bal-mély összekapcsolási fákkal kapcsolatban említett második előnyt, nevezetesen, hogy várhatóan hatékony terveket állítanak elő. Két példát közlünk erre vonatkozóan:

1. Ha egymenetes összekapcsolásokat használunk, és az építő reláció a bal oldalon van, akkor az egy időben szükséges memóriamennyiség feltehetően kisebb, mint ha egy jobb-mély vagy bozótszerű fát használnánk ugyanazokra relációkra.
2. Ha iterátorokkal megvalósított beágyazott ciklusú összekapcsolásokat használunk, akkor elkerüljük azt, hogy valamely közbülső relációt egynél többször kelljen létrehozni.

7.33. példa: Vegyük a 7.28.a) ábrán látható bal-mély fát, és tegyük fel, hogy mind a három összekapcsoláshoz egy egyszerű egymenetes összekapcsolást fogunk használni, mégpedig úgy, hogy a bal argumentum az építő reláció, vagyis a bal argumentumokat tartjuk a központi memóriában. Az $R \bowtie S$ kiszámításához az R relációt kell a központi memóriában tartanunk, és amint kiszámítottuk az $R \bowtie S$-t, annak eredményét is a központi memóriában kell tartanunk. Tehát $B(R) + B(R \bowtie S)$ számú központi memóriapufferre van szükségünk. Ha R-et vesszük a legkisebb relációnak, és egy kiválasztás eléggé kicsivé tette az R-et, akkor valószínűleg nem lesz probléma, hogy rendelkezésre álljon ennyi puffer.

Az $R \bowtie S$ kiszámítása után az eredményül kapott relációt össze kell kapcsolni a T-vel. Az R tárolásához használt pufferekre már nincs tovább szükség, így azok újra felhasználhatók a $(R \bowtie S) \bowtie T$ eredményének tárolásához. Hasonlóképpen, amikor ezt a relációt kapcsoljuk össze az U-val, az $R \bowtie S$ relációra már nincs tovább szükség, így az ehhez használt pufferek újra felhasználhatók a végső összekapcsolás eredményének tárolásához. Általánosan azt mondhatjuk, hogy egy egymenetes összekapcsolással kiszámított bal-mély összekapcsolási fa feltételezi, hogy legalább két ideiglenes reláció tárolásához szükséges hely mindig rendelkezésre áll a központi memóriában.

Most vizsgáljuk meg a 7.28.c) ábrán található jobb-mély fa egy hasonló megvalósítását. Elsőként az R relációt kell betölteni a központi memória puffereibe, mivel mindig a bal argumentum az építő reláció. Ezután létre kell hozni az $S \bowtie (T \bowtie U)$ relációt, és a gyökérben lévő összekapcsoláskor ezt kell használni vizsgáló relációként. Az $S \bowtie (T \bowtie U)$ kiszámításához be kell vinni az S-et pufferekbe, és ki kell számolni a $T \bowtie U$ összekapcsolást mint a hozzá tartozó vizsgáló relációt. A $T \bowtie U$ kiszámításához viszont először pufferekbe kell betölteni a T-t. Most tehát az R, S és T három relációt tároljuk egy időben a központi memóriában. Általánosan fogalmazva, ha egy n levéllel rendelkező jobb-mély összekapcsolási fát akarunk kiértékelni, akkor egyszerre $n-1$ relációt kell a központi memóriában tárolni.

Természetesen előfordulhat, hogy a $B(R) + B(S) + B(T)$ összméret kisebb, mint a bal-mély fa kiszámítása során a két közbülső lépéshez szükséges helymennyiség, amelyek rendre $B(R) + B(R \bowtie S)$ és $B(R \bowtie S) + B((R \bowtie S) \bowtie T)$.[9] Ahogy azonban a 7.32. példában rámutattunk, a több összekapcsolást tartalmazó lekérdezéseknél gyakran van egy kis reláció, amely lehet a legbalra eső argumentum egy bal-mély fában. Ha R kicsi, akkor az $R \bowtie S$-től azt várhatjuk, hogy lényegesen kisebb, mint az S, az $(R \bowtie S) \bowtie T$-től pedig azt, hogy kisebb, mint a T, ami csak további igazolását jelenti a bal-mély fák használatának. □

7.34. példa: Tegyük fel, hogy a 7.28. ábra négyes összekapcsolását beágyazott ciklusú összekapcsolásokkal szándékozzuk megvalósítani, és hogy az abban szereplő mindhárom összekapcsoláshoz egy-egy iterátort használunk. Az egyszerűség kedvéért azt is tegyük fel, hogy az R, S, T és U relációk mindegyike tárolt reláció, nem pedig kifejezés. Ha a 7.28.a) ábrán szereplő bal-mély fát használjuk, akkor a gyökérhez tartozó iterátor az $(R \bowtie S) \bowtie T$ bal argumentum egy, a központi memória méretének megfelelő darabját kapja meg. Majd ezt a darabot összekapcsolja a teljes U-val, de amennyiben az U egy tárolt reláció, az U-t csak végig kell olvasni, és nem kell előállítani. Amikor megkapja és a memóriába helyezi a bal argumentum következő darabját, újra be kell olvasni az U-t, de a beágyazott ciklusú összekapcsolásnál szükség van erre az ismétlésre, és nem kerülhető el, ha mindkét argumentum nagy.

[9] Vegyük észre, hogy egy kifejezésfa eredményének tárolási költségét nem számoljuk bele a költség becslésébe, ahogy ezt máskor sem tettük.

A pufferkezelő szerepe

Észreveheti az olvasó a különbséget a 6.14. és 6.17. példában látott megközelítések között, ahol az összekapcsoláshoz rendelkezésre álló központi memóriapufferek számára egy rögzített korlátot feltételeztünk, és az itteni rugalmasabb feltételezés között, hogy annyi puffer áll rendelkezésre, amennyi szükséges, de próbálunk nem „túl sokat" használni. Elevenítsük fel a 6.8. részben mondottakat, miszerint a pufferkezelő jelentős rugalmassággal rendelkezik a műveletekhez szükséges pufferek kiosztása ügyében. Ütközések állhatnak azonban elő, ha egyszerre túl sok puffer kerül kiosztásra, ami leronthatja a használt algoritmus feltételezett hatékonyságát.

Ehhez hasonlóan, az $(R \bowtie S) \bowtie T$ egy darabjának megszerzéséhez megkapjuk az $R \bowtie S$ egy darabját a memóriában, és átolvassuk a T-t. A T többszöri átolvasására szükség lehet, hacsak nem elkerülhető. Végül az $R \bowtie S$ egy darabjának megszerzése az R egy darabjának beolvasását és S-sel történő összehasonlítását igényli, esetleg többször. Mindezen tevékenységek során azonban csak tárolt relációkat olvasunk többször, és ez az ismételt olvasás akkor válik a beágyazott ciklusú összekapcsolás működésének részévé, amikor a központi memória nem elég nagy ahhoz, hogy egy teljes reláció elférjen benne.

Hasonlítsuk most össze a bal-mély fához tartozó iterátorok viselkedését a 7.28.c) ábrán szereplő jobb-mély fára vonatkozó iterátorok viselkedésével. A gyökérhez tartozó iterátor az R egy darabjának beolvasásával kezd. Majd meg kell konstruálnia a teljes $S \bowtie (T \bowtie U)$ relációt, és azt össze kell hasonlítania az R megkapott darabjával. Amikor az R következő darabját olvassuk be a memóriába, az $S \bowtie (T \bowtie U)$ relációt újra létre kell hozni. Az R minden egyes további darabjának beolvasásakor újra és újra elő kell állítani ugyanezt a relációt.

Természetesen azt megtehetnénk, hogy az $S \bowtie (T \bowtie U)$ relációt egyszer létrehozzuk és eltároljuk vagy a memóriában, vagy lemezen. Ha lemezen tároljuk el, akkor a bal-mély fának megfelelő tervhez képest extra lemez I/O-műveleteket használunk, ha pedig a memóriában tároljuk, akkor belefutunk a memória túlhasználatának a 7.33. példában tárgyalt problémájába. □

7.6.4. Dinamikus programozás az összekapcsolási sorrend és csoportosítás megválasztására

Több reláció összekapcsolásakor a sorrend megválasztására három lehetőségünk van:

1. Az összes lehetséges sorrendet figyelembe vesszük.
2. Egy részhalmazt veszünk figyelembe.
3. Használunk egy heurisztikát egy sorrend kiválasztására.

Ebben a részben a felsorolás egy ésszerű megközelítését tárgyaljuk, amit *dinamikus programozásnak* nevezünk. Ez vagy arra használható, hogy az összes sorrendet megvizsgáljuk, vagy arra, hogy csak bizonyos részhalmazokat vizsgáljunk meg, mint amilyen például a bal-mély fákra leszűkített sorrendek részhalmaza. A 7.6.6. részben egy olyan elfogadható heurisztikát nézünk majd meg, amely egyetlen sorrend kiválasztására szolgál. A dinamikus programozás egy általános algoritmusminta.[10] A dinamikus programozás mögött az az alapötlet áll, hogy kitöltünk egy táblázatot a költségekről úgy, hogy csak a sikeres következtetéshez szükséges minimális információt őrizzük meg.

Tegyük fel, hogy végre akarjuk hajtani az $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$ összekapcsolást. A dinamikus programozás algoritmusában létrehozunk egy olyan táblázatot, amelyben van egy bejegyzés az n reláció közül egyet vagy többet tartalmazó minden egyes részhalmazhoz. Ebben a táblázatban az alábbiakat helyezzük el:

1. Ezen relációk összekapcsolásának becsült mérete. Ennek meghatározásához a 7.4.6. részben szereplő formulát használhatjuk.
2. Ezeknek a relációknak az összekapcsolásához szükséges legkisebb költség. A példáinkban a közbülső relációk méreteinek összegét fogjuk használni (nem beleértve magukat az R_i relációkat és a táblázat aktuális bejegyzéséhez tartozó összes reláció összekapcsolását). Emlékezzünk vissza, hogy a köztes relációk méretei adják a legegyszerűbb használható becslést a lemez I/O-műveletek, CPU-használat és egyéb tényezők költségére. Más, bonyolultabb becslést is használhatnánk azonban, mint például a teljes lemez I/O-műveletek száma, ha hajlandók és képesek lennénk a szükséges extra számítások elvégzésére. Ha a lemez I/O-műveleteket vagy a futási idő más becslését használjuk, akkor figyelembe kell venni az adott összekapcsoláshoz használt algoritmust is, mivel a különböző algoritmusok különböző költséggel járnak. Ezeket a kérdéseket a dinamikus programozási technikák alapjainak megtanulása után fogjuk megtárgyalni.
3. Az a kifejezés, amelyik a legkisebb költséget adja. Ez a kifejezés a kérdéses relációkat kapcsolja össze, valamilyen csoportosításban. Opcionálisan korlátozhatjuk magunkat a bal-mély kifejezésekre, amikor is a kifejezés csak egy sorrendje a relációknak.

A táblázatot a részhalmaz méretére vonatkozó indukció alapján konstruáljuk meg. Két változat létezik, attól függően, hogy a fák összes lehetséges alakjait figyelembe kívánjuk-e venni, vagy csak a bal-mély fákat. Különbség van ugyanis a táblázat megalkotásának módjában, amely különbséget majd akkor magyarázzuk el, amikor a táblázat megkonstruálásának az indukciós lépését tárgyaljuk.

Alap: Az egyetlen R relációhoz tartozó bejegyzés az R méretéből, egy 0 költségből és abból a formulából áll, ami maga az R. Relációk egy $\{R_i, R_j\}$ párjára is könnyű ki-

[10] A dinamikus programozás általános tárgyalása megtalálható a következő irodalomban is: A. V. Aho, J. E. Hopcroft and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1984.

számítani a bejegyzést. A költség 0, mivel nincsenek közbülső relációk. A méret becslését a 7.4.6. részben szereplő szabály adja meg: az R_i és R_j méreteit összeszorozzuk, majd az R_i és R_j minden közös attribútuma esetén, ha van ilyen, osztunk az attribútumhoz tartozó értékhalmazok méretei közül a nagyobbal. A formula vagy $R_i \bowtie R_j$, vagy $R_j \bowtie R_i$. A 7.6.1. részben bevezetett elvnek megfelelően az R_i és R_j közül a kisebbet vesszük bal argumentumnak.

Indukció: Most építhetjük tovább a táblázatot, kiszámítva a bejegyzéseket minden részhalmazhoz, amelyek mérete 3, 4 és így tovább, amíg el nem jutunk az egyetlen n méretű részhalmaz bejegyzéséhez. Egy ilyen bejegyzés mondja meg, hogy melyik a legjobb módszer az ott szereplő összes reláció összekapcsolásának kiszámítására. Megadja továbbá annak a módszernek a becsült költségét is, amire szükségünk van, amikor a későbbi bejegyzéseket számoljuk. Azt kell megnézni, hogy hogyan számítunk ki egy k relációból álló $\mathcal{R}$ halmazhoz tartozó bejegyzést.

Ha csak bal-mély fákat szándékozunk figyelembe venni, akkor a k relációból álló $\mathcal{R}$ minden egyes R relációja esetén azt a lehetőséget vesszük, amikor az $\mathcal{R}$-hez tartozó összekapcsolást úgy számoljuk ki, hogy először kiszámítjuk az $\mathcal{R} - \{R\}$-hez tartozó összekapcsolást, majd ezt összekapcsoljuk az R-rel. Az $\mathcal{R}$-hez tartozó összekapcsolás költsége az $\mathcal{R} - \{R\}$-hez tartozó költségnek és az utóbbi összekapcsolás méretének összege lesz. A legkisebb költséget eredményező R-et választjuk. Az $\mathcal{R}$-hez tartozó kifejezés bal argumentuma az $\mathcal{R} - \{R\}$-hez tartozó legjobb összekapcsolási kifejezés, a jobb argumentuma pedig az R lesz. Az $\mathcal{R}$-hez tartozó méret az lesz, amit a 7.4.6. rész formulája ad.

Ha az összes fát figyelembe akarjuk venni, akkor a relációk egy $\mathcal{R}$ részhalmazához tartozó bejegyzés kiszámítása valamivel összetettebb. Meg kell vizsgálni az összes lehetséges módját az $\mathcal{R}$ halmaz $\mathcal{R}_1$ és $\mathcal{R}_2$ teljesen elkülönülő részhalmazokra történő particionálásának. Minden ilyen particionálás esetén vesszük az alábbiak összegét:

1. Az $\mathcal{R}_1$ és $\mathcal{R}_2$ legjobb költségei.
2. Az $\mathcal{R}_1$ és $\mathcal{R}_2$ méretei.

A legjobb költséget adó partíció esetén ezt az összeget az $\mathcal{R}$-hez tartozó költségnek vesszük. Az $\mathcal{R}$-hez tartozó formula az $\mathcal{R}_1$-hez és az $\mathcal{R}_2$-höz tartozó legjobb sorrendű összekapcsolások összekapcsolása lesz.

7.35. példa: Tekintsük az R, S, T és U négy reláció összekapcsolását. Az egyszerűség kedvéért feltételezzük, hogy mindegyiknek 1000 sora van. A relációk attribútumait és az egyes relációkban az attribútumokhoz tartozó értékhalmazok becsült méreteit a 7.29. ábra adja meg.

Az egyelemű halmazokra vonatkozó méreteket, költségeket és legjobb terveket a 7.30. ábrán látható táblázat tartalmazza. Minden egyes relációra a méret a megadott 1000, a költség 0, mivel nincsenek közbülső relációk, a legjobb (és az egyetlen) kifejezés pedig maga a reláció.

$R(a, b)$	$S(b, c)$	$T(c, d)$	$U(d, a)$
$V(R, a) = 100$			$V(U, a) = 50$
$V(R, b) = 200$	$V(S, b) = 100$		
	$V(S, c) = 500$	$V(T, c) = 20$	
		$V(T, d) = 50$	$V(U, d) = 1000$

7.29. ábra. *Paraméterek a 7.35. példához*

	$\{R\}$	$\{S\}$	$\{T\}$	$\{U\}$
Méret	1000	1000	1000	1000
Költség	0	0	0	0
Legjobb terv	R	S	T	U

7.30. ábra. *Az egyelemű halmazokhoz tartozó táblázat*

Vegyük sorra most a relációpárokat. Mindegyikhez 0 költség tartozik, hiszen két reláció összekapcsolásakor még mindig nincsenek közbülső relációk. Két lehetséges terv létezik egy-egy párhoz, a két reláció valamelyike lehet a bal argumentum, de mivel mindegyik reláció ugyanolyan méretű, nincs szempont a kezünkben, ami alapján a tervek közül választhatnánk. Az ábécérendnek megfelelő első relációt vesszük hát bal argumentumnak mindegyik esetben. Az eredményül kapott relációk méreteit a szokásos képlet alapján számoljuk ki. A 7.31. ábrán bemutatott táblázat összegzi mindezeket az eredményeket. Megjegyezzük, hogy az 1 M (Mega) jelentése: 1 000 000, az olyan „összekapcsolások" mérete, amelyek valójában szorzatok.

	$\{R, S\}$	$\{R, T\}$	$\{R, U\}$	$\{S, T\}$	$\{S, U\}$	$\{T, U\}$
Méret	5000	1 M	10 000	2000	1M	1000
Költség	0	0	0	0	0	0
Legjobb terv	$R \bowtie S$	$R \bowtie T$	$R \bowtie U$	$S \bowtie T$	$S \bowtie U$	$T \bowtie U$

7.31. ábra. *A relációpárokhoz tartozó táblázat*

Nézzük most a négy reláció közül hármat magában foglaló összekapcsolásokhoz tartozó táblázatot. Három reláció összekapcsolásának kiszámításakor először ki kell választani közülük kettőt, amelyeket összekapcsolunk. Az eredmény méretének becslését a szokásos formulával számoljuk ki, ennek a számításnak a részleteitől most eltekintünk. Emlékezzünk vissza, hogy ugyanazt a méretet kapjuk, függetlenül az összekapcsolás kiszámításának módjától.

Minden relációhármas esetén a költség az egyetlen közbülső relációnak – az elsőként kiválasztott két reláció összekapcsolásának – a mérete. Mivel azt akarjuk, hogy ez a költség a lehető legkisebb legyen, megvizsgáljuk a három relációból származó összes lehetséges relációpárt, és a legkisebb méretet eredményező párt választjuk.

A formula meghatározásakor először a két választott relációt csoportosítjuk egybe, melyek még lehetnek vagy bal, vagy jobb argumentumok. Tegyük fel, hogy csak a bal-mély fákat vesszük figyelembe, vagyis a első két reláció összekapcsolását mindig bal argumentumként használjuk. Mivel két reláció összekapcsolásának becsült mérete

minden esetben legalább 1000 (az egyes relációk mérete), ha megengednénk nem bal-mély fákat, akkor mindig az egyedül álló relációt választanánk bal argumentumnak a példánkban. A hármasokhoz tartozó táblázatot a 7.32. ábra mutatja.

	$\{R, S, T\}$	$\{R, S, U\}$	$\{R, T, U\}$	$\{S, T, U\}$
Méret	10 000	50 000	10 000	2000
Költség	2 000	5 000	1 000	1000
Legjobb terv	$(S \bowtie T) \bowtie R$	$(R \bowtie S) \bowtie U$	$(T \bowtie U) \bowtie R$	$(T \bowtie U) \bowtie S$

7.32. ábra. *A relációhármasokhoz tartozó táblázat*

Példaként vizsgáljuk meg az $\{R, S, T\}$ hármashoz tartozó számításokat. Meg kell néznünk az ezekből képezhető három pár mindegyikét. Ha az $R \bowtie S$-sel kezdünk, akkor a költség ennek a relációnak a mérete, ami 5000, amint ez a 7.31. ábrán látható, a párokra vonatkozó táblázat alapján tudjuk. Az $R \bowtie T$ kezdés 1 000 000-t ad költségként, ha pedig az $S \bowtie T$-vel kezdünk, akkor a költség 2000. A három lehetőség közül a legutóbbi jelenti a legalacsonyabb költséget, így azt a tervet választjuk. Ez a választás nemcsak az $\{R, S, T\}$ oszlop költség rovatában tükröződik, hanem a legjobb terv sorban is, ahol is az a terv jelenik meg, amelyik először az S-et és a T-t csoportosítja össze.

Most azt a helyzetet vizsgáljuk meg, amikor mind a négy relációt összekapcsoljuk. Ennek a relációnak a becsült mérete 100 sor, az igazi költség lényegében a közbülső relációk létrehozásában áll. Emlékezzünk azonban, hogy tervek összehasonlításakor a végeredmény méretét egyébként sem vesszük figyelembe soha.

A mind a négy relációt magában foglaló összekapcsolás kiszámításának két általános módja van:

1. Kiválasztunk és a lehető legjobb módon összekapcsolunk hármat, majd az eredményt összekapcsoljuk a negyedikkel,
2. A négy relációt kétszer két párra osztjuk, összekapcsoljuk a párokat, majd az eredményeket összekapcsoljuk.

Természetesen, ha csak bal-mély fákkal foglalkozunk, akkor a második típusú terveket kizárjuk, hiszen azok bozótszerű fákat eredményeznek. A 7.33. ábrán látható táblázat összefoglalja az összekapcsolások hét lehetséges csoportosítását, amelyek a 7.31. és 7.32. ábrákon szereplő legjobb csoportosításokra épülnek.

Csoportosítás	*Költség*
$((S \bowtie T) \bowtie R) \bowtie U$	12 000
$((R \bowtie S) \bowtie U) \bowtie T$	55 000
$((T \bowtie U) \bowtie R) \bowtie S$	11 000
$((T \bowtie U) \bowtie S) \bowtie R$	3 000
$(T \bowtie U) \bowtie (R \bowtie S)$	6 000
$(R \bowtie T) \bowtie (S \bowtie U)$	2 000 000
$(S \bowtie T) \bowtie (R \bowtie U)$	12 000

7.33. ábra. *Összekapcsolások csoportosításai és azok költségei*

Nézzük például a 7.33. ábrán található első formulát, amely azt ábrázolja, hogy először összekapcsoljuk az R, S és T relációkat, majd az így kapott eredményt összekapcsoljuk az U-val. A 7.32. ábra táblázatából tudjuk, hogy az R, S és T összekapcsolásának a legjobb módja az, ha először az S-et és a T-t kapcsoljuk össze. Ennek a kifejezésnek a bal-mély alakját használtuk, és hogy a bal-mély formát folytassuk, az U-t jobb oldalról kapcsoltuk hozzá. Ha csak bal-mély fákat engedünk meg, akkor ez a kifejezés és reláció sorrend az egyetlen választási lehetőség. Ha megengednénk bozótszerű fákat is, akkor az U-t bal oldalról kapcsolnánk, ugyanis kisebb, mint a másik három összekapcsolása. Az általunk kapott összekapcsolás költsége 12 000, ami az $(S \bowtie T) \bowtie R$ költségének és méretének az összege, amelyek rendre 2000 és 10 000.

A 7.33. ábra utolsó három kifejezése további lehetőségeket ad, amennyiben bozótszerű fákat is figyelembe veszünk. Ezeket úgy kapjuk, hogy először két relációpárt kapcsolunk össze. Az utolsó sor például azt a megoldást mutatja, hogy először elvégezzük az $R \bowtie U$ és az $S \bowtie T$ összekapcsolásokat, majd összekapcsoljuk azok eredményeit. Ennek a kifejezésnek a költsége a két pár méretének és költségének az összegeként adódik. Mindkét költség 0, amint ez minden pár esetében így kell hogy legyen, a méretek pedig rendre 10 000, illetve 2000. Mivel általában a kisebb relációt választjuk bal argumentumnak, végül az $(S \bowtie T) \bowtie (R \bowtie U)$ kifejezést kapjuk.

Azt látjuk ebben a példában, hogy a legkisebb költség a negyedik formulához tartozik: $((T \bowtie U) \bowtie S) \bowtie R$. Ezt a kifejezést választjuk ki az összekapcsolás kiszámítására, és ennek költsége 3000. Mivel ez egy bal-mély fa, ez a logikai lekérdezésterv kerül kiválasztásra, függetlenül attól, hogy a mi dinamikus programozási stratégiánk mindenféle típusú tervet figyelembe vesz, vagy csak a bal-mély terveket. □

7.6.5. Dinamikus programozás részletesebb költségfüggvényekkel

Egy dinamikus programozási algoritmusban leegyszerűsíti a számításokat az, ha költségbecslésként reláció méreteket használunk. Ennek az egyszerűsítésnek egy hátránya azonban az, hogy a számolás során az összekapcsolásoknak nem a tényleges költségeit vesszük figyelembe. Erre egy szélsőséges példa, amikor egy $R(a, b) \bowtie S(b, c)$ összekapcsolásban az R relációnak egyetlen sora van, az S relációnak pedig van egy indexe a b összekapcsolási attribútumra, ekkor ugyanis az összekapcsolás szinte 0 időbe kerül. Másrészről viszont, ha az S-nek nincs indexe, akkor végig kell azt olvasni, ami $B(S)$ lemez I/O-műveletet igényel, még akkor is, ha az R egyelemű. Egy olyan költségbecslés, ami csak az R, S és $R \bowtie S$ méreteit veszi figyelembe, nem tud különbséget tenni e két eset között, vagyis a csoportosításban az $R \bowtie S$ használatának költségét vagy túlbecsüljük, vagy alulbecsüljük.

Nem nehéz azonban a dinamikus programozási algoritmust úgy módosítani, hogy az összekapcsolási algoritmusokat is számításba vegye. Először is, a használt költségbecslés a lemez I/O-műveletek száma lesz, vagy valamilyen más általunk előnyben részesített futási idő egység. Az $\mathcal{R}_1 \bowtie \mathcal{R}_2$ költségének kiszámításakor összeadjuk az $\mathcal{R}_1$ költségét, az $\mathcal{R}_2$ költségét és e két reláció összekapcsolásának legkisebb költségét, a legjobb rendelkezésre álló algoritmust használva. Mivel az utóbbi költség rendszerint

függ az $\mathcal{R}_1$ és $\mathcal{R}_2$ méreteitől, ezeknek a méreteknek a becslését szintén ki kell számolni, ahogy ezt a 7.35. példában tettük.

A dinamikus programozás egy még hatékonyabb változata a 7.5.4. részben említett Selinger-féle optimalizálásra épül. Ekkor az egyes relációhalmazok esetén, amelyek összekapcsolásra számot tarthatnak, nemcsak egy költséget tartunk meg, hanem több költséget. Emlékezzünk vissza, hogy a Selinger-féle optimalizálás az összekapcsolás eredményének előállításakor nemcsak annak legkisebb költségét veszi figyelembe, hanem azokat a legkisebb költségeket is, amelyek a reláció különféle „érdekes" rendezettségű változatainak előállításához szükségesek. Ezek az érdekes rendezettségek lehetnek olyanok, amelyek egy későbbi rendezéses öszekapcsolás során előnyösen használhatók, vagy olyanok, amelyek felhasználhatók a teljes lekérdezés végeredményének előállításakor, ha a felhasználó valamilyen sorrendnek megfelelő rendezettségben várja a végeredményt. Amikor rendezett relációkat kell előállítani, rendezéses összekapcsolás – legyen az egymenetes vagy többmenetes – használatát figyelembe kell venni mint egy lehetőséget, ha viszont egy eredmény rendezettsége nem értékes szempont, akkor a tördelő összekapcsolások minden esetben legalább olyan jók, mint a megfelelő rendezéses összekapcsolások.

7.6.6. Egy mohó algoritmus az összekapcsolási sorrend kiválasztására

Ahogy a 7.35. példa is mutatja, még a dinamikus programozás gondosan korlátozott keresése is exponenciális számú számításhoz vezet, mint az összekapcsolandó relációk számának egy függvénye. Öt vagy hat reláció összekapcsolásakor az optimális sorrend megtalálásához indokolt egy olyan alapos módszert használni, mint amilyen a dinamikus programozás vagy az elágazás-és-korlátozás keresés. Amikor azonban az összekapcsolások száma túlnő ezen, vagy ha az alapos kereséshez szükséges időt nem akarjuk ráfordítani, akkor használhatunk heurisztikus összekapcsolási sorrendet a lekérdezésoptimalizálóban.

A leggyakrabban választott heurisztika valamilyen *mohó* (greedy) algoritmus, ahol egy adott ponton döntést hozunk az összekapcsolási sorrendre vonatkozóan, és soha nem lépünk vissza vagy vizsgáljuk felül az egyszer már meghozott döntéseket. Egy olyan mohó algoritmust vizsgálunk meg, amely csak bal-mély fákat választ ki. A „mohóság" alapja az az elv, hogy a köztes relációk méretét a lehető legalacsonyabban akarjuk tartani a fa minden szintjén.

Alap: Kezdjük azzal a relációpárral, amelyek összekapcsolásának becsült mérete a legkisebb. Ezeknek a relációknak az összekapcsolás lesz az *aktuális fa* (current tree).

Indukció: A aktuális fában még nem szereplő relációk közül keressük meg azt, amelyiket az aktuális fával összekapcsolva a legkisebb becsült méretű relációt kapjuk. Az új aktuális fa bal argumentuma a régi aktuális fa, jobb argumentuma pedig a kiválasztott reláció lesz.

Összekapcsolás szelektivitása

Hasznos lehet úgy szemlélni az olyan heurisztikákat, mint amilyen a bal-mély fát kiválasztó mohó algoritmus, hogy minden R reláció rendelkezik egy *szelektivitással,* amikor összekapcsoljuk az aktuális fával, ami a következő hányados:

$$\frac{\text{az összekapcsolás eredményének mérete}}{\text{az aktuális fa eredményének mérete}}$$

Mivel rendszerint egyik relációnak sem ismerjük a pontos méretét, ezeket a méreteket becsüljük, mint ahogy ezt az előzőekben is tettük. Az összekapcsolási sorrend egy mohó megközelítése az, hogy a legkisebb szelektivitással rendelkező relációt választjuk ki.

Ha például egy összekapcsolási attribútum kulcsa az R-nek, akkor a szelektivitás legfeljebb 1, ami általában egy kedvező szituáció. Megjegyezzük, hogy a 7.29. ábra statisztikájából ítélve, a d attribútum kulcs az U-ban, és a többi relációnak nincs kulcsa, ami megmagyarázza, hogy miért az a legjobb kezdés, hogy a T-t összekapcsoljuk az U-val.

7.36. példa: Alkalmazzuk a mohó algoritmust a 7.35. példa relációira. Az alaplépés az, hogy megkeressük azt a relációpárt, amelyeknek az összekapcsolása a legkisebb méretű. A 7.31. ábrát megnézve azt látjuk, hogy ez a $T \bowtie U$ összekapcsolásra igaz, amelynek költsége 1000. Vagyis $T \bowtie U$ az „aktuális fa".

Most azt nézzük meg, hogy az R vagy az S relációt kapcsoljuk-e össze a fával következőként. Összehasonlítjuk tehát a $(T \bowtie U) \bowtie R$, illetve a $(T \bowtie U) \bowtie S$ méretét. A 7.32. ábra azt mondja, hogy a második, amelynek mérete 2000, jobb, mint az első, amelynek mérete 10 000. Az új aktuális fa ennek megfelelően a $(T \bowtie U) \bowtie S$ lesz.

Nincs más lehetőségünk, mint hogy az utolsó lépésben az R-t kapcsoljuk össze az eddigivel, aminek során a teljes költség 3000 lesz, ami egyenlő a két közbülső reláció méretének összegével. Vegyük észre, hogy a mohó algoritmus ugyanazt a fát adja eredményül, mint amit a 7.35. példában a dinamikus programozási algoritmus kiválasztott. Léteznek azonban olyan példák, amikor a mohó algoritmus nem találja meg a legjobb megoldást, a dinamikus programozási algoritmus viszont garantáltan megtalálja a legjobbat; lásd 7.6.4. feladat. □

7.6.7. Feladatok

7.6.1. feladat: Vegyük a 7.4.1. feladatban szereplő relációkat, és adjuk meg a dinamikus programozási táblázat bejegyzéseit, amelyek a lehetséges összekapcsolási sorrendek kiértékeléséhez tartoznak, figyelembe véve:

a) Csak bal-mély fákat engedünk meg.
b) Minden fa megengedett.

Melyek a legjobb választások az egyes esetekben?

7.6.2. feladat: Ismételjük meg a 7.6.1. feladatot az alábbi változtatásokkal:

i) A Z sémája $Z(d, a)$-ra változik.
ii) $V(Z, a) = 100$.

7.6.3. feladat: Ismételjük meg a 7.6.1. feladatot a 7.4.2. feladat relációival.

* **7.6.4. feladat:** Vegyük az $R(a, b)$, $S(b, c)$, $T(c, d)$ és $U(a, d)$ relációk összekapcsolását, ahol az R-nek és az U-nak 1000 sora van, az S-nek és a T-nek pedig 100 sora van. Továbbá, minden reláció minden attribútumának 100 értéke van, kivéve a c attribútumot, amelyre $V(S, c) = V(T, c) = 10$.

a) Milyen sorrendet választ ki a mohó algoritmus? Mi a hozzá tartozó költség?
b) Melyik az optimális összekapcsolási sorrend, és mi a hozzá tartozó költség?

7.6.5. feladat: Hány fa létezik

a) hét reláció,
b) nyolc reláció

összekapcsolása esetén? Ezek közül hány se nem bal-mély, se nem jobb-mély?

! **7.6.6. feladat:** Tegyük fel, hogy össze akarjuk kapcsolni az R, S, T és U relációkat a 7.28. ábrán látható fastruktúrák valamelyikének megfelelően, és minden közbülső relációt a memóriában akarunk tartani mindaddig, ameddig szükség van rájuk. A szokásos feltevésünket követve, a négy reláció összekapcsolásának eredményét valamilyen további folyamat felhasználja amint azt előállítottuk, így ehhez a relációhoz nincs szükség memóriára. A tárolt relációkhoz és a köztes relációkhoz szükséges blokkok számával [pl. $B(R)$ vagy $B(R \bowtie S)$] kifejezve adjunk alsó korlátot M-re, a szükséges memóriablokkok számára, ha a kiértékelés az alábbi alapján történik:

* a) 7.28.a) ábrán szereplő bal-mély fa,
b) 7.28.b) ábrán szereplő bozótszerű fa,
c) 7.28.c) ábrán szereplő jobb-mély fa.

Milyen feltételezések alapján következtethetünk arra, hogy egy fa biztosan kevesebb memóriát használ, mint valamelyik másik?

*! **7.6.7. feladat:** A táblázat hány bejegyzését kell kitöltenünk, ha k reláció összekapcsolásakor dinamikus programozást használunk az összekapcsolási sorrend kiválasztására?

7.7. A fizikai lekérdezésterv kiválasztásának befejezése

A lekérdezést elemeztük, átalakítottuk egy kiindulási logikai lekérdezéstervvé, és a 7.3. részben leírt transzformációk segítségével feljavítottuk a logikai lekérdezéstervet. A fizikai lekérdezésterv kiválasztási folyamatának részeként felsoroljuk a lehetséges választásokat, és becsüljük a hozzájuk tartozó költségeket, amit a 7.5. részben tárgyaltunk. A 7.6. rész központi kérdése a sok relációt magában foglaló összekapcsolások felsorolása, költségbecslése és összekapcsolási sorrendje volt. Mintegy ennek kiterjesztéseként, hasonló technikákat használhatunk egyesítések, metszetek vagy bármilyen más kommutatív/asszociatív művelet esetén.

Hátravan még néhány lépés ahhoz, hogy a logikai tervet egy teljes fizikai lekérdezéstervvé alakítsuk. Az alapvető témakörök, amelyekre még ki kell térnünk ebben a részben, a következők:

1. A lekérdezésterv műveleteit megvalósító algoritmusok kiválasztása, feltéve hogy az algoritmus kiválasztása még nem történt meg valamilyen korábbi lépésben. Utóbbira példa egy összekapcsolási sorrend megválasztása dinamikus programozással.
2. Annak eldöntése, hogy a köztes eredményeket mikor *materializáljuk* (előállítjuk teljes egészében és lemezen tároljuk), illetve mikor *„futószalagosítjuk"* (csak a központi memóriában állítjuk elő, és nem feltétlenül tartjuk egyben az egészet egyszerre).
3. Jelölésrendszer a fizikai lekérdezésterv operátorai számára. Ez magában foglalja a tárolt relációk elérési módszereivel, illetve a relációs algebra operátorait megvalósító algoritmusokkal kapcsolatos részleteket.

Az operátorok algoritmusainak kiválasztását nem fogjuk teljes egészében megtárgyalni. Ehelyett a két legfontosabb operátor ide vonatkozó kérdéseit nézzük meg: a kiválasztást a 7.7.1. részben és az összekapcsolásokat a 7.7.2. részben. Ezután, a 7.7.3. résztől a 7.7.5. részig, a futószalagos technika és a materializáció közötti választás kérdését vesszük szemügyre. A fizikai lekérdezéstervekkel kapcsolatos jelölésrendszert a 7.7.6. részben mutatjuk be.

7.7.1. Kiválasztási eljárás megválasztása

Amikor fizikai lekérdezéstervet választunk, az egyik legfontosabb lépés az, hogy minden egyes kiválasztás operátorhoz algoritmust jelölünk ki. A 6.3.1. részben beszéltünk a $\sigma_C(R)$ operátor triviális megvalósításáról, amikor is a teljes R relációhoz hozzáférünk, és megnézzük, hogy mely sorok elégítik ki a C feltételt. Majd a 6.7.2. részben azt a lehetőséget vizsgáltuk, amikor a C feltétel „attribútum egyenlő konstans" alakú volt, és volt egy index az adott attribútumra. Ha ez adott, akkor a feltétel-

nek megfelelő sorokat anélkül is megtalálhatjuk, hogy az R relációt egyáltalán megnéznénk. Tekintsük most ennek a problémának az általánosítását, nevezetesen amikor van egy olyan kiválasztási feltételünk, amely több feltétel konjunkciója (AND-je), amelyek között vannak „attribútum egyenlő konstans" alakúak és más összehasonlítások attribútumok és konstansok között, mint például a <.

Ha nincsenek több attribútumra vonatkozó multidimenzionális indexek, akkor a használandó stratégia egy vagy több olyan attribútum választását foglalja magában, amely

a) Rendelkezik indexszel, és
b) A kiválasztásban szereplő részfeltételek valamelyikében egy konstanssal van összehasonlítva.

Ekkor ezeket az indexeket használjuk az egyes feltételeket kielégítő sorok halmazainak a beazonosítására. A 4.2.3. és 5.1.5. rész tárgyalta azt, hogy az indexekből megkapott sormutatókat hogyan használhatjuk arra, hogy csak az olyan sorokat találjuk meg, amelyek az összes feltételnek eleget tesznek, még mielőtt azokat a sorokat a lemezről beolvasnánk.

Az egyszerűség kedvéért, több indexnek az ilyen módon történő használatát nem vizsgáljuk. Ehelyett inkább csak az alábbi típusú algoritmusokra korlátozzuk vizsgálódásunkat:

1. Egy $A\theta\ c$ alakú összehasonlítást használ, ahol A egy indexszel rendelkező attribútum, c egy konstans és θ egy összehasonlító operátor (= vagy <).
2. Visszaadja az 1.-ben szereplő összehasonlítást kielégítő sorokat, használva a 6.2.1. részben tárgyalt indexolvasás operátort.
3. A 2.-ben kapott minden sorra megvizsgálja, hogy kielégíti-e a kiválasztási feltétel fennmaradó részét. Az ezt a lépést végrehajtó fizikai operátort Filternek fogjuk nevezni. A sorok kiválasztásához használt feltétel paramétere ennek az operátornak, nagy hasonlóságot mutatva így a relációs algebra σ operátorával.

Az ilyen típusú algoritmusokon túl szükség van egy olyan algoritmusra is, amely nem használ indexet, hanem a teljes relációt végigolvassa (a táblaolvasás fizikai operátor segítségével), és az egyes sorokat átadja a Filter operátornak a kiválasztási feltétel teljesülésének ellenőrzése céljából.

Hogy egy adott kiválasztást az algoritmusok közül melyikkel valósítjuk meg, azt úgy döntjük el, hogy becsüljük az adatok olvasásának költségét az egyes megoldások esetén. A lehetséges algoritmusok költségeinek összehasonlításakor már nem használható tovább a közbülső relációk méretein alapuló egyszerűsített költségbecslés. Ennek az az oka, hogy most a logikai lekérdezésterv egyetlen lépésének megvalósításával foglalkozunk, és a közbülső relációk függetlenek a megvalósítástól.

Újra a lemez I/O-műveletek számával fogunk tehát dolgozni, csakúgy mint a 6. fejezetben, amikor szintén algoritmusokat és hozzájuk tartozó költségeket tárgyaltunk. A korábbiakhoz hasonlóan azzal az egyszerűsítéssel élünk, hogy csak az adatblokkok

elérési költségét vesszük figyelembe, az indexblokkét nem. Ne feledjük el, hogy a szükséges indexblokkok száma általában sokkal kisebb, mint a szükséges adatblokkok száma, így a lemez I/O-költség e közelítése elég pontos.

Most pedig felvázoljuk, hogy hogyan lehet becsülni a különböző algoritmusok költségét. Feltesszük, hogy adott a $\sigma_C(R)$ művelet, ahol a C feltétel egy vagy több ÉS-sel összekötött összehasonlításból áll. Példaként az $a = 10$ és $b < 20$ összehasonlításokat használjuk mint az egyenlőségi feltétel, illetve az egyenlőtlenségi feltétel reprezentánsait.

1. A táblaolvasási algoritmus és egy hozzá társított szűrési lépés együttes költsége:

   a) $B(R)$, ha az R nyalábolt, és
   b) $T(R)$, ha az R nem nyalábolt.

2. Egy algoritmus, amely egy egyenlőséget vizsgáló összehasonlítást vesz, mint például $a = 10$, ahol az a attribútumhoz létezik index, először indexolvasást használ az illeszkedő sorok megtalálására, majd egy szűrést hajt végre a megkapott sorokon, hogy ellenőrizze, hogy azok a teljes C feltételt kielégítik-e. Az ilyen algoritmus költsége:

   a) $B(R)/V(R, a)$, ha az index nyalábolt, és
   b) $T(R)/V(R, a)$, ha az index nem nyalábolt.

3. Egy algoritmus, amely egy egyenlőtlenségi összehasonlítást vesz, mint például $b < 20$, ahol a b attribútumhoz létezik index, először indexolvasást használ az illeszkedő sorok megtalálására, majd egy szűrést hajt végre a megkapott sorokon, hogy ellenőrizze, hogy azok a teljes C feltételt kielégítik-e. Az ilyen algoritmus költsége:

   a) $B(R)/3$, ha az index nyalábolt,[11] és
   b) $T(R)/3$, ha az index nem nyalábolt.

7.37. példa: Tekintsük a $\sigma_{x=1 \text{ AND } y=2 \text{ AND } z<5}(R)$ kiválasztást, ahol az $R(x, y, z)$ a következő paraméterekkel rendelkezik: $T(R) = 5000$, $B(R) = 200$, $V(R, x) = 100$ és $V(R, y) = 500$. Tegyük fel továbbá, hogy az R nyalábolt, és az x, y és z attribútumok mindegyikéhez létezik index, de csak a z-hez tartozó index nyalábolt. Ennek a kiválasztásnak a megvalósítását a következő lehetőségek közül választhatjuk ki:

1. Táblaolvasás és azt követő szűrés. A költség $B(R)$, azaz 200 lemez I/O-művelet, mivel az R nyalábolt.
2. Használjuk az x-hez tartozó indexet és az indexolvasás operátort az $x = 1$ egyenlőséget kielégítő sorok megtalálására, majd használjuk a szűrés operátort annak

[11] Emlékezzünk a feltételezésünkre, miszerint a tipikus egyenlőtlenség a sorok 1/3-át adja vissza, a 7.4.3. részben tárgyalt indokoknak megfelelően.

ellenőrzésére, hogy az $y = 2$ és $z < 5$ összehasonlítások is teljesülnek-e ezekre a sorokra. Mivel körülbelül $T(R)/V(R, x) = 50$ olyan sor van, amelyre $x = 1$, és az index nem nyalábolt, körülbelül 50 lemez I/O-műveletre van szükség.

3. Használjuk az y-hoz tartozó indexet és az indexolvasás operátort az $y = 2$ egyenlőséget kielégítő sorok megtalálására, majd használjuk a szűrés operátort annak ellenőrzésére, hogy az $x = 1$ és $z < 5$ összehasonlítások is teljesülnek-e ezekre a sorokra. Ennek a nem nyalábolt indexnek a használata esetén a költség $T(R)/V(R, y)$, azaz 10 lemez I/O-művelet.
4. Használjuk a z-hez tartozó nyalábolt indexet és az indexolvasás operátort a $z < 5$ egyenlőtlenséget kielégítő sorok megtalálására, majd ezekre a sorokra alkalmazzuk a szűrés operátort, hogy lássuk, hogy az $x = 1$ és $y = 2$ feltételek is teljesülnek-e. A lemez I/O-műveletek száma körülbelül $B(R)/3 = 67$.

Azt látjuk, hogy a legkisebb költségű algoritmus a harmadik, és ennek a becsült költsége 10 lemez I/O-művelet. A kiválasztáshoz tartozó legjobb fizikai terv tehát először megkeresi az összes sort, amelyre $y = 2$, majd szűri azokat a másik két feltételnek megfelelően. □

7.7.2. Összekapcsolási eljárás megválasztása

A 6. fejezetben láttuk a különböző összekapcsolási algoritmusokhoz tartozó költségeket. Ha építünk arra a feltételezésre, hogy ismerjük (vagy becsüljük) az összekapcsolás végrehajtásához rendelkezésre álló pufferek számát, akkor alkalmazhatjuk a 6.5.8. részben szereplő képleteket a rendezéses összekapcsolásokra, a 6.6.7. részben szereplőket a tördelő összekapcsolásokra, valamint a 6.7.3. és 6.7.3. részben szereplőket az indexes összekapcsolásokra.

Ha viszont nem vagyunk biztosak benne vagy nem ismerjük a lekérdezés végrehajtása során rendelkezésre álló pufferek számát (mert nem tudjuk, hogy mi egyebet végez még az adatbázis-kezelő rendszer ugyanabban az időben), vagy nem vagyunk birtokában a méretre vonatkozó fontos paraméterek becsült értékeinek, mint amilyenek például a $V(R, a)$-k, akkor még mindig van néhány elv, amelyet egy összekapcsolási módszer megválasztásakor alkalmazhatunk. Hasonló gondolatmenet érvényes más bináris operátorokra – mint amilyen az egyesítés –, valamint a teljes relációra vonatkozó, a γ és a δ unáris operátorokra is.

- Az egyik megközelítés az, hogy az egymenetes összekapcsolást alkalmazzuk, azt remélve, hogy a pufferkezelő elég puffert tud az összekapcsolásnak szentelni, vagy legalábbis közel jár ehhez, így az ütközések nem eredményeznek jelentős költséget. Egy másik lehetőség az (csak összekapcsolásoknál, más bináris operátoroknál nem), hogy beágyazott ciklusú összekapcsolást választunk, azt remélve, hogy még ha a bal argumentum nem is kap elég puffert ahhoz, hogy egyszerre elférjen a memóriában, nem kell majd túl sok darabra osztani azt, és a kapott összekapcsolás még mindig elfogadható hatékonyságú lesz.

- Egy rendezéses összekapcsolás akkor jó választás, ha az alábbiak valamelyike teljesül:

  1. Az egyik vagy mindkét argumentum már rendezett az összekapcsolási attribútum(ok) szerint.
  2. Kettő vagy több összekapcsolás van ugyanazzal az összekapcsolási attribútummal, mint például az $(R(a, b) \bowtie S(a, c)) \bowtie T(a, d)$ összekapcsolás esetében, ahol az R és az S relációk a szerinti rendezése maga után vonja azt, hogy az $R \bowtie S$ eredménye az a szerint rendezett lesz, így az közvetlenül használható egy második rendezéses összekapcsolásban.

- Ha van alkalmunk indexet használni, mint például az $R(a, b) \bowtie S(b, c)$ összekapcsolásnál, ahol az R várhatóan kicsi (esetleg egy kulcs alapján történő kiválasztás eredménye, ami csak egy sort eredményez), és az $S.b$ összekapcsolási attribútumra létezik index, akkor indexes összekapcsolást válasszunk.
- Ha nem áll módunkban már rendezett relációkat vagy indexeket használni, és többmenetes összekapcsolásra van szükség, akkor valószínűleg a tördelő módszer a legjobb választás, mert a szükséges menetek száma a kisebb argumentum méretétől függ, és nem mindkét argumentumtól.

7.7.3. Futószalagosítás és materializáció

Az utolsó fő téma, amit egy fizikai lekérdezésterv megválasztása kapcsán meg kell beszélni, az az eredmények futószalagosítása. Egy lekérdezésterv végrehajtásának naiv módja az, hogy kialakítjuk a műveletek megfelelő sorrendjét (tehát egy művelet nem kerül végrehajtásra mindaddig, amíg az alatta szereplő argumentumok végrehajtása meg nem történt), és mindegyik művelet eredményét lemezen tároljuk mindaddig, ameddig egy másik művetnek szüksége van rá. Ezt a stratégiát *materializációnak* nevezzük, mivel minden közbülső reláció lemezen létrejön („testet ölt").

Egy lekérdezésterv végrehajtásának kifinomultabb és általában hatékonyabb módja

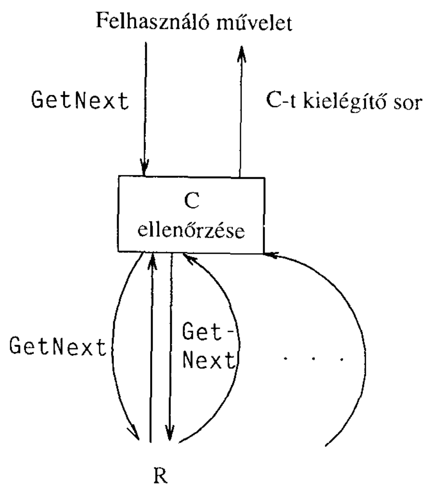
> **Materializáció a memóriában**
>
> Bárki el tudja képzelni, hogy létezik egy közbülső megközelítés a futószalagosítás és a materializáció között, amikor egy művelet teljes eredményét a központi memória puffereiben (nem lemezen) tároljuk, mielőtt az azt felhasználó műveletnek átadódik. Az ilyen műveletmódot futószalagosításnak tekintjük, ahol is első teendőként a felhasználó művelet elrendezi a memóriában a teljes relációt vagy annak egy nagy darabját. Az ilyen viselkedésre példa egy olyan kiválasztás, amelynek eredménye tovább adódik, mint bizonyos összekapcsolási algoritmusok – ilyen az egyszerű egymenetes összekapcsolás, a többmenetes tördelő összekapcsolás vagy a rendezéses összekapcsolás – valamelyikének bal (építő) argumentuma.

az, amikor több művelet fut egyszerre. Egy adott művelet által előállított sorok közvetlenül átadódnak annak a műveletnek, amelyik használja azokat anélkül, hogy köztes sorokat bármikor is tárolnánk lemezen. Ezt a módszert *futószalagosításnak* nevezzük, és általában iterátorok egy hálózatával valósítjuk meg (lásd a 6.2.6. részt), amelyek függvényei a megfelelő időben hívják egymást. A futószalagosításnak nyilván megvan a maga előnye, hiszen lemez I/O-műveleteket takarít meg, van azonban egy hátránya is. Mivel egyszerre több műveletnek kell a központi memórián osztozni, megeshet, hogy magas lemez I/O-művelet számú algoritmusokat kell választani, vagy ütközések fordulhatnak elő, ami a futószalagosítás által nyert lemez I/O-művelet megtakarításokat visszaveszi, vagy esetleg még többet használ fel.

7.7.4. Unáris műveletek futószalagosítása

Az unáris műveletek – kiválasztás és vetítés – kiváló jelöltek arra, hogy a futószalagosítást alkalmazzuk rajtuk. Mivel ezek a műveletek egyszerre egy sort dolgoznak fel, soha nincs szükség egy blokknál többre a bemenet számára, és egy blokknál többre a kimenet számára. A 6.10. ábrán szemléltettük ezt a műveletmódot.

Egy futószalagosított unáris műveletet megvalósíthatunk iterátorok segítségével, ahogy ezt a 6.2.6. részben megtárgyaltuk. A futószalagra kerülő eredményt felhasználó művelet a `GetNext()` függvényt hívja, amikor egy újabb sorra van szüksége. Egy vetítés esetében egyszer meg kell hívni a `GetNext()`-et a forrássorokra, megfelelően vetíteni kell az adott sort, és visszaadni az eredményt a felhasználó művelet számára. Egy σ_C vetítésnél (ami technikailag a `Filter(C)` fizikai operátor) viszont előfordulhat, hogy többször kell a forrásra meghívni a `GetNext()`-et amíg talál olyan sort, amely kielégíti a C feltételt. Ezt az utóbbi folyamatot a 7.34. ábra szemlélteti.



7.34. ábra. *Egy futószalagosított kiválasztás végrehajtása iterátorok használatával*

7.7.5. Bináris műveletek futószalagosítása

A bináris műveletekből származó eredményeket is lehet futószalagosítani. Egy puffert használunk arra a célra, hogy az eredményt a felhasználó műveletnek átadjuk, mégpedig blokkonként. Az eredmény kiszámításához és az eredmény felhasználásához szükséges további pufferek száma azonban az eredmény méretétől és a lekérdezésben szereplő további relációk méreteitől függően változik. Az idevonatkozó problémákat és lehetőségeket egy összetett példán keresztül szemléltetjük.

7.38. példa: Nézzünk fizikai lekérdezésterveket az alábbi kifejezéshez:

$$(R(w, x) \bowtie S(x, y)) \bowtie U(y, z)$$

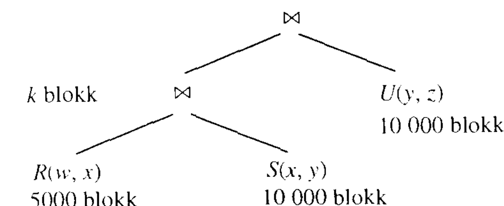
A következő feltételezésekkel élünk:

1. Az R 5000 blokkot foglal el, az S és az U pedig egyaránt 10 000 blokkot foglalnak el.
2. Az $R \bowtie S$ közbülső eredmény által elfoglalt blokkok számát k-val jelöljük. A k-t becsülhetnénk az R-ben és S-ben előforduló x-értékek száma, valamint a (w, x, y) soroknak az R-beli (w, x) sorokhoz és az S-beli (x, y) sorokhoz viszonyított mérete alapján. Látni szeretnénk azonban, hogy mi történik, ha k változik, így nyitva hagyjuk ezt a konstanst.
3. Mindkét összekapcsolást tördelő összekapcsolással valósítjuk meg, mégpedig k-tól függően egymenetessel vagy kétmenetessel.
4. 101 puffer áll rendelkezésre. Ezt a számot, mint eddig is, mesterkélten alacsonyra vettük.

A kifejezést és a paramétereket a 7.35. ábra mutatja.

Nézzük először az $R \bowtie S$ összekapcsolást. Egyik reláció sem fér el a központi memóriában, ezért egy kétmenetes tördelő összekapcsolásra van szükségünk. Ahhoz, hogy a kisebb R reláció egyes kosarait 100 blokkra korlátozzuk, legalább 50 kosárra szükség van.[12] Ha pontosan 50 kosarat használunk, akkor az $R \bowtie S$ tördelő összekapcsolás második menete 51 puffert használ, így szabadon marad 50 puffer, amit az $R \bowtie S$ eredményének az U-val való összekapcsolásához lehet használni.

Tegyük most fel, hogy $k \leq 49$, vagyis az $R \bowtie S$ eredménye legfeljebb 49 blokkot foglal el. Ekkor az $R \bowtie S$ eredményét 49 pufferbe futószalagosíthatjuk, a kikeresés



7.35. ábra. *Logikai lekérdezésterv és paraméterek a 7.38. példához*

céljából tördelőtáblába rendezhetjük, és marad egy blokk arra, hogy az U egyes blokkjait sorra beolvassuk. A második összekapcsolást tehát egy egymenetes összekapcsolással végrehajthatjuk. A lemez I/O-műveletek száma:

a) Az R és S kétmenetes tördelő összekapcsolásának végrehajtásához: 45 000.
b) Az $(R \bowtie S) \bowtie U$ egymenetes tördelő összekapcsolásban az U beolvasásához: 10 000.

Ez összesen 55 000 lemez I/O-művelet.

Most azt tegyük fel, hogy $k > 49$, de $k \leq 5000$. Az $R \bowtie S$ eredményét még mindig futószalagosíthatjuk, de egy másik stratégiát kell használni, amikor is ezt a relációt az U-val egy 50 kosaras, kétmenetes összekapcsolással kapcsoljuk össze.

1. Mielőtt elkezdjük az $R \bowtie S$-t, az U-t széttördeljük 50 darab 200 blokkos kosarakba.
2. Következő lépésként elvégezzük az R és az S kétmenetes tördelő összekapcsolását 51 kosarat használva, csakúgy mint az előbb. Most azonban amint előállítjuk az $R \bowtie S$ egy sorát, elhelyezzük azt a fennmaradó 50 puffer valamelyikében, amelyek az $R \bowtie S$-nek az U-val történő összekapcsolásához szükséges 50 kosár kialakítására valók. Ezeket a puffereket lemezre írjuk, amikor megtelnek, ahogy ez a kétmenetes tördelő összekapcsolások esetében szokásos.
3. Végezetül kosaranként összekapcsoljuk az $R \bowtie S$-t az U-val. Minthogy $k \leq 5000$, az $R \bowtie S$ kosarai legfeljebb 100 blokk méretűek lesznek, ez az összekapcsolás tehát megvalósítható. Az a tény, hogy az U kosarai 200 blokk méretűek, nem jelent problémát, hiszen a kosarak egymenetes összekapcsolásában az $R \bowtie S$ kosarait használjuk építő relációként, és az U kosarait vizsgáló relációként.

Az ehhez a futószalagosított összekapcsoláshoz tartozó lemez I/O-műveletek száma:

a) az U beolvasásához és sorainak kosarakba történő írásához: 20 000,
b) az $R \bowtie S$ kétmenetes tördelő összekapcsolás végrehajtásához: 45 000,
c) az $R \bowtie S$ kosarainak kiírásához: k.
d) az $R \bowtie S$ és az U kosarainak beolvasásához a végső összekapcsolásban: k + 10 000.

Az összköltség tehát 75 000 + $2k$. Vegyük észre, hogy a folytonosság nyilvánvalóan megszakad, amikor a k 49-ről 50-re nő, az utolsó összekapcsolást ugyanis egymenetesről kétmenetesre kellett változtatni. A gyakorlatban nem változna meg a költség ilyen hirtelen, mivel akkor is használhatnánk az egymenetes összekapcsolást, ha nem volna elég puffer, és némi ütközés előfordulna.

Végül nézzük meg, hogy mi a helyzet, ha $k > 5000$. Most a rendelkezésre álló 50 pufferben nem végezhetünk egy kétmenetes összekapcsolást, ha az $R \bowtie S$ eredményét futószalagosítjuk. Használhatnánk egy hárommenetes összekapcsolást, de ez mindkét argumentumnál blokkonként 2 extra lemez I/O-műveletet igényelne, ami 20 000 + $2k$-val több lemez I/O-műveletet jelentene. Jobban tesszük, ha inkább eltekintünk az $R \bowtie S$ futószalagosításától. Az összekapcsolások kiszámításának váza ebben az esetben így néz ki:

1. Kiszámítjuk az $R \bowtie S$-t egy kétmenetes tördelő összekapcsolást használva, és az eredményt eltároljuk lemezen.
2. Az $R \bowtie S$-t összekapcsoljuk az U-val, szintén egy kétmenetes tördelő összekapcsolást használva. Vegyük észre, hogy mivel $B(U)$ = 10 000, a 100 blokktárolót használó kétmenetes tördelő összekapcsolás végrehajtható, függetlenül attól, hogy milyen nagy a k. Technikailag az U-nak a megfelelő összekapcsolás bal argumentumaként kellene szerepelnie a 7.35. ábrán, ha úgy döntünk, hogy az U-t választjuk építő relációnak a tördelő összekapcsolásban.

Az ehhez az algoritmushoz szükséges lemez I/O-műveletek száma:

a) Az R és az S kétmenetes összekapcsolásához: 45 000.
b) Az $R \bowtie S$ eredményének eltárolásához: k.
c) Az U-nak és az $R \bowtie S$-nek a kétmenetes tördelő összekapcsolásához: 30 000 + $3k$.

A k tartománya	*Futószalagosítás vagy materializáció*	*Az utolsó összekapcsolás algoritmusa*	*Lemez I/O-műveletek száma*
$k \leq 49$	Futószalagosítás	Egymenetes	55 000
$50 \leq k \leq 5000$	Futószalagosítás	50 kosaras, kétmenetes	75 000 + $2k$
$5000 < k$	Materializáció	100 kosaras, kétmenetes	75 000 + $4k$

7.36 ábra. *Összekapcsolási algoritmusok költségei az $R \bowtie S$ méretének függvényében*

A teljes költség így 75 000 + $4k$, ami kevesebb, mint annak a költsége, ha az utolsó lépesben egy hárommenetes összekapcsolást használunk. A három algoritmust a 7.36. ábrán található táblázatban foglaltuk össze. □

7.7.6. Fizikai lekérdezéstervekkel kapcsolatos jelölések

Sok példát láttunk olyan operátorokra, amelyek arra használhatók, hogy fizikai lekérdezésterveket képezzenek. Általában a logikai terv minden operátora a fizikai terv egy vagy több operátorává alakul át, és a logikai terv minden egyes levelből (tárolt relációk) a fizikai tervben valamilyen olvasási operátor lesz, amely az adott relációra vonatkozik. A materializációt pedig az eltárolandó köztes eredményre alkalmazott valamilyen `Store` operátor jelezné, amit egy megfelelő olvasási művelet követne (rendszerint `TableScan`, mivel a köztes relációkhoz nem létezik index, hacsak explicit módon létre nem hozunk egyet), amint az eltárolt eredményhez az azt felhasználó művelet hozzáfér. Az egyszerűség kedvéért azonban a fizikai lekérdezésterveknél mi úgy fogjuk jelezni egy bizonyos közbülső reláció materializálásának tényét, hogy a fában egy kettős vonallal áthúzzuk azt az élt, amelyik az adott relációt és azt felhasználó műveletet összeköti. Az összes többi élnél feltételezzük, hogy futószalagos technikát alkalmazunk a sorok előállítója és felhasználója között.

Most pedig felsoroljuk a fizikai lekérdezéstervekben általában előforduló különféle operátorokat. A relációs algebrával ellentétben, amely eléggé szabványos jelölésrendszerrel rendelkezik, a fizikai lekérdezéstervekhez az egyes adatbázis-kezelő rendszerek saját belső jelölésrendszert használnak.

Operátorok a levelekhez

Minden egyes *R* relációt, amely a logikai lekérdezésterv fájában egy levél operandus, valamilyen olvasás operátorra cserélünk. A választási lehetőségek a következők:

1. `TableScan(R)`: Az *R* sorait tartalmazó összes blokkot beolvassa tetszőleges sorrendben.
2. `SortScan(R,L)`: Az *R* sorait az *L* listában szereplő attribútum(ok) szerint rendezve olvassa be.
3. `IndexScan(R,C)`: Itt a *C* egy $A\theta\ c$ alakú feltétel, ahol az *A* egy attribútum, a θ az = vagy < összehasonlító operátor, és a *c* egy konstans. Az *R* sorait egy az *A* attribútumhoz létező indexen keresztül érjük el. Ha a θ összehasonlítás nem =, akkor az indexnek olyannak kell lennie, amelyik támogatja a tartományra vonatkozó kérdéseket, mint amilyen például egy B-fa.
4. `IndexScan(R,A)`: Itt az *A* az *R* egy attribútuma. A teljes *R* relációt egy az *R.A*-hoz tartozó indexen keresztül kapjuk meg. Ez az operátor úgy viselkedik, mint a `TableScan`, de hatékonyabb lehet bizonyos körülmények között, ha az *R* nem nyalábolt és/vagy a blokkjait nem könnyű megtalálni.

Fizikai operátorok a kiválasztáshoz

Ha az *R* egy tárolt reláció, akkor egy $\sigma_C(R)$ logikai operátor gyakran vegyül, vagy részben vegyül, az *R* relációhoz történő hozzáférési eljárással. Más kiválasztásokat, ahol az argumentum nem tárolt reláció vagy nem áll rendelkezésre alkalmas index, megfelelő fizikai `Filter` operátorokkal fogunk helyettesíteni. Idézzük fel a kiválasztás megvalósításának megválasztására szolgáló stratégiát, amit a 7.7.1. részben tárgyaltunk. A kiválasztás különböző megvalósításaihoz a következő jelöléseket fogjuk használni:

1. A $\sigma_C(R)$-t egyszerűen helyettesíthetjük a `Filter(C)` operátorral. Ennek a választásnak akkor van értelme, ha az *R*-hez nem létezik index, pontosabban ha egyetlen *C*-ben előforduló attribútumra sincs index. Ha *R* – a kiválasztás argumentuma – egy közbülső reláció, amit a futószalagos technikán keresztül kap meg a kiválasztás, akkor a `Filter` mellett nincs más operátorra szükség. Ha *R* egy tárolt vagy materializált reláció, akkor az *R* eléréséhez kell egy operátor. `TableScan` vagy `SortScan`. A `SortScan` mellett maradhatunk, ha a $\sigma_C(R)$ eredménye később egy olyan operátornak adódik át, amely rendezett argumentumot vár.

2. Ha a *C* feltétel $A\theta\ c$ `AND` *D* alakú, ahol *D* egy másik feltétel, és az *R.A*-ra létezik index, akkor a következőket tehetjük:

   a) Az *R* eléréséhez használjuk az `IndexScan(R,Aθc)` operátort, és
   b) A $\sigma_C(R)$ helyett használjuk a `Filter(D)` operátort.

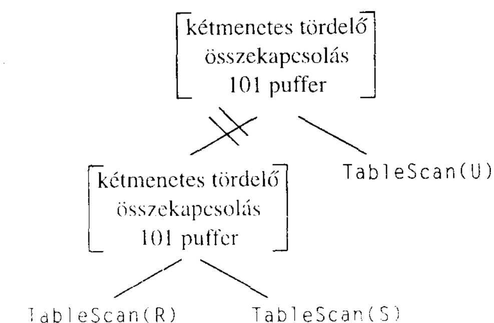
Fizikai operátorok a rendezéshez

A fizikai lekérdezéstervben bárhol rendezhetünk egy relációt. Bevezettük már a `SortScan(R,L)` operátort, amely beolvas egy *R* relációt, és előállítja annak rendezett formáját az *L* attribútumlistának megfelelően. Ha rendezéses algoritmust használunk egy művelethez, mondjuk összekapcsoláshoz vagy csoportosításhoz, akkor van egy kiindulási fázis, amikor az argumentumot valamilyen attribútumlistának megfelelően rendezzük. Nem tárolt operandusok esetén általában egy explicit `Sort(L)` operátort használunk a rendezés elvégzésére. Ugyanez az operátor a fizikai lekérdezésterv fájának tetején is használható, amennyiben az eredeti lekérdezés `ORDER BY` záradékot tartalmaz, és rendezni kell az eredményt, megfelelve ezzel a relációs algebránk τ operátorának.

Egyéb relációs algebrai műveletek

Az összes többi művelet helyettesíthető megfelelő fizikai operátorral. Adhatunk olyan megnevezéseket ezeknek az operátoroknak, amelyek mutatják az alábbiakat:

1. A végrehajtandó műveletet, például összekapcsolást vagy csoportosítást.
2. A szükséges paramétereket, például egy théta-összekapcsolásban a feltételt, vagy egy csoportosításban az attribútumlistát.
3. Az algoritmushoz használt általános stratégiát: rendezéses, tördelő, vagy index alapú bizonyos összekapcsolásoknál.
4. Menetek számára vonatkozó döntést: egymenetes, kétmenetes vagy sokmenetes



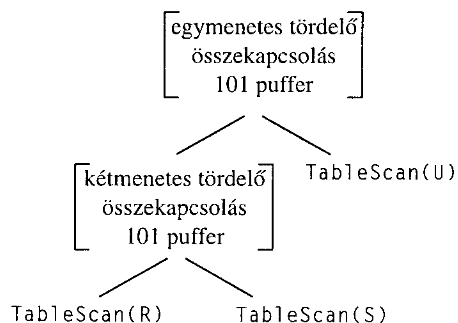
7.37. ábra. *Egy fizikai terv a 7.38. példából*

(rekurzív, ami annyi menetet használ, amennyi az aktuális adatok esetén szükséges). Azt is megtehetjük, hogy ezt a választást a futási időre halasztjuk.

5. A művelethez szükséges pufferek várható száma.

7.39. példa: A 7.38. példában a $k > 5000$ esetre kidolgozott fizikai tervet a 7.37. ábra mutatja. Ebben a tervben mindhárom relációt a `TableScan` operátor segítségével érjük el. Használunk egy kétmenetes tördelő összekapcsolást az első összekapcsoláshoz, materializáljuk az eredményt, majd használunk egy kétmenetes tördelő összekapcsolást a második összekapcsoláshoz. A materializációt jelző kettős vonalból következik, hogy a felső összekapcsolás bal argumentumát is egy `TableScan` operátor segítségével szerezzük meg, valamint hogy az első összekapcsolás eredményét a `Store` operátort használva tároljuk el.

A 7.38. példában a $k \leq 49$ esetre adott fizikai terv a 7.38. ábrán látható. Vegyük észre, hogy a második összekapcsolás az eddigitől eltérő számú menetet és puffert használ, továbbá nem egy materializált, hanem egy futószalagosított bal argumentumot használ. □



7.38. ábra. *Egy másik fizikai terv arra az esetre, amikor az $R \bowtie S$ várhatóan nagyon kicsi*

7.40. példa: Tekintsük a 7.37. példa kiválasztási műveletét, ahol úgy döntöttünk, hogy az a legjobb megoldás, hogy az y szerinti indexet használjuk az $y = 2$ feltételt kielégítő sorok megtalálására, majd ezekre a sorokra ellenőrizzük az $x = 1$ és $z < 5$ további feltételeket. A fizikai lekérdezésterv a 7.39. ábrán látható. A levél mutatja azt, hogy az R-et az ő y szerinti indexén keresztül érjük el, és az $y = 2$ összehasonlításnak eleget tevő sorokat keressük vissza. A szűrési operátor (`Filter`) mondja meg azt, hogy a kiválasztást azzal fejezzük be, hogy a megkapott sorok közül kiválasztjuk azokat, amelyekre $x = 1$ és $z < 5$. □

```
Filter(x=1 AND z<5)
        |
        |
 IndexScan(R, y=2)
```

7.39. ábra. *Egy kiválasztás kifejtése úgy, hogy a legmegfelelőbb indexet használja*

7.7.7. Fizikai operátorok sorrendbe állítása

A fizikai lekérdezéstervekkel kapcsolatos utolsó téma a műveletek sorrendje. A fizikai lekérdezéstervet általában egy fával ábrázoljuk, és a fák mondanak valamit a műveletek sorrendjéről, hiszen az adatoknak alulról felfelé kell haladni a fában. A bozótszerű fákban azonban lehetnek olyan belső csomópontok, amelyek sem nem ősei, sem nem leszármazottjai egymásnak, ezért a belső csomópontok kiértékelésének sorrendje esetleg nem mindig világos. Ráadásul, mivel iterátorokat használhatunk műveletek futószalagtechnikával történő megvalósítására, elképzelhető, hogy különböző csomópontok futási idői átfedik egymást, és nincs is értelme a csomópontok „sorrendjéről" beszélni.

Ha a materializációt a kézenfekvő tárol-és-később-visszakeres módon valósítjuk meg, és a futószalagosítást iterátorokkal valósítjuk meg, akkor kialakíthatunk egy előre rögzített eseménysorozatot, amely mentén a fizikai lekérdezéstervek egyes műveleteit végrehajtjuk. Az események sorrendjét, ami egy fizikai lekérdezésterv fájában közvetett módon benne van, a következő szabályok foglalják össze:

1. Vágjuk szét a fát részfáira az olyan éleknél, amelyek materializációt képviselnek. A részfákat egyenként hajtjuk végre.
2. A részfák közötti sorrendet alulról felfelé, balról jobbra állapítsuk meg. Pontosan szólva, végezzük el a teljes fa egy előrendezéses bejárását. A részfákat annak a sorrendnek megfelelően rendezzük, ahogyan az előrendezéses bejárás a részfákat elhagyja.
3. Hajtsuk végre az egyes részfák összes csomópontját iterátorok hálózatát használva. Így az egy részfán belül szereplő összes csomópontot egyidejűleg hajtjuk végre, `GetNext` hívásokkal a bennük szereplő operátorok között, meghatározva ezáltal az események pontos sorrendjét.

Ezt a stratégiát követve a lekérdezésoptimalizáló futtatható kódot tud generálni a lekérdezéshez, ami várhatóan függvényhívások egy sorozata lesz.

7.7.8. Feladatok

7.7.1. feladat: Vegyünk egy $R(a, b, c, d)$ relációt, amelynek van egy nyalábolt indexe az a-ra, és egy-egy nem nyalábolt indexe az összes többi attribútumra. A lényeges paraméterek a következők: $B(R) = 1000$, $T(R) = 5000$, $V(R, a) = 20$, $V(R, b) = 1000$, $V(R, c) = 5000$ és $V(R, d) = 500$. Adjuk meg az alábbi kiválasztásokhoz tartozó legjobb lekérdezéstervet (index- vagy táblaolvasás és azt követő szűrés) a lemez I/O-művelet költséggel együtt:

* a) $\sigma_{a = 1 \text{ AND } b = 2 \text{ AND } d = 3}(R)$

b) $\sigma_{a = 1 \text{ AND } b = 2 \text{ AND } c \geq 3}(R)$

c) $\sigma_{a = 1 \text{ AND } b \leq 2 \text{ AND } c \geq 3}(R)$

! 7.7.2. feladat: A $B(R)$, $T(R)$, $V(R, x)$ és $V(R, y)$ segítségével fejezzük ki az alábbi feltételeket, amelyek egy R-en végrehajtott kiválasztás költségére vonatkoznak:

* a) Jobb, ha indexolvasást egy x-re létező nem nyalábolt indexszel és egy x-et egy konstanssal egyenlővé tévő összehasonlítással használunk, mint ha egy y-ra létező nem nyalábolt indexszel és egy y-t egy konstanssal egyenlővé tévő összehasonlítással tennénk.
b) Jobb, ha indexolvasást egy x-re létező nem nyalábolt indexszel és egy x-et egy konstanssal egyenlővé tévő összehasonlítással használunk, mint ha egy y-ra létező nyalábolt indexszel és egy y-t egy konstanssal egyenlővé tévő összehasonlítással végeznénk el.
c) Jobb, ha indexolvasást egy x-re létező nem nyalábolt indexszel és egy x-et egy konstanssal egyenlővé tévő összehasonlítással használunk, mint ha egy y-ra létező nem nyalábolt indexszel és egy $y > C$ alakú összehasonlítással tennénk, ahol C egy konstans.

7.7.3. feladat: Hogyan változnának meg az arra vonatkozó következtetések, hogy mikor alkalmazzunk futószalagosítást a 7.38. példában, ha az R reláció mérete nem 5000 blokk lenne, hanem:

a) 2000 blokk,
! b) 10 000 blokk,
! c) 100 blokk.

! 7.7.4. feladat: Tegyük fel, hogy az $(R(a, b) \bowtie S(a, c)) \bowtie T(a, d)$ kifejezést a jelzett sorrendnek megfelelően akarjuk kiszámítani. $M = 101$ központi memóriapufferrel rendelkezünk, és $B(R) = 2000$. Mivel az a összekapcsolási attribútum ugyanaz mindegyik összekapcsolásnál, úgy döntünk, hogy az első $R \bowtie S$ összekapcsolást egy kétmenetes rendezéses összekapcsolással valósítjuk meg. A második összekapcsoláshoz annyi menetet fogunk használni, amennyi szükséges, szétdarabolva a T-t bizonyos számú a szerint rendezett részlistára, és összefésülve azokat az $R \bowtie S$ összekapcsolásból származó rendezett és futószalagosított soraival. A $B(T)$ milyen értékei esetén kellene a T-nek az $R \bowtie S$-sel történő összekapcsolásához a következőt választani:

* a) Egy egymenetes összekapcsolást: a T-t beolvassuk a memóriába, és sorait összehasonlítjuk az $R \bowtie S$ soraival, amint azok előállnak.
b) Egy kétmenetes összekapcsolást: T-hez rendezett részlistákat hozunk létre, és mindegyik rendezett részlistához fenntartunk egy puffert a memóriában, mialatt az $R \bowtie S$ sorait előállítjuk.

7.8. Összefoglalás

- *Lekérdezések fordítása:* A fordítás egy lekérdezést fizikai lekérdezéstervvé alakít át, ami egy műveletsorozat, amelyet a lekérdezés-végrehajtó motorral lehet megvalósítani. A lekérdezésfordítás alapvető lépései: elemzés, szemantikus ellenőrzés, az előnyben részesített logikai lekérdezésterv (algebrai kifejezés) kiválasztása és abból a legjobb fizikai terv generálása.
- *Az elemző:* Egy SQL-lekérdezés feldolgozása során az első lépés a lekérdezés elemzése, csakúgy mint bármilyen programozási nyelvben írt kód esetén. Az elemzés eredménye egy elemzőfa, amelynek csomópontjai az SQL elemeinek felelnek meg.
- *Szemantikus ellenőrzés:* Az előfeldolgozó megvizsgálja az elemzőfát, ellenőrzi, hogy az attribútumok, relációnevek és típusok értelmesek-e, és feloldja az attribútumhivatkozásokat, ha ugyanaz az attribútum több relációban is előfordul.
- *Átalakítás logikai lekérdezéstervvé:* A lekérdezésfeldolgozónak a szemantikai szempontból ellenőrzött elemzőfát át kell alakítania egy algebrai kifejezéssé. A relációs algebrába történő átfordítás túlnyomó része kézenfekvő, az alkérdések azonban problémát jelentenek. A szokásos megközelítésben egy kétargumentumú kiválasztást vezetünk be, ami az alkérdést a kiválasztás feltételébe teszi, és azután megfelelő transzformációkat alkalmazunk, amelyek lefedik a szokásos speciális eseteket.
- *Algebrai transzformációk:* Egy logikai lekérdezéstervet sokféle módon átalakíthatunk egy jobb tervvé algebrai szabályok felhasználásával. A 7.2. rész felsorolja a alapvető szabályokat.
- *A logikai lekérdezésterv kiválasztása:* A lekérdezésfeldolgozónak ki kell választania azt a lekérdezéstervet, amely leginkább esélyes arra, hogy egy hatékony fizikai tervhez vezessen. Az algebrai transzformációk alkalmazásán túl, érdemes az asszociatív és kommutatív operátorokat csoportosítani, különösen az összekapcsolásokat. Ezáltal a fizikai lekérdezésterv a legjobb sorrendet és csoportosítást tudja megválasztani ezekhez a műveletekhez.
- *Relációk méreteinek becslése:* Amikor kiválasztjuk a legjobb logikai tervet, vagy amikor meghatározzuk az összekapcsolások vagy más asszociatív-kommutatív műveletek sorrendjét, a közbülső relációk becsült méreteit használjuk a legvégén kiválasztott fizikai terv tényleges futási idő vagy lemez I/O-műveletek költségének helyettesítésére. Ha ismerjük – vagy becsüljük – a relációk méreteit (sorok száma), valamint a különböző értékek számát minden reláció minden attribútumára vonatkozóan, akkor ez segít abban, hogy a köztes relációk méreteire jó becsléseket kapjunk.
- *Hisztogramok:* Néhány rendszer hisztogramokat tart fenn bizonyos attribútumok értékeiről. Ez az információ arra használható, hogy a közbülső relációk méreteire így jobb becsléseket kapjunk, mint a fejezetben hangsúlyozott egyszerű módszerekkel.
- *Költség alapú optimalizálás:* A legjobb fizikai terv kiválasztásakor szükség van arra, hogy minden egyes lehetséges terv költségét becsülni tudjuk. Különböző stratégiákat lehet arra használni, hogy előállítsunk minden vagy néhány lehetséges fizikai tervet, ami egy adott logikai tervet valósít meg.

- *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 erednek. 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 leg-

> **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.

fontosabb 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.

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:

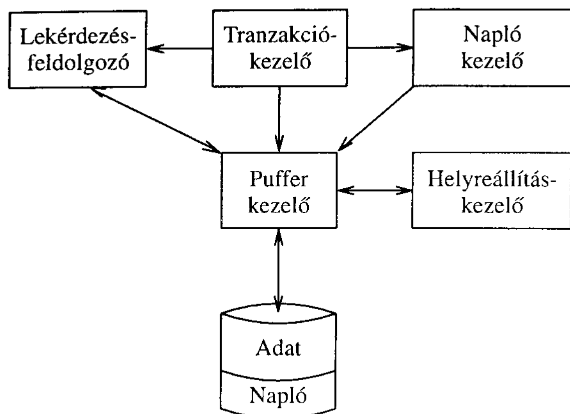
1. Jelzéseket ad át a naplókezelőnek (alább részletezzük) úgy, hogy a szükséges információ „naplóbejegyzés" formában a naplóban tárolható legyen.
2. Biztosítja, hogy a párhuzamosan végrehajtott tranzakciók ne zavarhassák egymás működését („ütemezés"; lásd a 9.1. részben).

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 adatbázisállapotot 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 memóriapuffer(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 felthető 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 érté-

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

kére. 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. □

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:

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.

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*

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 *helyreállításkezelő* (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 adatbáziselem 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 `COMMIT`ot 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:

~~1~~ 9	~~2~~ 10	~~3~~ 11	4	5	6	7	8

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 $T_1, \ldots, T_k$ tranzakciók mindegyikének normális vagy abnormális befejeződését, nem tiltva közben újabb tranzakciók indítását.
3. Ha a $T_1, \ldots, T_k$ 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 $(T_1, \ldots, T_k)$> 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 $(T_1, \ldots, T_k)$> naplóbejegyzésig megtaláljuk. Ennél a <START CKPT $(T_1, \ldots, T_k)$> naplóbejegyzésnél megállhatunk, a még korábbiakat már nem kell használnunk, azokat el is dobhatjuk.
- Amennyiben a <START CKPT $(T_1, \ldots, T_k)$> naplóbejegyzéssel találkozunk előbb, az azt jelenti, hogy a katasztrófa az ellenőrzőpont-képzés közben fordult elő. Ennek következtében $T_1, \ldots, T_k$ 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 ellenőrzőpont-képzést 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 kiíródtak. 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 ellenőrzőpontképzés 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 adatbáziselemnek, 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 ($T_1, \ldots, T_k$)> 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 ($T_1, \ldots, T_k$)> 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 ($T_1, \ldots, T_k$)> 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 ($S_1, \ldots, S_m$)> naplóbejegyzésig[12] vissza kell keresnünk, és helyre kell állítanunk az olyan befejeződött tranzakciók tevékenységének eredményeit, melyek ez utóbbi <START CKPT ($S_1, \ldots, S_m$)> 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. □

[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.

Minthogy a tranzakciók több ellenőrzőpont készítésekor is aktívak lehetnek, célszerű lehet, hogy a <START CKPT ($T_1, \ldots, T_k$)> 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.

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 ($T_1, \ldots, T_k$)> naplóbejegyzést, ahol $T_1, \ldots, T_k$-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 adatbázisállapot 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 máso-

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.

latot (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.

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-

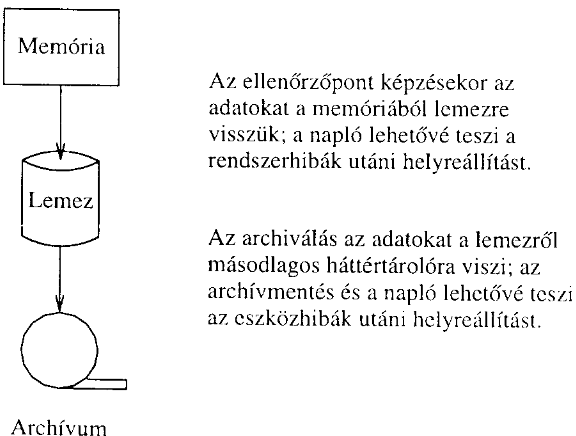
nak (a teljes adatbázishoz viszonyítva) csak azt a kis részét érintik, amely az utolsó mentés óta változott meg, s így ezek kevesebb helyet igényelnek és gyorsabban menthetők, mint a teljes mentés hely- és időigénye.

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.