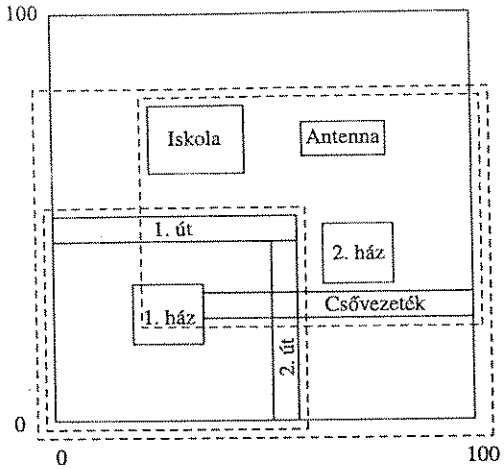
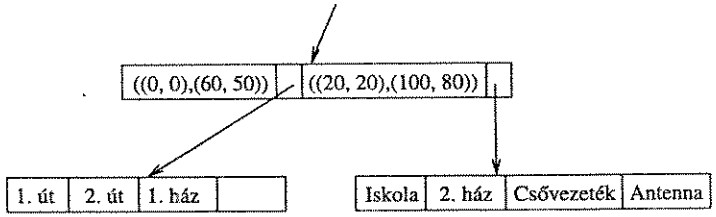


5.20. ábra. *Objektumhalmaz szétvágása*

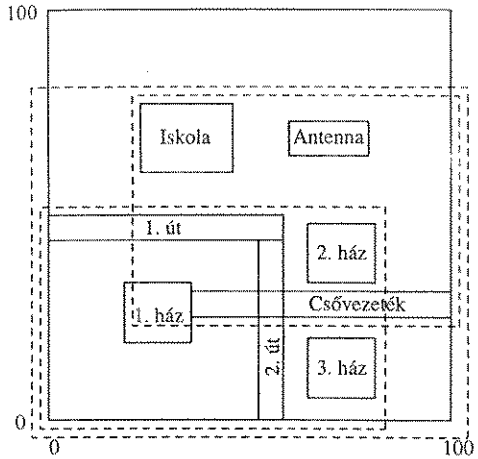
vonallal jelzett téglalapok), amely minimálissá teszi az átfedést, miközben olyan egyenletesen vágja szét a levelet, amennyire csak lehetséges.

Az 5.21. ábrán azt mutatjuk meg, hogy a két új levél hogyan illeszkedik be az R-fába. E csomópontok szülője tartalmazza a mutatókat mindkét levélre, és emellett a mutatókhoz kapcsoltan az adott levelet befedő téglalap alakú terület bal alsó és jobb felső sarkának koordinátáit is. □



5.21. ábra. *Egy R-fa*

5.19. példa: Tegyük fel, hogy beszúrtunk egy újabb házat a 2. ház alá, a bal alsó sarok koordinátái (70, 5), a jobb felsőé pedig (80, 15). Mivel a ház nincs benne teljesen egyik levél régiójában sem, választanunk kell egy régiót, amit megnövelünk. Ha az alsó alrégiót növeljük meg, ami az 5.21. ábrán az első levélnek felel meg, akkor 1000 négyzetegységgel növeljük meg a területet, mivel 20 egységnyit növeljük jobb felé. Ha a másik alrégiót növeljük meg, az alját lejjebb húzva 15 egységgel, akkor a hozzáadott terület 1200 négyzetegység. Az elsőt részesítjük előnyben, és az új, megváltozott régiók az 5.22. ábrán láthatók. Az 5.21. ábra felső csomópontjában lévő régió leírását is meg kell változtatnunk ((0, 0), (60, 50))-ről ((0, 0), (80, 50))-re. □



5.22. ábra. *Terület kiterjesztése az új adat elhelyezéséhez*

5.3.9. Feladatok

5.3.1. feladat: Mutassunk többszintű indexet az 5.10. ábra adataihoz, ha az indexek (az adott sorrendben):

a) a sebesség és a memória,
b) a memória és a merevlemez,
c) a sebesség, a memória és a merevlemez.

5.3.2. feladat: Helyezzük el az 5.10. ábra adatait egy *kd*-fában. Tegyük fel, hogy két rekord fér egy blokkba. Minden egyes szinten válasszunk egy határoló értéket, amely olyan egyenletesen osztja szét a rekordokat, amennyire csak lehetséges. A vágandó attribútumok sorrendje legyen a következő:

a) a sebesség és a memória (felváltva),
b) a sebesség, a memória és a merevlemez (felváltva),
c) amelyik attribútum a legegyenletesebb vágást eredményezi az egyes csomópontokra.

5.3.3. feladat: Tegyük fel, hogy van egy $R(x, y, z)$ relációnk, ahol az x és y attribútumpár együtt alkotja a kulcsot. Az x attribútum 1 és 100, az y pedig 1 és 1000 közötti értéket vehet fel. Minden egyes x-hez 100 különböző értékű y rekord, és minden egyes y-hoz 10 különböző értékű x rekord tartozik. Vegyük észre, hogy így 10 000 rekord van az R-ben. Többkulcsos indexet akarunk használni, ami segít nekünk megválaszolni a következő alakú lekérdezéseket:

```
SELECT z
FROM R
WHERE x = C AND y = D;
```

ahol C és D konstans. Tegyük fel, hogy a blokkok tíz kulcs-mutató párt tudnak tárolni, és sűrű indexeket akarunk létrehozni minden szinten, esetleg ritka magasabb szintű indexeket felettük, emiatt minden index egyetlen blokkal indul. Szintén feltesszük, hogy kezdetben minden index- és adatblokk a lemezen van.

* a) Hány lemez I/O-művelet szükséges a fenti formájú lekérdezés megválaszolásához, ha az x szerinti az első index?
b) Hány lemez I/O-művelet szükséges a fenti formájú lekérdezés megválaszolásához, ha az y szerinti az első index?
! c) Tegyük fel, hogy 11 blokkot tudunk folyamatosan a memóriában tartani. Mely blokkokat választanánk, és az x-et vagy az y-t tennénk az első indexnek, ha a továbbiakban szükséges lemez I/O-műveletek számát minimalizálni szeretnénk?

5.3.4. feladat: Az 5.3.3.a) feladat struktúrája esetén, hány lemez I/O-művelet szükséges a $20 \leq x \leq 35$ és $200 \leq y \leq 350$ tartománylekérdezés megválaszolásához? Tegyük fel, hogy az adatok egyenletesen oszlanak el, azaz a várható számú pontot találjuk bármilyen tartományon belül.

5.3.5. feladat: Az 5.13. ábrán látható fa esetén, mely új pontok kerülnének:

* a) a (30, 260) pontot tartalmazó blokkba?
b) az (50, 100) és az (50, 120) pontokat tartalmazó blokkba?

5.3.6. feladat: Mutassuk meg az 5.15. ábrán látható fa lehetséges kibővítéseit, ha beszúrjuk előbb a (20, 110), majd azután a (40, 400) pontokat.

! **5.3.7. feladat:** Említettük, hogy ha egy *kd*-fa teljesen kiegyensúlyozott, és végrehajtunk egy „lekérdezés részleges egyezéssel" típusú kérést, ahol a két attribútum egyikének az értéke adott, akkor kereséskor körülbelül négyzetgyök n számú levelet kell bejárnunk az n levélből.

a) Magyarázzuk meg, hogy miért?
b) Ha a fa felváltva vág a d dimenzióban, és ezek közül a dimenziók közül m-nek az értékét megadtuk, akkor a levelek mekkora részét kell várhatóan végignézni?
c) Milyen a b) teljesítménye a particionált tördelőtáblához viszonyítva?

5.3.8. feladat: Helyezzük el az 5.10. ábra adatait egy quad-fába, amelynek a dimenziói: a sebesség és a memória. Tegyük fel, hogy a sebesség 100-tól 500-ig mehet, és a memória 0-tól 256-ig.

5.3.9. feladat: Ismételjük meg az 5.3.8. feladatot, hozzáadva harmadik dimenzióként a merevlemezt, ami 0 és 32 közötti lehet.

*! **5.3.10. feladat:** Tételezzük fel, hogy egy quad-fa középpontját szabadon megválaszthatjuk, ekkor vajon feloszthatók-e mindig olyan résznegyedekre a negyedek, amelyekben azonos számú pont van (illetve amilyen egyenletesesen csak lehet, ha a negyedben lévő pontok száma nem osztható 4-gyel)? Igazoljuk a választ.

! **5.3.11. feladat:** Tegyük fel, hogy van egy adatbázisunk, amiben 1 000 000 régió van, amelyek átfedhetik egymást. Egy R-fa csomópontjaiban (blokkjaiban) 100 régió-mutató pár fér el. Bármely csomópontban ábrázolt régiónak 100 alrégiója van, és az átfedések ezen régiók között olyanok, hogy a 100 alrégió összege 150%-a az eredeti régió méretének. Ha egy „hol-vagyok-én" lekérdezést hajtunk végre egy adott pontra, mennyi a visszakeresendő blokkok várható száma?

! **5.3.12. feladat:** Az 5.22. ábrán szereplő R-fában egy új régió vagy abba az alrégióba kerülhet, amelyik az iskolát tartalmazza, vagy abba, amelyik a 3. házat. Adjuk meg azokat a téglalap alakú régiókat, amelyekre az iskolát tartalmazó alrégiót részesítenénk előnyben (azaz, ez a választás minimalizálja az alrégió méretnövekedését).

5.4. Bittérképindexek

Nézzünk most egy olyan indexet, amely sokban különbözik az eddig tárgyalt típusúaktól. Kezdjük azzal, hogy egy állandó számú rekordból álló állományt tekintünk, amelyben a rekordok sorszámai 1, 2, ..., n. Továbbá az állomány adatstruktúrája olyan, amely lehetővé teszi, hogy könnyen megtaláljuk az i-edik rekordot, tetszőleges i-re.

Egy F mezőhöz tartozó *bittérképindex* tulajdonképpen n hosszú bitvektorok olyan gyűjteménye, amelyben minden, az F mezőben előfordulható értékhez tartozik egy bitvektor. A v értékhez tartozó vektor az i-edik helyen 1-et tartalmaz, ha az i-edik rekordban az F mező értéke v, és 0-t, ha nem.

5.20. példa: Vegyünk egy állományt, amelyben a rekordoknak két mezője van: F és G, amelyek egész, illetve szöveg típusúak. Az aktuális állomány 6 rekordot tartalmaz, amelyek 1-től 6-ig vannak számozva, és a következők az értékei sorban: (30, `foo`), (30, `bar`), (40, `baz`), (50, `foo`), (40, `bar`), (30, `baz`).

Az első mezőhöz, F-hez tartozó bittérképnek három bitvektora lesz, mindegyik 6 hosszú. Az első, amely a 30 értékhez tartozik: 110001, mivel az első, a második és a hatodik rekordban $F = 30$. A másik kettő, a 40 és 50 értéhez tartozó rendre: 001010 és 000100.

A G-hez tartozó bittérképindexnek szintén három bitvektora lesz, mivel három különböző érték fordul elő benne. A három bitvektor:

Érték	*Vektor*
`foo`	100100
`bar`	010010
`baz`	001001

Mindegyik esetben az 1-esek mutatják, hogy melyik rekordban fordul elő a megfelelő szöveg. □

5.4.1. Indítékok a bittérképindexekhez

Először úgy tűnik, hogy a bittérképindexek túl sok helyet igényelnek, különösen, ha sok különböző érték tartozik egy mezőhöz, mivel a bitek száma összesen a rekordszám és az értékek számának szorzata. Például, ha a mező egy kulcs, és n rekordunk van, akkor n^2 bit szükséges az összes bitvektorhoz az adott mező esetén. Azonban tömörítést használva a bitek számát közelíthetjük n-hez, függetlenül a különböző értékek számától, ahogy azt az 5.4.2. részben majd fogjuk látni.

Szintén gyanítható, hogy gondok lesznek a bittérképindexek kezelésével is. Például kihasználják, hogy a rekordok száma ugyanaz marad egész idő alatt. Hogyan fogjuk megtalálni az i-edik rekordot, amikor az állományhoz hozzáadódnak illetve törlődnek rekordok? Hasonlóan, egy mezőérték feltűnhet vagy eltűnhet. Hogyan találjuk meg hatékonyan egy érték bittérképét? Ezeket és a kapcsolódó kérdéseket az 5.4.4. részben tárgyaljuk.

Viszont a bittérképindexek előnye az, hogy a lekérdezések részleges egyezéssel sok esetben nagyon hatékonyan válaszolhatók meg a segítségével. Bizonyos értelemben a kosarak előnyeit kínálják, amit a 4.16. példában tárgyaltunk, ahol megkaptuk a `Film` azon sorait, amelyekben néhány attribútum értéke adott volt anélkül, hogy először vissza kellett volna nyernünk minden egyes attribútumra az összes megfelelő rekordot. Egy példával illusztráljuk a lényeget.

5.21. példa: Visszautalunk a 4.16. példára, ahol a

```
Film(cím, év, hossz, gyártó)
```

relációt kérdeztük le a következő lekérdezéssel:

```
SELECT cím
FROM Film
WHERE gyártó = 'Disney' AND
      év = 1995;
```

Tegyük fel, hogy a `gyártó` és az `év` attribútumon is van bittérképindex. Akkor vehetjük az `év = 1995` és `gyártó = 'Disney'` vektorok metszetét, azaz a vektorok bitenkénti AND művelettel vett eredményét, ami azt a vektort eredményezi, amelyben az i-edik pozíción akkor és csak akkor van 1-es, ha az i-edik `Film` sor egy olyan filmhez tartozik, amit a Disney gyártott 1995-ben.

Ha vissza tudjuk nyerni a `Film` sorait a sorszámuk alapján, akkor csak azokat a blokkokat kell beolvasnunk, amelyek egy vagy több ilyen sort tartalmaznak, pontosan úgy, ahogy a 4.16. példában is tettük. Ahhoz, hogy a két bitvektor metszetét vegyük, be kell őket olvasni a memóriába, ami egy lemezműveletet igényel minden egyes blokkra, amelyet a két vektor egyike elfoglal. Mint említettük, később érintjük mindkét témát: az 5.4.4. részben a rekordok elérését a sorszámuk alapján, és az 5.4.2. részben annak biztosítását, hogy a bitvektorok ne foglaljanak el túl sok helyet. □

A bittérképindexek a tartománylekérdezések megválaszolását is segíthetik. A következőkben megnézünk egy példát, amely egyaránt mutatja azt, hogy hogyan használjuk őket tartománylekérdezéshez, valamint azt is részleteiben bemutatjuk rövid bitvektorokkal, hogy hogyan használhatók a bitvektorok bitenkénti AND és OR műveletei arra, hogy megtaláljuk a választ a lekérdezésünkre anélkül, hogy meg kellene néznünk egyetlen rekordot is azokon kívül, amelyekre szükségünk van.

5.22. példa: Tekintsük az aranyékszer-adatokat, amelyeket az 5.7. példában vezettünk be. Tegyük fel, hogy a példa tizenkét pontja az alábbi, 1-től 12-ig számozott rekord:

1: (25, 60)	2: (45, 60)	3: (50, 75)	4: (50, 100)
5: (50, 120)	6: (70, 110)	7: (85, 140)	8: (30, 260)
9: (25, 400)	10: (45, 350)	11: (50, 275)	12: (60, 260)

Az első elemre, az életkorra hét különböző érték van, így az életkor bittérképindexe a következő hét vektorból áll:

25: 100000001000	30: 000000010000	45: 010000000100
50: 001110000010	60: 000000000001	70: 000001000000
85: 000000100000		

A fizetés komponensre tíz különböző érték van, így a fizetés bittérképindexnek a következő tíz bitvektora van:

60: 110000000000	75: 001000000000	100: 000100000000
110: 000001000000	120: 000010000000	140: 000000100000
260: 000000010001	275: 000000000010	350: 000000000100
400: 000000001000		

Tegyük fel, hogy meg akarjuk találni azokat az ékszervásárlókat, akiknek életkora a 45–55, fizetése pedig a 100–200 tartományba esik. Először a tartományba eső életkor értékekhez tartozó bitvektorokat keressük meg, ebben a példában csak két ilyen van: 010000000100 és 001110000010 a 45, illetve az 55 értékhez tartozó. Ha vesszük a bitenkénti OR-művelet eredményét, akkor egy új bitvektorunk lesz, amelyben az i-edik helyen akkor és csak akkor van 1, ha az életkor a rekordban a kívánt tartományba esik. Ez a bitvektor a 011110000110.

Azután megkeressük azokat a bitvektorokat, amelyek a 100 és 200 ezer közé eső fizetésértékekhez tartoznak. Négy ilyen van, a megfelelő fizetésértékek: 100, 110, 120 és 140; a bitenkénti OR eredménye pedig: 000111100000.

Az utolsó lépés a bitenkénti AND értékét venni ennek a két vektornak, amiket az OR-művelettel számoltunk ki. Azaz:

011110000110 AND 000111100000 = 000110000000

Így arra jutottunk, hogy csak a negyedik és az ötödik rekord van a kívánt tartományban, amelyek az (50, 100) és az (50, 120) pontok. □

5.4.2. Tömörített bittérképek

Tegyük fel, hogy egy n rekordot tartalmazó állomány F mezőjén van egy bittérképindexünk, és m különböző érték fordul elő az állományban az F mezőben. Ekkor az index összes bitvektorában a bitek száma mn. Ha a blokkok mondjuk 4096 bájt hosszúak, akkor 32 768 bit fér egy blokkba, tehát a szükséges blokkok száma: $mn/32768$. Ez a szám lehet kicsi az egész állomány tárolásához szükséges blokkok számához viszonyítva, de minél nagyobb az m értéke, annál több helyet foglal le a bittérképindex.

Viszont, ha az m nagy, az 1-es a bitvektorokban nagyon ritka lesz, pontosabban annak valószínűsége, hogy egy tetszőleges bit 1-es: $1/m$. Ha az 1-es ritka, akkor lehetőségünk van úgy kódolni a bitvektorokat, hogy átlagosan sokkal kevesebb, mint n bitet tartalmazzanak. Egy szokásos módszer a *szakaszhosszkódolásnak* vagy sorkifejtő kódolásnak nevezett, ahol egy szakaszt – ami i darab egymás utáni 0 bitből, majd egy ezeket követő 1-esből áll – az i egész szám valamilyen megfelelő bináris kódjával ábrázolunk. Majd egymás után rakjuk a kódokat az összes szakaszra, és az így kapott bitsorozat a bitvektor kódolt változata.

Elképzelhető, hogy az i egészet egyszerűen úgy ábrázoljuk, hogy bináris számként írjuk fel. Azonban ez az egyszerű szerkezet nem mindig megfelelő, mert nem lehet a kódok sorozatát a benne foglalt szakaszok hosszának egyértelmű meghatározásával szétszedni (lásd a „A bináris számok nem megfelelőek a szakaszhosszkódoláshoz" című bekeretezett részt). Így az i egész szám kódja, ami a szakasz hosszát mutatja, bonyolultabb kell legyen, mint az egyszerű bináris ábrázolás.

A sok lehetséges kódolási szerkezet közül egyet fogunk használni. Létezik jobb, bonyolultabb szerkezet, ami az itt elért tömörítés mértékét majdnem kétszeresére tudja javítani, de csak akkor, ha a jellemző szakaszhossz nagyon nagy. A szerkezetünknél először meg kell határoznunk, hogy az i binárisan ábrázolva hány bitből áll. Ez a j szám, ami közelítőleg $\log_2 i$, unárisan ábrázolva $j - 1$ darab 1-esből és egy 0-sból áll. Aztán folytathatjuk az i bináris értékével.[3]

5.23. példa: Ha $i = 13$, akkor $j = 4$, azaz 4 bit kell az i bináris ábrázolásához. Így az i kódoltan 1110-val kezdődik. Ezt követi az i binárisan, vagyis 1101. Tehát a 13 kódolva 11101101.

Az $i = 1$ kódolva 01, és az $i = 0$ kódolva 00. Mindkét esetben $j = 1$, tehát egyetlen 0 a kezdet, és ezt a 0-t követi az i-t ábrázoló egy bit. □

Ha egymás mögé rakjuk a kódolt egészek sorozatait, mindig vissza tudjuk állítani a szakaszhosszak sorozatát, és ezért a eredeti bitvektor visszaállítható. Tegyük fel, hogy átnéztünk már valahány kódolt bitet, és most egy bitsorozat elején vagyunk, amely egy bizonyos i egész szám kódja. Továbbmegyünk az első 0-ig, így meghatározzuk a j értékét. Azaz, j egyenlő azon bitek számával, amennyit le kell olvasnunk, amíg elérünk az első 0-hoz (beleértve ezt a 0-t is a bitek számába). Ha már ismerjük a j értékét, akkor vegyük a következő j bitet, ez adja kettes számrendszerben ábrázolva az i egész számot. Továbbá, ha végignéztük az i-t ábrázoló biteket, akkor tudjuk, hol van a következő egész kódjának a kezdete, így meg tudjuk ismételni az eljárást.

[3] Ténylegesen, a $j = 1$ esetet leszámítva (azaz, ha $i = 0$, vagy $i = 1$), biztosak lehetünk abban, hogy az i kettes számrendszerben felírva 1-gyel kezdődik. Így számonként megspórolhatunk 1 bitet, ha ezt az 1-est elhagyjuk, és csak a maradék $j - 1$ bitet használjuk.

A bináris számok nem megfelelőek a szakaszhosszkódoláshoz

Tegyük fel, hogy az i darab 0-ból, és utána egy 1-esből álló szakaszhoz az i egész szám bináris értékét rendeljük. Akkor a 000101 bitvektor két szakaszból áll, amelyeknek a hossza 3, illetve 1. Ezek az egészek binárisan ábrázolva 11 és 1, tehát a 000101 szakaszhosszkódolásának eredménye 111. Azonban, hasonló számítás mutatja, hogy a 010001 bitvektor kódja szintén 111, és a 010101 a harmadik olyan, amelynek a kódja szintén 111. Így a 111 nem dekódolható egyértelműen bitvektorrá.

5.24. példa: Fejtsük vissza a 11101101001011 sorozatot. Az elején kezdve, a 4-edik biten találjuk az első 0-t, tehát $j = 4$. A következő 4 bit: 1101, tehát megállapíthatjuk, hogy az első egész a 13. A 001011 maradt, amit vissza kell fejtenünk.

Mivel az első bit 0, tudjuk, hogy a következő bit magát az egészet ábrázolja, ez a szám a 0. Így eddig a 13, 0 sorozatot fejtettük vissza, és a maradék visszafejtendő sorozat a 1011.

Az első 0-t a második pozíción találjuk, amiből következik, hogy az utolsó két bit ábrázolja az utolsó egészet, ami 3. A szakaszhosszak teljes sorozata tehát 13, 0, 3. Ezekből a számokból felépíthetjük a tényleges bitvektort: 0000000000000110001. □

Gyakorlatilag minden bitvektor, amit így dekódolunk, 1-essel végződik, és a záró 0-kat nem állítjuk vissza. Mivel feltételezhetően ismerjük az állományban lévő rekordok számát, a további 0-kat hozzá tudjuk adni. Azonban, mivel a 0 egy bitvektorban azt jelenti, hogy a megfelelő rekord nincs benne a kívánt halmazban, nem is kell tudnunk a rekordok teljes számát, és figyelmen kívül hagyhatjuk a záró 0-kat.

5.25. példa: Alakítsunk át néhány, az 5.22. példában szereplő bitvektort a mi szakaszhosszkódunkra. Az első három (25, 30, 45) életkorhoz tartozó vektorok: 100000001000, 000000010000, illetve 010000000100. Ezek közül az elsőhöz a (0, 7) szakaszhosszsorozat tartozik. A 0 kódja 00, a 7 kódja 110111. Így a 25 éves életkor bitvektora a 00110111 sorozattá alakul.

Hasonlóan, a 30 éves életkornak csak egy szakasza van, hét 0-val. Tehát ennek a kódja: 110111. A 45 éves kor bitvektorának két szakasza van (1, 7). Mivel az 1 kódja 01, és mint meghatároztuk, a 7 kódja 110111, a harmadik bitvektor kódja: 01110111. □

A tömörítés az 5.25. példában nem nagy. Azonban nem láthatjuk az igazi előnyöket, ha n, a rekordok száma kicsi. Hogy méltányolni tudjuk a kódolás értékét, tegyük

fel, hogy $m = n$, vagyis a mezőnek, amelyen a bittérképindexet létrehozzuk, minden értéke egyedi. Megjegyezzük, hogy egy i hosszú szakasz kódja körülbelül $2 \log_2 i$ bit. Mindegyik bitvektorban egyetlen 1-es van, tehát egyetlen szakaszból áll, és annak a szakasznak a hossza nem nagyobb, mint n. Így a $2 \log_2 n$ bit egy felső korlát ebben az esetben a bitvektor kódjának hosszára.

Mivel n bitvektor van az indexben (mert $m = n$), az indexet alkotó bitek teljes száma legfeljebb $2n \log_2 n$. Megjegyezzük, hogy kódolás nélkül n^2 bit kellett volna. Ha $n > 4$, akkor $2n \log_2 n < n^2$, és ahogy n nő a $2n \log_2 n$ tetszőlegesen kisebb lesz, mint n^2.

5.4.3. Műveletek szakaszhosszkódolt bitvektorokon

Ha bitenkénti AND vagy OR műveleteket kell végrehajtanunk kódolt bitvektorokon, nemigen van más választásunk, mint visszafejteni őket, és a műveletet az eredeti bitvektorokon hajtani végre. Azonban nem kell az egész dekódolást egyszerre végrehajtani. A tömörítési szerkezet, amit leírtunk, lehetővé teszi, hogy csak egy szakaszt fejtsünk vissza egyszerre, és így meg tudjuk határozni, hogy hol a következő 1-es mindegyik, a műveletben résztvevő bitvektorban. Ha OR a művelet, az eredménybe is 1-est teszünk azon a pozíción, ha AND a művelet, akkor és csak akkor teszünk 1-est, ha mindkét operandusban a következő 1-es ugyanazon a pozíción van. A leírt algoritmus bonyolult, de egy példa kellően világossá teheti.

5.26. példa: Tekintsük az 5.25. példában a 25 és 30 életkorra kapott kódolt bitvektorokat: 00110111, illetve 110111. Az első szakaszukat könnyen dekódolhatjuk; azt kapjuk, hogy 0, illetve 7. Azaz az első 1-es a 25-höz tartozó bitvektorban az első pozíción fordul elő, míg a 30 esetén a bitvektorban az első 1-es a nyolcadik helyen van. Így egy 1-est képezünk az első pozícióra.

Aztán dekódolnunk kell a 25 életkorhoz tartozó következő szakaszt, mivel az a bitvektor adhat még egyest a 8-as pozíció előtt, ahol a 30-hoz tartozó bitvektorban egyes van. Azonban a 25-ös életkor következő szakasza 7, ami azt jelenti, hogy a bitvektorban a következő 1-es a 9-dik helyen van. Tehát hat 0-t helyezünk el és egy 1-est a 8-dik pozícióra, ami a 30-hoz tartozó bitvektorból jön. Ez a bitvektor nem járul hozzá több 1-essel az eredményhez. Az egyes a 9-dik pozícióra a 25-höz tartozó bitvektor alapján kerül, és ez a bitvektor sem ad további 1-eseket.

Arra jutottunk, hogy a két bitvektor OR műveletének eredménye 100000011. Az eredeti bitvektorok hosszát nézve, ami 12, láthatjuk, hogy ez nincs teljesen rendben, ugyanis három záró 0 bit lemaradt. Ha tudjuk, hogy az állományban a rekordok száma 12, hozzáadhatjuk azokat a 0-kat a végéhez. Azonban érdektelen, hogy hozzáragasztjuk-e azokat a 0-kat, mert csak 1-es bit esetén kell beolvasnunk rekordot. Ebben a példában nem fogjuk beolvasni a 10 és 12 közötti rekordokat semmiképpen. □

5.4.4. Bittérképindexek kezelése

Leírtuk a bittérképindexek műveleteit anélkül, hogy három fontos kérdést említettünk volna:

1. Amikor keresünk egy bitvektort egy adott értékhez, vagy bitvektorokat, amik megfelelnek egy tartományba eső értékeknek, hogyan tehetjük ezt hatékonyan?
2. Ha kiválasztottunk egy rekordhalmazt, ami válasz a kérdésünkre, hogyan nyerhetjük vissza azokat a rekordokat hatékonyan?
3. Ha rekordok beszúrása vagy törlése megváltoztatja az adatállományt, hogyan igazítjuk hozzá a változásokhoz egy adott mező bittérképindexét?

Bitvektorok keresése

Az első kérdés megválaszolható olyan technikák alapján, amiket már tanultunk. Gondoljunk úgy a bitvektorokra, mint rekordokra, amelyeknek a kulcsa a bitvektornak megfelelő érték (bár maga az érték nincs benne a „rekordban"). Ekkor bármilyen másodlagos indexelési technika hatékonyan támogatja az értékekhez tartozó bitvektorok elérését. Például, ha B-fát használunk, amelynek levelei kulcs-mutató párokat tartalmaznak; a mutató a kulcsértékhez tartozó bitvektorhoz vezet. A B-fa gyakran jó választás, mert könnyen támogatja a tartománylekérdezéseket, de a tördelőtáblák vagy az indexszekvenciális állományok szintén választhatók.

A bitvektorokat szintén tárolni kell valahol. Legjobb úgy elképzelni őket, mint változó hosszú rekordokat, mivel általában nőni fognak, ahogy egyre több rekordot adunk az adatállományhoz. Ha a bitvektorok – esetleg tömörített formában – jellemzően rövidebbek, mint egy blokk, akkor megfontolandó, hogy többet rakjunk egy blokkba, és átrendezzük őket, ha szükséges. Ha a bitvektorok tipikusan hosszabbak egy blokknál, akkor meggondolandó, hogy blokkok láncában tároljuk egyesével őket. A 3.4. rész technikái hasznosak lehetnek.

Rekordok keresése

Most gondoljuk át a második kérdést: ha egyszer meghatároztuk, hogy szükségünk van az adatállomány k-adik rekordjára, hogyan találjuk meg? Ismét csak alkalmazhatók a már ismert technikák. Képzeljük el a k-adik rekordot úgy, mint aminek k a keresési kulcsa (bár ez a kulcs nincsen benne a rekordban). Ekkor létrehozhatunk egy másodlagos indexet az adatállományhoz, aminek a keresési kulcsa a rekordszám.

Ha nincs rá ok, hogy az állományt valami más módon szervezzük, a rekordsorszámot akár az elsődleges index keresési kulcsaként is használhatjuk, ahogy a 4.1. részben tárgyaltuk. Ekkor az állomány szervezése különösen egyszerű, mivel a rekordsorszámok sosem változnak (még törléskor sem), és az új rekordokat csak az adatállomány végéhez kell adnunk. Így az adatállomány blokkjait teljesen tele lehet rakni

ahelyett, hogy az állomány közepén a beszúrások számára külön helyet kellene hagyni, mint ahogy szükséges volt a 4.6. részben, az indexszekvenciális állományok általános eseténél.

Adatállományok módosításának kezelése

Két szempontból jelentenek problémát az adatállomány-módosítások a bittérképindexre:

1. A rekordsorszámok nem változhatnak, ha egyszer kiosztották azokat.
2. Az adatállomány változásai szükségessé teszik a bittérképindex változtatását is.

Az 1. pont következménye, hogy ha töröljük az i rekordot, a legegyszerűbb „nyugdíjazni" a számát, a helyére pedig egy „sírkövet" tenni az adatállományban. A bittérképindexet szintén változtatni kell, mivel a bitvektorban, amelyben 1-es van az i-edik helyen, az 1-est 0-ra kell cserélni. Megjegyezzük, hogy meg tudjuk találni a megfelelő bitvektort, mert tudjuk, hogy milyen érték volt az i-edik rekordban a törlés előtt.

Következőként tekintsük át az új rekord beszúrását. Tárolhatjuk a következő lehetséges rekordszámot, és ezt rendeljük hozzá az új rekordhoz. Azután minden bittérképindexhez meg kell határoznunk az új rekord hozzá tartozó mezőjében lévő értéket, és az ahhoz az értékhez tartozó bitvektort módosítani kell úgy, hogy a végéhez hozzáteszünk egy 1-est. Gyakorlatilag ennek az indexnek az összes többi bitvektora kap egy 0-t a végére, de ha olyan tömörítési eljárást használunk, mint az 5.4.2. részben, akkor nem szükséges a tömörített értékeket változtatni.

Speciális eset, ha az új rekord olyan értéket tartalmaz az indexelt mezőben, ami még nem fordult elő. Ez esetben ehhez az értékhez szükségünk van egy új bitvektorra, és ezt a bitvektort és a megfelelő értéket be kell szúrni abba a másodlagos indexstruktúrába, amit egy adott értékhez tartozó bitvektor visszakereséséhez használunk.

Végül nézzük az adatállomány i-edik rekordjának a módosítását, amikor egy olyan mező változik mondjuk v értékről w-re, amelyhez van bittérképindex. Meg kell találnunk a v érték bitvektorát, és az i-edik pozíción az 1-est 0-ra kell váltani. Ha van bitvektor a w-hez, akkor a 0-t az i-edik helyen 1-re kell változtatni. Ha még nincs bitvektor a w-hez, akkor létrehozunk egyet, ahogy az előző bekezdésben leírtuk azt az esetet, amikor a beszúrás egy új értéket eredményez.

5.4.5. Feladatok

5.4.1. feladat: Az 5.10. ábra adataival mutassuk meg a bittérképindexeket a következő attribútumokra:

* a) sebesség,
b) memória és
c) merevlemez,

mind i) nem tömörített alakban, mind ii) tömörített alakban, az 5.4.2. rész szerkezetét használva.

5.4.2. feladat: Az 5.22. példa bittérképeit használva keressük meg azokat az ékszervásárlókat, akiknek az életkora a 25–40 tartományban van, és a fizetésük 0 és 100 közötti.

5.4.3. feladat: Tekintsünk egy 1 000 000 rekordot tartalmazó állományt, amely az F mezőjében m különböző értéket tartalmaz.

a) Hány bájtos az F mező bittérképindexe az m függvényében?
! b) Tegyük fel, hogy az 1-től 1 000 000-ig számozott rekordokban az F mező értékei round-robin módon adottak, így minden egyes érték előfordul bármely m egymás utáni rekordban. Hány bájtot használna fel egy tömörített index?

!! **5.4.4. feladat:** Az 5.4.2. részben említettük, hogy csökkenteni lehetne a bitek számát $2 \log_2 i$-ről – amennyit abban a részben használtunk az i szám kódolásához –, közel $\log_2 i$-ig. Mutassuk meg, hogyan lehet tetszőlegesen megközelíteni ezt a határt, amennyiben az i elég nagy. *Tanács:* Az i bináris kódjának hosszát unárisan kódoltuk. Tudnánk-e a kód hosszát binárisan kódolni?

5.4.5. feladat: Kódoljuk a következő bittérképeket az 5.4.2. részben használt szerkezetet használva:

* a) 0110000000100000100.
b) 1000001000000010011101.
c) 0001000000000010000010000.

*! **5.4.6. feladat:** Rámutattunk, hogy a tömörített bittérképindexek közelítőleg $2n \log_2 n$ bitet használnak fel egy n rekordos állomány esetén. Hasonlítsuk össze ezt a bitmennyiséget egy B-fa-index által felhasznált bitek számával. Emlékezzünk rá, hogy a B-fa-index mérete függ a kulcs és a mutató hosszától, és (bizonyos mértékig) a blokk méretétől is. Használhatunk azonban ésszerű becsléseket ezekre a paraméterekre a számításainkban. Miért részesítjük esetleg előnyben a B-fákat, még akkor is, ha több helyet foglalnak le, mint a tömörített bittérképek?

5.5. Összefoglalás

- *Többdimenziós adatok:* Sok alkalmazás, mint például a térképészeti adatbázisok vagy az eladási és készletnyilvántartó adatok, felfoghatók úgy, mint pontok egy két- vagy többdimenziós térben.
- *Többdimenziós indexeket igénylő lekérdezések:* Azok a lekérdezéstípusok, amelyeket a többdimenziós adatokon támogatni kell, többek között: a lekérdezések részle-

ahelyett, hogy az állomány közepén a beszúrások számára külön helyet kellene hagyni, mint ahogy szükséges volt a 4.6. részben, az indexszekvenciális állományok általános eseténél.

Adatállományok módosításának kezelése

Két szempontból jelentenek problémát az adatállomány-módosítások a bittérképindexre:

1. A rekordsorszámok nem változhatnak, ha egyszer kiosztották azokat.
2. Az adatállomány változásai szükségessé teszik a bittérképindex változtatását is.

Az 1. pont következménye, hogy ha töröljük az i rekordot, a legegyszerűbb „nyugdíjazni" a számát, a helyére pedig egy „sírkövet" tenni az adatállományban. A bittérképindexet szintén változtatni kell, mivel a bitvektorban, amelyben 1-es van az i-edik helyen, az 1-est 0-ra kell cserélni. Megjegyezzük, hogy meg tudjuk találni a megfelelő bitvektort, mert tudjuk, hogy milyen érték volt az i-edik rekordban a törlés előtt.

Következőként tekintsük át az új rekord beszúrását. Tárolhatjuk a következő lehetséges rekordszámot, és ezt rendeljük hozzá az új rekordhoz. Azután minden bittérképindexhez meg kell határoznunk az új rekord hozzá tartozó mezőjében lévő értéket, és az ahhoz az értékhez tartozó bitvektort módosítani kell úgy, hogy a végéhez hozzáteszünk egy 1-est. Gyakorlatilag ennek az indexnek az összes többi bitvektora kap egy 0-t a végére, de ha olyan tömörítési eljárást használunk, mint az 5.4.2. részben, akkor nem szükséges a tömörített értékeket változtatni.

Speciális eset, ha az új rekord olyan értéket tartalmaz az indexelt mezőben, ami még nem fordult elő. Ez esetben ehhez az értékhez szükségünk van egy új bitvektorra, és ezt a bitvektort és a megfelelő értéket be kell szúrni abba a másodlagos indexstruktúrába, amit egy adott értékhez tartozó bitvektor visszakereséséhez használunk.

Végül nézzük az adatállomány i-edik rekordjának a módosítását, amikor egy olyan mező változik mondjuk v értékről w-re, amelyhez van bittérképindex. Meg kell találnunk a v érték bitvektorát, és az i-edik pozíción az 1-est 0-ra kell váltani. Ha van bitvektor a w-hez, akkor a 0-t az i-edik helyen 1-re kell változtatni. Ha még nincs bitvektor a w-hez, akkor létrehozunk egyet, ahogy az előző bekezdésben leírtuk azt az esetet, amikor a beszúrás egy új értéket eredményez.

5.4.5. Feladatok

5.4.1. feladat: Az 5.10. ábra adataival mutassuk meg a bittérképindexeket a következő attribútumokra:

* a) sebesség,
b) memória és
c) merevlemez,

mind i) nem tömörített alakban, mind ii) tömörített alakban, az 5.4.2. rész szerkezetét használva.

5.4.2. feladat: Az 5.22. példa bittérképeit használva keressük meg azokat az ékszervásárlókat, akiknek az életkora a 25–40 tartományban van, és a fizetésük 0 és 100 közötti.

5.4.3. feladat: Tekintsünk egy 1 000 000 rekordot tartalmazó állományt, amely az F mezőjében m különböző értéket tartalmaz.

a) Hány bájtos az F mező bittérképindexe az m függvényében?
! b) Tegyük fel, hogy az 1-től 1 000 000-ig számozott rekordokban az F mező értékei round-robin módon adottak, így minden egyes érték előfordul bármely m egymás utáni rekordban. Hány bájtot használna fel egy tömörített index?

!! **5.4.4. feladat:** Az 5.4.2. részben említettük, hogy csökkenteni lehetne a bitek számát $2 \log_2 i$-ről – amennyit abban a részben használtunk az i szám kódolásához –, közel $\log_2 i$-ig. Mutassuk meg, hogyan lehet tetszőlegesen megközelíteni ezt a határt, amennyiben az i elég nagy. *Tanács:* Az i bináris kódjának hosszát unárisan kódoltuk. Tudnánk-e a kód hosszát binárisan kódolni?

5.4.5. feladat: Kódoljuk a következő bittérképeket az 5.4.2. részben használt szerkezetet használva:

* a) 0110000000100000100.
b) 1000001000000100 1101.
c) 000100000000001000001000 0.

*! **5.4.6. feladat:** Rámutattunk, hogy a tömörített bittérképindexek közelítőleg $2n \log_2 n$ bitet használnak fel egy n rekordos állomány esetén. Hasonlítsuk össze ezt a bitmenynyiséget egy B-fa-index által felhasznált bitek számával. Emlékezzünk rá, hogy a B-fa-index mérete függ a kulcs és a mutató hosszától, és (bizonyos mértékig) a blokk méretétől is. Használhatunk azonban ésszerű becsléseket ezekre a paraméterekre a számításainkban. Miért részesítjük esetleg előnyben a B-fákat, még akkor is, ha több helyet foglalnak le, mint a tömörített bittérképek?

5.5. Összefoglalás

- *Többdimenziós adatok:* Sok alkalmazás, mint például a térképészeti adatbázisok vagy az eladási és készletnyilvántartó adatok, felfoghatók úgy, mint pontok egy két- vagy többdimenziós térben.
- *Többdimenziós indexeket igénylő lekérdezések:* Azok a lekérdezéstípusok, amelyeket a többdimenziós adatokon támogatni kell, többek között: a lekérdezések részle-

ges egyezéssel (a dimenziók egy részhalmazára megadott értékeknek megfelelő pontok), a tartománylekérdezések (az egyes dimenziókra megadott tartományokon belüli pontok), a legközelebbi szomszéd (egy adott ponthoz legközelebbi pont), és a „hol-vagyok-én" (régió vagy régiók, amelyek egy adott pontot tartalmaznak).

- *Legközelebbi szomszéd-lekérdezések végrehajtása:* Sok adatstruktúra lehetővé teszi a legközelebbi szomszéd-lekérdezések végrehajtását úgy, hogy végrehajtunk egy tartománylekérdezést a kiszemelt pont körül, és megnöveljük a tartományt, ha nincs pont abban a tartományban. Óvatosnak kell lenni, mert hiába találunk pontot egy téglalap alakú tartományban, az nem feltétlenül zárja ki annak a lehetőségét, hogy van közelebbi pont a téglalapon kívül.
- *Rácsos állományok:* A rácsos állomány felszeleteli a pontok terét minden egyes dimenzió mentén. A rácsvonalak távolsága lehet különböző, és dimenziónként lehet különböző számú rácsvonal. A rácsos állományok jól támogatják a tartománylekérdezéseket, a lekérdezéseket részleges egyezéssel, és a legközelebbi szomszéd-lekérdezéseket, legalábbis, ha az adatok viszonylag egyenletes eloszlásúak.
- *Particionált tördelőtáblák:* Egy particionált tördelőfüggvény minden egyes dimenzióból a kosár számának néhány bitjét képezi. A részleges egyezéses lekérdezéseket jól támogatják, és hatékonyságuk nem függ az adatok egyenletes eloszlásától.
- *Többkulcsos indexek:* Egy egyszerű többdimenziós struktúra, amelynek van egy gyökere, ami index az egyik attribútumon, és ez elvezet egy második attribútumon lévő indexek gyűjteményéhez, amik egy harmadik attribútum indexeihez vezethetnek és így tovább. A tartomány és a legközelebbi szomszéd-lekérdezésekhez hasznosak.
- *kd-fák:* Ezek a fák olyanok, mint a bináris keresőfák, de a különböző szinteken különböző attribútumok szerint ágaznak el. A részleges egyezéses, a tartomány és a legközelebbi szomszéd-lekérdezéseket egyaránt jól támogatják. A facsomópontok meggondolt blokkokba csomagolása szükséges ahhoz, hogy alkalmassá tegyük a másodlagos tárolókon végzett műveletekhez.
- *Quad-fák:* A quad-fa a többdimenziós kockát „negyedekre" osztja, és rekurzívan tovább osztja a „negyedeket" ugyanolyan módon, ha túl sok pont van bennük. A részleges egyezéses, a tartomány és a legközelebbi szomszéd-lekérdezéseket támogatják.
- *R-fák:* Az ilyen fajta fa általában régiók gyűjteményét ábrázolja, nagyobb régiók hierarchiájába csoportosítva azokat. Segíti a „hol-vagyok-én" lekérdezéseket, és ha az elemi régiók ténylegesen pontok, más – ebben a fejezetben tanulmányozott – lekérdezéseket is támogat.
- *Bittérképindexek:* A többdimenziós lekérdezéseket egy olyan index támogatja, amely a pontokat vagy a rekordokat rendezi, és bitvektorokkal jelöli azoknak a rekordoknak a helyét, amelyeknek egy adott értéke van a megfelelő attribútumon. Ezek az indexek a tartomány-, legközelebbi szomszéd- és a részleges egyezéses lekérdezéseket támogatják.
- *Tömörített bittérképek:* Azért, hogy helyet takarítsunk meg, a bittérképindexeket – amelyek gyakran nagyon kevés 1-est tartalmazó vektorokból állnak – tömörítjük a szakaszhosszkódolást használva.

5.6. Irodalomjegyzék

A legtöbb ebben a fejezetben tárgyalt adatstruktúra az 1970-es években és az 1980-as évek elején végzett kutatások terméke. A *kd*-fákat [2] vezeti be. A másodlagos tárolókra megfelelő módosítások [3]-ban és [13]-ban jelennek meg. A particionált tördelést és annak használatát részleges egyezéses lekérdezésekre [12] és [5] tárgyalja. Viszont az 5.2.8. feladat tervezési elképzelése [14]-ből származik.

A rácsos állományok [9]-ben jelennek meg. A quad-fák [6]-ban találhatók meg. Az R-fákat [8] vezeti be, és két jól ismert kiterjesztése [15] és [1].

A bittérképindexnek érdekes a története. Egy Nucleus nevű cég, amelyet Ted Glaser alapított, szabadalmaztatta az ötletét és kifejlesztett egy adatbázis-kezelő rendszert, amelyikben az indexstruktúra és az adatábrázolás is bittérképindex volt. A vállalkozás az 1980-as évek végén csődbe ment, de az ötletet napjainkban is beépítették több nagy kereskedelmi adatbázisrendszerbe. Az első publikáció ebben a témában [10]. Az ötlet legújabb kiterjesztése [11].

A többdimenziós tárolási struktúráknak számos összegzése van. Az egyik legkorábbi [4]. A legújabb áttekintések [16]-ban és [7]-ben találhatók. Az előbbi több, egyéb jelentős adatbázis témakörről szóló tanulmányt is tartalmaz.

1. N. Beekmann, H.–P. Kriegel, R. Schneider, and B. Seeger, „The R*-tree: an efficient and robust access method for points and rectangles," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1990), pp. 322–331.
2. J. L. Bentley, „Multidimensional binary search trees used for associative searching," *Comm. ACM* **18**:9 (1975), pp. 509–517.
3. J. L. Bentley, „Multidimensional binary search trees in database applications," *IEEE Trans. on Software Engineering* **SE-5**:4 (1979), pp. 333–340.
4. J. L. Bentley and J. H. Friedman, „Data structures for range searching," *Computing Surveys* **13**:3 (1979), pp. 397–409.
5. W. A. Burkhard, „Hashing and trie algorithms for partial match retrieval," *ACM Trans. on Database Systems* **1**:2 (1976), pp. 175–187.
6. R. A. Finkel and J. L. Bentley, „Quad trees, a data structure for retrieval on composite keys," *Acta Informatica* **4**:1 (1974), pp. 1–9.
7. V. Gaede and O. Gunther, „Multidimensional access methods," *Computing Surveys* **30**:2 (1998), pp. 170–231.
8. A. Guttman, „R-trees: a dynamic index structure for spatial searching," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1984), pp. 47–57.
9. J. Nievergelt, H. Hinterberger, and K. Sevcik, „The grid file: an adaptable, symmetric, multikey file structure," *ACM Trans. on Database Systems* **9**:1 (1984), pp. 38–71.
10. P. O'Neil, „Model 204 architecture and performance," *Proc. Second Intl. Workshop on High Performance Transaction Systems,* Springer-Verlag, Berlin, 1987.
11. P. O'Neil and D. Quass, „Improved query performance with variant indexes," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1997), pp. 38–49.

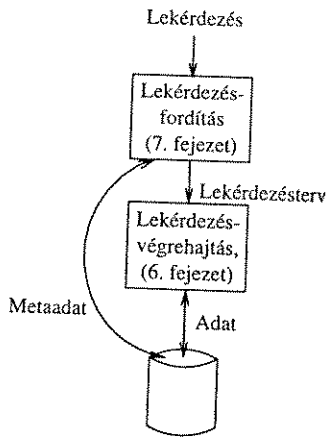
12. R. L. Rivest, „Partial match retrieval algorithms," *SIAM J. Computing* 5:1 (1976), pp. 19–50.
13. J. T. Robinson, „The K-D-B-tree: a search structure for large multidimensional dynamic indexes," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1981), pp. 10–18.
14. J. B. Rothnie Jr. and T. Lozano, „Attribute based file organization in a paged memory environment," *Comm. ACM* **17**:2 (1974), pp. 63–69.
15. T. K. Sellis, N. Roussopoulos, and C. Faloutsos, „The R+-tree: a dynamic index for multidimensional objects," *Proc. Intl. Conf. on Very Large Databases* (1987), pp. 507–518.
16. C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari, *Advanced Database Systems,* Morgan-Kaufmann, San Francisco, 1997.

6. fejezet

Lekérdezések végrehajtása

Az előző fejezetekben megismerhettük azokat az adatszerkezeteket, amelyek olyan alapvető adatbázis-műveleteket támogatnak, mint például sorok megtalálása megadott keresési kulcs alapján. Most már készen állunk arra, hogy ezeket az adatszerkezeteket felhasználjuk a lekérdezések megválaszolására szolgáló hatékony algoritmusok készítésekor. A lekérdezésfeldolgozás átfogó témáját a 7. fejezet mutatja be. A *lekérdezésfeldolgozó* (query processor) egy relációs adatbázis-kezelő komponenseinek azon csoportja, amelyik a felhasználó lekérdezéseit, valamint adatmódosító utasításait lefordítja adatbázis-műveletekre, és végre is hajtja ezeket a műveleteket. Mivel az SQL a lekérdezések igen magas szintű megfogalmazását teszi lehetővé a számunkra, ezért a lekérdezésfeldolgozónak még igen sok részletet kell megadnia a lekérdezés végrehajtására vonatkozólag. Ráadásul egy lekérdezés naiv végrehajtási stratégiája olyan végrehajtási algoritmust eredményezhet, amely a szükségesnél jóval több időt vesz igénybe.

A 6.1. ábrán láthatjuk a 6. és a 7. fejezetek közötti témamegosztást. Ebben a fejezetben a lekérdezés végrehajtására koncentrálunk, ami tulajdonképpen az adatbázis



6.1. ábra. *A lekérdezésfeldolgozó fontosabb részei*

adatait manipuláló algoritmusok összessége. A relációs algebra áttekintésével fogunk kezdeni. A legtöbb adatbázis-kezelő rendszer ezt vagy valami hasonló jelölést használ a felhasználó által SQL-ben megfogalmazott lekérdezések belső ábrázolására. A relációs algebra olyan – az olvasó számára talán már ismert – műveleteket tartalmaz, mint az összekapcsolás és az egyesítés. Az SQL azonban inkább az adatok multihalmaz modelljét alkalmazza és nem a halmaz modellt. Ráadásul SQL-ben vannak olyan műveletek is, amelyek a klasszikus relációs algebrának nem részei, mint például az összesítés, a csoportosítás és a rendezés. Az SQL-lekérdezések ábrázolásában betöltött szerepe alapján át kell tehát újra gondolnunk ezt a relációs algebrát.

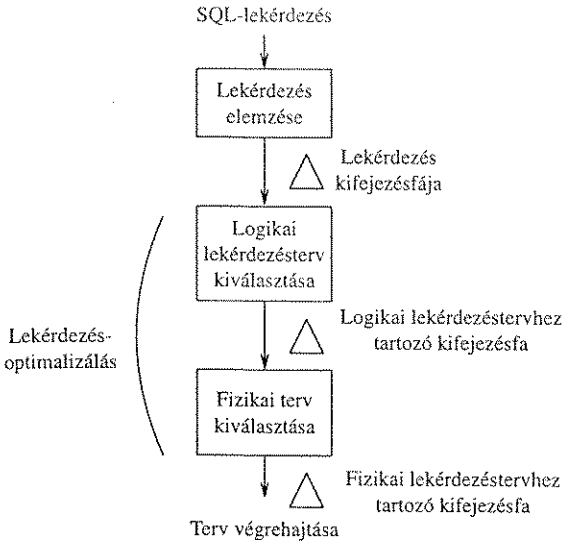
A relációs algebra használatának egyik előnye, hogy megkönnyíti a lekérdezések különböző lehetséges formáinak kifejezését. Az egy lekérdezéshez tartozó különböző algebrai kifejezéseket *logikai lekérdezésterveknek* (logical query plans) nevezzük. Ezeket a terveket gyakran kifejezésfákkal ábrázoljuk. Ebben a könyvben is ezt a jelölésmódot használjuk majd.

Ebben a fejezetben rendszerezzük a relációs algebrai műveletek végrehajtására szolgáló főbb módszereket. Ezek a módszerek az alapstratégiában különböznek egymástól. A legfontosabb megközelítések a végigpásztázás, a tördelés, a rendezés és az indexelés. A módszerek abban is különböznek, hogy milyen előfeltételezést használnak a rendelkezésre álló memória méretéről. Egyes algoritmusok feltételezik, hogy a műveletben érintett relációk közül legalább az egyik befér a memóriába. Más algoritmusok azt feltételezik, hogy a művelet valamennyi argumentuma túl nagy ahhoz, hogy beférjen a memóriába, és ezen algoritmusok költsége és szerkezete jelentősen eltér az előzőektől.

A lekérdezésfordítás áttekintése

A lekérdezésfordítás három fontosabb lépésre osztható fel, ahogyan az a 6.2. ábrán is látható:

a) *Elemzés* (parsing), amelynek során egy – a lekérdezést és annak szerkezetét jellemző – *elemző fát* (parse tree) építünk fel.
b) *Lekérdezésátírás* (query rewrite), amelynek során az elemző fát átkonvertáljuk egy kezdeti lekérdezéstervvé, amely rendszerint a lekérdezésnek egy algebrai megvalósítása. Ezt a kezdeti tervet később átalakítjuk egy olyan ekvivalens tervvé, amelynek végrehajtási ideje várhatóan kisebb lesz.
c) *Fizikai terv előállítása* (physical plan generation), amelyben a b) pontban megkapott absztrakt lekérdezéstervet, amelyet gyakran *logikai lekérdezéstervnek* (logical query plan) nevezünk, átalakítjuk *fizikai lekérdezéstervvé* (physical query plan) oly módon, hogy a logikai terv valamennyi operátorának megvalósítására kiválasztunk egy algoritmust, és meghatározzuk ezen operátorok végrehajtási sorrendjét. A fizikai tervet egy kifejezésfával ábrázoljuk, éppúgy mint az elemzés eredményét és a logikai tervet. A fizikai terv olyan részleteket is tartalmaz, mint a lekérdezésben szereplő relációkhoz történő hozzáférés, illetve, hogy egy relációt kell-e rendezni, és ha igen, mikor.



6.2. ábra. *A lekérdezésfordítás áttekintése*

A b) és c) részeket együtt gyakran nevezzük *lekérdezésoptimalizálásnak* (query optimizer), és általában ezek a lépések a lekérdezésfordítás legnehezebb részei. A 7. fejezetet a lekérdezésoptimalizálásának szenteljük. Itt megtanulhatjuk, hogy miként kell kiválasztani azt a „lekérdezéstervet", amelynek a legrövidebb a végrehajtási ideje. A legjobb lekérdezésterv kiválasztásához a következőket kell eldöntenünk:

1. A lekérdezéssel ekvivalens algebrai formák közül melyik vezet a lekérdezés megválaszolásának leghatékonyabb algoritmusához?
2. A kiválasztott forma operátorainak megvalósításához melyik algoritmust használjuk?
3. Az operátorok milyen formában adják át egymásnak az adatokat? A csővezetékek módszerét használva, memóriapufferekben avagy lemezen?

Valamennyi döntés az adatbázis metaadataitól függ. A lekérdezésoptimalizáló számára rendelkezésre álló jellegzetes metaadatok a következőket foglalják magukban: az egyes relációk mérete, olyan statisztikák, mint például egy attribútum különböző értékeinek gyakorisága, bizonyos indexek létezése, az adatok elhelyezkedése a lemezen.

6.1. Algebrai megközelítés

Ahhoz, hogy a lekérdezések végrehajtására szolgáló jó algoritmusokról beszélhessünk, először szükségünk van a lekérdezéseket alkotó elemi műveletek jelölésének kidolgozására. Számos SQL-lekérdezés kifejezhető a klasszikus „relációs algebrát" alkotó néhány operátorral, és még az objektumorientált lekérdezőnyelvek is lényegében

ugyanazokat a műveleteket végzik el, amelyek a relációs algebrában megtalálhatók. Az SQL-nek és más lekérdezőnyelveknek is vannak azonban olyan lehetőségei, amelyek nem fejezhetők ki a klasszikus relációs algebra segítségével. Éppen ezért, miután ismertettük ezt az algebrát, be fogunk vezetni az SQL lehetőségeit szolgáló operátorokat, olyanokat mint a csoportosítás, a rendezés, és az ismétlődések kiküszöbölése.

A relációs algebrát ráadásul eredetileg úgy tervezték, mintha a relációk halmazok lennének. SQL-ben azonban a relációk *multihalmazok*. Ez azt jelenti, hogy ugyanaz a sor többször is megjelenhet egy SQL-relációban. Ezért úgy vezetjük be a relációs algebrát, mint egy multihalmazokon értelmezett algebrát. A relációs algebrai operátorok a következők:

- *Egyesítés, metszet és különbség*: Halmazokon végezve megegyeznek a hagyományos halmazműveletekkel. Multihalmazokon végezve van néhány különbség, amit a 6.1.1. részben fogunk bemutatni. Ezek az operátorok megfelelnek a `UNION`, az `INTERSECT` és az `EXCEPT` SQL-operátoroknak.
- *Kiválasztás:* Ez az operátor egy új relációt eredményez egy régi reláció bizonyos sorainak kiválasztásával. A kiválasztás valamilyen feltételen vagy predikátumon alapul. Nagyjából megfelel egy SQL-lekérdezés `WHERE` záradékának.
- *Vetítés:* Ez az operátor egy új relációt eredményez egy régi reláció bizonyos oszlopainak kiválasztásával, hasonló módon, mint egy SQL-lekérdezés `SELECT` záradéka. Ki fogjuk terjeszteni a klasszikus relációs algebra eme operátorát azzal, hogy engedélyezzük az attribútumok átnevezését, valamint olyan attribútumok létrehozását, amelyeket a régi reláció attribútumaival és konstansokkal végzett műveletekkel hozunk létre éppúgy, mint az SQL `SELECT` záradékában.
- *Szorzat:* Ez az operátor tulajdonképpen a halmaz alapú Descartes-szorzat (vagy keresztszorzat), amely úgy hoz létre sorokat, hogy a két reláció sorait az összes lehetséges módon összepárosítja. Ez megfelel az SQL `FROM` záradékában felsorolt relációk listájának, amelyek szorzata alkotja azt a relációt, amelyre a `WHERE` záradék feltételét és a `SELECT` záradék vetítését alkalmazzuk.
- *Összekapcsolás:* Több különböző típusú összekapcsolási operátor van, amelyek megfelelnek az SQL2-szabvány `JOIN`, `NATURAL JOIN` és `OUTER JOIN` operátorainak. Ezekről a 6.1.5. részben olvashatunk majd.

Mindezek mellett kiegészítjük a relációs algebrát a következő operátorokkal, amelyeket abból a célból vezettünk be, hogy az összes lehetséges SQL-lekérdezés optimalizálását tárgyalni tudjuk.

- *Ismétlődések kiküszöbölése.* Ezzel az operátorral halmazt készíthetünk egy multihalmazból éppúgy, mint az SQL `SELECT` záradékának `DISTINCT` kulcsszavával.
- *Csoportosítás.* Ezt az operátort azzal a céllal terveztük, hogy utánozza egy SQL `GROUP BY` hatását, valamint az olyan összesítő operátorokét (összeg, átlag stb.), amelyek egy `SQL SELECT` záradékban előfordulhatnak.
- *Rendezés.* Ez az operátor az SQL `ORDER BY` záradékának hatását reprezentálja. Használatos továbbá más operátorokhoz (pl. összekapcsolás) tartozó rendezés alapú algoritmusok részeként.

6.1.1. Egyesítés, metszet és különbség

Ha a relációk halmazok lennének, akkor az $\cup$, $\cap$ és $-$ operátorok a megszokott operátorok volnának. Van azonban két fontos különbség az SQL-relációk és a halmazok között:

a) A relációk multihalmazok.
b) A relációk rendelkeznek sémával, ami nem más, mint az oszlopok neveinek megfelelő attribútumhalmaz.

A b) problémával könnyű megbirkózni. Az egyesítésnél, metszetnél és a különbségnél megköveteljük, hogy a két argumentum reláció sémája megegyezzen. Ily módon az eredményül kapott reláció sémája is ugyanaz lesz, mint az argumentumoké.

Az a) azonban néhány új definíciót tesz szükségessé, mivel az egyesítés, a metszet és a különbség egy kissé másképpen működik multihalmazokon, mint halmazokon. Az eredmény felépítésére vonatkozó szabályok a következőképpen módosulnak:

1. Az $R \cup S$ esetén egy t sor annyiszor fordul elő az eredményben, ahányszor előfordul az R-ben plusz ahányszor előfordul az S-ben.
2. Az $R \cap S$ esetén egy t sor annyiszor fordul elő az eredményben, amennyi az R-ben és az S-ben levő előfordulások minimuma.
3. Az $R - S$ esetén egy t sor annyiszor fordul elő az eredményben, ahányszor előfordul az R-ben mínusz ahányszor előfordul az S-ben, de soha nem kevesebbszer, mint nulla.

Figyeljük meg, hogyha az R és az S történetesen halmazok, vagyis egyikben sem jelenik meg egy elem kétszer, akkor az $R \cap S$, illetve $R - S$ eredménye pontosan ugyanaz, mint amit a halmazokon értelmezett operátoroktól elvárunk. Azonban még ha az R és S halmazok lennének is, a multihalmazos $R \cup S$ egyesítésnek akkor is lehet olyan végeredménye, ami nem halmaz. Nevezetesen, ha egy elem egyaránt megjelenik R-ben és S-ben is, akkor kétszer fog megjelenni az $R \cup S$-ben, míg a halmaz alapú egyesítésben csak egyszer szerepelne.

6.1. példa: Legyen az $R = \{A, B, B\}$ és $S = \{C, A, B, C\}$ két multihalmaz. Ekkor:

- $R \cup S = \{A, A, B, B, B, C, C\}$.
- $R \cap S = \{A, B\}$.
- $R - S = \{B\}$.

Az egyesítésben azért szerepel kétszer az A, mert az R és az S egyaránt tartalmaz egy A-t; és azért szerepel benne három B, mivel az R két B-t tartalmaz és az S egyet. Az egyesítés elemeit rendezett sorrendben tüntettük fel, de ne feledjük, hogy a sorrend nem számít, és a multihalmaz hét elemének sorrendjét tetszőleges módon permutálhatjuk.

A metszetben azért szerepel egy A, mert az R és az S egyaránt egy A-t tartalmaz,

így az A előfordulásainak minimuma 1. B-ből is egy szerepel, mert míg az R két B-t tartalmaz, az S csak egyet. A C egyáltalán nem jelenik meg a metszetben, habár kétszer is szerepel az S-ben, de egyszer sem szerepel az R-ben.

És végül a különbség nem tartalmaz A-t, habár az A megjelenik az R-ben, de az S-ben is megjelenik legalább annyiszor, mint az R-ben. A különbség egy B-t tartalmaz, mert az R-ben kétszer szerepel, az S-ben egyszer, és $2 - 1 = 1$. C-t sem tartalmaz, mivel a C nem szerepel az R-ben. Ily módon az $R - S$ szempontjából lényegtelen, hogy a C megjelenik-e az S-ben. □

A multihalmazokon végzett műveletek fenti szabályai függetlenek attól, hogy az R és S elemei sorok, objektumok vagy valami mások. A relációs algebránál azonban feltételezzük, hogy az R és S relációk, ily módon sémával rendelkeznek (ami tulajdonképpen egy olyan attribútumlista, amely a relációk oszlopainak nevét tartalmazza). Az egyesítés, metszet és különbség képzésénél megköveteljük, hogy a két reláció sémája megegyezzen. Az eredmény ugyanezzel a sémával fog rendelkezni, tehát az eredmény szintén egy reláció.

Alapértelmezésben, a `UNION`, `INTERSECT` és `EXCEPT` SQL-műveletek kiküszöbölik az eredményből az ismétlődéseket, még akkor is, ha az argumentum relációk tartalmaznak ismétlődéseket. Ezeknek a műveleteknek a multihalmazos változatát SQL-ben az `ALL` kulcsszó segítségével képezhetjük, ilyen például a `UNION ALL`. Megjegyzendő, hogy ezen műveletek alapértelmezett változatai SQL-ben nem biztos, hogy halmaz alapúak. Ezek inkább multihalmaz alapú műveletek, amelyeket követ a 6.1.6. részben bemutatott, ismétlődések kiküszöbölésére szolgáló δ-művelet.

Ebben a fejezetben bemutatjuk a megfelelő algoritmusokat az egyesítés, metszet és különbség műveletek halmaz alapú és multihalmaz alapú változataihoz egyaránt. A félreértések elkerülése érdekében úgy különböztetjük meg ezt a kétfajta operátort, hogy egy megfelelő alsó indexszel jelöljük azok típusát. S-sel jelöljük a halmaz (angolul „set") alapú műveleteket, B-vel a multihalmaz (angolul „bag") alapú műveleteket. Így például $\cup_S$ a halmaz alapú egyesítés, és $-_B$ a multihalmazos különbség. Ha egy operátornak nincs alsó indexe, akkor alapértelmezésben a multihalmazos változatot vesszük. Kivételt jelent majd a 7.2. rész, amelyben algebrai törvényszerűségeket adunk meg, és ahol az a célunk, hogy amikor nincs alsó index, akkor a törvényszerűség legyen érvényes a művelet mindkét verziójára.

6.1.2. Kiválasztás

A $\sigma_C(R)$ *kiválasztás* tartalmaz egy R relációt és egy C feltételt. A C feltétel magában foglalhat:

1. Konstansokra és/vagy attribútumokra alkalmazott aritmetikai (pl. +, *) vagy karakterlánc (pl. összefűzés, `LIKE`) operátorokat.
2. Az 1. segítségével felépített kifejezések összehasonlítását, pl. $a < b$ vagy $a + b = 10$.
3. A 2. segítségével felépített kifejezésekre alkalmazott `AND`, `OR` és `NOT` logikai összekapcsolásokat.

Alkérdések a `WHERE` záradékban

Az itt bevezetett σ operátor sokkal erőteljesebb, mint a relációs algebra hagyományos kiválasztás operátora, mivel megengedtük a σ alsó indexében az `AND`, `OR` és `NOT` logikai operátorokat. Azonban még ez a σ operátor sincs olyan erőteljes, mint a `WHERE` záradék az SQL-ben, mivel ott szerepelhetnek alkérdések és bizonyos relációkra értelmezett logikai operátorok, mint például az `EXISTS`. Egy alkérdést tartalmazó feltételt teljes relációkon dolgozó operátorral kell kifejeznünk, míg a σ operátor alsó indexe konkrét sorok tesztelésére alkalmazandó.

Relációs algebrában egy operátorban szereplő valamennyi reláció az operátor explicit argumentuma, és nem alsó indexben megjelenő paraméter. Szükség van tehát a relációs algebrában az alkérdések kezelésére, olyan operátorok segítségével, mint a $\bowtie$ (összekapcsolás), amelyben az alkérdés relációja összekapcsolódik a külső lekérdezés relációjával. E témát elhalasztjuk a 7.1. részig. A 7.3.2. részben olyan kiválasztás operátorokról is fogunk beszélni, amelyek engedélyezik az alkérdéseket mint explicit argumentumokat.

A $\sigma_C(R)$ kiválasztás az R azon sorainak multihalmazát eredményezi, amelyek teljesítik a C feltételt. Az eredmény reláció sémája ugyanaz, mint az R reláció sémája.

6.2. példa: Legyen $R(a, b)$ a következő reláció:

a	b
0	1
2	3
4	5
2	3

A $\sigma_{a \geq 1}(R)$ eredménye:

a	b
2	3
4	5
2	3

Figyeljük meg, hogy egy olyan sor, amelyik teljesíti a feltételt, ugyanannyiszor jelenik meg az eredményben, mint magában a relációban, míg egy olyan sor, amelyik nem teljesíti a feltételt, mint például a (0, 1), egyáltalán nem jelenik meg.

A $\sigma_{b \geq 3 \text{ AND } a+b \geq 6}(R)$ eredménye:

a	b
4	5

□

6.1.3. Vetítés

Ha R egy reláció, akkor a $\pi_L(R)$ az R reláció L listára történő *vetítése*. A klasszikus relációs algebrában L az R (bizonyos) attribútumainak egy listája. Kiterjesztjük a vetítés operátort, hogy hasonlóvá váljon az SQL SELECT záradékához. A mi vetítési listánk a következő típusú elemeket tartalmazhatja:

1. Az R egy attribútumát.
2. Egy $x \rightarrow y$ kifejezést, ahol x és y attribútumnevek. Az $x \rightarrow y$ kifejezés az L listában azt jelenti, hogy vesszük az R reláció x attribútumát és átnevezzük y-ra; azaz az eredmény relációban ennek az attribútumnak a neve y lesz.
3. Egy $E \rightarrow z$ kifejezést, ahol E az R attribútumait, konstansokat, aritmetikai operátorokat és karakterlánc operátorokat tartalmazó kifejezés, és z az új neve annak az attribútumnak, amelyet az E-ben szereplő számítások eredményeznek. Például, az $a + b \rightarrow x$, mint a lista egy eleme, az a és b attribútumok összegét reprezentálja x néven. A $c \parallel d \rightarrow e$ elem a c és d elemek összefűzését jelenti (amelyek feltehetően karakterlánc-értékűek), e néven.

A vetítés eredményét úgy számoljuk ki, hogy vesszük sorban az R valamennyi sorát. Kiértékeljük az L listát, oly módon, hogy behelyettesítjük a sorok komponenseit az L-ben felsorolt megfelelő attribútumokba, és alkalmazzuk az L operátorait ezekre az értékekre. Az eredmény egy olyan reláció, amelynek sémája megegyezik az L listában felsorolt attribútumokkal, az átnevezéseket figyelembe véve. Az R valamennyi sora létrehoz egy sort az eredményben. Az R ismétlődő sorai természetesen ismétlődő sorokat hoznak létre az eredményben, de az eredmény akkor is tartalmazhat ismétlődő sorokat, ha az R nem tartalmazott ilyet.

6.3. példa: Legyen R a következő reláció:

a	b	c
0	1	2
0	1	2
3	4	5

A $\pi_{a,\, b + c \rightarrow x}(R)$ eredménye:

a	x
0	3
0	3
3	9

Az eredmény sémája két attribútumot tartalmaz. Az egyik az a, amely az R első attribútuma, átnevezés nélkül. A második az R második és harmadik attribútumának az összege, x néven.

Egy másik példát véve, a $\pi_{b - a \rightarrow x,\, c - b \rightarrow y}(R)$ eredménye:

x	y
1	1
1	1
1	1

Figyeljük meg, hogy a vetítési listában megadott számítások történetesen ugyanazt az (1, 1) sort eredményezték a (0, 1, 2) sorra és a (3, 4, 5) sorra egyaránt. Ezért az (1, 1) sor háromszor jelenik meg az eredményben. □

6.1.4. Relációk szorzata

Ha R és S két reláció, akkor az $R \times S$ *szorzat* egy olyan reláció, amelynek sémája az R és az S attribútumaiból áll. Ha mindkét sémában van, mondjuk, egy a nevű attribútum, akkor a szorzat sémájában az attribútumok neveiként az $R.a$, illetve $S.a$ jelöléseket használjuk.

A szorzat sorait az összes olyan sorok alkotják, amelyeket úgy kapunk, hogy veszszük az R egy sorát, és annak elemeit kiegészítjük az S egyik sorával. Ha egy r sor n-szer jelenik meg az R-ben, az s pedig m-szer szerepel S-ben, akkor a szorzatban az rs sor nm-szer jelenik meg.

6.4. példa: Legyen $R(a, b)$ a következő reláció:

a	b
0	1
2	3
2	3

és az $S(b, c)$ reláció legyen a következő:

b	c
1	4
1	4
2	5

Ekkor az $R \times S$ eredménye a 6.3. ábrán látható reláció. Figyeljük meg, hogy az R valamennyi sorát párosítottuk az S valamennyi sorával, tekintet nélkül az ismétlődésekre. Így például a (2, 3, 1, 4) sor négyszer jelenik meg, mivel az őt alkotó komponensek az R, illetve az S ismétlődő sorai. Figyeljük meg továbbá, hogy a szorzat sémájában szerepel az $R.b$ és az $S.b$ attribútum is, a két reláció b attribútumának megfelelően. □

a	$R.b$	$S.b$	c
0	1	1	4
0	1	1	4
0	1	2	5
2	3	1	4
2	3	1	4
2	3	2	5
2	3	1	4
2	3	1	4
2	3	2	5

6.3. ábra. *Az R és S relációk szorzata*

6.1.5. Összekapcsolások

Számos olyan hasznos „összekapcsolás" operátor van, amely egy szorzatból és az azt követő kiválasztásból és vetítésből épül fel. Ezek az operátorok explicit módon megtalálhatók az SQL2-szabványban, mint a relációk kombinálásának módjai a FROM záradékban. Az összekapcsolások azonban sok olyan gyakori SQL-lekérdezés hatását is reprezentálják, amelyek FROM záradéka egy vagy több relációt tartalmaz, és amelyek WHERE záradékában ezen relációk attribútumaira alkalmazott egyenlőségek vagy összehasonlítások szerepelnek.

A legegyszerűbb és legelterjedtebb a *természetes összekapcsolás* (natural join). Az R és S relációk természetes összekapcsolását $R \bowtie S$ szimbólummal jelöljük. Ez a kifejezés a $\pi_L(\sigma_C(R \times S))$ rövidítése, ahol:

1. C egy olyan feltétel, amely megköveteli az R és S azonos nevű attribútumainak egyenlőségét.
2. Az L egy attribútumlista, amely az R és S összes attribútumát tartalmazza, kivételt az azonos nevű attribútumok képeznek, amelyek csak egy példányban szerepelnek ebben a listában. Ha $R.x$ és $S.x$ a két egyenlővé tett attribútum, akkor a vetítés eredményében csak egy x attribútum fog szerepelni. Ezt a hagyományokhoz híven, függetlenül attól, hogy az $R.x$-et vagy az $S.x$-et választjuk, átnevezzük x-re.

6.5. példa: Ha az $R(a, b)$ és $S(b, c)$ a 6.4. példában bevezetett relációk, akkor az $R \bowtie S$ a $\pi_{a,\, R.b \to b,\, c}(\sigma_{R.b = S.b}(R \times S))$ kifejezést jelenti. Ez azt jelenti, hogy mivel az R és S relációkban b az egyetlen közös attribútumnév, ezért a kiválasztás csak ezt a két attribútumot teszi egyenlővé. Most az $R.b$-t választottuk, és ezt neveztük át b-re, de tetszés szerint választhattuk volna az $S.b$-t is.

Egy természetes összekapcsolás eredményét megkaphatjuk úgy is, hogy sorban alkalmazzuk a $\times$, σ és π operátorokat. Könnyebb azonban „egy lépésben" kiszámolni a természetes összekapcsolást. Számos módszer létezik arra vonatkozóan, hogy miként találjuk meg az R és S relációk azon sorpárjait, amelyek megegyeznek az összes azonos nevű attribútumon. Ezekre a sorpárokra képezzük az eredménysorokat, amelyek

minden attribútumon megegyeznek ezekkel a sorokkal. A 6.4. példa R és S relációit használva például azt találjuk, hogy azon sorpárok, amelyekre b értéke megegyezik, az R (0, 1) sorából és az S két (1, 4) sorából tevődnek össze. Az $R \bowtie S$ eredménye tehát:

a	b	c
0	1	4
0	1	4

Figyeljük meg, hogy két összekapcsolandó sorpár van; ezek történetesen az S azonos sorait tartalmazzák, ez az oka annak, hogy az eredményben kétszer jelenik meg a (0, 1, 4). □

Az összekapcsolás másik formája a *théta-összekapcsolás*. Az R és S relációk esetén az $R \underset{C}{\bowtie} S$ a $\sigma_C(R \times S)$ kifejezés rövidítése. Ha a C feltétel egyetlen tagot tartalmaz, amely $x = y$ alakú, ahol x az R attribútuma, y pedig az S attribútuma, akkor ezt az összekapcsolást *egyenlőség alapú összekapcsolásnak* (equijoin) nevezzük. Jegyezzük meg, hogy a természetes összekapcsolással ellentétben az egyenlőség alapú összekapcsolás nem tartalmaz vetítést, még akkor sem, ha az eredményben két vagy több egyforma oszlop szerepel.

6.6. példa: Legyenek az $R(a, b)$ és $S(b, c)$ a 6.4. és 6.5. példában bevezetett relációk. Ekkor az $R \underset{a + R.b < c + S.b}{\bowtie} S$ megegyezik a $\sigma_{a + R.b < c + S.b}(R \times S)$ kifejezéssel. Ez azt jelenti, hogy az összekapcsolás feltétele megköveteli, hogy az R sorában a komponensek összege kisebb legyen, mint az S sorában a komponensek összege. Az eredmény tartalmazza az $R \times S$ összes sorát, kivéve azt, amelyben az R sora (2, 3) és az S sora (1, 4), mivel itt az R sorbeli összeg nem kisebb, mint az S sorbeli összeg. Az eredmény relációt a 6.4. ábrán láthatjuk.

a	$R.b$	$S.b$	c
0	1	1	4
0	1	1	4
0	1	2	5
2	3	2	5
2	3	2	5

6.4. ábra. *A théta-összekapcsolás eredménye*

Legyen egy másik példa az $R \underset{b=b}{\bowtie} S$ egyenlőség alapú összekapcsolás kiszámítása. Megállapodás szerint, a théta-összekapcsolásban egyenlővé tett attribútumok közül az első a bal oldali argumentumhoz tartozik, míg a második a jobb oldali argumentumhoz, azaz ez a kifejezés ugyanaz, mint az $R \underset{R.b = S.b}{\bowtie} S$. Az eredmény ugyanaz, mint az

A „théta-összekapcsolás" jelentése

Történelmileg valamennyi összekapcsolás egy egyszerű feltételt tartalmazott, amely a két argumentum reláció egy-egy attribútumát hasonlította össze. Egy ilyen összekapcsolás általános formáját $R \underset{x\theta y}{\bowtie} S$ alakban írtuk fel, ahol θ a következő hat aritmetikai operátor egyikét jelenti: $=$, $\neq$, $<$, $\leq$, $>$ vagy $\geq$. Mivel az összehasonlítást a θ szimbólum jelölte, ezért ez a művelet „théta-összekapcsolás" néven vált ismertté. Manapság a terminológiát megtartottuk, habár a théta-összekapcsolás feltétele már nem csak attribútumok egyszerű összehasonlítása lehet, hanem bármilyen, kiválasztásban engedélyezett feltétel. Mindazonáltal a gyakorlatban kétségkívül azok a théta-összekapcsolások vannak túlsúlyban, amelyek két attribútumot hasonlítanak össze, ezek közül is főleg az egyenlőségen alapuló összehasonlítások.

$R \bowtie S$ eredménye, kivéve, hogy mindkét, egyenlővé tett attribútum megmarad. Tehát ennek az egyenlőség alapú összekapcsolásnak az eredménye:

a	*R.b*	*S.b*	*c*
0	1	1	4
0	1	1	4

□

6.1.6. Ismétlődések kiküszöbölése

Szükségünk van egy olyan operátorra, amely egy multihalmazt halmazzá alakít, az SQL `DISTINCT` kulcsszavának megfelelően. Erre a célra a $\delta(R)$-t használjuk, amely visszaadja azt a halmazt, ami az R relációban egyszer vagy többször előforduló sorokból egyetlen példányt tartalmaz.

6.7. példa: Ha az R reláció megfelel a 6.4. példa ugyanolyan nevű relációjának, akkor a $\delta(R)$ eredménye:

a	*b*
0	1
2	3

Figyeljük meg, hogy a (2, 3) sor, amely kétszer jelent meg az R relációban, a $\delta(R)$-ben csak egyszer szerepel. □

Emlékezzünk vissza, hogy az SQL `UNION`, `INTERSECT` és `EXCEPT` operátorai alapértelmezés szerint kiküszöbölik az ismétlődéseket, viszont mi úgy definiáltuk az $\cup$, $\cap$ és $-$ operátorokat, hogy megfeleljenek az alapértelmezés szerinti multihalmazos meghatározásnak. Ezért, ha egy olyan SQL-kifejezést szeretnénk algebrai kifejezéssé alakítani, mint az R `UNION` S, akkor azt kell írnunk, hogy $\delta(R \cup S)$.

6.1.7. Csoportosítás és összesítés

SQL-ben lehetőségek egész családja működik együtt az olyan lekérdezések támogatására, amelyek „csoportosítást és összesítést" használnak:

1. *Összesítő operátorok* (aggregation operators). Az öt operátor, az `AVG`, `SUM`, `COUNT`, `MIN` és `MAX` annak az attribútumnak az átlagát, összegét, a benne található elemek számát, minimumát, illetve maximumát határozzák meg, amelyre alkalmazzuk őket. Ezek az operátorok a `SELECT` záradékokban jelennek meg.
2. *Csoportosítás.* Egy SQL-lekérdezés `GROUP BY` záradéka által a `FROM` és `WHERE` záradékok alapján felépített reláció csoportosítva lesz a `GROUP BY` záradékban felsorolt attribútum (vagy attribútumok) alapján. Ezek után az összesítések a csoportokra készülnek el.
3. Egy `HAVING` záradékot kötelezően egy `GROUP BY` záradéknak kell megelőznie, és egy olyan feltételt fogalmaz meg (ez a feltétel összesítéseket és a csoportosítás attribútumait is érintheti), amelyet egy csoportnak teljesítenie kell ahhoz, hogy a lekérdezés eredményének részét képezze.

A csoportosításokat és az összesítéseket általában együtt kell megvalósítani és optimalizálni. Ily módon egyetlen olyan γ operátort fogunk bevezetni a kiterjesztett relációs algebránkba, amely a csoportosítás és összesítés hatását reprezentálja. A γ operátor segít a `HAVING` záradék megvalósításában is, amelyet a γ-t követő kiválasztás és vetítés képvisel.

A γ operátor alsó indexét egy L lista képezi, amelynek valamennyi eleme a következők egyike:

a) A reláció egy attribútuma, amelyre a γ-t alkalmazzuk; ez az attribútum egyike a lekérdezés `GROUP BY` listájának. Ezt az elemet *csoportosító* (grouping) attribútumnak nevezzük.
b) A reláció egyik attribútumára alkalmazott összesítő operátor. Ahhoz, hogy az eredményben névvel lássuk el az összesítés eredményeként előálló attribútumot, egy nyilat és egy új nevet írunk az összesítés után. Ez az elem a lekérdezés `SELECT` záradékának egyik összesítését reprezentálja. A megfelelő attribútumot *összesített* (aggregated) attribútumnak nevezzük.

A $\gamma_L(R)$ kifejezés által visszaadott reláció a következőképpen épül fel:

1. Az R sorait *csoportokba* osztjuk szét. Valamennyi csoport azokból a sorokból épül fel, amelyek az L lista csoportosított argumentumaira egy bizonyos értékkel ren-

δ a γ egy speciális esete

Technikailag a δ operátor redundáns. Ha $R(a_1, a_2, ..., a_n)$ egy reláció, akkor $\delta(R)$ ekvivalens a $\gamma_{a_1, a_2, ..., a_n}(R)$ kifejezéssel. Ez azt jelenti, hogy az ismétlődések kiküszöböléséhez a reláció összes attribútumán csoportosítunk, és nem végzünk összesítést. Így mindegyik csoport megfelel egy olyan sornak, amely egyszer vagy többször szerepel R-ben. Mivel a γ eredménye minden csoporthoz pontosan egy sort tartalmaz, ezért ennek a csoportosításnak az eredménye az, hogy kiküszöböli az ismétlődéseket. Azonban, mivel a δ egy igen elterjedt és fontos operátor, külön fogunk vele foglalkozni az algebrai törvényszerűségek tárgyalásakor, valamint az operátorok megvalósítására szolgáló algoritmusokban.

delkeznek. Ha nincsenek csoportosított attribútumok, akkor a teljes R reláció lesz egy csoport.

2. Minden egyes csoportra képezünk egy olyan sort, amely a következőket tartalmazza:

   i) a csoportosított attribútumok értékeit az adott csoportra és
   ii) a csoport összes sorára vonatkozó összesítéseket, amelyeket az L lista összesített attribútumai specifikálnak.

6.8. példa: Tegyük fel, hogy adott a következő reláció:

```
SzerepelBenne(cím, év, színészNév)
```

és szeretnénk megkapni azon színészeket, akik legalább három filmben szerepeltek, azzal az évszámmal együtt, amikor először szerepeltek. A következő SQL-lekérdezés pontosan ezt adja meg:

```
SELECT színészNév, MIN(év) AS minÉv
FROM SzerepelBenne
GROUP BY színészNév
HAVING COUNT(cím) >= 3;
```

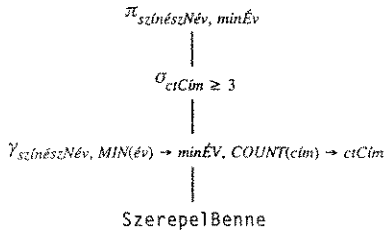
Az ekvivalens algebrai kifejezés csoportosítást fog végezni a `színészNév` attribútumon. Minden bizonnyal ki kell számolnunk minden csoportra a `MIN(év)` összesítést. Ahhoz azonban, hogy meg tudjuk ítélni, hogy melyik csoport teljesíti a `HAVING` záradékot, ki kell számolnunk a `COUNT(cím)` összesítést is valamennyi csoportra.

A csoportosító kifejezéssel kezdjük:

$\gamma_{színészNév,\ MIN(év) \to minÉv,\ COUNT(cím) \to ctCím}$(SzerepelBenne)

A kifejezés eredményének első két oszlopára a lekérdezés eredménye miatt van

szükség. A harmadik oszlop egy kiegészítő attribútum, amelyet `ctCím`-nek neveztünk el. Ez azért szükséges, mert minden sorra alkalmaznunk kell a `HAVING` záradékban szereplő feltételt. Ez azt jelenti, hogy a lekérdezéshez tartozó algebrai kifejezést azzal folytatjuk, hogy kiválasztjuk a `ctCím >= 3` feltételnek megfelelő sorokat, és az eredményt levetítjük az első két oszlopra. A lekérdezés reprezentációját a 6.5. ábrán láthatjuk. Ez egy egyszerűbb formájú kifejezésfa (lásd 6.1.9. részt), ahol egymás után négy operátort láthatunk, mindegyik az előző operátor alatt foglal helyet. □



6.5. ábra. *A 6.8. példához tartozó algebrai kifejezésfa*

Még a `HAVING` záradék nélküli SQL csoportosító lekérdezések között is van olyan, amelyet nem lehet kifejezni egyetlen γ operátorral. A `FROM` záradék például több relációt is tartalmazhat, és ezeket először egyesíteni kell egy szorzat operátorral. Ha a lekérdezésnek van egy `WHERE` záradéka, akkor ennek a záradéknak a feltételét egy σ operátorral ki kell fejezni, vagy esetleg a relációk szorzatát egy összekapcsolássá kell alakítani. Előfordulhat továbbá, hogy a `GROUP BY` záradék egyik attribútuma nem szerepel a `SELECT` listában. A 6.8. példában például elhagyhatjuk a `színészNév` attribútumot a `SELECT` záradékból, habár a hatás egy kissé furcsa lenne: kapnánk egy évszámokból álló listát, de semmi nem jelölné, hogy melyik év melyik színésznek felel meg. Ebben az esetben a `GROUP BY` összes attribútumát felsoroljuk a γ listájában, majd ezután alkalmazunk egy vetítést, amely eltávolítja azokat a csoportosító attribútumokat, amelyek nem jelennek meg a `SELECT` záradékban.

6.1.8. Rendezés

A τ operátort fogjuk használni egy reláció rendezéséhez. Ezt az operátort az SQL `ORDER BY` záradékának megvalósítására lehet használni. A rendezés egy fizikai lekérdezésterv operátorának szerepét is betölti, hiszen a relációs algebra sok más operátora gyorsabbá tehető, ha először rendezünk egy vagy több argumentumban szereplő relációt.

Pontosabban fogalmazva, a $\tau_L(R)$ kifejezés, ahol R egy reláció, L pedig az R bizonyos attribútumainak listája, pontosan az R relációt adja, de az R sorai rendezettek az L által megadott módon. Ha L az $a_1, a_2, ..., a_n$ lista, akkor az R sorai először az a_1 attribútum értékei szerint vannak rendezve. Egyforma a_1 értékek esetén az a_2 értékei

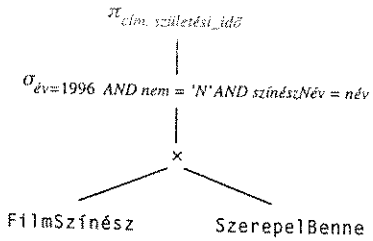
számítanak. Azok a sorok, amelyek megegyeznek az a_1 és a_2 értékeiken, az a_3 értékek szerint kerülnek rendezésre és így tovább. Azok a sorok, amelyek még az a_n attribútumon is megegyeznek, tetszőleges sorrendbe helyezhetők. Éppúgy, mint az SQL esetén, feltételezzük, hogy az alapértelmezett rendezési sorrend növekvő, de ez csökkenőre változtatható az attribútum utáni DESC kulcsszóval.

6.9. példa: Ha az R egy olyan reláció, amelynek sémája $R(a, b, c)$, akkor a $\tau_{c,b}(R)$ rendezi az R sorait a c értékük szerint, és az azonos c értékű sorokat a b értékük szerint. Azok a sorok, amelyek mind a b, mind a c attribútumon megegyeznek, tetszőleges sorrendbe helyezhetők. □

A τ operátor szabálytalan abban a tekintetben, hogy a mi relációs algebránkban ez az egyetlen olyan operátor, amelynek eredménye nem halmaz, hanem sorok egy listája. Ily módon egy algebrai kifejezésben csak utolsó operátorként van értelme beszélni a τ operátorról. Ha egy másik relációs algebrai operátort alkalmazunk a τ után, akkor a τ eredményét halmazként vagy multihalmazként kezeljük, és nem számít a sorok sorrendje. Gyakran használjuk azonban a τ-t fizikai lekérdezéstervekben, amelyek operátorai nem ugyanazok, mint a relációs algebrai operátorok. Sok későbbi operátor veszi hasznát annak, ha egy vagy több argumentum rendezett, és lehet, hogy ők maguk is rendezett eredményt állítanak elő.

6.1.9. Kifejezésfák

Több relációs algebrai operátort használhatunk egyetlen kifejezésben, ha egy vagy több operátor eredményére (eredményeire) alkalmazunk egy másik operátort. Ily módon éppúgy, mint bármely más algebra esetén, az operátorok egymás utáni alkalmazását egy *kifejezésfa* (expression tree) formájában rajzolhatjuk fel. A fa leveleit relációk nevei alkotják, és a belső csúcsokat olyan operátorok alkotják, amelyek akkor nyernek értelmet, amikor alkalmazzuk a gyermeke vagy gyermekei által reprezentált relációkra. A 6.5. ábrán például egy olyan egyszerű kifejezésfát láthattunk, amelyben három egyoperandusú (unáris) operátort alkalmaztunk egymás után. Sok kifejezésfa tartalmaz azonban kétoperandusú (bináris) operátorokat, az ilyen kifejezésfák több ággal rendelkeznek.



6.6. ábra. *A 6.10. példa SQL-lekérdezésének egyik lehetséges logikai terve*

6.10. példa: Tegyük fel, hogy rendelkezésünkre állnak a következő relációk:

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

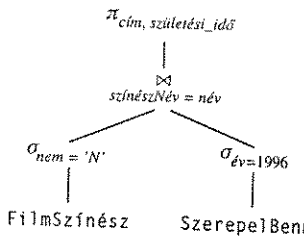
amelyekből szeretnénk kinyerni azokat a filmcímeket, a színésznők születésni idejével együtt, amelyekben az 1996-ban megjelent filmekben szereplő színésznők játszanak:

```
SELECT cím, születési_idő
FROM FilmSzínész, SzerepelBenne
WHERE év = 1996 AND
      nem = 'N' AND
      színészNév = név;
```

Ez azt jelenti, hogy összekapcsoljuk a FilmSzínész és SzerepelBenne relációkat, felhasználva azt a feltételt, hogy a színész neve mindkét relációban azonos. Ezután kiválasztjuk azokat a sorokat, amelyben a megjelenés éve 1996 és a színész neme nő.

A fentihez hasonló egyszerű SQL-lekérdezést az elemző (lásd 6.2. ábra) egy olyan logikai lekérdezéstervvé alakít, amelynek első lépése a FROM utáni relációk egyesítése a szorzat operátor felhasználásával. A következő lépés a WHERE záradéknak megfelelő kiválasztás végrehajtása, és az utolsó lépés ennek levetítése a SELECT záradékban szereplő listára. A fenti lekérdezésnek megfelelő algebrai kifejezést a 6.6. ábrán láthatjuk.

Több olyan kifejezés is létezik, amelyik *ekvivalens* a 6.6. ábra kifejezésével, abban az értelemben, hogy a FilmSzínész és SzerepelBenne relációk bármely előfordulására ezen kifejezések eredménye ugyanaz lesz. A 6.7. ábrán egy ilyen ekvivalens kifejezésre adunk példát. Ez a kifejezés jelentősen eltér a 6.6. ábrán látható tervtől. Először is észrevehetjük, hogy a WHERE záradék színészNév = név feltételét ebben az esetben a szorzatra alkalmazzuk, amely ily módon egy egyenlőségen alapuló összekapcsolássá válik. A 6.7. ábrán egy kiválasztás és egy szorzat összekapcsolássá történő átalakítását alkalmaztuk. Az összekapcsolások általában kevesebb sort eredményeznek, éppen ezért, ha lehet választani, akkor inkább az összekapcsolást választjuk a szorzat helyett.



6.7. ábra. *Egy másik, valószínűleg jobb logikai lekérdezésterv*

Másodszor azt vehetjük észre, hogy a WHERE záradékban szereplő két feltételt szétszedtük két σ-műveletté, és ezeket a műveleteket lejjebb „csúsztatjuk" a fában, egészen a megfelelő relációkig. A $\sigma_{\text{év} = 1996}$ kiválasztást például közvetlenül a SzerepelBenne relációra alkalmazzuk, mivel ez az egyetlen olyan reláció, amely év attribútumot visz a 6.6. ábra szorzatába. Általános szabály, hogy érdemes a kiválasztást (rendszerint) a lehető leghamarabb elvégezni. Mivel a szorzások és az összekapcsolások jellegzetesen több időt vesznek igénybe mint a kiválasztások, ezért a relációk méretének mihamarabbi csökkentése sokkal jobban lecsökkenti az összekapcsoláshoz szükséges időt, mint amennyire megnöveli a kiválasztáshoz szükséges időt. A relációk méretének mihamarabbi csökkentését úgy érhetjük el, ha a kiválasztást minél lejjebb csúsztatjuk a fában úgy, ahogyan az a 6.7. ábrán látható. A logikai lekérdezéstervek tökéletesítésének általános témaköréhez vissza fogunk még térni a 7.2. részben. □

6.1.10. Feladatok

6.1.1. feladat: Adott két reláció:

$R(a, b)$: {(0, 1), (2, 3), (0, 1), (2, 4), (3, 4)}
$S(a, b)$: {(0, 1), (2, 4), (2, 5), (3, 4), (0, 2), (3, 4)}

Számoljuk ki a következőket:

* a) $R \cup_S S$.
b) $R \cup_B S$.
c) $R \cap_S S$.
d) $R \cap_B S$.
e) $R -_S S$.
f) $R -_B S$.
g) $S -_S R$.
h) $S -_B R$.
* i) $\pi_{a+b,\, a^2,\, b^2}(R)$
j) $\pi_{a+1,\, b-1}(S)$
* k) $\sigma_{a<b \text{ AND } (a+b>a\times b \text{ OR } a+b\geq 6)}(R)$.
l) $\sigma_{a<b \text{ AND } (a+b>a\times b \text{ OR } a+b\geq 6)}(S)$.
m) $\sigma_{a>1 \text{ OR } b>4 \text{ OR } b=2}(R)$.
n) $\sigma_{a>1 \text{ OR } b>4 \text{ OR } b=2}(S)$.

6.1.2. feladat: Adott három reláció:

$R(a, b)$: {(0, 1), (2, 3), (0, 1), (2, 4), (3, 4)}
$S(b, c)$: {(1, 2), (1, 2), (2, 5), (3, 5), (4, 5)}
$T(c, d)$: {(2, 3), (3, 4), (4, 5), (5, 6)}

Összekapcsolás jellegű operátorok

Van néhány olyan operátor, amelyeket együttesen az összekapcsolás különböző változatainak tekintünk. Ezek jelölési módja és definíciója következik most:

1. $R \ltimes S$, az R és S relációk *félig összekapcsolása* (semijoin), amely az R reláció azon t sorainak multihalmaza, amelyre létezik legalább egy olyan sor az S-ben, amely megegyezik a t-vel az R és S valamennyi közös attribútumán.
2. $R \overline{\ltimes} S$, az R és S relációk *félig anti összekapcsolása* (antisemijoin), amely az R reláció azon t sorainak multihalmaza, amelyek *nem* egyeznek meg az S egyetlen sorával sem az R és S közös attribútumain.
3. $R \overset{\circ}{\bowtie} S$, az R és S relációk *külső összekapcsolása* (outerjoin), amely az $R \bowtie S$ soraiból áll, amelyekhez még hozzávesszük az R, illetve az S lógó sorait (egy sort akkor nevezünk *lógó* sornak, ha nem kapcsolható össze a másik reláció egyetlen sorával sem). Az ily módon hozzáadott sorokat ki kell egészítenünk speciális *null* szimbólumokkal mindazon attribútumok helyén, amelyekkel ezek a sorok nem rendelkeznek, de az eredménysorok igen. (Az SQL-ben a NULL érték szerepel, a feladatainkban a $\perp$ szimbólum látja majd el ezt a feladatot.)
4. $R \overset{\circ}{\bowtie}_B S$, az R és S relációk *bal oldali külső összekapcsolása* (left outerjoin), amely olyan mint a külső összekapcsolás, de most csak az R lógó sorait egészítjük ki $\perp$ értékekkel, és adjuk hozzá az eredményhez.
5. $R \overset{\circ}{\bowtie}_J S$, az R és S relációk *jobb oldali külső összekapcsolása* (right outerjoin), amely olyan mint a külső összekapcsolás, de most csak az S lógó sorait egészítjük ki $\perp$ értékekkel, és adjuk hozzá az eredményhez.

Számoljuk ki a következőket:

* a) $R \bowtie S$.
b) $S \bowtie T$.
! c) $R \bowtie T$.
d) $R \underset{R.b<S.b}{\bowtie} S$.
* e) $R \underset{a+d=5}{\bowtie} T$.
f) $R \underset{b=c}{\bowtie} T$.
* g) $\gamma_{a,SUM(b)}(R)$.
h) $\gamma_{c,MIN(b)}(S)$.
i) $\delta(R)$.
j) $\tau_{b,a}(R)$.

! 6.1.3. feladat: Az „Összekapcsolás jellegű operátorok" című bekeretezett részben definiált öt operátorra adjunk meg olyan kifejezéseket, amelyek csak a most definiált relációs algebra szabványos operátorait használják. A különböző külső összekapcsolá-

soknál használhatunk speciális $N(a_1, a_2, \ldots, a_n)$ „null relációkat", amelyek egy sorból állnak, és minden elemük $\perp$.

* a) Félig összekapcsolás.
b) Félig anti összekapcsolás.
* c) Bal oldali külső összekapcsolás.
d) Jobb oldali külső összekapcsolás.
e) Külső összekapcsolás.

6.1.4. feladat: Írjuk fel a következő összekapcsolásokat olyan kifejezések segítségével, amelyek kiválasztást, vetítést és szorzatot tartalmaznak.

a) $R(a, b, c, d) \bowtie S(b, d, e)$.
b) $R(a, b, c) \underset{a+d=10 \text{ OR } b=S.c}{\bowtie} S(c, d)$.

! **6.1.5. feladat:** Az f unáris operátort *idempotensnek* nevezzük, ha tetszőleges R relációra $f(f(R)) = f(R)$. Ez azt jelenti, hogy f többszöri alkalmazása ugyanazt eredményezi mint az f egyszeri alkalmazása. A következő operátorok közül melyek idempotensek? Magyarázzuk meg, hogy miért, vagy adjunk ellenpéldát.

* a) δ.
* b) π_L.
c) σ_C.
d) γ_L.
e) τ.

6.1.6. feladat: A következő „filmes" relációkat felhasználva:

```
Film(cím, év, hossz, stúdióNév)
FilmSzínész(név, cím, nem, születési_idő)
SzerepelBenne(cím, év, színészNév)
Stúdió(név, cím)
```

írjuk át a következő lekérdezéseket kifejezésfákká, felhasználva az ebben a részben bemutatott algebrai operátorokat.

a)
```
SELECT cím
  FROM Film, Stúdió
  WHERE stúdióNév = név AND cím = 'Elfújta a szél';
```

b)
```
(SELECT név FROM FilmSzínész)
    UNION
(SELECT színészNév FROM SzerepelBenne);
```


c)
```
(SELECT név FROM FilmSzínész)
    UNION ALL
(SELECT színészNév FROM SzerepelBenne);
```

d)
```
SELECT színészNév, SUM(hossz)
  FROM Film NATURAL JOIN SzerepelBenne
  GROUP BY színész
  HAVING COUNT(*) >= 3;
```

6.2. Bevezetés a fizikai lekérdezésterv-operátorok világába

A fizikai lekérdezéstervek operátorokból épülnek fel, amelyek mindegyike a terv egy lépését valósítja meg. A fizikai operátorok gyakran a relációs algebra egyik operátorának konkrét megvalósításai. Szükségünk van azonban olyan feladatokhoz is fizikai operátorokra, amelyek nem kapcsolhatók a relációs algebra egyik operátorához sem. Gyakran kell például „beolvasni" egy táblát, ami azt jelenti, hogy egy relációs algebrai kifejezés egyik operandus relációjának összes sorát betöltjük a memóriába. Ebben a részben bevezetjük a fizikai lekérdezésterveket alkotó építőköveket. A későbbi részek olyan komplex algoritmusokat tartalmaznak, amelyekkel hatékonyan megvalósíthatók a relációs algebrai operátorok. Ezek az algoritmusok szintén jelentős részét képezik a fizikai lekérdezésterveknek. Ebben a részben bevezetjük az „iterátor" fogalmát is, amely egy olyan fontos módszer, melynek segítségével megvalósulhat a fizikai lekérdezést felépítő operátorok közötti adatcsere.

6.2.1. Táblák átvizsgálása

Egy fizikai lekérdezéstervben valószínűleg a legalapvetőbb dolog egy R reláció teljes tartalmának a beolvasása. Ez a lépés elengedhetetlen, amikor például az R relációt egyesítjük vagy összekapcsoljuk egy másik relációval. Ennek az operátornak az egyik változata tartalmaz egy egyszerű predikátumot, ilyenkor az R relációnak csak azokat a sorait olvassuk be, amelyek kielégítik a predikátumot. Két alapvető megközelítés létezik egy R reláció sorainak megtalálására.

1. Sok esetben az R relációt a másodlagos memória területén tároljuk, és sorait blokkokba szervezzük. Az R sorait tartalmazó blokkok ismertek a rendszer számára, és lehetséges a blokkok egymás utáni beolvasása. Ezt a műveletet nevezzük *táblaátvizsgálásnak*.
2. Ha létezik egy index az R valamelyik attribútumára, akkor használhatjuk ezt az indexet az R sorainak beolvasásához. Az R egy ritka indexét (lásd 4.1.3. részt) például használhatjuk arra, hogy elvezessen bennünket az R-et tartalmazó valamennyi

blokkhoz, még akkor is, ha egyébként nem is tudjuk, hogy melyek ezek a blokkok. Ezt a műveletet nevezzük *index alapú átvizsgálásnak*.

A 6.7.2. részben, amikor a σ operátor megvalósításáról lesz szó, ismét vissza fogunk térni az index alapú átvizsgálásra. Most csak egy fontos megjegyzés teszünk: az indexet nem csak arra használhatjuk, hogy segítségével beolvassuk a reláció *összes* sorát, hanem arra is, hogy csak azokat a sorokat olvassuk be, amelyek egy konkrét értékkel rendelkeznek az index keresési kulcsát alkotó attribútumon, illetve attribútumokon. (Esetenként azokat a sorokat is kereshetjük, amelyekre ezek az attribútumok egy konkrét értéktartományba tartoznak.)

6.2.2. Rendezés a táblák átvizsgálásakor

Több oka is lehet annak, amiért rendezni szeretnénk egy relációt mialatt beolvassuk a sorait. Egyik oka az lehet, hogy a lekérdezésnek van `ORDER BY` záradéka, amely megköveteli, hogy a reláció rendezett legyen. Másik oka az lehet, hogy a relációs algebrai műveletek implementálására szolgáló algoritmusok közül több is megköveteli, hogy az argumentum relációk közül az egyik vagy akár mindegyik rendezett reláció legyen. Ezek az algoritmusok a 6.5. részben jelennek meg, de máshol is találkozhatunk velük.

A *rendezéses átvizsgálás* nevű fizikai lekérdezésterv-operátor veszi az R relációt azon attribútumok specifikációjával együtt, amelyeken el kell végezni a rendezést, és előállítja a rendezett R relációt. A rendezéses átvizsgálás megvalósítására több lehetőség is létezik:

a) Ha szeretnénk előállítani az a attribútumon rendezett R relációt, és létezik egy B-fa-index az a attribútumra, vagy az R egy olyan indexszekvenciális fájl, amely az a szerint van rendezve, akkor az index bejárása lehetővé teszi a rendezett R reláció előállítását.
b) Ha a rendezni kívánt R reláció elég kicsi ahhoz, hogy beférjen a memóriába, akkor táblaátvizsgálással vagy indexátvizsgálással kinyerhetjük a tábla sorait, és ezután alkalmazhatunk egyet a hatékony, memóriában rendező algoritmusok közül. A memóriában történő rendezéssel több könyv is foglalkozik, nekünk most nem szándékunk ennek bemutatása.
c) Ha az R túl nagy ahhoz, hogy beférjen a memóriába, akkor jó választás lehet a 2.3.3. részben bemutatott többmenetes összefésülés. Ahelyett azonban, hogy a végleges, rendezett R relációt visszatennénk a lemezre, inkább előállítjuk a rendezett R egy blokkját, amikor a sorokra szükség van.

6.2.3. A fizikai operátorok kiszámításának modellje

Egy lekérdezés általában néhány relációs algebrai műveletből áll, míg a megfelelő fizikai lekérdezésterv néhány fizikai operátorból áll. Egy fizikai operátor általában egy relációs algebrai operátor megvalósítása, de amint azt a 6.2.1. részben is láthattuk, vannak olyan fizikaiterv-operátorok is, amelyek nem jelennek meg a relációs algebrában; ilyenek például a beolvasási műveletnek megfelelő fizikai operátorok.

Mivel egy jó lekérdezésfeldolgozónál lényeges a fizikaiterv-operátorok okos megválasztása, ezért meg kell tudnunk becsülni valamennyi operátor „költségét". Egy művelet költségének méréséhez a lemez I/O-műveletek számát fogjuk használni. Ez a mérési mód megfelel a 2.3.1. részben bemutatott szemléletnek, mely szerint az adatok lemezről történő beolvasása hosszabb, mint bármilyen más hasznos tevékenység, amely a már memóriában levő adatokkal történik. Egy fontos kivétel, amikor a lekérdezés megválaszolása hálózaton keresztül valósul meg. Az elosztott lekérdezések feldolgozásának költségéről a 6.10. és a 10.4.4. részekben lesz szó.

Amikor ugyanannak a műveletnek a különböző algoritmusait hasonlítjuk össze, akkor egy első látásra talán meglepő feltevéssel fogunk élni:

- Feltételezzük, hogy egy tetszőleges operátor argumentumai a lemezen találhatók, viszont az eredmény a memóriában marad.

Ha az operátor egy lekérdezés végső eredményét állítja elő, és az eredményt kiírjuk lemezre, akkor ennek költsége csak a válasz méretétől függ, és attól nem, hogy miként számoltuk ki az eredményt. Egyszerűen hozzáadhatjuk az utolsó kiírás költségét a lekérdezés teljes költségéhez. Több alkalmazásban azonban az eredményt egyáltalán nem tároljuk lemezen, hanem kinyomtatjuk vagy átadjuk valamilyen adatformátumokkal foglalkozó programnak. Ily módon a kimenet lemez I/O-költsége vagy nulla, vagy attól függ, hogy egy általunk ismeretlen alkalmazás mit tesz az adatokkal.

Hasonlóképpen egy olyan operátor eredményét, amelyik a lekérdezésnek részét képezi, gyakran szintén nem írjuk ki a lemezre. A 7.7.3. részben lesz majd szó a „csővezetékek módszeréről", ahol egy operátor eredményét a memóriában építjük fel, valószínűleg pillanatok alatt, és argumentumként továbbadjuk egy másik operátornak. Ebben a helyzetben soha sem kell az eredményt kiírnunk lemezre, és ráadásul megtakarítjuk az argumentum lemezről történő beolvasásának költségét annál az operátornál, amelyik ezt az eredményt argumentumként használja. Ez a megtakarítás kiváló lehetőséget nyújt a lekérdezésoptimalizáló számára.

6.2.4. A költségbecslés paraméterei

Most bemutatjuk egy operátor költségbecsléséhez használatos paramétereket. A költségbecslés elengedhetetlen, amikor az optimalizálónak el kell döntenie, hogy melyik lekérdezésterv végrehajtása lenne a leggyorsabb. A 7.5. részben láthatjuk majd ennek a költségbecslésnek a kiaknázását.

Szükségünk van egy olyan paraméterre, amelyik az operátor által használt memóriaterületet képviseli, és szükségünk van olyan paraméterekre is, amelyeket az argumentum(ok) méretének becslésére használunk. Tegyük fel, hogy a memória olyan pufferekből áll, amelyek mérete ugyanakkora, mint a lemezblokkok mérete. Ekkor egy konkrét operátor végrehajtásához rendelkezésre álló memóriapufferek számát M-mel fogjuk jelölni. Emlékezzünk vissza, hogy amikor egy operátor költségét megbecsüljük, akkor nem számoljuk bele a kimenet előállításának költségét – legyen az felhasznált memória vagy lemez I/O-művelet; ily módon az M csak a bemenet és az operátor közbeeső eredményeinek tárolására szolgál.

Gyakran tekinthetünk úgy az M-re, mint a teljes memóriára vagy mint a memória legnagyobb részére éppúgy, ahogyan a 2.3.4. részben tettük. Látni fogunk azonban olyan eseteket is, amikor több művelet osztozik a memórián, így az M jóval kisebb lehet, mint a teljes memória. Ahogyan azt majd a 6.8. részben is látni fogjuk, valójában egy művelethez rendelkezésre álló pufferek száma nem feltétlenül egy megjósolható konstans érték, hanem esetenként a végrehajtás során dől el, az egyidejűleg futó egyéb folyamatoktól függően. Ha ez így van, akkor az M valójában csak egy becslése a művelethez rendelkezésre álló pufferek számának. Ha a becslés hibás, akkor a tényleges végrehajtási idő különbözni fog az optimalizáló által megjósolt végrehajtási időtől. Még az is előfordulhat, hogy a kiválasztott fizikai lekérdezésterv más lett volna, ha a lekérdezésoptimalizáló tudta volna, hogy a végrehajtás alatt mennyi lesz a ténylegesen elérhető pufferek száma.

A következőkben bevezetjük azokat a paramétereket, amelyek az argumentum relációkhoz történő hozzáférés költségét becsülik meg. Ezek a paraméterek a reláció adatainak méretét és eloszlását becsülik meg, és a rendszer bizonyos időnként újra és újra kiszámítja őket, hogy ezzel segítse a lekérdezésoptimalizálót a fizikai operátorok kiválasztásában.

Az egyszerűség kedvéért feltételezzük, hogy a lemezen levő adatokhoz blokkonként férhetünk hozzá, és egyszerre egy blokk kerül beolvasásra. A gyakorlatban persze, ha képesek vagyunk a reláció több blokkját is egyszerre beolvasni, akkor a 2.4. részben bemutatott technikákkal gyorsíthatunk az algoritmuson, és ezáltal egyszerre több egymást követő blokkot is beolvashatunk. Lássuk most a három paramétercsaládot, a B-t, T-t és V-t:

- Amikor egy R reláció méretével foglalkozunk, akkor a leggyakrabban azzal vagyunk elfoglalva, hogy vajon hány blokkba fér el az R reláció összes sora. Ezt a számot $B(R)$-rel fogjuk jelölni, vagy egyszerűen csak B-vel, ha egyértelmű, hogy az R relációról van szó. Általában feltételezzük, hogy az R reláció *nyalábolt* (clustered), azaz B darab blokkban (vagy legalábbis megközelítően B darab blokkban) van tárolva. Ahogyan azt a 4.1.6. részben láthattuk, a gyakorlatban előfordulhat, hogy az R tárolására használt valamennyi blokk egy kis részét üresen akarjuk hagyni, gondolván az R-be történő majdani beszúrásokra. Mindazonáltal a B egy kellően jó megközelítése lesz azon blokkok számának, amelyeket be kell olvasni a lemezről ahhoz, hogy R minden sorát megkapjuk. A továbbiakban B-vel ezt a közelítő blokkszámot fogjuk jelölni.

- Néha szükségünk lesz arra, hogy ismerjük az R sorainak számát. Ezt $T(R)$-rel fogjuk jelölni, vagy egyszerűen csak T-vel, ha egyértelmű, hogy az R relációról van szó. Ha arra vagyunk kíváncsiak, hogy az R hány sora fér el egy blokkban, akkor használhatjuk a T/B hányadost. Vannak olyan megvalósítások is, ahol egy relációt olyan blokkokban tárolunk, amelyekben más relációk sorai is helyet kapnak. Ebben az esetben az egyszerűség kedvéért feltesszük, hogy az R minden egyes sorához külön lemezolvasás szükséges, és ilyenkor a T-t használjuk arra, hogy megbecsüljük az R beolvasásához szükséges lemez I/O-műveletek számát.
- Végezetül előfordul néha, hogy egy reláció valamelyik oszlopában található különböző értékek számára szeretnénk hivatkozni. Ha R egy reláció és a az egyik attribútuma, akkor a $V(R, a)$ jelenti az R reláció a oszlopában található különböző értékek számát. Általánosabban, ha $[a_1, a_2, \ldots, a_n]$ egy attribútumlista, akkor a $V(R,[a_1, a_2, \ldots, a_n])$ jelenti az R reláció $a_1, a_2, \ldots, a_n$ attribútumaihoz tartozó oszlopokban található különböző n-esek számát. Másfelől, ez nem más, mint a $\delta(\pi_{a_1, a_2, \ldots, a_n}(R))$ sorainak a száma.

6.2.5. Az átvizsgáló operátorok I/O-költsége

A bevezetett paraméterek egyszerű alkalmazásaként az eddig bemutatott valamennyi táblaátvizsgáló operátorra felírhatjuk a szükséges lemez I/O-műveletek számát. Ha az R reláció nyalábolt, akkor a táblaátvizsgáló operátorhoz szükséges lemez I/O-műveletek száma megközelítőleg B. Hasonlóképpen, ha R befér a memóriába, akkor a rendezéses beolvasást megvalósíthatjuk úgy, hogy beolvassuk R-et a memóriába, itt végrehajtunk rajta egy rendezést, és ez így ismét csak B lemez I/O-műveletet igényel.

Ha az R nyalábolt, de kétfázisú többmenetes összefésüléses rendezést igényel, akkor a 2.3.4. részben leírtaknak megfelelően körülbelül $3B$ lemez I/O-műveletre lesz szükségünk. Ezek a lemez I/O-műveletek egyenletesen oszlanak meg az R részlistákba történő beolvasása, a részlisták kiírása és a részlisták újrabeolvasása között. Emlékezzünk vissza, hogy az eredmény végső kiírásával nem foglalkozunk. Nem foglalkozunk a felhalmozott kimeneti adatok által elfoglalt memóriatartománnyal sem. Ehelyett inkább feltételezzük, hogy mindegyik kimeneti blokkot rögtön felhasználja egy másik művelet, vagy egyszerűen csak lemezre íródnak.

Ha azonban az R nem nyalábolt, akkor a szükséges lemez I/O-műveletek száma általában jóval nagyobb. Ha az R szét van osztva más relációk sorai között, akkor az R-hez tartozó táblaátvizsgálás során annyi blokkot kell beolvasni, ahány sora van az R-nek, vagyis az I/O-költség megegyezik T-vel. Hasonlóképpen, ha szeretnénk rendezni az R-et, és az R befér a memóriába, akkor T darab lemez I/O-műveletre van szükségünk ahhoz, hogy a teljes R relációt beolvassuk a memóriába. És végül, ha az R nem nyalábolt, és kétfázisú rendezést igényel, akkor kezdetben T darab lemez I/O-műveletre van szükség a részcsoportok beolvasásához. A részlistákat azonban már tárolhatjuk (és később be is olvashatjuk) nyalábolt formában, így ez a lépés csak $2B$ lemez I/O-műveletet igényel. Egy nagyméretű nem nyalábolt reláció rendezéses átvizsgálásának teljes költsége tehát $T + 2B$.

És végül nézzük meg egy index alapú átvizsgálás költségét. Egy R reláció indexe általában jóval kevesebb blokkot foglal el, mint $B(R)$. Ily módon a teljes R reláció átvizsgálása, amelyhez legalább B darab lemez I/O-műveletre van szükség, sokkal több lemez I/O-műveletet fog igényelni, mint a teljes index megvizsgálása. Éppen ezért, noha az index alapú átvizsgáláshoz szükség van a relációnak és az indexnek a megvizsgálására egyaránt, a továbbiakban a következő becslést fogjuk alkalmazni:

- Továbbra is a B, illetve T költségbecslést használjuk egy nyalábolt, illetve nem nyalábolt reláció index segítségével történő teljes beolvasásához.

Ha azonban csak az R egy részéhez akarunk hozzáférni, akkor gyakran elkerülhetjük a teljes index és a teljes R végigkeresését. Az indexek ilyen jellegű felhasználásának elemzését a 6.7.2. részre halasztjuk.

6.2.6. Fizikai operátorok megvalósításához használatos iterátorok

Több fizikai operátor megvalósítható *iterátorként*, ami nem más, mint három függvény olyan együttese, amely lehetővé teszi, hogy a fizikai operátor eredményének felhasználója soronként férjen hozzá az eredményhez. Egy művelet iterátorát felépítő három függvény a következő:

1. `Open`. Ez a függvény elindítja a sorok kinyerésének folyamatát, de nem ad vissza egyetlen sort sem. Inicializálja a művelethez szükséges adatszerkezeteket, és meghívja az `Open` függvényt a művelet argumentumaira.
2. `GetNext`. Ez a függvény visszaadja az eredmény következő sorát, és a helyzethez igazítja az adatszerkezeteket úgy, hogy lehetővé váljon további sorok kinyerése. Az eredmény következő sorának kinyeréséhez általában egyszer vagy többször meghívja az argumentum(ok)ra a `GetNext` függvényt. Ez a függvény beállít egy olyan jelzést is, amelyből kiderül, hogy sikerült-e sort előállítania, avagy nincs már több előállítandó sor. Erre a célra a `Found` nevű logikai típusú változót fogjuk használni, amelynek értéke akkor és csak akkor lesz igaz, ha a függvény egy új sort adott vissza.
3. `Close`. Ez a függvény befejezi az iterálást, miután az összes sort vagy az összes felhasználó által kívánt sort sikerült kinyerni. Általában meghívja a `Close` függvényt az operátor argumentumaira.

Amikor az iterátorokról és azok függvényeiről beszélünk, akkor az `Open`, `GetNext` és `Close` szavakra úgy tekintünk, mint metódusok túlterhelt neveire. Ez azt jelenti, hogy ezeknek a metódusoknak több különböző megvalósítása létezik, attól függően, hogy melyik „osztályra" alkalmazzuk a metódust. Ebben az esetben feltételezzük, hogy valamennyi fizikai operátorra létezik egy olyan osztály, melynek objektumai olyan relációk, amelyeket az operátor előállíthat. Ha az R egy ilyen osztály tagja, akkor az R iterátorának függvényeire az `R.Open()`, `R.GetNext()` és `R.Close()` jelöléseket használjuk.

Miért éppen iterátorok?

A 7.7. részben látjuk majd, hogy a lekérdezéstervekben használt iterátorok miként támogatják a hatékony végrehajtást. Az iterátorok ellentétesek a *materializáló* stratégiával, ahol valamennyi operátor eredményét teljes egészében előállítjuk – és ezt vagy lemezen vagy a memóriában tároljuk, ha ez utóbbi megengedett. Ha iterátorokat használunk, akkor egyszerre több művelet is aktív. Az operátorok sorokat adnak át egymásnak, ami csökkenti a tárolási szükségletet. Természetesen nem mindegyik fizikai operátor tudja hasznosítani az iterációs vagy „csővezetékes" megközelítést, ahogyan azt majd látni is fogjuk. Bizonyos esetekben majdnem mindent az `Open` függvénynek kell megvalósítania, ami gyakorlatilag egyenértékű a materializációval.

6.11. példa: Talán a legegyszerűbb iterátor az, amelyik a táblaátvizsgálás-operátort valósítja meg. Tegyük fel, hogy szeretnénk végrehajtani a `TableScan(R)`-t, ahol R egy olyan nyalábolt reláció, amelynek blokkjaihoz kényelmesen hozzáférhetünk. Ily módon feltételezhetjük, hogy a „vegyük az R következő blokkját" feladatot nem kell részletesen körülírnunk, mivel ezt a háttértároló könnyen megvalósítja. Továbbá fel-

```
Open(R) {
   b := az R első blokkja;
   t := a b blokk első sora;
   Found := TRUE;
}

GetNext(R) {
   IF (t túl van a b blokk utolsó során) {
      legyen b a következő blokk;
      IF (nincs következő blokk) {
         FOUND := FALSE;
         RETURN;
      }
      ELSE /* b egy új blokk*/
         t := a b blokk első sora;
   }
/*készen állunk arra, hogy visszaadjuk a t sort és továbblépjünk*/
   oldt := t;
   legyen t a b blokk következő sora;
   RETURN oldt;
}

CLOSE(R) {
}
```

6.8. ábra. *A táblaátvizsgálás-operátor iterátora*

tételezzük, hogy egy blokkon belül egy rekordokból (sorokból) álló könyvtárat találunk, ezáltal könnyű kinyerni egy blokk következő sorát avagy megmondani, hogy elérkeztünk az utolsó sorhoz.

A 6.8. ábra az iterátor három függvényét vázolja. Elképzelünk egy *b* blokkmutatót és egy *t* sormutatót, amelyek egy *b* blokk *t* sorára mutatnak. Feltételezzük, hogy mindkét mutató képes „túlmutatni" az utolsó blokkon, illetve egy blokk utolsó során, és, hogy azonosítani lehet az ilyen helyzeteket. Figyeljük meg, hogy a `Close` ebben az esetben semmit nem végez. A gyakorlatban egy iterátor `Close` függvénye többféleképpen is megtisztíthatja az ABKR belső szerkezeteit. Értesítheti a pufferkezelőt, hogy egy bizonyos pufferre nincs tovább szükség, vagy értesítheti a tranzakciókezelőt, hogy egy reláció olvasása befejeződött. □

6.12. példa: Most lássunk egy olyan példát, amikor az iterátor a legtöbb munkát az `Open` függvényével végzi. Az operátor a rendezéses átvizsgálás, ahol beolvassuk az *R* reláció sorait, de rendezett sorrendben adjuk őket vissza. Tegyük fel továbbá, hogy az *R* elég nagy, tehát kétfázisú, többmenetes összefésüléses rendezést kell használnunk úgy, mint a 2.3.4. részben.

Még az első sort sem tudjuk visszaadni mindaddig, amíg meg nem vizsgáltuk az *R* valamennyi sorát. Ily módon az `Open`-nek legalább a következőket meg kell tennie:

1. Beolvassa az *R* valamennyi sorát memória méretű részenként, rendezi őket, majd eltárolja őket lemezen.
2. Inicializálja az adatszerkezeteket a második (összefésülő) fázishoz, és betölti valamennyi részlista első blokkját a memóriabeli struktúrákba.

Ezután, a `GetNext` folyamatosan „versenyezteti" a részlisták elején álló sorokat, hogy kiválassza azt a sort, amely valamennyi részlistát tekintve az első lesz. Ha a győztes részlista kiürül, akkor a `GetNext` újra betölti a hozzá tartozó puffert. □

6.13. példa: Végül nézzünk meg egy egyszerű példát arra, hogy más iterátorok meghívásával miként kombinálhatjuk az iterátorokat. Ez nem egy tipikusan jó példa arra, hogy miként alkalmazható egyszerre több iterátor. Ezzel még várnunk kell addig, amíg más fizikai operátorok (pl. kiválasztás és összekapcsolás) algoritmusaival nem foglalkozunk, mert azok jobban kiaknázzák az iterátorok nyújtotta lehetőségeket.

Az általunk választott művelet a multihalmazos egyesítés, $R \cup S$, amelyben először előállítjuk az *R* összes sorát majd az *S* összes sorát, tekintet nélkül az ismétlődésekre. Feltesszük, hogy az `R.Open`, `R.GetNext` és `R.Close` függvények léteznek, és ezek alkotják az *R*-hez tartozó iterátort, és ugyanígy az *S* reláció függvényei is léteznek. Ezek a függvények lehetnek az *R* és *S* relációkra alkalmazott táblabeolvasás függvényei, ha ezek tárolt relációk, illetve lehetnek olyan iterátorok, amelyek az *R* és *S* kiszámításához más iterátorok egész rendszerét meghívják. Az egyesítés iterátor függvényeit a 6.9. ábrán vázoltuk. Ezeknek a függvényeknek egyik finomsága az, hogy egy `CurRel` nevű megosztott változót használnak, ami vagy *R* vagy *S*, attól függően, hogy melyik az aktuális, éppen olvasott reláció. □

6.3. Adatbázis-műveletek egymenetes algoritmusai

Az alábbiakban megkezdjük egy, a lekérdezések optimalizálásában különösen fontos témának a tanulmányozását: Hogyan is végezzük el egy logikai lekérdezésterv egyedi lépéseit – például egy összekapcsolást vagy egy kiválasztást? A logikai lekérdezésterv fizikai lekérdezéstervvé való átalakítási folyamatának alapvető fontosságú lépése az egyes operátorok algoritmusának kiválasztása. Noha az operátorok megvalósítására vonatkozó algoritmusokra számos javaslat született, ezek nagyjából három csoportba sorolhatók:

1. Rendezésen alapuló módszerek. Ezekkel főként a 6.5. rész foglalkozik.
2. Tördelésen alapuló módszerek. Lásd többek között a 6.6. és a 6.10. részeket.
3. Index alapú módszerek. Ilyenek főként a 6.7. részben szerepelnek.

A fentiek mellett az operátorok algoritmusait „nehézségi fok" és költség szempontjából három fokozatra oszthatjuk fel:

a) Bizonyos módszereknél az adatokat csak egyszer kell lemezről beolvasni. Ezeket nevezzük *egymenetes* algoritmusoknak, és ezek képezik ennek a szakasznak a tár-

```
Open(R,S) {
   R.Open();
   CurRel := R;
}

GetNext(R,S) {
   IF (CurRel = R) {
      t := R.GetNext();
      IF (Found) /* R nem ürült ki */
         RETURN t;
      ELSE /* R kiürült */ {
         S.Open();
         CurRel := S;
      }
   }
   /* itt S-ből kell olvasni */
   RETURN S.GetNext();
/* Vegyük észre, hogy ha S kiürült, akkor a Found-ot az S.GetNext
FALSE-ra állítja, ami a GetNext számára is a helyes művelet */
}

Close(R,S) {
   R.Close();
   S.Close();
}
```

6.9. ábra. *Az egyesítés iterátor felépítése komponenseiből*

gyát. Általában véve csak akkor működnek, ha a művelet argumentumainak legalább egyike belefér a memóriába, bár léteznek kivételes esetek is, főleg a kiválasztás és a vetítés terén, amint azt a 6.3.1. szakasz taglalja.

b) Vannak olyan módszerek, amelyek olyan adatokra is működnek, amelyek túl nagyok ahhoz, hogy beférjenek a rendelkezésre álló memóriába, de az elképzelhető legnagyobb adathalmazokra már nem működnek. Egy ilyen algoritmusra példa a 2.3.4. rész kétfázisú összefésülő rendezése. Ezeket a *kétmenetes* algoritmusokat úgy jellemezhetjük, hogy az adatokat az első alkalommal lemezről kell beolvasni, aztán következik valamilyen típusú feldolgozásuk, majd az összes – vagy majdnem az összes – adatot lemezre kell írni, és ekkor következik a második menetben a második olvasás a további feldolgozáshoz. Az ilyen algoritmusokkal a 6.5. és a 6.6. részekben találkozhatunk.

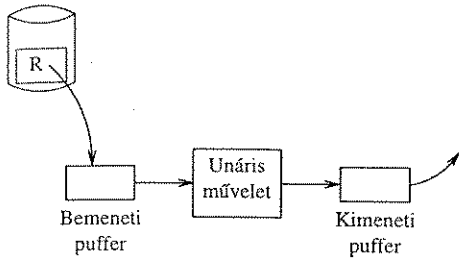
c) Végül vannak olyan módszerek, amelyek az adatok méretétől függetlenül működnek. Az ilyen algoritmusok három vagy annál is több menettel dolgoznak, és tulajdonképpen a kétmenetes algoritmusok természetes, rekurzív általánosításai. A többmenetes algoritmusokkal a 6.9. részben foglalkozunk.

Ebben a részben az egymenetes módszerekre fogunk összpontosítani. Ugyanakkor mind itt, mind a későbbi fejezetekben az operátorokat durván szólva a következő három csoport egyikébe fogjuk sorolni:

1. *Soronkénti, unáris műveletek.* Ezek a műveletek – a kiválasztás és a vetítés – nem igénylik, hogy a teljes reláció egyszerre a memóriában legyen (sőt még azt sem, hogy annak jelentős része ott legyen). Így a blokkokat egyenként olvashatjuk be, egyetlen memóriapuffert használva, majd megadhatjuk a kimenetet.
2. *Unáris, teljes relációs műveletek.* Az ilyen egy argumentumos műveleteknél az összes sort (vagy legalábbis a sorok nagy részét) egyszerre kell a memóriában látnunk. Ezért az egymenetes algoritmusok a körülbelül M méretű vagy annál kisebb relációkra vannak korlátozva. E csoport általunk vizsgált műveletei a γ és a δ.
3. *Bináris, teljes relációs műveletek.* Az összes többi művelet ebbe a csoportba tartozik: az egyesítés, a metszet és a különbség halmaz- és multihalmaz-változatai, az összekapcsolások és a szorzatok. Látni fogjuk majd, hogy ezen műveletek mindegyikénél az argumentumok legalább egyikének mérete nem lehet nagyobb M-nél akkor, ha egymenetes algoritmust akarunk használni.

6.3.1. Soronkénti műveletek egymenetes algoritmusai

A $\sigma(R)$ és a $\pi(R)$ egymenetes műveletek algoritmusai egyértelműek, függetlenül attól, hogy a reláció belefér-e a memóriába vagy nem. R blokkjait egyenként olvassuk be a bemeneti pufferbe, a műveleteket minden soron elvégezzük, majd a kiválasztott vagy a vetített sorokat a 6.10. ábrán bemutatottak szerint kivisszük a kimeneti pufferbe. Mivel a kimeneti puffer lehet egy másik operátor bemeneti puffere, vagy az előző adatokat küldhet a felhasználóhoz vagy egy alkalmazáshoz, ezért a kimeneti puffert



6.10. ábra. *Az R reláción végrehajtott kiválasztás vagy vetítés*

nem vesszük számításba a helyigény tekintetében. Eszerint a bemeneti pufferre – B-től függetlenül – csak az $M \geq 1$ korlátot követeljük meg.

A folyamat lemez I/O-műveletigénye attól függ, hogy az R argumentum reláció hogyan áll rendelkezésre. Ha R kezdetben lemezen van, úgy a költség megegyezik azzal, ami az R-re vonatkozó táblabeolvasás vagy index alapú beolvasás költsége. E költségekkel a 6.2.5. részben foglalkoztunk. A költség jellemzően B, ha R nyalábolt, illetve T, ha R nem nyalábolt. Ezzel együtt szeretnénk emlékeztetni az olvasót arra a fontos kivételre, amikor a művelet a kiválasztás, a feltétel pedig egy indexszel rendelkező attribútumot hasonlít össze egy konstanssal. Ebben az esetben az indexet felhasználhatjuk arra, hogy az R-et tartalmazó blokkoknak csupán egy részhalmazát kelljen beolvasnunk, ami a teljesítményt gyakran jelentősen növeli.

6.3.2. Unáris, teljes relációs műveletek egymenetes algoritmusai

Térjünk most át azokra az unáris műveletekre, amelyek egyszerre nem csak egy sorra alkalmazandók, hanem inkább a teljes relációkra: ilyen az ismétlődések kiküszöbölése (δ) és a csoportosítás (γ).

Az extra pufferek felgyorsíthatják a műveleteket

Noha a soronkénti műveletek éppenséggel működhetnek egyetlen bemeneti és egyetlen kimeneti puffer használatával is, sok esetben meggyorsíthatjuk a feldolgozást több bemeneti puffer lefoglalásával, amint azt a 6.10. ábrán is szemléltetni próbáltuk. Az alapötlet először a 2.4.1. részben merült fel. Ha R-et a cilindereken belül egymást követő blokkokon tároljuk, akkor egy teljes cilindert úgy olvashatunk be a pufferekbe, hogy közben cilinderenként csak egyszer kell a fejbeállási időt és a keresési időt kivárnunk. Hasonlóképpen, ha a művelet kimenetét lehet tele cilindereken tárolni, úgy az írással szinte semmi időt sem pazarlunk el.

Ismétlődések kiküszöbölése

Az ismétlődések kiküszöböléséhez megtehetjük, hogy R blokkjait egyenként olvassuk be, viszont minden egyes sornál el kell döntenünk a következőket:

1. A sor most fordul-e elő először, amikor is átmásolhatjuk azt a kimenetbe, vagy
2. A sorral korábban már találkoztunk, és ebben az esetben nem szabad azt kiírnunk.

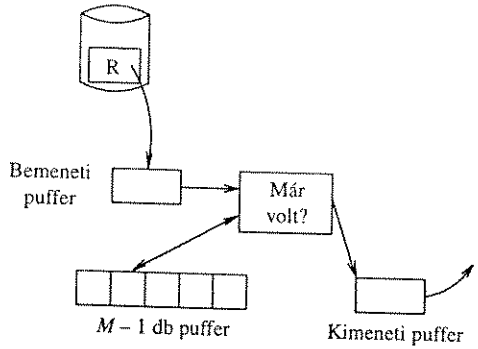
A döntés támogatására kell egy másolatot tartanunk a memóriában az összes általunk már látott sorról, amint azt a 6.11. ábra mutatja. Egy memóriapuffer tartalmazza R sorainak egy blokkját, míg a fennmaradó $M-1$ puffert használhatjuk arra, hogy tartalmazzák a már előfordult minden egyes sor egy másolatát.

A már előfordult sorok tárolásakor ügyelnünk kell arra, hogy milyen elsődleges memória-adatstruktúrát használunk. Naiv módon egyszerűen felsorolhatnánk, listázhatnánk a már előfordult sorokat. Amikor R egy új sorát vesszük, összehasonlítjuk azt a már előfordult összessel, és ha az előbbi nem egyezik meg az ismert sorok egyikével sem, akkor az új sort egyrészt kitesszük a kimenetbe, másrészt pedig hozzáadjuk a már előfordult sorok memóriabeli listájához.

Ugyanakkor, ha az elsődleges memóriában eleve n darab sor van, akkor minden egyes új sor n-nel arányos processzoridőt igényel, így a teljes művelet végrehajtásának processzoridő-igénye n^2-tel lesz arányos. Mivel n nagyon nagy is lehet, ez a komoly időigény megkérdőjelezheti azt a feltételezésünket, miszerint csupán a lemez I/O-műveletek ideje volt jelentős. Másképpen szólva, olyan elsődleges memóriastruktúrára van szükségünk, amely lehetővé teszi az alábbi műveletek mindegyikét:

1. Új sor hozzáadása.
2. Annak meghatározása, hogy egy adott sor már szerepelt-e.

A fenti műveleteket közel konstans idő alatt kellene elvégezni, a memóriában aktuálisan tárolt sorok n számától lényegében függetlenül. Számos ilyen struktúra is-



6.11. ábra. *Memória kezelése ismétlődések egymenetes kiküszöbölésénél*

mert. Használhatunk például egy tördelőtáblát nagyszámú kosárral vagy a kiegyensúlyozott bináris keresési fa valamelyik formáját[1]. Az említett struktúrák mindegyike igényel még némi további helyet a sorok tárolásához felhasználtak mellett. Például egy memóriabeli tördelőtáblának szüksége van még helyre a kosarakból álló tömb számára, illetve a kosarakon belül a sorok láncolására szolgáló mutatók számára. Ugyanakkor a plusz helyigény általában kicsi a sorok tárolásához szükséges helyhez képest. Ennek értelmében tehát azt az egyszerűsítő feltevést tesszük, hogy nincs szükség további külön tárra, és a sorok memóriában történő tárolásához szükséges helyre összpontosítjuk a figyelmünket.

A feltevés értelmében az elsődleges memória $M-1$ rendelkezésre álló pufferében annyi sort tárolhatunk, amennyi elfér R $M-1$ blokkjában. Ha azt akarjuk elérni, hogy R minden különálló sorának egy példánya elférjen az elsődleges memóriában, úgy $B(\delta(R))$ nem lehet nagyobb $M-1$-nél. Mivel várakozásunk szerint M sokkal nagyobb 1-nél, a továbbiakban általában a következő egyszerűbb közelítéssel fogunk élni:

- $B(\delta(R)) \leq M$.

Vegyük észre, hogy általánosságban nem számíthatjuk ki $\delta(R)$ méretét anélkül, hogy ne számítanánk ki magát $\delta(R)$-t. Amennyiben ez utóbbi méretet alulbecsülnénk, azaz $B(\delta(R))$ valójában nagyobb lenne M-nél, úgy ezért komoly árat fizetnénk (ki-be olvasás formájában), mivel az R különböző sorait tartalmazó blokkokat gyakran kellene ki- és bevinni az elsődleges memóriába.

Csoportosítás

Egy γ_L csoportosító művelet nulla vagy több csoportosító attribútumot, és feltehetőleg egy vagy több összesített attribútumot ad vissza. Ha az elsődleges memóriában minden egyes csoportnak – pontosabban szólva a csoportosító attribútumok minden értékének – egy bejegyzést hozunk létre, akkor R sorait blokkonként egymás után beolvashatjuk. Egy csoport *bejegyzése* (entry) a csoportosító attribútumok értékeiből és az egyes összesítések értékeiből képzett kumulált értékekből áll. A kumulált érték – egy esetet leszámítva – egyértelmű:

- Egy `MIN(a)` vagy egy `MAX(a)` összesítés esetén jegyezzük fel az a attribútumnak a csoport bármely soránál látott minimális vagy maximális értékét. Szükség szerint változtassuk meg ezt a minimumot vagy maximumot minden alkalommal, amikor a csoport egy sorát megvizsgáltuk.

[1] A megfelelő elsődleges memóriastruktúrák tárgyalását lásd Aho, A. V., J. E. Hopcroft, J. D. Ullman: *Adatstruktúrák és algoritmusok* (Data Structures and Algorithms, Addison-Wesley, 1984) c. művében. Érdemes kiemelni, hogy n elem tördeléssel történő feldolgozásához $O(n)$ időre van szükség, míg a kiegyensúlyozott fával történő feldolgozás időigénye $O(n \log n)$; a mi céljainkra mindkettő elegendően megközelíti a lineáris időt.

Nem nyalábolt adatokkal végzett műveletek

Tartsuk szem előtt, hogy az egy művelethez szükséges lemez I/O-műveletek számát azzal a feltételezéssel számítottuk ki, hogy a művelet tárgyát képező relációk nyaláboltak. Abban az (egyébként ritkán előforduló) esetben, ha R nem nyalábolt, előfordulhat, hogy nem $B(R)$, hanem inkább $T(R)$ lemez I/O-műveletekre van szükség R összes sorának a beolvasásához. Vegyük azonban észre, hogy minden olyan relációt, amelyet egy operátor eredményeképpen kaptunk, nyaláboltnak tételezhetünk fel, mivel semmi oka nincs annak, hogy egy időleges relációt nem nyalábolt formában tároljunk.

- Minden `COUNT` összesítésre adjunk hozzá egyet az értékhez a csoport minden egyes előforduló sora esetén.
- `SUM(a)` esetében adjuk hozzá az a attribútum értékét az addig kumulált összeghez.
- A problémás eset az `AVG(a)`. Két kumulációt kell fenntartanunk: a csoport sorainak számlálását és a sorok a értékeinek az összegét. Ezeket a `COUNT`, illetve a `SUM` összesítés szabályai szerint számítjuk ki. Miután R minden sorát végignéztük, az átlag kiszámításához az összeg (SUM) és a számosság (COUNT) hányadosát képezzük.

Ha R összes sorát beolvastuk a bemeneti pufferbe, és azok már hozzájárultak csoportjaik összesítéséhez, akkor elkészíthetjük a kimenetet az egyes csoportokhoz tartozó sorok kiírásával. Vegyük észre, hogy nem hozhatjuk létre a γ művelet kimenetét addig, amíg az utolsó sort meg nem néztük. Ez az észrevétel azt is jelenti, hogy a szóban forgó algoritmus nem igazán illik bele az iterátor sémába; a teljes csoportosítást el kell végezni az `Open` függvénnyel még azt megelőzően, hogy a `GetNext` az első sort visszaadhatná.

Annak érdekében, hogy az egyes sorok memórián belüli feldolgozása hatékony legyen, olyan elsődleges memória-adatstruktúrát kell használnunk, amelynek segítségével a csoportosító attribútumok értékeinek ismeretében megkaphatjuk az egyes csoportok bejegyzését. Amint azt az előzőekben a γ művelettel kapcsolatban bemutattuk, ezt a célt jól szolgálják az olyan memóriabeli adatstruktúrák, mint a tördelőtáblák vagy a kiegyensúlyozott fák. Nem szabad azonban elfelejtenünk, hogy ennek a struktúrának a keresési kulcsa csak a csoportosító attribútumokból áll.

A fenti egymenetes algoritmushoz szükséges lemez I/O-műveletek száma B, amint annak lennie is kell egy unáris operátor bármely egymenetes algoritmusa esetében. A szükséges memóriapufferek M száma nem áll semmilyen egyszerűen leírható kapcsolatban B-vel, bár elmondható, hogy M általában kisebb B-nél. A gondot az okozza, hogy a csoportok bejegyzése lehet R sorainál rövidebb vagy akár hosszabb is, a csoportok száma pedig bármi lehet, ami R sorainak számánál kisebb vagy azzal egyenlő. A legtöbb esetben azonban a csoportbejegyzések nem hosszabbak R sorainál, és a csoportok száma sokkal kisebb a sorokénál.

6.3.3. Bináris műveletek egymenetes algoritmusai

Foglalkozzunk most a bináris műveletekkel: ezek az egyesítés, a metszet, a különbség, a szorzat és az összekapcsolás. Az összekapcsolások tárgyalásának egyszerűsítése céljából csak a természetes összekapcsolással foglalkozunk. Egy egyenlőségen alapuló összekapcsolás – az attribútumok megfelelő átnevezését követően – hasonló módon valósítható meg, a théta-összekapcsolásokat pedig felfoghatjuk úgy, mint egy szorzatot vagy egyenlőségen alapuló összekapcsolást, amelyet egy olyan kiválasztás követ, amit az egyenlőséges összekapcsolásban nem lehet kifejezni.

Létezik egy kivételes művelet – a multihalmazos egyesítés –, amelyet egy nagyon egyszerű egymenetes algoritmussal elvégezhetünk. $R \cup_B S$ kiszámításához először R minden sorát kitesszük a kimenetbe, majd ezt követően lemásoljuk S minden sorát, ahogy azt a 6.13. példában tettük. A lemez I/O-műveletek száma $B(R) + B(S)$, amint annak egy R és S operandusú egymenetes algoritmus esetén lennie is kell; az $M = 1$ pedig elegendő feltétel, függetlenül attól, hogy R és S mekkorák.

Más bináris műveletek esetén az R és S operandusok közül a kisebbiket kell beolvasni az elsődleges memóriába, majd olyan alkalmas adatstruktúrát kell kialakítani, hogy a sorokat mind beilleszteni, mind kikeresni könnyű legyen, amint azt a 6.3.2. részben ismertettük. Csakúgy mint korábban, itt is elegendő egy tördelőtáblázat vagy egy kiegyensúlyozott fa használata. A struktúra felépítéséhez kevés helyre van szükség (a sorok által amúgy is elfoglalt hely mellett), amit a továbbiakban elhanyagolunk. Így az R és S relációkon egy menetben végrehajtandó bináris művelettel szemben hozzávetőleges a következő követelmény adódik:

- $\min((B(R), B(S)) \leq M$.

A fenti egyenlőtlenség azt tételezi fel, hogy egy puffert használunk a nagyobb reláció blokkjainak beolvasására, míg körülbelül M darab pufferre van szükség a teljes kisebbik reláció és a hozzá tartozó memóriabeli adatstruktúra befogadására.

Az alábbiakban megadjuk a különféle műveletekhez tartozó részleteket. Minden alkalommal feltesszük, hogy a relációk közül R a nagyobb, és S-et tartjuk az elsődleges memóriában.

Halmazegyesítés

S-et beolvassuk $M-1$ memóriapufferbe, majd olyan keresési struktúrát építünk fel, amelyben a keresési kulcs maga a teljes sor. Az összes beolvasott sort ezután a kimenetbe is bemásoljuk. Ezután R minden egyes blokkját egyenként beolvassuk az M-edik pufferbe. Az R-ben lévő minden t sorra megnézzük, hogy az benne van-e S-ben, és ha nem, akkor t-t a kimenetbe másoljuk.

Halmazmetszet

S-et beolvassuk $M-1$ pufferbe, és olyan keresési struktúrát építünk fel, amelyben a teljes sorok alkotják a keresési kulcsot. Beolvassuk R minden blokkját, majd minden t sorára megnézzük, hogy az S-ben is szerepel-e. Ha igen, akkor t-t a kimenetbe másoljuk, ha pedig nem, akkor figyelmen kívül hagyjuk.

Halmazkülönbség

Mivel a különbség nem kommutatív operátor, különbséget kell tennünk $R -_S S$ és $S -_S R$ között. Továbbra is feltételezzük, hogy R a nagyobbik reláció. Mindkét esetben S-et beolvassuk $M-1$ pufferbe, majd olyan keresési struktúrát építünk fel, amelyben teljes sorok alkotják a keresési kulcsot.

$R -_S S$ kiszámításához beolvassuk R minden blokkját, és egyenként megvizsgáljuk az adott blokk t sorait. Ha t S-ben van, akkor figyelmen kívül hagyjuk, ha pedig nincs S-ben, akkor bemásoljuk a kimenetbe.

$S -_S R$ kiszámításához ismét beolvassuk R blokkjait, és egymás után megvizsgáljuk a t sorokat. Ha t S-ben van, akkor töröljük azt S memóriabeli másolatából, ha pedig nincs, akkor nem csinálunk semmit. Miután R minden egyes sorát megvizsgáltuk, bemásoljuk a kimenetbe S megmaradó sorait.

Multihalmazmetszet

S-et beolvassuk $M-1$ pufferbe, majd minden egyes különböző sorhoz egy *számlálót* (count) rendelünk, amely kezdetben azt mutatja, hogy a szóban forgó sor hányszor fordul elő S-ben. Egy t sor több példányát nem többször, külön-külön tároljuk, hanem csak egyetlen másolatát, és ehhez társítjuk azt a számlálót, ami megegyezik t előfordulásainak számával.

Az említett struktúra valamivel több helyet igényelhetne mint $B(S)$ darab blokk, ha kevés számú ismétlődés fordulna elő, bár gyakran az a helyzet, hogy S kellően tömör. Így továbbra is azt tesszük fel, hogy a $B(S) \leq M-1$ elegendő feltétel egy egymenetes algoritmus működéséhez, bár ez a feltétel inkább csak közelítés.

A következőkben beolvassuk R minden egyes blokkját, és annak minden t sorára megnézzük, hogy a sor előfordul-e S-ben. Ha nem, akkor figyelmen kívül hagyjuk, t nem jelenhet meg a metszetben. Ha azonban t megjelenik S-ben, és a hozzá tartozó számláló még mindig pozitív, akkor t-t a kimenetbe tesszük, és a számlálót eggyel csökkentjük. Ha t megjelenik S-ben, de számlálója elérte a nullát, akkor nem tesszük a kimenetbe, hiszen ez azt jelenti, hogy t-nek már annyi példányát írtuk át a kimenetbe, ahányszor az S-ben megtalálható volt.

Multihalmazkülönbség

$S -_B R$ kiszámításához S sorait beolvassuk a memóriába, majd megszámoljuk minden egyes különböző sor előfordulási gyakoriságát, ahogy azt a multihalmazmetszetnél is tettük. Amikor R-et beolvassuk, minden t sornál megnézzük, hogy az előfordul-e S-ben, és ha igen, akkor csökkentjük a hozzá tartozó számlálót. Ha ez megtörtént, akkor átmásoljuk a kimenetbe az összes olyan memóriabeli sort, amelynek a számlálója pozitív, és a sort annyiszor másoljuk, amennyi az említett számláló.

$R -_B S$ kiszámításához S sorait megint beolvassuk a memóriába, majd megszámoljuk a különböző sorok előfordulásának gyakoriságát. Egy c számlálójú t sorra gondolhatunk úgy, mintha c darab okunk lenne arra, hogy t-t ne másoljuk a kimenetbe R sorainak beolvasásakor. Más szavakkal, amikor R egy t sorát beolvassuk, megnézzük, hogy az S-ben szerepel-e. Ha nem, akkor t-t átmásoljuk a kimenetbe. Ha viszont t szerepel S-ben, akkor megnézzük a hozzá tartozó aktuális c számlálót. Ha $c = 0$, akkor t-t a kimenetbe másoljuk. Ha viszont $c > 0$, akkor t-t nem másoljuk a kimenetbe, hanem c-t csökkentjük eggyel.

Szorzat

S-et beolvassuk az elsődleges memória $M-1$ pufferébe. Itt nincs szükség semmilyen speciális adatstruktúrára. Ezt követően beolvassuk R minden egyes blokkját, majd R minden sorára t-t összefűzzük S minden egyes sorával a memóriában. Minden összefűzött sor úgy kerül a kimenetre, ahogyan előállt.

Vegyük észre, hogy ez az algoritmus minden R-beli sorra jelentős processzoridőt igényelhet, mert minden sort $M-1$ darab sorokkal teli blokkhoz kell párosítani. Ugyanakkor a kimenet mérete is nagy, és azt várhatjuk, hogy az eredmény lemezre írásához vagy a kimenet más jellegű feldolgozásához szükséges idő meghaladja majd a kimenet létrehozásához szükséges processzoridőt.

Természetes összekapcsolás

Itt és a többi összekapcsolási algoritmusnál éljünk azzal a konvencióval, hogy $R(X, Y)$-t kapcsoljuk össze $S(Y, Z)$-vel, ahol Y jelöli R és S összes közös attribútumát, X jelöli R összes olyan attribútumát, amelyek nincsenek benne S sémájában, végül pedig Z jelöli S összes olyan attribútumát, amelyek nincsenek benne R sémájában. Továbbra is feltesszük, hogy S a kisebbik reláció. A természetes összekapcsolás kiszámítását a következőképpen végezzük el:

1. Olvassuk be S összes sorát, és alkossunk belőlük egy olyan memóriabeli keresési struktúrát, amelyben Y attribútumai alkotják a keresési kulcsot. Szokás szerint egy tördelőtáblázat vagy egy kiegyensúlyozott fa jó példája az ilyen struktúráknak. E célra használjunk $M-1$ memóriablokkot.

Mi van akkor, ha M ismeretlen?

Noha az algoritmusokat úgy mutattuk be, mintha M, a rendelkezésre álló memóriablokkok száma rögzített és eleve ismert lenne, nem szabad elfelejtenünk, hogy M nem egy esetben ismeretlen, hacsak nem veszünk figyelembe olyan triviális korlátokat, mint mondjuk a gép teljes memóriája. Éppen ezért egy lekérdezés-optimalizáló az egymenetes és kétmenetes algoritmusok közötti választáskor megpróbálhatja megbecsülni M-et, majd döntését erre a becslésre alapozhatja. Ha az optimalizáló téved, akkor a tévedés ára vagy pufferek ki-be olvasása a lemez és a memória között (ha M-et túlbecsültük), vagy szükségtelenül sok menet elvégzése (ha M-et alulbecsültük).

Van emellett még néhány olyan algoritmus is, amely rugalmasan alkalmazkodik, mihelyt a memória kisebb lesz a vártnál. Az ilyenek például egymenetes algoritmusként működnek addig, amíg ki nem futnak a helyből, ezután viszont már kétmenetes algoritmusként viselkednek. Néhány ilyen megközelítést ismertetnek a 6.6.6. és a 6.8.3. részek.

2. Olvassuk be R minden egyes blokkját a fennmaradó egyetlen memóriapufferbe, R minden egyes t sorára keressük meg a keresési struktúrával S azon sorait, amelyek megegyeznek t-vel Y összes attribútumán. S minden egyes megegyező sorára alkossunk egy sort a t-vel való összekapcsolással, majd tegyük az eredményként kapott sort a kimenetbe.

Ugyanúgy mint az összes többi bináris, egymenetes algoritmusnál, ennél is $B(R) + B(S)$ lemez I/O-műveletre van szükség az operandusok beolvasásához. Az algoritmus mindaddig működik, amíg $B(S) \leq M - 1$, illetve ha közelítőleg $B(S) \leq M$. A többi tanulmányozott algoritmushoz hasonlóan a memóriabeli keresési struktúra által igénybe vett pluszhelyet itt sem vettük figyelembe, mivel ez csak csekély többletet jelent.

A természetes összekapcsolástól különböző összekapcsolásokra itt nem fogunk kitérni. Emlékeztetünk rá, hogy az egyenlőségen alapuló összekapcsolást lényegében ugyanúgy kell elvégezni, mint a természetes összekapcsolást, de figyelembe kell vennünk, hogy a két reláció „azonos" attribútumai esetleg más névvel szerepelnek. Egy egyenlőségen alapuló összekapcsolástól (equijoin) különböző théta-összekapcsolás helyettesíthető egy olyan egyenlőségen alapuló összekapcsolással vagy szorzattal, amit egy kiválasztás követ.

6.3.4. Feladatok

6.3.1. feladat: Az alábbi műveletek mindegyikére írjunk olyan iterátort, amelyik a jelen fejezetben bemutatott algoritmust használja.

* a) Vetítés.
* b) Ismétlődések kiküszöbölése (δ).

c) Csoportosítás (γ_L).
* d) Halmazegyesítés.
e) Halmazmetszet.
f) Halmazkülönbség.
g) Multihalmazmetszet.
h) Multihalmaz-különbség.
i) Szorzat.
j) Természetes összekapcsolás.

6.3.2. feladat: A 6.3.1. feladatban szereplő összes operátorról döntsük el, hogy azok vajon *blokkolóak-e,* ami alatt azt értjük, hogy az első kimenet addig nem jöhet létre, amíg az összes bemenetet be nem olvastuk. Másképpen szólva egy *blokkoló* operátor olyan, aminek az összes fontos feladatát az `Open` végzi.

* **6.3.3. feladat:** Mutassuk meg, hogy mik lennének a csoportok bejegyzései, ha a 6.1.6.d) feladat lekérdezésében megvalósítanánk a γ operátort.

6.3.4. feladat: A 6.14. ábra összefoglalja az ebben és a következő fejezetben szereplő algoritmusok memória- és lemez I/O-művelet igényét. Azt is feltételeztük azonban, hogy az összes argumentum nyalábolt. Hogyan változnának meg a bejegyzések, ha az egyik vagy akár mindkét argumentum nem lenne nyalábolt?

! **6.3.5. feladat:** A 6.1.3. feladatban öt összekapcsolásszerű operátort definiáltunk. Adjunk meg mindegyikükre egymenetes algoritmust:

* a) $R \ltimes S$, feltéve, hogy R befér a memóriába.
* b) $R \ltimes S$, feltéve, hogy S befér a memóriába.
c) $R \overline{\ltimes} S$, feltéve, hogy R befér a memóriába.
d) $R \overline{\ltimes} S$, feltéve, hogy S befér a memóriába.
* e) $R \bowtie_L S$, feltéve, hogy R befér a memóriába.
f) $R \bowtie_L S$, feltéve, hogy S befér a memóriába.
g) $R \bowtie_R S$, feltéve, hogy R befér a memóriába.
h) $R \bowtie_R S$, feltéve, hogy S befér a memóriába.
i) $R \bowtie S$, feltéve, hogy R befér a memóriába.

6.4. Beágyazott ciklusú összekapcsolások

Mielőtt a később sorra kerülő szakaszokban áttérnénk az összetettebb algoritmusokra, foglalkozzunk először az összekapcsolás operátor „beágyazott ciklusú"-nak nevezett algoritmuscsaládjával. Ezek az algoritmusok bizonyos értelemben „másfél menetesek", mivel minden változatban a két argumentum egyikéhez tartozó sorokat csak egyszer olvassuk be, a másik argumentumot viszont többször olvassuk. Beágyazott ciklusú összekapcsolások használhatók bármekkora méretű relációra, nem szükséges, hogy az egyik reláció elférjen a memóriában.

6.4.1. Sor alapú beágyazott ciklusú összekapcsolás

Kezdjük a tárgyalást a beágyazott ciklusú témakör legegyszerűbb változatával, ahol a ciklusok a kérdéses relációk minden egyes sorára ismétlődnek. Ebben a *sor alapú beágyazott ciklusú összekapcsolásnak* nevezett algoritmusban a

$R(X, Y) \bowtie S(Y, Z)$

összekapcsolást a következőképpen számítjuk ki:

```
FOR S minden egyes s sorára DO
   FOR R minden egyes r sorára DO
      IF r és s összekapcsolható egy t sorrá THEN
         t kiírása;
```

Ha nem figyelünk arra, hogy hogyan pufferezzük az R és S relációk blokkjait, akkor a fenti algoritmus akár $T(R)T(S)$ számú lemez I/O-műveletet is igényelhet. Ugyanakkor sok esetben az algoritmus módosításával sokkal alacsonyabb költség is elérhető. Ilyen eset például az, amikor az R-beli összekapcsoló attribútum(ok)ra indexet tudunk használni, hogy megkeressük R sorai között azokat, amelyek megfelelnek S egy adott sorának, és ilyenkor nem kell a teljes R relációt beolvasnunk. Az index alapú összekapcsolásokat a 6.7.3. részben ismertetjük. Egy második javítási lehetőség sokkal figyelmesebben veszi szemügyre azt, hogy R és S sorai miként vannak megosztva a blokkok között, és a lehető legtöbb memóriát használja fel a lemez I/O-műveletek számának csökkentésére, amikor a belső ciklusban végighalad. Ezt a blokk alapú beágyazott ciklusú összekapcsolási változatot a 6.4.3. részben tárgyaljuk.

6.4.2. Egy iterátor a sor alapú beágyazott ciklusú összekapcsoláshoz

A beágyazott ciklusú összekapcsolás egyik előnye az, hogy jól beleillik az iterátoros megközelítésbe, és így egyes esetekben segít elkerülni a köztes relációk lemezen való tárolását. Ezt majd a 7.7.3. részben fogjuk látni részletesebben. Az $R \bowtie S$ iterátora könnyen felépíthető R és S iterátoraiból, amelyeket a 6.2.6. részhez hasonlóan most is `R.Open`, ... stb.-vel jelölünk. A beágyazott ciklusú összekapcsolás három iterátor függvényének a kódja a 6.12. ábrán látható. Feltételezzük, hogy sem az R, sem az S reláció nem üres.

6.4.3. Egy algoritmus a blokk alapú beágyazott ciklusú összekapcsoláshoz

Javíthatunk a 6.4.1. részben megismert sor alapú beágyazott ciklusú összekapcsoláson, ha $R \bowtie S$-t a következő módon számítjuk ki:

1. Mindkét argumentum relációnál megvalósítjuk a blokkonkénti hozzáférést.
2. A lehető legtöbb memóriát használjuk az S reláció, azaz a külső ciklushoz tartozó reláció sorainak tárolására.


```
Open(R,S) {
   R.Open();
   S.Open();
   s := S.GetNext();
}

GetNext(R,S) {
   REPEAT {
      r := R.GetNext();
      IF (NOT Found) { /*R az aktuális
            s-re kiürült */
         R.Close();
         s := S.GetNext();
         IF (NOT Found) RETURN; /* R és S
            is kiürült */
         R.Open();
         r := R.GetNext();
      }
   }
   UNTIL(r és s összekapcsolható);
   RETURN r és s összekapcsolva;
}

Close(R,S) {
   R.Close();
   S.Close();
}
```

6.12. ábra. *Sor alapú beágyazott ciklusú összekapcsolás iterátor függvényei*

Az 1. pont biztosítja, hogy amikor a belső ciklusban végigmegyünk R sorain, akkor R beolvasásához a lehető legkevesebb lemez I/O-műveletet használjuk fel. A 2. pont révén lehetőségünk nyílik arra, hogy R minden egyes beolvasott sorát ne csupán egy S-beli sorral kapcsoljuk össze, hanem annyival, amennyi csak elfér a memóriában.

Ugyanúgy mint a 6.3.3. részben, tegyük fel, hogy $B(S) \leq B(R)$, de emellett tegyük fel még azt is, hogy $B(S) > M$, azaz egyik reláció sem fér be teljesen a memóriába. Ismétlődően beolvassuk S-nek $M-1$ darab blokkját a memóriapuffereibe. S-nek a memóriában lévő sorai számára létrehozunk egy olyan keresési struktúrát, amelynek kulcsa megegyezik R és S közös attribútumaival. Ezt követően végigvesszük R összes blokkját, azokat egyenként a memória legutolsó blokkjába beolvasva. Ha ez megtörtént, akkor R blokkjának összes sorát összehasonlítjuk S éppen memóriában lévő blokkjainak összes sorával. Az összekapcsolódó sorok esetén az összekapcsolt sort a kimenetbe tesszük. A most ismertetett algoritmus beágyazott ciklusú struktúráját a 6.13. ábra formális bemutatása jól szemlélteti.

A 6.13. ábra programja látszólag három egymásba ágyazott ciklust tartalmaz. Ha azonban a kódot a helyes absztrakciós szinten nézzük, akkor valójában csak két cik-

```
FOR S minden M-1 blokkból álló darabjára DO BEGIN
   olvassuk be ezeket a blokkokat a memóriapufferekbe;
   a sorokat szervezzük egy keresési struktúrába,
   melynek kulcsa az R és S közös attribútumai;
   FOR R minden egyes b blokkjára DO BEGIN
      olvassuk be b-t a memóriába;
      FOR b minden t sorára DO BEGIN
         keressük meg S azon sorait a memóriában,
         amelyek kapcsolódnak t-vel;
         írjuk ki e sorok t-vel való összekapcsolását;
      END ;
   END ;
END ;
```

6. 13. ábra. *A beágyazott ciklusú összekapcsolás algoritmusa*

lust találunk. Az első, a külső ciklus S sorain fut végig, a másik két ciklus pedig R sorain fut. Ez utóbbi folyamatot annak hangsúlyozására tüntettük fel két ciklusból állóként, hogy az a sorrend, amelyben R sorait végigvesszük, nem tetszőleges. Ezeket a sorokat blokkonként kell végigvennünk (a második ciklus szerepe), és egy blokkon belül annak összes sorát végignézzük, mielőtt átlépnénk a következő blokkra (a harmadik ciklus szerepe).

6.14. példa: Tegyük fel, hogy $B(R) = 1000$, $B(S) = 500$ és legyen $M = 101$. 100 darab memóriablokkot fogunk használni S-nek 100 blokkos darabokban történő pufferezésére, így a 6.13. ábra külső ciklusát ötször kell végrehajtani. Minden egyes iteráció alkalmával 100 lemez I/O-művelettel olvassuk be S egy darabját, majd R-et teljes egészében be kell olvasnunk a második ciklusban, amihez 1000 lemez I/O-műveletre van szükség. Így az összes lemez I/O-műveletek száma 5500.

Vegyük észre, hogy ha R és S szerepét felcseréltük volna, akkor az algoritmus valamivel több lemez I/O-műveletet használt volna fel. Ekkor 10 alkalommal hajtódna végre a külső ciklus, alkalmanként 600 lemez I/O-műveletet végezve, azaz az összes I/O-műveletek száma 6000 lenne. Általában igaz, hogy a kisebb relációnak a külső ciklusban való használata némi előnyt jelent. □

A 6.13. ábra algoritmusát néha „beágyazott blokkos összekapcsolás"-nak nevezik. Mi továbbra is megmaradunk az egyszerű *beágyazott ciklusú összekapcsolás* (nested-loop join) névnél, hiszen a beágyazott ciklusú séma gyakorlatban leggyakrabban megvalósított változatáról van szó. Ha szükség van a 6.4.1. rész sor alapú beágyazott ciklusú összekapcsolásától való megkülönböztetésre, akkor a 6.13. ábra sémájára mint „blokk alapú beágyazott ciklusú összekapcsolás"-ra fogunk hivatkozni.

6.4.4. A beágyazott ciklusú összekapcsolás elemzése

A 6.14. példa elemzése megismételhető tetszőleges $B(R)$, $B(S)$ és M esetén. Tegyük fel, hogy S a kisebbik reláció, és a darabok, vagyis a külső ciklus iterációinak száma $B(S)/(M-1)$. Minden iteráció során S-nek $M-1$ blokkját és R-nek $B(R)$ számú blokkját olvassuk be. A lemez I/O-műveletek száma tehát

$$\frac{B(S)}{M-1}\left(M-1+B(R)\right)$$

vagy átalakítva

$$B(S)+\frac{B(S)B(R)}{M-1}$$

Feltéve, hogy mind M, mind $B(S)$ és $B(R)$ nagy, és közülük M a legkisebb, a fenti formula $B(S)B(R)/M$-mel közelíthető. Ez más szavakkal azt jelenti, hogy a költség a két reláció méretének szorzata és a rendelkezésre álló memória hányadosával arányos. Sokkal jobban járunk akkor, ha mindkét reláció nagy, bár észrevehető, hogy megfelelően kis példák esetén (mint pl. a 6.14. volt) egy beágyazott ciklusú összekapcsolás költsége nem sokkal haladja meg egy egymenetes összekapcsolását, amely ez esetben 1500 lemez I/O-művelet lenne. Valóban, ha $B(S) \leq M-1$, akkor a beágyazott ciklusú összekapcsolás a 6.3.3. rész egymenetes összekapcsolási algoritmusával azonossá válik.

Noha általában véve a beágyazott ciklusú nem a rendelkezésünkre álló lehető leghatékonyabb összekapcsolási algoritmus, meg kell jegyeznünk azt is, hogy néhány korai ABKR esetében ez volt az egyetlen elérhető típus. Még manapság is szükség van rá bizonyos esetekben, hatékonyabb összekapcsolási algoritmusok szubrutinjaként, például amikor az egyes relációk nagyszámú sorához az összekapcsoló attribútum(ok) ugyanazon értéke tartozik. Olyan esetre, amikor a beágyazott ciklusú összekapcsolás alapvető fontosságú, a 6.5.5. részben láthatunk példát.

6.4.5. Az eddigi algoritmusok összefoglalása

A 6.3. és a 6.4. részekben tárgyalt algoritmusok memóriaszükségleteit és lemez I/O-művelet igényét a 6.14. ábra szemlélteti. A γ és a δ műveletek memóriaigénye valójában a feltüntetettnél bonyolultabb, az $M = B$ pedig csak durva közelítés. A γ esetén M a csoportok számával növekszik, míg a δ esetén M növekedése a különböző sorok számának növekedését követi.

Operátor	*Szükséges M kb.*	*Lemez I/O-műveletek*	*Rész*
σ, π	1	B	6.3.1.
γ, δ	B	B	6.3.2.
$\cup, \cap, -, \times, \bowtie$	min $(B(R), B(S))$	$B(R) + B(S)$	6.3.3.
$\bowtie$	bármely $M \geq 2$	$B(R)B(S)/M$	6.4.3.

6.14. ábra. *Az egymenetes és a beágyazott ciklusú algoritmusok memória és lemez I/O-művelet követelményei*

6.4.6. Feladatok

6.4.1. feladat: Adjuk meg a blokk alapú beágyazott ciklusú összekapcsolás három iterátor függvényét.

* **6.4.2. feladat:** Tegyük fel, hogy $B(R) = B(S) = 10\,000$, és $M = 1000$. Számítsuk ki egy beágyazott ciklusú összekapcsolás lemez I/O-művelet költségét.

6.4.3. feladat: A 6.4.2. feladatban szereplő relációk esetén M milyen értéke esetén nem lenne szükség több mint

a) 100 000
! b) 25 000
! c) 15 000

lemez I/O-műveletre $R \bowtie S$ beágyazott ciklusú összekapcsolási algoritmussal történő kiszámításához?

! **6.4.4. feladat:** Ha R és S közül az egyik sem nyalábolt, akkor úgy tűnik, mintha a beágyazott ciklusú összekapcsoláshoz mintegy $T(R)T(S)/M$ lemez I/O-műveletre lenne szükség.

a) Hogyan csökkenthető jelentősen ez a költség?
b) Ha R és S közül csak az egyik nem nyalábolt, hogyan hajtanánk végre a beágyazott ciklusú összekapcsolást? Vizsgáljuk meg mind a két esetet, vagyis amikor a nagyobb, illetve amikor a kisebb reláció a nem nyalábolt.

! **6.4.5. feladat:** A 6.12. ábrán bemutatott iterátor nem fog helyesen működni, ha akár R, akár S üres. Írjuk át a függvényeket olyan formába, hogy azok működjenek akkor is, ha a relációk egyike vagy mindkettő üres.

6.5. Rendezésen alapuló kétmenetes algoritmusok

Ebben az alfejezetben elkezdjük a többmenetes algoritmusok tanulmányozását. Ezeket olyan relációkon végzett relációs algebrai műveletek végrehajtására használjuk, amelyeknél a relációk mérete nagyobb, mint amit a 6.3. rész egymenetes algoritmusai még kezelni képesek. Konkrétan a *kétmenetes algoritmusokra* fogunk összpontosítani, amelyekben az operandus reláció adatait először beolvassuk a memóriába, ott valamilyen módon feldolgozzuk azokat, majd kiírjuk a lemezre őket. Végül a művelet befejezéséhez ismét beolvassuk az adatokat a memóriába. Ezt az alapötletet természetesen kiterjeszthetjük tetszőleges számú menetre, és ilyenkor az adatokat többször olvassuk be a memóriába. Mi most mégis a kétmenetes algoritmusokra fogunk koncentrálni, mégpedig a következők miatt:

a) Két menet rendszerint elegendő, még a nagyon nagy relációk esetén is.
b) A kettőnél több menetre történő általánosítás nem okoz nehézséget; az ilyen kiterjesztéseket a 6.9. részben tárgyaljuk majd.

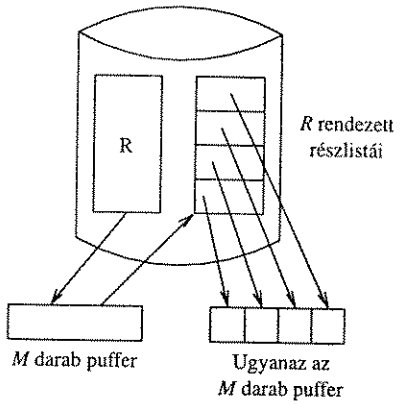
Ebben a részben a rendezést fogjuk vizsgálni, mint a relációs műveletek végrehajtására szolgáló eszközt. Az alapötlet a következő. Tegyük fel, hogy van egy nagy relációnk, R, amelyre $B(R)$ nagyobb M-nél, vagyis a rendelkezésre álló memóriapufferek számánál. Ekkor egymás után többször ismételve a következőket hajthatjuk végre:

1. Beolvassuk R-nek M darab blokkját a memóriába.
2. A memóriában lévő M blokkot rendezzük valamilyen hatékony rendező algoritmus segítségével. Egy ilyen algoritmus által igénybe vett processzoridő a lineárisnál alig meredekebben nő a memóriában lévő sorok számának függvényében, vagyis azt várhatjuk, hogy a rendezési idő nem fogja meghaladni az 1. lépésbeli lemez I/O-műveletekhez szükséges időt.
3. A rendezett listát kiírjuk M darab lemezblokkba. A blokkok tartalmára úgy fogunk hivatkozni, mint R *rendezett részlistáinak* egyike.

Az összes bemutatásra kerülő algoritmus ezt követően egy második menetet használ a rendezett részlisták valamilyen módon történő összefésülésére, hogy a kívánt műveletet végrehajtsa.

6.5.1. Ismétlődések kiküszöbölése rendezés segítségével

A $\delta(R)$ művelet két menetben való végrehajtásához R sorait a fent leírtak szerint részlistákba rendezzük. Ezt követően a rendelkezésre álló memóriát arra használjuk fel, hogy minden egyes rendezett részlistáról beletegyünk egy blokkot, ahogyan azt a



6.15. ábra. *Kétmenetes algoritmus az ismétlődések kiküszöbölésére*

2.3.4. részben a többutas összefésüléses rendezésnél is tettük. Most azonban a részlisták sorainak rendezése helyett a sorok egy-egy példányát mindig kitesszük a kimenetbe, és az összes többi vele azonos sort figyelmen kívül hagyjuk. A folyamat vázlatát a 6.15. ábra mutatja.

Kicsit részletesebben leírva úgy történik a dolog, hogy vesszük az egyes blokkok első, még nem vizsgált sorát, és megkeressük közöttük a rendezés szerinti elsőt; jelölje ezt mondjuk t. Ezt a sort egyszer kimásoljuk a kimenetbe, majd az összes blokk elejéről eltávolítjuk t összes példányát. Ha egy blokk kiürül, akkor pufferébe behozzuk ugyanazon részlista következő blokkját, és ha abban a blokkban is szerepel t, akkor azt is eltávolítjuk.

6.15. példa: Az egyszerűség kedvéért tegyük fel, hogy a sorok maguk egész számok, és egy blokkba mindössze két sor fér be. Legyen még $M = 3$, vagyis az elsődleges memóriában három blokk található. Az R reláció 17 sorból áll:

2, 5, 2, 1, 2, 2, 4, 5, 4, 3, 4, 2, 1, 5, 2, 1, 3.

Az első hat sort beolvassuk a memória három blokkjába, rendezzük őket, majd az R_1 részlista formájában kiírjuk őket. Hasonlóan a 7-től a 12-ig levő sorokat is beolvassuk, rendezzük, majd az R_2 részlistába kiírjuk. Végül az utolsó öt sort is ugyanígy rendezzük, aminek eredménye az R_3 részlista.

A második menet elindításához behozzuk a memóriába a három részlista mindegyikének első blokkját (két sorát). A helyzet ekkor a következő:

Részlista	*Memóriában*	*Lemezen vár*
R_1:	1 2	2 2, 2 5
R_2:	2 3	4 4, 4 5
R_3:	1 1	2 3, 5

A memóriában lévő három blokk első soraira pillantva azt látjuk, hogy a rendezés szerint az első az 1. Ennek megfelelően az 1-et egyszer bemásoljuk a kimenetbe, majd az összes 1-et eltávolítjuk a memória blokkjaiból. Miután ezt megtettük, R_3 blokkja kiürül, így behozhatjuk a következő blokkot – a 2 és a 3 sorokkal – ugyanarról a részlistáról. Ha ebben a blokkban több 1-es is szerepelt volna, akkor azokat is eltávolítanánk. A helyzet most így fest:

Részlista	*Memóriában*	*Lemezen vár*
R_1:	2	2 2, 2 5
R_2:	2 3	4 4, 4 5
R_3:	2 3	5

Most a 2 a legkisebb sor a listák elején, és ez ráadásul minden listában szerepel. Ezért a 2 egy példányát kitesszük a kimenetbe, és az összes 2-es sort eltávolítjuk a memóriabeli blokkokból. Ezzel R_1 blokkja kiürül, és a memóriába bekerül annak a részlistának a következő blokkja. A blokkban vannak 2-es sorok, amelyek eltávolítása után az R_1 blokkja ismét kiürül. A memóriába ezekután bekerül a részlista harmadik blokkja, aminek 2-es sorát ismét eltávolítjuk. A kialakult helyzet az alábbi lesz:

Részlista	*Memóriában*	*Lemezen vár*
R_1:	5	
R_2:	3	4 4, 4 5
R_3:	3	5

Most a 3 lesz kiválasztva legkisebb sorként, egy másolatát kitesszük a kimenetbe, R_2 és R_3 blokkjait kiürítjük, majd újra beolvassuk a lemezről, ami után a helyzet az alábbi:

Részlista	*Memóriában*	*Lemezen vár*
R_1:	5	
R_2:	4 4	4 5
R_3:	5	

A példa befejezéseként a következő lépésben a 4-es sort választjuk ki, ami után elfogy az R_2 lista nagyobbik része is. Az utolsó lépésben minden lista egyetlen 5-ös sort tartalmaz, amit egyszer kiviszünk a kimenetbe, majd eltávolítjuk a bemeneti pufferekből. □

A fenti algoritmus által végrehajtott lemez I/O-műveletek száma, ismét csak elhanyagolva a kimenet kezelését, a következő:

1. $B(R)$ R minden egyes blokkjának beolvasásához, a rendezett részlisták létrehozásakor.
2. $B(R)$ az egyes rendezett részlisták lemezre írásához.
3. $B(R)$ a részlisták minden egyes blokkjának megfelelő időben való beolvasásához.

Ezek szerint az algoritmus teljes költsége $3B(R)$, a 6.3.2. részben ismertetett egymenetes algoritmus $B(R)$ költségével szemben.

Ugyanakkor az egymenetes algoritmushoz képest sokkal nagyobb állományokat tudunk kezelni a kétmenetes algoritmussal. Feltéve, hogy M darab memóriablokk áll rendelkezésre, egyenként éppen M darab blokkból álló részlistákat hozunk létre. A második menetben aztán minden egyes részlistának egy memóriablokkra van szüksége, így M-nél nem lehet több részlista, és ezek mindegyike M blokk hosszúságú. A kétmenetes algoritmus alkalmazásának feltétele tehát $B \leq M^2$, szemben az egymenetes algoritmus $B \leq M$ korlátjával. Ezt úgy is kifejezhetjük, hogy $\delta(R)$ kétmenetes algoritmussal történő kiszámításához $B(R)$ memóriablokk helyett csak $\sqrt{B(R)}$ blokkra van szükség.

6.5.2. Csoportosítás és összesítés rendezés segítségével

A $\gamma_L(R)$ kétmenetes algoritmusa egészen hasonló a 6.5.1. részben megismert $\delta(R)$-re vonatkozó algoritmushoz. Az alábbiakban ennek lépéseit összegezzük:

1. R sorait M blokkonként beolvassuk a memóriába. Minden M darab blokkot rendezünk, rendezési kulcsként az L csoportosító attribútumait használva. Az egyes rendezett részlistákat egyenként lemezre írjuk.
2. Minden egyes részlistához egy darab memóriablokkot használva, első lépésként az egyes részlisták első blokkját betöltjük a hozzá tartozó pufferbe.
3. Egymás után ismétlődően mindig újra megkeressük a rendezési kulcs (a csoportosító attribútumok) szerinti legkisebb értéket a pufferek sorra következő sorai között. Ez a v érték alkotja majd a következő csoportot, amelyre a következőket tesszük:
   a) Előkészítjük a csoport L listában szereplő összesítéseinek a kiszámítását. Csakúgy mint a 6.3.2. részben, most is a sorok számát és az értékek összegét tartjuk nyilván az átlag helyett.
   b) A v keresési kulccsal összehasonlítva megvizsgáljuk a sorok mindegyikét, és folyamatosan gyűjtjük a szükséges összesítéseket.
   c) Ha egy puffer kiürül, akkor beolvassuk a helyére ugyanannak a részlistának a következő blokkját.

Ha nincs több sor, amelyik a v keresési kulccsal rendelkezik, akkor kiírunk egy olyan sort a kimenetbe, amelyik az L csoportosító attribútumaiból, valamint a csoportra hozzájuk kiszámított összesítések értékeiből áll.

Csakúgy mint a δ-ra vonatkozó algoritmus, ez a kétmenetes γ algoritmus is $3B(R)$ lemez I/O-műveletet igényel, és használható mindaddig, amíg $B(R) \leq M^2$.

6.5.3. Az egyesítés egy rendezésen alapuló algoritmusa

Ha multihalmaz-egyesítésre van szükségünk, akkor a 6.3.3. rész egymenetes algoritmusa – ahol egyszerűen lemásoltuk a két relációt – az argumentumok méretétől függetlenül működik, így $\cup_B$-hez nincs szükség kétmenetes algoritmus használatára. Ugyanakkor $\cup_S$ egymenetes algoritmusa csak akkor működik, ha legalább az egyik reláció kisebb, mint a rendelkezésre álló memória, így a halmazegyesítéshez kétmenetes algoritmusra is szükség lehet. Az általunk most bemutatott módszer a metszet és a különbség műveletek halmaz- és multihalmaz-változataira is működik, ahogyan azt a 6.5.4. részben majd látni fogjuk. $R \cup_S S$ kiszámításához a következőket kell tennünk:

1. R sorait M darab blokkonként beolvassuk a memóriába, a sorokat rendezzük, majd a kapott rendezett részlistákat visszaírjuk a lemezre.
2. Ugyanezt elvégezzük S-re az S reláció rendezett részlistáinak létrehozásához.
3. R és S minden egyes részlistájához veszünk egy memóriapuffert, majd a megfelelő részlista első blokkját oda betöltjük.

4. Újra és újra megkeressük a pufferekben az első még ott lévő t sort. Bemásoljuk t-t a kimenetbe, majd t összes előfordulását eltávolítjuk a pufferekből (ha R és S halmazok, akkor legfeljebb két ilyen előfordulás lehet). Ha egy puffer kiürül, akkor azt feltöltjük a megfelelő részlista következő blokkjával.

Megfigyelhetjük, hogy R és S minden sorát kétszer olvassuk be a memóriába: egyszer, amikor a részlistákat hozzuk létre, és másodszor valamelyik részlista részeként. A sort egyszer egy újonnan létrehozott részlista részeként a lemezre is kiírjuk. A lemez I/O-művelet költsége így $3(B(R) + B(S))$.

Az algoritmus addig működik, amíg a két reláción belüli részlisták együttes száma nem haladja meg M-et, mert minden egyes részlistához egy pufferre van szükségünk. Mivel minden részlista hossza M blokk, a két reláció mérete nem lehet nagyobb M^2-nél, azaz $B(R) + B(S) \leq M^2$.

6.5.4. A metszet és a különbség rendezésen alapuló algoritmusai

Akár a halmaz-, akár a multihalmaz-változatra van szükségünk, az algoritmusok lényegében azonosak a 6.5.3. részben tárgyalttal, csupán abban van különbség, ahogyan a rendezett részlisták elején álló t sor példányait kezeljük. Az eddigiekhez hasonlóan létrehozzuk az R és S argumentum relációkra az M blokkból álló rendezett részlistákat. Ezután minden részlistához egy memóriapuffert használunk, amelyet kezdetben a részlista első blokkjával töltünk fel.

Ezután újra meg újra megvizsgáljuk az összes pufferben maradó sor közül a legkisebb t sort. Meghatározzuk R összes t-vel azonos sorának a számát, majd ugyanezt megtesszük S-re is. Ehhez ismét az szükséges, hogy a puffereket újra feltöltsük arról a részlistáról, amelynek aktuálisan pufferelt blokkja kiürült. A következőkben azt adjuk meg, hogy hogyan döntjük el, hogy t a kimenetbe kerüljön-e, és ha igen, hányszor:

- Ha a művelet a halmazmetszet, úgy t-t akkor írjuk ki, ha R-ben és S-ben is előfordul.
- Ha a művelet a multihalmazmetszet, úgy t-t annyiszor tesszük a kimenetbe, amennyi az R-beli és S-beli előfordulásainak a minimuma. Vegyük észre, hogy t-t nem írjuk a kimenetbe, ha ezen számosságok bármelyike nulla, azaz, ha t nem szerepel mindkét relációban.
- Ha a művelet a halmazkülönbség, $R -_S S$, akkor t-t csak akkor tesszük a kimenetbe, ha R-ben előfordul, de S-ben nem.
- Ha a művelet a multihalmaz-különbség, $R -_B S$, akkor t-t annyiszor írjuk a kimenetbe, ahányszor előfordul R-ben mínusz ahányszor előfordul S-ben. Természetesen ha t S-ben legalább annyiszor fordul elő mint R-ben, akkor nem kerül a kimenetbe.

6.16. példa: Használjuk ugyanazokat a feltételezéseket mint a 6.15. példában: Legyen $M = 3$, a sorok egészek, és egy blokkba két sor fér be. Az adatok majdnem azonosak lesznek az említett péda adataival. Itt azonban két argumentumra van szükségünk, ezért feltesszük, hogy R-nek 12, S-nek pedig 5 sora van. Mivel a memóriába 6 sor fér

be, az első menetben R-ből két részlistát kapunk, nevezzük ezeket R_1-nek és R_2-nek, S-ből pedig egyetlen rendezett részlista lesz, legyen ez S_1.[2] A rendezett részlisták létrehozását követően (amelyeket a 6.15. példához hasonlóan hoztunk létre a rendezetlen adatokból) az alábbi lesz a helyzet:

Részlista	*Memóriában*	*Lemezen vár*
R_1:	1 2	2 2, 2 5
R_2:	2 3	4 4, 4 5
S_1:	1 1	2 3, 5

Tegyük fel, hogy az $R\_{B}S$ multihalmaz-különbséget szeretnénk kiszámítani. Azt találjuk, hogy a memóriapufferekben található legkisebb sor az 1, ezért megnézzük, hogy az 1 hányszor fordul elő az R, illetve az S részlistáiban. Azt látjuk, hogy az 1 R-ben egyszer, S-ben pedig kétszer szerepel. Mivel az R-beli előfordulások száma nem nagyobb az S-belinél, az 1-es sort nem másoljuk a kimenetbe. Mivel az 1 számolásakor S_1 első blokkja kiürült, betöltjük S_1 következő blokkját, ami a következőt szituációhoz vezet:

Részlista	*Memóriában*	*Lemezen vár*
R_1:	2	2 2, 2 5
R_2:	2 3	4 4, 4 5
S_1:	2 3	5

Most azt látjuk, hogy a 2-es a megmaradóak között a legkisebb sor, megszámoljuk tehát, hogy R-ben hányszor szerepel – ötször –, majd ugyanezt megtesszük S-re is, ahol egyszer találjuk meg. A 2-es sort tehát négyszer írjuk a kimenetbe. A számolások elvégzése során kétszer kell újratöltenünk R_1 pufferét, ami után a helyzet a következő:

Részlista	*Memóriában*	*Lemezen vár*
R_1:	5	
R_2:	3	4 4, 4 5
S_1:	3	5

Ezután foglalkozzunk a 3-as sorral. Azt találjuk, hogy mind R-ben, mind S-ben egyszer fordul elő. Ezért 3 nem kerül a kimenetbe. Példányainak a pufferekből való eltávolítása után ez lesz a helyzet:

Részlista	*Memóriában*	*Lemezen vár*
R_1:	5	
R_2:	4 4	4 5
S_1:	5	

[2] Mivel S befér a memóriába, használhatnánk éppenséggel a 6.3.3. rész egymenetes algoritmusát is. A kétmenetes megközelítés használata csupán illusztrálásra szolgál.

A 4-es sor R-ben háromszor fordul elő, S-ben viszont egyszer sem, ezért a 4-et háromszor írjuk a kimenetbe. Végül 5 kétszer szerepel R-ben, S-ben pedig egyszer, ezért egyszer tesszük a kimenetbe. A teljes kimenet ezek után: 2, 2, 2, 2, 4, 4, 4, 5. □

Az algoritmusok ezen családjának elemzése megegyezik a halmazegyesítés algoritmusra a 6.5.3. részben leírtakkal:

- $3(B(R) + B(S))$ lemez I/O-művelet szükséges.
- Körülbelül $B(R) + B(S) \leq M^2$ az algoritmus működésének korlátja.

6.5.5. Egy egyszerű rendezésen alapuló összekapcsolási algoritmus

A rendezést többféle módon lehet nagy relációk összekapcsolására használni. Mielőtt rátérnénk az összekapcsolási algoritmusok vizsgálatára, hadd jegyezzük meg, hogy összekapcsolások számításakor felmerülhet egy olyan probléma, amely az eddig vizsgált bináris műveletekkel kapcsolatban nem okozott gondot. Egy összekapcsolás alkalmával a két reláció azon sorainak a száma, amelyek az összekapcsolás alapjául szolgáló attribútumokon megegyeznek, és ezért egyidejűleg kell hogy a memóriában legyenek, meghaladhatja a memóriába beférő mennyiséget. A szélsőséges példa talán az lehet, amikor az összekapcsolás alapjául szolgáló attribútum(ok)nak csupán egyetlen értéke van, és így az egyik reláció minden sora összekapcsolódik a másik reláció minden sorával. Ebben az esetben nincs más választásunk, mint hogy vegyük az azonos attribútumértékekkel bíró két sorhalmaznak egy beágyazott cikusú összekapcsolását.

Annak érdekében, hogy ezt az eshetőséget elkerüljük, megpróbálhatjuk az algoritmus más célra felhasznált memóriaigényét csökkenteni, és ezáltal több puffert tudunk elérhetővé tenni az összekapcsolódó sorok befogadására. Ebben a szakaszban azt az algoritmust mutatjuk be, amely a lehető legtöbb puffert teszi elérhetővé a közös értékkel rendelkező sorok tárolására. A 6.5.7. részben megnézünk majd egy másik rendezésen alapuló algoritmust is, amely ugyan kevesebb lemez I/O-műveletet használ, de problémákba ütközhet akkor, ha nagyszámú olyan sor van, amelyek az összekapcsolás attribútumain megegyeznek.

Tegyük fel, hogy az $R(X, Y)$ és az $S(Y, Z)$ relációkat szeretnénk összekapcsolni, és ehhez M darab memóriablokk áll rendelkezésünkre. Ekkor a következőket tesszük:

1. Rendezzük R-et egy kétfázisú többutas összefésüléssel, amelyben Y a rendezési kulcs.
2. S-et hasonló módon rendezzük.
3. Összefésüljük a rendezett R és S relációkat. Ehhez általában csak két puffert használunk, egyet R és egyet S éppen aktuális blokkjára. Az alábbi lépéseket többször megismételjük:

   a) Megkeressük az Y összekapcsolási attribútumoknak azt a legkisebb y értékét, amely éppen az R és S blokkok elején található.

b) Ha y nem jelenik meg a másik reláció elején, akkor az y rendezési kulcsú sor(oka)t eltávolítjuk.
c) Egyébként azonosítjuk mindkét relációban az összes y rendezési kulcsú sort. Ha szükséges, addig olvassuk be a rendezett R és S blokkjait, amíg biztosak nem leszünk benne, hogy már egyik relációban sincs y értékű sor. Erre a célra összesen M puffert használhatunk fel.
d) A kimenetbe írjuk az összes olyan sort, amely R és S közös Y értékkel – jelen esetben éppen y-nal – rendelkező sorainak összekapcsolásával kialakítható.
e) Ha bármelyik relációban már nincs több megvizsgálatlan sor a memóriában, akkor annak pufferét újra feltöltjük.

6.17. példa: Vegyük ismét a 6.14. példa R és S relációit. Emlékeztetünk rá, hogy a relációk 1000, illetve 500 blokkot foglalnak el, és összesen $M = 101$ memóriapufferünk van. Ha egy relációra kétfázisú többutas összefésüléses rendezést végzünk, akkor minden blokkra négy lemez I/O-műveletet végzünk, mindkét fázisban kettőt-kettőt. R és S rendezéséhez tehát $4(B(R) + B(S))$ lemez I/O-művelet szükséges, ami esetünkben éppen 6000.

Amikor az összekapcsolódó sorok megkereséséhez összefésüljük a rendezett R és S relációkat, akkor R és S minden egyes blokkját még egyszer beolvassuk (ez már az ötödik I/O), ami további 1500 lemez I/O-műveletet jelent. Az összefésülés során általában mindössze kettőt használunk a 101 memóriablokkból. Ugyanakkor ha szükség van rá azt is megtehetjük, hogy R és S közös Y értékkel – ez esetben y-nal – rendelkező sorainak befogadására felhasználjuk mind a 101 blokkot. Így elegendő az a feltétel, hogy R és S közös Y értékkel bíró sorai semmilyen y esetén se foglaljanak el összesen 101 blokknál többet.

Vegyük észre, hogy ebben az algoritmusban az összes végrehajtott lemez I/O-műveletek száma 7500, szemben a 6.14. példában szereplő beágyazott ciklusú összekapcsolás 5500 műveletével. Tudjuk azonban, hogy a beágyazott ciklusú algoritmus természeténél fogva négyzetes, így futási ideje $B(R)B(S)$-sel arányos, míg a rendezéses összekapcsolás I/O-művelet költsége lineáris, vagyis a futási ideje $(B(R) + B(S))$-sel arányos. Csupán a konstans tényezők értéke és a példa kis mérete (az egyes relációk csak 5, illetve 10-szer nagyobbak annál, mint ami még beférne a memóriapufferekbe) okozza, hogy a beágyazott ciklusú összekapcsolás előnyösebb. Sőt, a 6.5.7. részben látni fogjuk, hogy a rendezéses összekapcsolást általában lehetséges $3(B(R) + B(S))$ lemez I/O-művelettel elvégezni, ami esetünkben 4500-at jelenene, ami már alatta van a beágyazott ciklus költségének. □

Ha van olyan y Y érték, amelyre az ezzel az értékkel rendelkező sorok nem férnek be M pufferbe, akkor az előző algoritmust módosítanunk kell.

1. Ha az egyik reláció, legyen ez mondjuk R, y Y értékkel rendelkező sorai beférnek $M - 1$ darab pufferbe, akkor olvassuk be R ezen blokkjait pufferekbe, majd egyenként olvassuk be S y értékű sorait a fennmaradó pufferbe. Valójában ilyenkor a 6.3.3. rész egymenetes összekapcsolását végezzük el azokra a sorokra, amelyek Y értéke éppen y.

2. Ha mindkét relációnak több y Y értékű sora van annál, hogy azok $M - 1$ pufferbe beférjenek, akkor használjuk fel az M puffert, és hajtsunk végre egy beágyazott ciklusú összekapcsolást a két reláció y Y értékű sorain.

Vegyük észre, hogy mindkét esetben előfordulhat, hogy az egyik reláció sorait beolvassuk, majd figyelmen kívül hagyjuk őket, és így később újra be kell olvasnunk azokat. Például az 1. esetben először lehet, hogy S azon sorainak beolvasásával próbálkozunk, amelyeknek Y értéke y, és azt találjuk, hogy azok nem férnek be $M - 1$ pufferbe. Majd ezután megpróbáljuk R-nek ugyanezen Y értékű sorait beolvasni, és ezek a sorok már befférnek az $M - 1$ pufferbe.

6.5.6. Az egyszerű rendezéses összekapcsolás elemzése

Amint azt a 6.17. példában megfigyeltük, algoritmusunk az argumentum reláció minden blokkjára öt lemez I/O-műveletet végez. Kivételt képezne az az eset, ha olyan sok sor lenne azonos Y értékkel, hogy a szóban forgó sorokat valamilyen speciális módon kellene összekapcsolnunk. Ebben az esetben a további lemez I/O-műveletek száma attól függ, hogy csak az egyik avagy mindkét reláció olyan sok azonos Y értékű sorral rendelkezike, hogy azok maguk már $M - 1$-nél több puffert igényelnek. Az összes esetet itt nem vesszük végig részletesen; a feladatokban megadunk néhány kidolgozásra szánt példát.

Azt is meg kell vizsgálnunk, hogy mekkorának kell M-nek lennie ahhoz, hogy az egyszerű rendezéses összekapcsolás működjön. Az elsődleges korlát az, hogy végre kell tudnunk hajtani R-en és S-en a kétfázisú, többutas összefésüléses rendezéseket. Ahogyan ezt a 2.3.4. részben már megvizsgáltuk, ezeknek a rendezéseknek az elvégzéséhez az szükséges, hogy $B(R) \leq M^2$ és $B(S) \leq M^2$ teljesüljön. Ha ezzel készen vagyunk, akkor már nem fogunk kifogyni a pufferekből, noha – amint már említettük – esetleg el kell majd térnünk az egyszerű összefésüléstől, ha az azonos Y értékkel rendelkező sorok nem férnek be M pufferbe. Összefoglalva, ha ilyen bonyodalmak nem merülnek fel, akkor:

- Az egyszerű rendezéses összekapcsolás $5(B(R) + B(S))$ lemez I/O-műveletet használ.
- Működéséhez $B(R) \leq M^2$ és $B(S) \leq M^2$ teljesülése szükséges.

6.5.7. Egy hatékonyabb rendezésen alapuló összekapcsolás

Ha nem kell aggódnunk az összekapcsolási attribútumokon azonos értékkel bíró sorok igen nagy száma miatt, akkor blokkonként 2 lemez I/O-műveletet megtakaríthatunk azáltal, hogy a rendezések második fázisát kombináljuk magával az összekapcsolással. Az ilyen algoritmusokat egyszerűen rendezéses összekapcsolásnak hívjuk. További ismert elnevezések még az „összefésüléses összekapcsolás" és a „rendezéses-összefésüléses összekapcsolás". Az $R(X, Y) \bowtie S(Y, Z)$ összekapcsolást M darab memóriablokkot használva a következőképpen számíthatjuk ki:

1. Y-t rendezési kulcsként használva mind R-re, mind S-re M méretű rendezett részlistákat hozunk létre.
2. Az egyes részlisták első blokkjait behozzuk egy-egy pufferbe, ehhez feltesszük, hogy összesen nincs M-nél több részlista.
3. A részlisták soron következő sorai között újra meg újra megkeressük a legkisebb Y értéket, y-t. Mindkét reláció sorai között beazonosítjuk az y értékkel rendelkezőket, ehhez esetleg betesszük azokat az M szabad puffer némelyikébe, feltéve, hogy M-nél kevesebb részlista van. A kimenetbe tesszük az összes olyan R-beli és S-beli sorok összekapcsolását, amelyek az Y attribútum(ok)on y értékkel rendelkeznek. Ha közben bármelyik részlista puffere kiürül, akkor azt lemezről ismét feltöltjük.

6.18. példa: Tekintsük ismét a 6.14. példabeli feladatot: 101 puffer használatával szeretnénk összekapcsolni az egyenként 1000, illetve 500 blokkos R és S relációkat. R-et és S-et felosztjuk 10, illetve 5 darab, egyenként 100 blokk hosszúságú részlistára, majd ezeket rendezzük.[3] Ezután 15 puffert a részlisták aktuális blokkjainak a befogadására használunk. Ha olyan helyzettel kerülünk szembe, ahol sok sornak van azonos Y értéke, ott a fennmaradó 86 puffert használhatjuk ezeknek a soroknak a tárolására. Ha még ennél is több ilyen sor van, akkor valamilyen speciális algoritmust kell használnunk, mint amilyen pl. a 6.5.5. rész végén szerepelt.

Feltéve, hogy az algoritmust nem kell módosítanunk a sok azonos Y értékű sor miatt, adatblokkonként három lemez I/O-műveletet kell végeznünk. Ezek közül kettő a rendezett részlisták létrehozására szolgál. Ezt követően az egyes rendezett részlisták minden egyes blokkját még egyszer beolvassuk a memóriába a többutas összefésülési folyamatban. A lemez I/O-műveletek teljes száma így tehát 4500. □

A fenti rendezéses összekapcsolási algoritmus – amennyiben alkalmazható – hatékonyabb a 6.5.5. rész algoritmusánál. A 6.18. példa kapcsán megjegyeztük, hogy a lemez I/O-műveletek száma $3(B(R) + B(S))$. Az algoritmust használhatjuk olyan adatokon, amelyek mérete megközelíti az előző algoritmusét. A rendezett részlisták mérete M blokk, és összesen legfeljebb M darab lehet belőlük. A $B(R) + B(S) \leq M^2$ korlát ennél fogva elégséges.

Elgondolkozhatunk rajta, hogy esetleg elkerülhetők-e mindazok a bonyodalmak, amelyek a nagyszámú azonos Y értékkel rendelkező sor esetén merülnek fel. A következőket érdemes számításba venni:

1. Néha biztosak lehetünk benne, hogy a probléma fel sem merül. Ha például R-nek Y egy kulcsa, akkor egy adott y Y érték R részlistáinak összes blokkjai között csak egyszer fordulhat elő. Amikor éppen y van soron, akkor az R-beli sort a helyén hagyhatjuk, és összekapcsolhatjuk azt S összes megfelelő sorával. Ha a folyamat során S részlistáinak blokkjai kiürülnek, úgy puffereik feltölthetők a következő blokkal. Ekkor egyáltalán nincs szükség pluszhelyre, R és S akárhány sora rendelkezik is az adott y Y értékkel. Ha Y R helyett S-nek kulcsa, akkor a gondolatmenet megismételhető R és S felcserélésével.
2. Ha $B(R) + B(S)$ sokkal kisebb M^2-nél, akkor az azonos Y értékkel rendelkező sorok számára rengeteg kihasználatlan pufferünk lesz, mint azt a 6.18. példa is jelezte.
3. Ha máshogy semmiképpen nem boldogulunk, akkor használhatunk egy beágyazott ciklusú összekapcsolást, leszűkítve az azonos Y értékkel rendelkező sorokra, ami ugyan plusz lemez I/O-műveleteket igényel, de a feladatot kifogástalanul elvégzi. Ezt a lehetőséget a 6.5.5. részben tárgyaltuk.

[3] Technikailag úgy is elrendezhettük volna a részlistákat, hogy mindegyiknek 101 blokk legyen a hossza, továbbá R és S utolsó részlistája 91, illetve 96 blokk hosszú legyen, de a költség ebben az esetben is pontosan ugyanannyi lenne.

6.5.8. A rendezésen alapuló algoritmusok összefoglalása

A 6.16. ábra a 6.5. részben ismertetett algoritmusok összefoglaló táblázatát mutatja. A 6.5.5. és a 6.5.7. részekben elmondottak értelmében akkor van szükség az idő- és a memóriakövetelmények módosítására, ha két olyan relációt kapcsolunk össze, amelyek nagyszámú sorára nézve az összekapcsolási attribútumok azonos értéket vesznek fel.

Operátor	*Szükséges* M *kb.*	*Lemez I/O-művelet*	*Rész*
γ, δ	$\sqrt{B}$	$3B$	6.5.1., 6.5.2.
$\cup, \cap, -$	$\sqrt{B(R+B(S))}$	$3(B(R) + B(S))$	6.5.3., 6.5.4.
$\bowtie$	$\sqrt{\max(B(R), B(S))}$	$5(B(R) + B(S))$	6.5.5.
$\bowtie$	$\sqrt{(B(R)+B(S))}$	$3(B(R) + B(S))$	6.5.7.

6.16. ábra. *A rendezés alapú algoritmusok memória- és lemez I/O-művelet követelményei*

6.5.9. Feladatok

6.5.1. feladat: A 6.15. példa feltevéseivel élve (blokkonként két sor stb.).

a) Mutassuk be az ismétlődések kiküszöbölésére vonatkozó kétmenetes algoritmus működését arra a harminc, egykomponensű sorból álló sorozatra, amelyben a 0, 1, 2, 3, 4 sorozat hatszor ismétlődik meg.

b) Mutassuk meg a csoportosítás kétmenetes algoritmusának működését a $\gamma_{aAVG(b)}(R)$ algoritmus kiszámításán keresztül. Az $R(a, b)$ reláció a harminc darab t_0–t_{29} sorból áll, a t_i sor csoportosító a komponense i mod 5, második b komponense pedig i.

6.5.2. feladat: Az alább felsorolt műveletek mindegyikére adjunk meg egy olyan iterátort, amely az ebben a fejezetben leírt algoritmust használja.

* a) Ismétlődések kiküszöbölése (δ).
b) Csoportosítás (γ_L).
* c) Halmazmetszet.
d) Multihalmaz-különbség.
e) Természetes összekapcsolás.

6.5.3. feladat: Ha $B(R) = B(S) = 10\,000$ és $M = 1000$, akkor mik lesznek a lemez I/O-művelet igényei a következő műveleteknek:

a) Halmazegyesítés.
* b) Egyszerű rendezéses összekapcsolás.
c) A 6.5.7. rész hatékonyabb rendezéses összekapcsolása.

! **6.5.4. feladat:** Tegyük fel, hogy egy, a mostani fejezetben tárgyalt algoritmus második menetének nincs szüksége az összes M pufferre, mert csak M-nél kevesebb részlista létezik. A feleslegessé váló pufferek kihasználásával hogyan tudunk lemez I/O-műveleteket megtakarítani?

! **6.5.5. feladat:** A 6.17. példával kapcsolatban megbeszéltük az R 1000 és az S 500 blokkból álló relációk összekapcsolását az $M = 101$ esetben. Arra is rámutattunk, hogy további lemez I/O-műveletekre is szükség lenne akkor, ha egy adott értékre annyi sor lenne, hogy egyik reláció sorai sem férnének be a memóriába. Számítsuk ki a szükséges lemez I/O-műveletek számát akkor, ha

* a) Csak két Y érték van, amelyek mindegyike R és S sorainak a felében fordul elő (emlékezzünk rá, hogy Y az összekapcsolás attribútumait jelölte).
b) Öt Y érték van, amelyek mindegyike egyformán valószínű mindkét relációban.
c) 10 darab Y érték van, amelyek mindegyike egyformán valószínű mindkét relációban.

! **6.5.6. feladat:** Ismételjük meg a 6.5.5. feladatot a 6.5.7. rész hatékonyabb rendezéses összekapcsolására.

6.5.7. feladat: Mennyi memóriára van szükségünk egy kétmenetes, rendezésen alapuló algoritmushoz, egyenként 10 000 blokkból álló relációk esetén, ha a művelet:

* a) δ.
b) γ.
c) Egy bináris művelet, mondjuk az összekapcsolás vagy az egyesítés.

6.5.8. feladat: Írjunk le egy kétmenetes rendezésen alapuló algoritmust a 6.1.3. feladat mind az öt összekapcsolásszerű operátorára.

! **6.5.9. feladat:** Tegyük fel, hogy a rekordok nagyobbak lehetnek a blokkoknál, azaz létezhetnek ún. átnyúló rekordok. Hogyan változnának meg ekkor a kétmenetes rendezéses algoritmusok memóriaigényei?

!! **6.5.10. feladat:** Előfordulhat, hogy megtakaríthatunk néhány lemez I/O-műveletet akkor, ha az utolsó részlistát a memóriában hagyjuk. Még annak is lehet értelme, hogy érdemes megpróbálni ebből hasznot húzni úgy, hogy M-nél kevesebb blokkot tartalmazó részlistákat használunk. Vajon hány lemez I/O-művelet takarítható meg így?

!! **6.5.11. feladat:** Az OQL mindig lehetővé teszi az objektumok tetszőleges, a felhasználó által megadott függvények szerinti csoportosítását. Sorokat csoportosíthatnánk például két attribútum összege szerint. Hogyan hajtanánk végre objektumok egy halmazán egy ilyen rendezésen alapuló csoportosítási műveletet?

6.6. Tördelésen alapuló kétmenetes algoritmusok

A tördelésen alapuló algoritmusok családja szintén a 6.5. részben bemutatott problémákat igyekszik megoldani. Az ilyen típusú algoritmusok mögött meghúzódó alapötlet a következő: ha az adatok mennyisége túl nagy ahhoz, hogy azokat az elsődleges memóriapuffereiben tároljuk, akkor végezzünk tördelést az argumentum(ok) összes sorára egy megfelelő tördelőkulcs segítségével. Az összes szokásos műveletre kiválasztható a tördelőkulcs oly módon, hogy a művelet végrehajtásakor együtt tekintendő soroknak ugyanaz legyen a tördelési értéke.

Ezt követően úgy végezzük el a műveletet, hogy egyszerre csak egy kosárral dolgozunk (illetve bináris műveletek esetén az azonos tördelési értékkel rendelkező kosárpárokon dolgozunk). Ezzel elérhetjük azt, hogy az operandus(ok) méretét a kosarak számával arányos mértékben csökkentjük. Ha a rendelkezésre álló pufferek száma M, akkor választhatjuk M-et a kosarak számának, és ezáltal egy M-es szorzóval növelhetjük az általunk kezelhető relációk méretét. Vegyük észre, hogy a 6.5. rész rendezésen alapuló algoritmusai az előzetes feldolgozással szintén egy M-es szorzót nyernek, viszont a rendezéses és a tördeléses megközelítések a hasonló mértékű megtakarítást egészen másféle módon érik el.

6.6.1. Relációk particionálása tördeléssel

Kezdjük a vizsgálódást azzal, hogy áttekintjük, miként particionálnánk az R relációt $M - 1$ darab nagyjából egyforma méretű kosárba, M puffer használatával. Feltesszük, hogy a h tördelőfüggvény argumentumait az R teljes sorai alkotják (azaz R összes attribútuma a tördelőkulcs részét képezi). Minden kosárhoz hozzárendelünk egy puffert. Az utolsó puffer fogadja az R blokkjait, egyszerre csak egyet. A blokk minden egyes t sorát a $h(t)$ kosárba tördeljük, majd bemásoljuk a megfelelő pufferbe. Ha a szóban forgó puffer már tele van, akkor az eredményt kiírjuk lemezre, és ugyanahhoz a kosárhoz egy másik blokkot inicializálunk. Végül minden egyes blokk utolsó kosarát is kiírjuk lemezre, ha az adott kosár nem üres. Az algoritmus további részleteit a 6.17. ábrán láthatjuk. Vegyük észre, hogy noha a sorok változó hosszúságúak lehetnek, az algoritmus azt feltételezi, hogy azok mindig befférnek egy üres pufferbe.

```
inicializáljunk M-1 kosarat M-1 üres puffer felhasználásával;
FOR az R minden egyes b blokkjára DO BEGIN
   olvassuk be a b blokkot az M-edik pufferbe;
   FOR a b blokk minden egyes t sorára DO BEGIN
      IF a h(t) kosárban nincs hely a t számára THEN
         BEGIN
            másoljuk ki a puffert lemezre;
            inicializáljunk egy új üres blokkot a pufferben;
         END;
      másoljuk a t sort h(t) kosárhoz tartozó pufferbe;
   END;
END;
FOR minden egyes kosárra DO
   IF az adott kosárhoz tartozó puffer nem üres THEN
      írjuk ki lemezre az adott puffert;
```

6.17. ábra. *Az R reláció particionálása M – 1 kosárba.*

6.6.2. Egy tördelésen alapuló algoritmus az ismétlődések kiküszöbölésére

A továbbiakban a relációs algebra olyan műveleteinek tördelésen alapuló algoritmusait fogjuk megvizsgálni, amelyek kétmenetes algoritmusokat igényelhetnek. Foglalkozzunk először az ismétlődések kiküszöbölésével, vagyis a $\delta(R)$ művelettel. Az R relációt $M-1$ kosárba tördeljük, amint az a 6.17. ábrán látható. Figyeljük meg, hogy az egyforma t sorok ugyanabba a kosárba kerülnek. A δ művelet így rendelkezik a következő, számunkra lényeges tulajdonsággal: a kosarakat vizsgálhatjuk egyenként, végrehajthatjuk δ-t egyenként az éppen aktuális kosárra, majd válaszként képezhetjük a $\delta(R_i)$-k egyesítését, ahol R_i az R reláció azon része, amelyik tördeléskor az i-edik kosárba kerül. A 6.3.2. rész egymenetes algoritmusa segítségével minden egyes R_i-ből kiküszöbölhetjük az ismétlődéseket, majd a kapott egyedi sorokat kiírjuk a kimenetre.

A módszer mindaddig működik, amíg az R_i-k egyenként elég kicsik ahhoz, hogy beférjenek a memóriába, és így lehetővé tegyék egymenetes algoritmus használatát. Minthogy azt feltételeztük, hogy a h tördelőfüggvény az R relációt egyforma méretű kosarakba tördeli, valamennyi R_i mérete hozzávetőleg $B(R)/(M-1)$ blokk lesz. Ha ez a szám nem nagyobb M-nél, azaz ha $B(R) \leq M(M-1)$, akkor a kétmenetes, tördelésen alapuló algoritmus működni fog. Amint azt a 6.3.2. részben megmutattuk, valójában csak annyi kell, hogy az egy kosárban található különböző sorok beférjenek M darab pufferbe, viszont abban nem lehetünk biztosak, hogy vannak-e egyáltalán ismétlődések. Így a $B(R) \leq M^2$ becslés, ahol M-et és $M-1$-et egyszerűen azonosnak vettük, megegyezik a δ művelet rendezésen alapuló, kétmenetes algoritmusánál adott becsléssel.

A lemez I/O-műveletek száma ugyancsak hasonló a rendezésen alapuló algoritmuséhoz. Az R minden blokkját egyszer olvassuk be a sorok tördelésénél, és minden kosár valamennyi blokkját kiírjuk lemezre. Ezt követően az egyes kosarak blokkjait ismételten beolvassuk annál az egymenetes algoritmusnál, amely az adott kosarat dolgozza fel. A lemez I/O-műveletek teljes száma tehát $3B(R)$.

6.6.3. Egy tördelésen alapuló algoritmus a csoportosításra és az összesítésre

A $\gamma_L(R)$ művelet végrehajtásához megint csak úgy kezdünk hozzá, hogy R összes sorát $M-1$ kosárba tördeljük. Most azonban ahhoz, hogy ugyanazon csoport összes sora ugyanabba a kosárba kerüljön, olyan tördelőfüggvényt kell választanunk, amely kizárólag az L lista csoportosító attribútumaitól függ.

Ha az R relációt kosarakba particionáltuk, akkor minden egyes kosárra külön-külön használhatjuk a γ művelet 6.3.2. részben megismert egymenetes algoritmusát. A 6.6.2. részben a δ kapcsán megbeszéltük, hogy az egyes kosarakat akkor tudjuk feldolgozni a memóriában, ha $B(R) \leq M^2$.

Ugyanakkor a második menetben az egyes kosarak feldolgozásánál csoportonként csak egy rekordra van szükségünk. Így tehát a kosarat egy menetben tudjuk kezelni még akkor is, ha annak mérete meghaladja M-et, feltéve, hogy azok a rekordok, amik a kosárban lévő csoportoknak felelnek meg, összesen nem igényelnek M-nél több puffert. Egy csoportnak megfelelő rekord rendszerint nem nagyobb R egy soránál. Ha ez így van, akkor $B(R)$-re jobb felső korlátot ad M^2 szorozva a csoportonkénti sorok átlagos számával.

Ennek következményeként, ha kevés csoport van, akkor tulajdonképpen a $B(R) \leq$ $\leq M^2$ szabálynak megfelelő relációknál jóval nagyobb R relációt is képesek vagyunk kezelni. Másrészről viszont, ha M nagyobb a csoportok számánál, akkor nem tudjuk feltölteni az összes kosarat. Az R méretére tehát a tényleges korlátozás M-nek egy bonyolult függvénye; a $B(R) \leq M^2$ csak egy egyszerű becslés. Végül pedig figyeljük meg, hogy γ-ra a lemez I/O-műveletek száma δ-hoz hasonlóan $3B(R)$.

6.6.4. Az egyesítés, a metszet és a különbség tördelésen alapuló algoritmusai

Ha bináris műveletről van szó, akkor ügyelnünk kell arra, hogy mindkét argumentum sorainak tördeléséhez ugyanazt a tördelőfüggvényt használjuk. Az $R \cup_S S$ kiszámításánál például mind az R, mind az S relációt egyenként $M-1$ kosárba tördeljük, jelölje ezeket $R_1, R_2, \ldots, R_{M-1}$, illetve $S_1, S_2, \ldots, S_{M-1}$. Ezek után minden i-re vesszük R_i és S_i halmaz alapú egyesítését, és az eredményt kiírjuk a kimenetre. Vegyük észre, hogy ha egy sor R-ben és S-ben egyaránt előfordul, akkor adott i-re a t sort R_i-ben és S_i-ben is megtaláljuk. Ily módon, amikor ezen két kosár egyesítését vesszük, akkor a t sort csak egyszer írjuk a kimenetbe, és így nincs lehetőség a végeredményben ismétlődések bekerülésére. A $\cup_B$ művelet esetén a 6.3.3. részben bemutatott egyszerű multihalmaz-egyesítési algoritmus a művelet legalkalmasabb megközelítése.

R és S metszetének, illetve különbségének kiszámításakor a $2(M-1)$ darab kosarat pontosan ugyanúgy hozzuk létre, mint a halmaz alapú egyesítés esetében, majd a megfelelő kosárpárokra alkalmazzuk a megfelelő egymenetes algoritmust. Figyeljük meg, hogy itt az összes algoritmus lemez I/O-művelet igénye $B(R) + B(S)$. Ehhez a mennyiséghez hozzá kell még adni azt a blokkonkénti két lemez I/O-műveletet, amely a két reláció sorainak tördeléséhez és a kosarak lemezen való tárolásához szükséges, így a lemez I/O-műveletek száma összesen $3(B(R) + B(S))$.

Az algoritmusok működéséhez az szükséges, hogy vehessük R_i és S_i egymenetes egyesítését, metszetét vagy különbségét, amelyeknek mérete körülbelül $B(R)/(M-1)$, illetve $B(S)/(M-1)$. Emlékezzünk rá, hogy ezen műveletek egymenetes algoritmusaihoz az kell, hogy a kisebbik operandus legfeljebb $M-1$ blokkot foglaljon el. Így a tördelésen alapuló kétmenetes algoritmusokhoz jó közelítéssel a $\min(B(R),B(S)) \leq M^2$ feltételnek kell teljesülnie.

6.6.5. A tördeléses összekapcsolási algoritmus

Ahhoz, hogy $R(X, Y) \bowtie S(Y, Z)$ összekapcsolást egy tördelésen alapuló kétmenetes algoritmussal számíthassuk ki, majdnem ugyanazt kell tennünk mint a 6.6.4. részben vizsgált többi bináris műveletnél. Az egyetlen különbség az, hogy tördelőkulcsként csak az összekapcsolási attribútumot, Y-t kell használnunk. Ekkor ugyanis biztosak lehetünk benne, hogy ha R és S sorai összekapcsolódnak, akkor adott i-re a megfelelő R_i és S_i kosarakba fognak kerülni. Ezt a *tördeléses összekapcsolás*[4] nevű algoritmust az egymáshoz rendelt kosárpárok egymenetes összekapcsolása teszi teljessé.

6.19. példa: Térjünk vissza a 6.14. példában megismert R és S relációk tárgyalására; ezek mérete egyenként 1000 és 500 blokk, a rendelkezésre álló elsődleges memóriapufferek száma pedig 101. Megtehetjük, hogy mindkét relációt egyenként 100 kosárba tördeljük, azaz R és S esetében az átlagos kosárméret 10, illetve 5 blokk. Mivel a kisebbik szám – ami most 5 – jóval kisebb a rendelkezésre álló pufferek számánál, a kosárpárok egymenetes összekapcsolása várhatóan nem fog akadályba ütközni.

A lemez I/O-műveletek száma az R és S kosarakba történő tördelése közben végzett beolvasáskor 1500; majd újabb 1500 I/O-műveletet kell a kosarak lemezre írásához, végül pedig 1500 I/O-művelet szükséges az egyes kosárpárok memóriába történő beolvasásához, amikor a megfeleltetett kosarak egymenetes összekapcsolását végezzük. A szükséges lemez I/O-műveletek száma tehát 4500, pont úgy mint a 6.5.7. szakasz hatékony rendezéses összekapcsolásánál. □

A 6.19. példát általánosítva kimondhatjuk, hogy:

- A tördeléses összekapcsoláshoz $3(B(R) + B(S))$ lemez I/O-művelet kell.
- A kétmenetes tördeléses összekapcsolás addig működik, amíg $\min(B(R), B(S)) \leq M^2$.

Az utóbbi kijelentés ugyanúgy indokolható, mint a többi bináris műveletnél: a kosárpárok egyik tagjának be kell férnie $M-1$ darab pufferbe.

[4] Néha a 'tördeléses összekapcsolás' kifejezést a 6.3.3. részben bemutatott egymenetes összekapcsolási algoritmus egy olyan változatának tartják fenn, amelyben a tördelőtáblát elsődleges memóriastruktúraként használják. Ilyenkor az itt bemutatott kétmenetes tördeléses összekapcsolás algoritmusra 'particionált tördeléses összekapcsolás' néven utalnak.

6.6.6. Lemez I/O-műveletek megtakarítása

Ha az első menetben több memória áll rendelkezésre, mint ami a kosarankénti egy blokk befogadásához szükséges, akkor módunkban áll lemez I/O-műveleteket megtakarítani. Az egyik lehetőség az, hogy minden kosárra több blokkot használunk, és azokat csoportként írjuk ki egymást követő lemezblokkokra. Szigorúan véve ez a technika ugyan nem takarít meg lemez I/O-műveletet, de a lemez I/O-műveleteket felgyorsítja, hiszen az íráskor keresési időt és fejbeállási időt spórolunk meg.

Van azonban számos trükk, amelyek használatával elkerülhető néhány kosár lemezre írása és újbóli beolvasása. Közülük a leghatékonyabb, az ún. *hibrid tördeléses összekapcsolás*, amely a következőképpen működik. Tegyük fel, hogy S a kisebb reláció, és hogy az $R \bowtie S$ összekapcsoláshoz k darab kosarat kell létrehoznunk, ahol k sokkal kisebb M-nél, azaz a rendelkezésre álló memóriánál. Amikor az S relációt tördeljük, választhatunk úgy, hogy a k kosár közül m darabot teljesen az elsődleges memóriában tartunk, míg a fennmaradó $k-m$ számú kosár mindegyikéhez csak egy blokkot tartunk fent az elsődleges memóriában. Ezt úgy tehetjük meg, ha a memóriában lévő kosarak várható mérete plusz az összes többi kosárra számított egy-egy blokk nem haladja meg M-et, azaz

$$\frac{mB(S)}{k} + k - m \leq M. \tag{6.1.}$$

Ez azért van így, mert egy kosár várható mérete $B(S)/k$, és a memóriában m kosár van.

Amikor a másik reláció, az R sorait olvassuk be, akkor ennek a relációnak kosarakba történő tördelésekor a következőket tartjuk a memóriában:

1. S azon m darab kosarát, amelyeket soha nem írtunk ki lemezre, és
2. R azon $k-m$ darab kosarából, amelyek S-beli megfelelőit lemezre írtuk, egyenként egy blokkot.

Ha R egy t sora az első m kosár valamelyikébe kerül a tördeléskor, akkor azt azonnal összekapcsoljuk a megfelelő S-beli kosár összes sorával, mintha csak egymenetes tördeléses összekapcsolásról lenne szó. Minden egyes sikeres összekapcsolás eredményét tüstént a kimenetbe írjuk. Az összekapcsolás megkönnyítéséhez elengedhetetlen, hogy az S egyes memóriabeli kosarait hatékony keresési struktúrába szervezzük, pontosan úgy, mint az egymenetes tördeléses összekapcsolásnál. Ha tördeléskor t egy olyan kosárba kerül, amelynek megfeleltetett S-beli kosár a lemezen van, úgy t a kosár memóriabeli blokkjába kerül, majd végül átjut a lemezre, ahogy az a kétmenetes tördelésen alapuló összekapcsolásra történik.

A második menet alkalmával szokás szerint összekapcsoljuk R és S egymásnak megfeleltetett kosarait. Most azonban nincs szükség azon kosárpárok összekapcsolására, amelyekre az S-beli kosár a memóriában maradt, hiszen ezeket a kosarakat már összekapcsoltuk, és az eredményt kiírtuk a kimenetbe.

A lemez I/O-művelet megtakarítás a memóriában maradó S kosarak blokkjainak a száma plusz a hozzájuk tartozó R kosarak blokkjainak a száma, szorozva kettővel.

Mivel a kosarak m/k hányada van a memóriában, a megtakarítás $2(m/k)(B(R) + B(S))$. Feladatunk tehát m/k maximalizálása a 6.1. egyenlőtlenség megszorítását figyelembe véve. Noha ez a probléma formálisan is megoldható, intuíciónk a kissé meglepő, de helyes választ adja: $m = 1$, miközben k legyen a lehető legkisebb.

A magyarázat az, hogy $k - m$ kivételével az összes elsődleges memóriabeli puffert felhasználhatjuk S sorainak (az elsődleges memóriában való) tárolására, és minél több ilyen sor van, annál kevesebb lemez I/O-műveletre van szükség. Tehát k-t, azaz a kosarak számát akarjuk minimalizálni. Ezt úgy tehetjük meg, hogy minden kosarat nagyjából a legnagyobbra veszünk úgy, hogy még éppen beférjen az elsődleges memóriába. Ez tehát M méretű kosarakat jelent, ily módon $k = B(S)/M$. Ha ez teljesül, akkor a fennmaradó elsődleges memóriában csak egy kosár számára van hely, azaz $m = 1$.

A valóságban a kosarakat $B(S)/M$-nél valamivel kisebbre kell méreteznünk, másképpen előfordulhat, hogy nem lesz elég helyünk a memóriában egy teli kosárnak és a többi $k - 1$ kosárhoz tartozó egy-egy blokknak. Az egyszerűség kedvéért tegyük fel, hogy k körülbelül $B(S)/M$ és $m = 1$; ekkor a lemez I/O-művelet megtakarítása:

$$\left(\frac{2M}{B(S)}\right)\big(B(R)+B(S)\big),$$

a teljes költség pedig

$$\left(3-\frac{2M}{B(S)}\right)\big(B(R)+B(S)\big).$$

képletekkel fejezhető ki.

6.20. példa: Vegyük elő ismét a 6.14. példa problémáját. Ott az egyenként 1000 és 500 blokkból álló R és S relációkat kellett összekapcsolnunk $M = 101$ mellett. Ha hibrid tördeléses összekapcsolást használunk, akkor az a legelőnyösebb, ha k, a kosarak száma nagyjából 500/101. Legyen ezért a $k = 5$. Ekkor az S soraiból képzett kosarak átlagosan 100 blokkosak lesznek. Ha megpróbálunk betenni egy ilyen kosarat, továbbá még négy plusz blokkot a másik négy kosár számára, akkor összesen 104 darab memóriablokkra lesz szükségünk, márpedig azt nem kockáztathatjuk meg, hogy a kosár túlcsorduljon a memóriában.

Arra jutunk tehát, hogy legyen inkább $k = 6$. Ekkor S első menetbeli tördelésekor öt pufferünk van az öt kosárra, és maximum 96 puffer a memóriabeli kosárra, a kosarak várható mérete ez alkalommal 500/6, azaz 83. Az S teljes beolvasása során az első menetben felhasznált lemez I/O-műveletek száma 500, majd újabb 500 – 83 = 417 műveletre lesz szükség az öt kosár lemezre írásához. Amikor R-rel foglalkozunk az első menetben, akkor az egész relációt be kell olvasnunk (1000 lemez I/O-művelet), valamint 6 kosarából ötöt ki kell írnunk (833 lemez I/O-művelet).

A második menetben beolvassuk az összes lemezre írt kosarat, ami 417 + 833 = 1250 további lemez I/O-műveletet jelent. A lemez I/O-műveletek teljes száma tehát 1500 R és S beolvasásakor, 1250 a relációk 5/6-ának írásakor, majd újabb 1250 ezen sorok beolvasásakor, azaz összesen 4000. Ez a szám összemérhető a szokványos tördeléses összekapcsolás vagy rendezéses összekapcsolás 4500 lemez I/O-művelet igényével. □

6.6.7. A tördelésen alapuló algoritmusok összefoglalása

A 6.18. ábra megadja az eddig tárgyalt algoritmusok memóriakövetelményeit és lemez I/O-művelet igényét. Csakúgy, mint más típusú algoritmusok esetében, itt is érdemes megfigyelni, hogy γ és δ becslései elég visszafogottak, mert ezek valójában inkább a csoportok és ismétlődések számától függenek, nem pedig az argumentum reláció sorainak számától.

Vegyük figyelembe, hogy a rendezésen alapuló, illetve a megfelelő tördelésen alapuló algoritmusok követelményei szinte azonosak. A két megközelítés közötti jelentős különbségeket az alábbiakban összegeztük:

1. A bináris műveletek tördelésen alapuló algoritmusainak a memória méretére vonatkozó követelménye a két argumentum közül csak a kisebbiktől függ, nem pedig az argumentum méretek összegétől, mint a rendezésen alapuló algoritmusoknál.
2. A rendezésen alapuló algoritmusok esetén néha módunk van rá, hogy az eredményt rendezett sorrendben kapjuk meg, és ebből a későbbiek folyamán hasznot húzzunk. Az eredményt használhatjuk később, mondjuk egy másik, rendezésen alapuló algoritmusban, vagy alkothatja az egy olyan lekérdezésre adandó választ is, amelyet amúgy is rendezett sorrendben kell visszaadnunk.
3. A tördelésen alapuló algoritmusok azonos kosárméret esetén működnek. Mivel egy kisebb mértékű méretbeli eltérés általában mindig van, ezért nem használhatunk olyan kosarakat, amelyek átlagosan M blokkot foglalnak el. Ehelyett meg kell elégednünk valamivel kisebb felső korláttal a kosarak méretére vonatkozóan. Ennek hatása akkor különösen jelentős, ha a tördelőkulcsok száma kicsi, pl. egy csoportosítás végrehajtásakor egy kevés csoporttal rendelkező reláción, vagy egy olyan összekapcsolás esetén, ahol az összekapcsolás attribútumai kevés különböző értékkel rendelkeznek.
4. A rendezésen alapuló algoritmusok esetén – ha erre kellőképpen odafigyelünk – a rendezett részlistákat a lemez egymást követő blokkjaiba tudjuk írni. Így a blok-

Operátor	*Szükséges* M *kb.*	*Lemez I/O-művelet*	*Rész*
γ, δ	$\sqrt{B}$	$3B$	6.6.2., 6.6.3.
$\cup, \cap, -$	$\sqrt{B(S)}$	$3(B(R) + B(S))$	6.6.4.
$\bowtie$	$\sqrt{B(S)}$	$3(B(R) + B(S))$	6.6.5.
$\bowtie$	$\sqrt{B(S)}$	$(3 - 2M/B(S)) \times (B(R) + B(S))$	6.6.6.

6.18. ábra. *Elsődleges memória és lemez I/O-művelet követelmények a tördelésen alapuló algoritmusokra. Bináris műveletek esetén feltételezzük, hogy* $B(S) \leq B(R)$

konkénti három lemez I/O-művelet egyikénél csupán jelentéktelen fejbeállási időre és rövid keresési időre van szükség. Ezzel a tördelésen alapuló algoritmusok I/O-műveletéhez képest lényegesen nagyobb sebességet érhetünk el.

5. A fentiek mellett, ha M sokkal nagyobb a rendezett részlisták számánál, akkor egy rendezett részlistáról egyszerre több egymást követő blokkot is beolvashatunk, ismét csak fejbeállási időt és keresési időt takarítva meg.
6. Másrészt, ha egy tördelésen alapuló algoritmusban a kosarak számát M-nél kisebbre tudjuk választani, akkor egyszerre egy kosár több blokkját is ki tudjuk írni. Ezzel viszont a tördelésnél ugyanazt az előnyt szerezzük meg az írási lépésnél, mint amit a rendezés esetén a második beolvasásnál kaphatunk, ahogyan azt a 6.5. részben meg is jegyeztük. Hasonlóan, a lemezt esetleg be tudjuk úgy osztani, hogy egy kosár végül egy sáv egymás utáni blokkjaiba kerüljön. Ha ez sikerül, akkor a kosarakat csekély fejbeállási idővel vagy keresési idővel lehet beolvasni, azaz olvasási hatékonyságuk hasonlít a rendezett részlistákkal kapcsolatban említett írási hatékonyságra (4.).

6.6.8. Feladatok

6.6.1. feladat: A hibrid tördeléses összekapcsolás ötlete, hogy egy kosarat a memóriában tárolunk, más műveletek esetén is alkalmazható. Mutassuk meg, hogy hogyan lehet megtakarítani egy kosár tárolását és beolvasását minden egyes relációra, ha tördelésen alapuló kétmenetes algoritmust írunk a következő műveletekre:

* a) δ.
b) γ.
c) $\cap_B$.
d) $-_S$.

6.6.2. feladat: Ha $B(S) = B(R) = 10\,000$, és $M = 1000$, akkor mi a hibrid tördeléses összekapcsolás lemez I/O-művelet igénye?

6.6.3. feladat: Írjunk olyan iterátorokat, amelyek a következő műveletek tördelésen alapuló kétmenetes algoritmusait valósítják meg: a) δ, b) γ, c) $\cap_B$, d) $-_S$, e) $\bowtie$.

*! **6.6.4. feladat:** Tegyük fel, hogy egy megfelelő méretű ($B(R) \leq M^2$) R reláción tördelésen alapuló kétmenetes csoportosítás műveletet hajtunk végre. A csoportok száma azonban olyan kicsi, hogy néhány csoport nagyobb M-nél, azaz egyben nem férnek bele a memóriába. Hogyan kell módosítani az általunk megadott algoritmust? (kell-e egyáltalán?)

! **6.6.5. feladat:** Tegyük fel, hogy egy olyan lemezt használunk, ahol a fejet 100 ms alatt lehet rámozgatni egy blokkra, egy blokk olvasási ideje pedig 1/2 ms. Így – ha a fej egyszer már pozicionálva van – k egymást követő blokk olvasása $k/2$ ms ideig tart. Tegyük fel, hogy az $R \bowtie S$ kétmenetes tördeléses összekapcsolást szeretnénk kiszámítani, amelyben $B(R) = 1000$, $B(S) = 500$ és $M = 101$. Az összekapcsolás felgyorsítása érdekében a lehető legkevesebb kosarat akarjuk használni (feltesszük, hogy a sorok a kosarak között egyenletesen oszlanak meg), és a lemez egymást követő blokkjaiba a lehető legtöbb blokkot szeretnénk írni. (És később majd olvasni is persze.) Egy véletlenszerű lemez I/O-művelet idejét 100,5 ms-nak véve, és $100 + k/2$ ms-ot számítva k darab egymást követő blokk lemezre írására vagy lemezről beolvasására, válaszoljuk meg az alábbi kérdéseket:

a) Mennyi időt vesznek igénybe a lemez I/O-műveletek?
b) Mennyi időt vesznek igénybe a lemez I/O-műveletek akkor, ha a hibrid tördeléses összekapcsolást használjuk a 6.20. példában leírtak szerint?
c) Ugyanezen feltételek mellett mennyi időt vesz igénybe egy rendezésen alapuló összekapcsolás, feltéve, hogy a rendezett részlistákat egymást követő lemezblokkokra írjuk?

6.7. Index alapú algoritmusok

Ha létezik index a reláció egy vagy több attribútumához, akkor elérhetővé válik néhány olyan algoritmus, amely index hiányában nem lenne kivitelezhető. Az index alapú algoritmusok különösen hasznosak a kiválasztás operátorra, de az összekapcsolás és az egyéb bináris műveletek algoritmusai is igen jól kihasználják az indexeket. Folytatjuk az indexszel rendelkező táblák elérésére szolgáló index alapú beolvasás művelet 6.2.1. részben megkezdett tárgyalását is. Ahhoz, hogy ezt a témakört igazán értékelni tudjuk, először teszünk egy kitérőt, és az ún. „nyalábolt" indexekkel foglalkozunk.

6.7.1. Nyalábolt és nem nyalábolt indexek

Emlékezzünk rá a 6.2.3. részből, hogy egy relációt akkor nevezünk nyaláboltnak, ha sorai nagyjából annyi blokkban vannak tárolva, ahányban azok minimálisan elférnének. Az eddig elvégzett összes elemzés azt feltételezte, hogy a relációk nyaláboltak.

Beszélhetünk e mellett még *nyalábolt indexekről* is, amelyek olyan attribútumon vagy attribútumokon értelmezett indexek, amelyeknél a keresési kulcs egy rögzített értékéhez tartozó sorok nagyjából annyi blokkban helyezkednek el, ahány blokkban minimálisan elférnének. Vegyük észre, hogy egy nem nyalábolt relációnak nem lehet nyalábolt indexe[5], viszont egy nyalábolt relációnak lehetnek nem nyalábolt indexei.

[5] Technikai értelemben, ha az indexet a reláció egy kulcsára definiáljuk, azaz az indexkulcs egy adott értékével csak egy sor létezik, akkor az index mindig „nyalábolt", még akkor is, ha a reláció nem nyalábolt. Ugyanakkor, ha index kulcsértékenként csak egy sor van, akkor a tömörségnek nincs semmi haszna, és egy ilyen index teljesítményértékelése ugyanazt adja, mintha nem nyalábolt index lenne.

6.21. példa: Az a attribútum szerint rendezett $R(a, b)$ reláció, amelyet rendezett sorrendben tárolunk, biztosan nyalábolt. Az a attribútumon értelmezett index is nyalábolt, hiszen az a egy adott a_1 értékére az összes ilyen értékkel rendelkező sor egymás után található. Tehát nyalábolt an jelennek meg a sorok a blokkokban, kivéve esetleg az első és az utolsó a_1 értéket tartalmazó blokkokat, amint azt a 6.19. ábrán is láthatjuk. Egy index a b attribútumon viszont valószínűleg nem nyalábolt, mivel az adott b értékkel rendelkező sorok bárhol elhelyezkedhetnek a fájlban, kivéve, ha az a és b értékei erősen korreláltak. □

6.19. ábra. *Egy nyalábolt indexnek az összes rögzített értékkel rendelkező sora a lehetséges minimális (vagy majdnem minimális) számú blokkban található*

6.7.2. Index alapú kiválasztás

A 6.2.1. részben megmutattuk, hogy miként lehet egy $\sigma_C(R)$ kiválasztást megvalósítani oly módon, hogy beolvassuk az R reláció összes sorát, és a C feltételt kielégítő sorokat kiírjuk a kimenetre. Ha R nem rendelkezik indexszel, akkor ez a legjobb, amit tehetünk. A művelet által felhasznált lemez I/O-műveletek száma $B(R)$, vagy esetleg $T(R)$, az R sorainak száma, ha R nem nyalábolt reláció[6]. Tegyük fel azonban, hogy a C feltétel $a = v$ formájú, ahol az a olyan attribútum, amelyen van indexe, v pedig egy érték. Ekkor rákereshetünk az index v értékére, és megkapjuk azokat a mutatókat, amelyek az R azon soraira mutatnak, ahol az a attribútum értéke éppen v. Ezek a sorok képezik $\sigma_{a=v}(R)$ eredményét, így mindössze annyit kell tennünk, hogy visszanyerjük azokat.

Ha az $R.a$ indexe nyalábolt, akkor a $\sigma_{a=v}(R)$ halmaz visszanyeréséhez szükséges lemez I/O-műveletek száma nagyjából $B(R)/V(R, a)$. A tényleges érték lehet, hogy ennél valamivel nagyobb, mert:

1. Gyakran előfordul, hogy az indexet nem tároljuk teljes egészében a memóriában, vagyis kell néhány lemez I/O-művelet az indexben történő kereséshez.
2. Lehet, hogy az összes olyan sor, amelyre $a = v$, belefér b számú blokkba, mégis előfordulhat, hogy $b + 1$ blokkban vannak tárolva, mert nem valamelyik blokk elején kezdődnek.
3. Bár az index nyalábolt, az $a = v$ értékű sorok átnyúlhatnak néhány további blokkba is. Lássunk két okot arra, hogy ez miért fordulhat elő:

   a) R blokkjait esetleg nem a legszorosabban raktuk össze, mert helyet akartunk hagyni R növekedésének a 4.1.6. részben tárgyaltak szerint.

[6] Idézzük fel a 6.2.3. részben használt jelöléseket: $T(R)$ jelöli az R reláció sorainak számát és $V(R, L)$ a $\pi_L(R)$ művelet egymástól különböző sorainak számát.

   b) R tárolása elképzelhető olyan sorokkal együtt is, amelyek nem tartoznak R-hez, mondjuk egy nyalábolt fájl elrendezésében.

A fentiek mellett persze kerekítenünk is kell, ha $B(R)/V(R, a)$ érték nem egész szám. Ez leginkább akkor fordul elő, ha a az R kulcsa, ilyenkor $V(R, a) = T(R)$, ami feltehetőleg sokkal nagyobb $B(R)$-nél, nekünk azonban mégis szükségünk van egy lemez I/O-műveletre a v kulcsértékkel rendelkező sor visszanyerésére, és ehhez még hozzájön az index eléréséhez szükséges valahány lemez I/O-művelet is.

Nézzük meg most, hogy mi történik, ha az $R.a$ indexe nem nyalábolt. Első közelítésben elmondható, hogy minden visszanyert sor más blokkban lesz, és $T(R)/V(R, a)$ számú sort kell elérnünk. Ezek szerint éppen $T(R)/V(R, a)$ adja a szükséges lemez I/O-műveletek számának becslését. Maga a tényleges szám ennél nagyobb is lehet, mert előfordulhat, hogy pár indexblokkot lemezről kell beolvasnunk; de lehet a szám ennél

A nyalábolás alapfogalmai

Az előzőekben három különböző, bár egymáshoz közel álló koncepciót ismertünk meg „nyalábolás" néven.

1. A 4.2.2. részben a „nyalábolt fájlszervezés"-ről beszéltünk, amelyben egy R reláció sorait együtt tároljuk egy másik S reláció olyan sorával, amely a közös attribútumon megegyezik az R ezen soraival. Az ott bemutatott példában a filmeket tároló reláció sorait csoportosítottuk a stúdió reláció azon sorával, amely a filmet készítő stúdió adatait tartalmazta.
2. A 6.2.3. részben „nyalábolt reláció"-ról beszéltünk, ami azt jelenti, hogy a reláció sorait olyan blokkokban tároljuk, amelyek kizárólag, vagy legalábbis főként arra hivatottak, hogy a szóban forgó relációt tárolják.
3. Ebben a fejezetben vezettük be a nyalábolt index fogalmát – ami egy olyan index, amelynél a keresési kulcs egy adott értékével rendelkező sorok olyan blokkokban fordulnak elő, amelyek lényegében pont ezzel a keresésikulcs-értékkel rendelkező sorok tárolására vannak fenntartva. Az adott értékkel rendelkező sorokat általában egymás után tároljuk, és csupán az adott értékű sorok első és az utolsó blokkjaiban lehetnek más keresésikulcs-értékű sorok.

A nyalábolt fájlszervezés egy példa arra, hogy miként lehet olyan nyalábolt relációnk, amelynek blokkjai nem kizárólag ennek a relációnak a sorait tartalmazzák. Tegyük fel, hogy az S reláció egy sorához az R több sora is hozzátartozik egy nyalábolt fájlban. Ebben az esetben – noha R sorai nem olyan blokkokban találhatók, amelyek kizárólag az R számára vannak fenntartva – a blokkok mégis „főként" az R számára vannak fenntartva, és az R relációt nyaláboltnak hívjuk. Másrészt viszont S jellemzően *nem* nyalábolt reláció, hiszen sorai általában főként inkább R-beli, mintsem S-beli soroknak szánt blokkokban találhatók.

kisebb is, ha egyes visszanyert sorok véletlenül ugyanabban a blokkban jelennek meg, és a szóban forgó blokk a memóriapufferben marad.

6.22. példa: Legyen $B(R) = 1000$, és legyen $T(R) = 20\,000$. Ezek szerint R-nek húszezer sora van, amelyek húszasával találhatók a blokkokban. Legyen a az R egyik attribútuma; tegyük fel, hogy létezik index az a-n, és vizsgáljuk meg a $\sigma_{a=0}(R)$ műveletet. Az alábbiakban felsorolunk néhány lehetőséget, és megadjuk a legrosszabb esetben szükséges lemez I/O-műveletek számát. Az indexblokkok elérésének költségét minden esetben elhanyagoljuk.

1. Ha R nyalábolt, de nem használjuk az indexet, akkor a költség 1000 lemez I/O-művelet. Ez azt jelenti, hogy R minden blokkját vissza kell nyernünk.
2. Ha R nem nyalábolt, és nem használjuk az indexet, akkor a költség 20 000 lemez I/O-művelet.
3. Ha $V(R, a) = 100$ és az index nyalábolt, akkor az index alapú algoritmus $1000/100 = 10$ lemez I/O-műveletet használ.
4. Ha $V(R, a) = 10$ és az index nem nyalábolt, akkor az index alapú algoritmus $20\,000/100 = 200$ lemez I/O-műveletet használ. Figyeljük meg, hogy ez a költség magasabb, mint az egész R reláció beolvasása abban az esetben, ha R nyalábolt, de az index nem.
5. Ha $V(R, a) = 20\,000$, azaz a kulcs, akkor az index alapú algoritmus 1 lemez I/O-műveletet igényel, plusz még azt, ami az index eléréséhez szükséges, függetlenül attól, hogy az index tömör vagy sem.

□

Az index alapú beolvasás mint elérési módszer más típusú kiválasztási művelet során is hasznos lehet.

a) Egy index – pl. egy B-fa – lehetővé teszi az egy adott tartományon belüli keresési-kulcs-értékek elérését. Ha az R reláció a attribútumára létezik ilyen index, akkor azt használhatjuk arra, hogy a $\sigma_{a \geq 10}(R)$, illetve $\sigma_{a \geq 10 \text{ AND } a \leq 20}(R)$ típusú kiválasztások esetén az R-nek csak a kívánt tartományba eso sorait olvassuk be.
b) Egy komplex C feltételre alapuló kiválasztást bizonyos esetekben megvalósíthatunk úgy, hogy egy index alapú beolvasás után egy másik kiválasztást végzünk a már beolvasott sorokon. Ha pl. a C feltétel $a = v$ AND C' formájú, ahol C' egy tetszőleges feltétel, akkor a kiválasztást két egymás utáni kiválasztásra bonthatjuk fel. Az első kiválasztás csak $a = v$ feltételt ellenőrzi, míg a második a C' feltételt ellenőrzi. Az első kiválasztásnál az index alapú beolvasás operátor használata látszik valószínűnek. A kiválasztás művelet ilyen jellegű kettéosztása csupán egyike annak a sok javításnak, amelyet egy lekérdezésoptimalizáló a logikai lekérdezésterven eszközölhet, és amellyel a 7.7.1. rész foglalkozik.

6.7.3. Összekapcsolás index segítségével

Az összes eddig vizsgált bináris művelet, továbbá a δ és a γ teljes relációs unáris műveletek is előnyösen használhatnak bizonyos indexeket. Ezen algoritmusok nagy részét feladatként hagyjuk, és most csak az összekapcsolásokra fogunk összpontosítani. Ezen belül is az $R(X, Y) \bowtie S(Y, Z)$ természetes összekapcsolást vizsgáljuk meg. Emlékezzünk rá, hogy az X, az Y és a Z attribútumok halmazait jelöli, bár itt gondolhatunk rájuk úgy is, mint konkrét attribútumokra.

Az első index alapú összekapcsolási algoritmushoz tegyük fel, hogy az S relációnak van egy indexe az Y attribútumo(ko)n. Ekkor az összekapcsolás kiszámításának egyik módja az, hogy megvizsgáljuk az R minden egyes blokkját, majd pedig a blokkokon belül minden egyes t sort. Jelöljük az Y attribútum(ok)nak megfelelő t komponenst vagy komponenseket t_Y-nal. Használjuk az indexet arra, hogy megkeressük az S azon sorait, amelyek Y komponense éppen t_Y. Ezek S-nek pontosan azok a sorai, amelyek összekapcsolódnak az R t sorával, ezért minden egyes ilyen sor t-vel való összekapcsolását kiírjuk a kimenetre.

A lemez I/O-műveletek száma több tényezőtől függ. Először is, feltéve, hogy R nyalábolt, $B(R)$ számú blokkot kell beolvasnunk ahhoz, hogy R minden sorát megkapjuk. Ha R nem nyalábolt, akkor akár $T(R)$ számú lemez I/O-műveletre is szükség lehet.

R minden egyes t sorára átlagosan az S reláció $T(S)/V(S, Y)$ számú sorát kell beolvasnunk. Ha S-nek van az Y-ra egy nem nyalábolt indexe, akkor a szükséges lemez I/O-műveletek száma $T(R)T(S)/V(S, Y)$, ha viszont az index nyalábolt, akkor mindössze $T(R)B(S)/V(S, Y)$ is elégséges[7]. Mindkét esetben előfordulhat, hogy hozzá kell még adnunk Y értékenként néhány lemez I/O-műveletet magának az indexnek a beolvasására.

Függetlenül attól, hogy R nyalábolt vagy sem, mindenképpen az S sorainak elérési költsége a döntő, így ennek az összekapcsolási módszernek a költségét $T(R)T(S)/V(S, Y)$-ként illetve $T(R)\big(\max\big(1, B(S)/V(S, Y)\big)\big)$-ként adhatjuk meg azokra az esetekre, ha az S indexe nem nyalábolt, illetve nyalábolt.

6.23. példa: Térjünk vissza az előző példák adataihoz: az $R(X, Y)$ és az $S(Y, Z)$ relációk egyenként 1000, ill. 500 blokkot foglalnak el. Tegyük fel, hogy mindkét relációból tíz sor fér egy blokkba, azaz $T(R) = 10\,000$ és $T(S) = 5000$. Tegyük fel továbbá, hogy $V(S, Y) = 100$, azaz S sorai között 100 különböző Y érték fordul elő.

Feltételezzük, hogy R nyalábolt, S-nek pedig van egy nyalábolt indexe az Y attribútumo(ko)n. Ekkor – az index elérésére használtakat leszámítva – az R blokkjainak beolvasásához szükséges lemez I/O-műveletek száma körülbelül 1000 (amit a fenti képletekben elhanyagoltunk), ehhez jön még az összekapcsolás költsége, ami $10\,000 \times 500/100 = 50\,000$. Ez a szám jóval meghaladja az ugyanezekkel az adatokkal végzett, korábban bemutatott módszerek költségét. Ez a költség még tovább nő, ha vagy az R reláció vagy S indexe nem nyalábolt. □

[7] Ne felejtsük azonban el, hogy $B(S)/V(S, Y)$ helyére 1-et kell írnunk, ha az előbbi hányados 1-nél kisebb lenne; lásd a 6.7.2. részt.

Noha a 6.23. példa alapján azt gondolhatnánk, hogy az index alapú összekapcsolás nem valami jó ötlet, vannak olyan helyzetek amikor az $R \bowtie S$ összekapcsolást érdemes ilyen módszerrel elvégezni. A leggyakoribb eset az, amikor R sokkal kisebb S-nél, $V(S, Y)$ pedig nagy. A 6.7.5. feladatban utalunk majd egy tipikus lekérdezésre, ahol az összekapcsolást megelőző kiválasztás során R egészen pici lesz. Ebben az esetben S legnagyobb részét az algoritmus egyszer sem vizsgálja meg, hiszen a legtöbb Y érték R-ben meg sem jelenik. A rendezésen, valamint a tördelésen alapuló összekapcsolási módszerek viszont legalább egyszer megvizsgálják S minden egyes sorát.

6.7.4. Összekapcsolások rendezett index segítségével

Ha az index egy B-fa vagy más olyan struktúra, amelyből egy reláció sorait könnyedén megkaphatjuk rendezett formában, akkor számos más lehetőségünk nyílik az index használatára. A legegyszerűbb ezek közül talán az, amikor $R(X, Y) \bowtie S(Y, Z)$ összekapcsolást akarjuk kiszámítani, és vagy az R vagy az S rendelkezik egy rendezett indexszel Y attribútumo(ko)n. Ekkor elvégezhetünk egy közönséges rendezéses összekapcsolást, viszont kihagyhatjuk azt a köztes lépést, amelyben az egyik relációt Y szerint rendezzük.

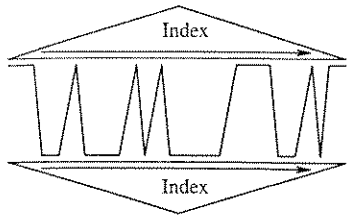
Szélsőséges esetben, amikor az R és az S egyaránt rendelkezik rendezett indexszel az Y attribútumo(ko)n, akkor a 6.5.5. rész egyszeru rendezésen alapuló összekapcsolásából csak a befejező lépést kell végrehajtanunk. Ezt a módszert néha *cikcakk-összekapcsolásnak* is szokták nevezni, mert oda-vissza ugrálunk az indexek között olyan Y értékeket keresve, amelyek közösek. Vegyük észre, hogy R-nek az olyan Y értékű sorait, amelyek S-ben nem fordulnak elő, egyszer sem kell beolvasnunk. Hasonlóan nem kell beolvasnunk egyszer sem S-nek azokat a sorait, amelyeknek Y értéke R-ben nem jelenik meg.

6.24. példa: Tegyük fel, hogy az $R(X, Y)$ és az $S(Y, Z)$ relációk rendelkeznek indexszel az Y attribútumo(ko)n. Egy egyszerű példaként legyenek R sorainak keresési kulcsai (Y értékei) sorrendben az 1, 3, 4, 4, 4, 5, 6 értékek, S keresésikulcs-értékei pedig legyenek 2, 2, 4, 4, 6, 7. R és S első kulcsaival kezdünk, esetünkben ezek az 1 és a 2. Mivel $1 < 2$, R első kulcsát átugorhatjuk, és rögtön rátérhetünk a második kulcsra, a 3-ra. Most S szóban forgó kulcsa kisebb R aktuális kulcsánál, így S két darab 2-esét átugorhatjuk, és jöhet a 4.

Ennél a pontnál R 3-as kulcsa kisebb S kulcsánál, így R kulcsát átugorjuk. Most mindkét aktuális kulcs a 4-es. Követjük mindkét relációban az összes 4-es kulcshoz tartozó mutatót, visszanyerjük a megfelelő sorokat, majd összekapcsoljuk őket. Figyeljük meg, hogy a relációnak egyetlen sorát sem olvastuk be addig, amíg a közös 4-es kulcsot el nem értük.

A 4-esekkel végezvén vesszük R 5-ös és S 6-os kulcsát. $5 < 6$, ezért átugrunk R következő kulcsára. Most mindkét kulcs 6-os, ezért visszanyerjük a megfelelő sorokat, és összekapcsoljuk őket. R mostanra kiürült, tehát tudjuk, hogy a két relációban már nincsenek további összekapcsolható sorpárok. □

Ha az indexek B-fák, akkor a két B-fa leveleit beolvashatjuk sorrendben balról, követve a rendszerbe épített mutatókat levélről levélre, a 6.20. ábrának megfelelően. Ha R és S nyalábolt, akkor egy adott kulcsnak megfelelő összes sor visszanyerése a két beolvasott relációrésszel arányos számú lemez I/O-műveletet eredményez. Megjegyezzük, hogy abban a szélsőséges esetben, amikor R és S olyan nagyszámú sorát olvassuk be, hogy egyik sem fér be a rendelkezésre álló memóriába, akkor egy, a 6.5.5. részben bemutatott ötlethez hasonló mentő ötlettel kell előrukkolnunk. A szokványos esetekben azonban a közös Y értékkel rendelkező sorok összekapcsolásához elegendő annyi lemez I/O-művelet, mint amennyi a relációk beolvasásához szükséges.



6.20. ábra. *Két indexet használó cikcakk-összekapcsolás*

6.25. példa: Folytassuk tovább a 6.23. példát, hogy lássuk, miként birkóznak meg ugyanezekkel az adatokkal a rendezés és indexelés kombinációját használó összekapcsolások. Tegyük fel először, hogy S rendelkezik egy indexszel az Y-on, és az index révén visszanyerhetjük S sorait Y attribútumo(ko)n rendezve. A mostani példában azt is feltesszük, hogy mindkét reláció és az index is nyalábolt. Az R relációhoz nem tételezünk fel indexet.

Feltételezzük, hogy 101 darab memóriablokk áll rendelkezésre. Ezeket használhatjuk arra, hogy létrehozzuk az 1000 blokkból álló R reláció 10 rendezett részlistáját. R egészének írásához és olvasásához 2000 lemez I/O-művelet szükséges. Ezután felhasználunk 11 memóriablokkot – tízet R részlistáihoz, egyet pedig S sorainak egy, az index segítségével visszanyert blokkjához. Nem vesszük figyelembe az index kezeléséhez szükséges lemez I/O-műveleteket és memóriablokkokat. Ha az index egy B-fa, akkor ezek a számok amúgy is kicsik lesznek. A második menetben beolvassuk R és S összes sorát, amihez 1500 lemez I/O-műveletet használunk fel, plusz még egy keveset, ami az indexblokkok egyenkénti beolvasásához kell. A lemez I/O-műveletek számát összesen tehát 3500-ra becsülhetjük, ami kevesebb, mint az eddig megvizsgált módszerek esetén volt.

Most pedig tegyük fel, hogy R és S egyaránt rendelkeznek indexszel az Y-on. Ekkor nincs szükség egyik reláció rendezésére sem. Mindössze 1500 lemez I/O-műveletet használva az indexek segítségével beolvashatjuk R és S blokkjait. Ha pedig csupán az indexekre támaszkodva meg tudjuk állapítani, hogy R vagy S jelentős része nem feleltethető meg a másik reláció sorainak, akkor a teljes költség jóval az 1500 lemez I/O-művelet alatt maradhat. Bárhogy is legyen azonban, ehhez hozzá kell még vennünk néhány lemez I/O-műveletet az indexek beolvasására. □

6.7.5. Feladatok

6.7.1. feladat: Tegyük fel, hogy az $R.a$ attribútumon van egy index. Írjuk le, hogy ezt az indexet miként lehet az alábbi műveletek végrehajtásának javítására használni. Milyen körülmények között lenne az index alapú algoritmus hatékonyabb a rendezésen vagy a tördelésen alapulónál?

* a) $R \cup_S S$ (tegyük fel, hogy R-ben és S-ben nincsenek ismétlődések, de lehetnek közös soraik).
b) $R \cap_S S$ (R és S ismét halmazok).
c) $\delta(R)$.

6.7.2. feladat: Tegyük fel, hogy $B(R) = 10\,000$ és $T(R) = 500\,000$. Legyen $R.a$-n index, és legyen $V(R, a) = k$ valamilyen k-val. Adjuk meg $\sigma_{a=0}(R)$ költségét k függvényében az alábbi feltételek mellett. Az indexhez való hozzáférés lemez I/O-műveletek száma elhanyagolható.

* a) Az index nyalábolt.
b) Az index nem nyalábolt.
c) R nyalábolt, az indexet pedig nem használjuk.

6.7.3. feladat: Ismételjük meg a 6.7.2. feladatot arra az esetre, amikor a művelet a $\sigma_{C \le a \text{ AND } a \le D}(R)$ tartomány lekérdezés. Tegyük fel, hogy C és D olyan konstansok, hogy az értékek k/10-ed része esik a tartományba.

! **6.7.4. feladat:** Ha R nyalábolt, de az $R.a$ index *nem*, akkor k-tól függően előnyösebb lehet egy olyan lekérdezés, amely táblabeolvasást végez R-en, illetve egy olyan, ami az indexet használja. Milyen k értékekre érdemesebb az indexet használni, ha a reláció és a lekérdezés:

a) A 6.7.2. feladatban szereplővel azonos.
b) A 6.7.3. feladatban szereplővel azonos.

* **6.7.5. feladat:** Tekintsük a következő SQL-lekérdezést:

```
SELECT születés_idő
FROM SzerepelBenne, FilmSzínész
WHERE cím = 'King Kong' AND színészNév = név;
```

A fenti lekérdezés a következő „filmes" relációkat használja:

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

A fentieket a relációs algebra nyelvére lefordítva, a lényeg egy egyenlőségen alapuló összekapcsolás a

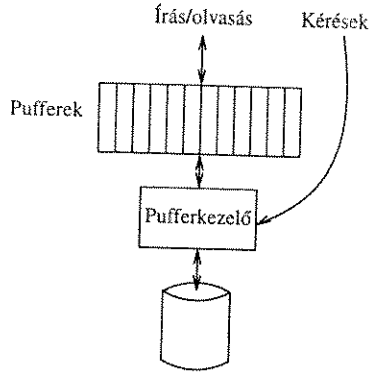
$\sigma_{\text{title} = \text{'King Kong'}}(\text{SzerepelBenne})$

és a FilmSzínész relációk között, amit az $R \bowtie S$ természetes összekapcsoláshoz hasonlóan lehet elvégezni. Mivel csak két 'King Kong' című film volt, $T(R)$ nagyon kicsi. Tegyük fel, hogy S – a FilmSzínész reláció – rendelkezik egy indexszel a név attribútumon. Hasonlítsuk össze ezen az $R \bowtie S$-en végrehajtott index-összekapcsolás költségét egy rendezésen vagy tördelésen alapulóval.

! **6.7.6. feladat:** A 6.25. példában megtárgyaltuk egy olyan $R \bowtie S$ összekapcsolás költségeit, amelyben az R és az S relációk egyikének vagy mindkettőjének volt rendezett indexe az összekapcsolási attribútum(ok)on. Az említett példában ismertetett módszerek azonban kudarcot vallhatnak akkor, ha túl nagy számú, az összekapcsolási attribútumo(ko)n azonos értékkel rendelkező sor van. Mik azok a korlátok (az azonos értékkel rendelkező sorok által elfoglalt blokkok számának tekintetében), amelyek teljesítése esetén a bemutatott módszereknek nincs szüksége további lemez I/O-műveletekre?

6.8. Pufferkezelés

Az eddigiekben feltettük, hogy a relációkon végzett műveletek számára rendelkezésre áll bizonyos számú memóriapuffer – ezek számát M-el jelöltük –, amelyben azok a szükséges adatokat tárolhatják. A gyakorlatban ezeket a puffereket ritkán foglaljuk le előre a műveletek számára, így M értéke a rendszer pillanatnyi állapotától függően változhat. A memóriapuffereket a *pufferkezelő* (buffer manager) teszi elérhetővé a processzek számára, így az adatbázison dolgozó lekérdezések számára is. A pufferkezelőnek kell arról gondoskodnia, hogy a processzek megkapják a számukra szükséges memóriát, és eközben a késedelmek és a kielégíthetetlen kérések száma minimális maradjon. A pufferkezelő szerepét a 6.21. ábrán láthatjuk.



6.21. ábra. *A pufferkezelő kezeli a lemezblokkok memóriába olvasására irányuló kéréseket*

6.8.1. A pufferkezelő működése

A pufferkezelő alapvetően kétféle módon működhet, amelyek a következők:

1. A pufferkezelő közvetlenül kezeli a memóriát (sok relációs adatbázis-kezelőben ezt a megoldást alkalmazzák), illetve
2. A pufferkezelő a virtuális memóriában foglal le puffereket, és az operációs rendszerre bízza annak eldöntését, hogy egy adott időpontban mely pufferek vannak ténylegesen a memóriában, és melyek vannak a lemezen egy úgynevezett lapcserélési területen. Ezt a lemezterületet az operációs rendszer tartja karban. (Sok, főként memóriában dolgozó adatbázis-kezelő és objektum alapú adatbázis-kezelő ezt a módszert alkalmazza.)

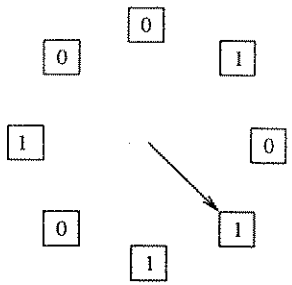
Bármelyik megoldást alkalmazza is az adatbázis-kezelő, mindkét esetben ugyanaz a probléma merül fel. Nevezetesen, a pufferkezelőnek úgy kell korlátoznia a használatban levő pufferek számát, hogy azok beleférjenek a rendelkezésre álló memóriába. Ha a pufferkezelő közvetlenül kezeli a memóriát, és a kérések meghaladják a rendelkezésre álló helyet, akkor ki kell választania egy puffert, amelyet kiürít úgy, hogy a tartalmát visszaírja a lemezre. Amennyiben a pufferelt blokk nem változott a beolvasás óta, úgy egyszerűen törölhető a memóriából, de ha változott akkor vissza kell írni a megfelelő helyre a lemezen. Ha a pufferkezelő a virtuális memóriában foglalja le a helyet, akkor lehetősége van arra, hogy több puffert foglaljon le, mint amennyi a tényleges memóriában elfér. Ha azonban az adatbázis-kezelő az összes ilyen puffert ténylegesen használja, akkor egy jól ismert operációs rendszerekre vonatkozó probléma merül fel, ami abban nyilvánul meg, hogy túl sok blokkmozgatás történik a memória és a lapcserélési terület között. Ilyenkor az operációs rendszer az idejének a nagy részét a blokkok cserélgetésével tölti, miközben kevés ténylegesen hasznos munkát végez.

Általában a pufferek számát egy paraméterként adhatjuk meg, amelyet az adatbázis-kezelő az elindulásakor állít be. Mostantól feltételezzük, hogy ez az érték úgy van beállítva, hogy a pufferek elfoglalják a rendelkezésre álló teljes memóriát, függetlenül attól, hogy a pufferek a tényleges vagy a virtuális memóriába kerülnek-e. A továbbiakban nem foglalkozunk azzal, hogy a pufferkezelő melyik működési módot követi, egyszerűen feltesszük, hogy van egy rögzített méretű *pufferterület* (buffer pool), amely a lekérdezések és más adatbázis-műveletek számára rendelkezésre álló pufferek egy halmaza.

6.8.2. Pufferkezelő stratégiák

A kritikus döntés, amelyet a pufferkezelőnek meg kell hoznia, amikor egy újólag kért blokkhoz szüksége van egy pufferre, hogy melyik korábbi blokkot dobja ki a pufferterületről. A leggyakrabban használt *puffercserélési stratégiák* talán ismerősek az olvasó számára az ütemezési eljárások más alkalmazási területeiről, mint amilyenek például az operációs rendszerek. Az említett stratégiák a következők:

- *Legrégebben használt (LRU).* Az LRU-szabály azt mondja, hogy dobjuk ki azt a blokkot, amelyre vonatkozóan a legrégebben nem történt írás vagy olvasás. Ez a módszer szükségessé teszi, hogy a pufferkezelő egy táblázatot tartson karban, amelyben az egyes pufferekbeli blokkok legutolsó hozzáférési időpontja szerepel. Szükség van még arra is, hogy minden egyes adatbázis-hozzáférési művelet egy bejegyzést tegyen ebbe a táblázatba. Ezeknek az információknak a karbantartása jelentős erőfeszítést igényel, az LRU mégis hatékony stratégia. Azokat a puffereket ugyanis, amelyeket már hosszabb idő óta nem használtak, kevésbé valószínű, hogy hamarabb szeretnék elérni, mint azokat, amelyekkel mostanában dolgoztak.
- *Elsőként bejövő-elsőként kimenő (FIFO).* A FIFO-eljárás alapján, ha egy pufferre van szükség egy új blokk számára, akkor azt a puffert ürítjük ki és használjuk fel, amelyikben a leghosszabb idő óta van ugyanaz a blokk. Ennél a módszernél a pufferkezelőnek csak azt az időpontot kell ismernie, amikor betöltötte azt a blokkot, amelyik jelenleg is a pufferben van. Ezt megoldhatja úgy, hogy egy bejegyzést tesz egy táblázatba, amikor a blokkot a lemezről beolvassa, és ezt később már nem kell módosítani a blokkhozzáférések esetén. A FIFO-módszer kevesebb karbantartási munkát igényel mint az LRU, de több esetben hoz hibás döntést. Egy olyan blokk például, amelyet újra meg újra használunk – mondjuk egy B-fa-index gyökérblokkja – előbb-utóbb a legrégebbi blokká válik a pufferben. Ilyenkor az eljárás szerint kiírjuk a lemezre, hogy nemsokkal később ismét beolvassuk egy másik pufferbe.
- *Az „óra" algoritmus.* Ez az algoritmus egy gyakran megvalósított, hatékony közelítése az LRU algoritmusnak. Képzeljük el, hogy a pufferek egy körbe vannak rendezve, ahogyan azt a 6.22. ábra mutatja. Van egy mutatónk, amelyik az egyik pufferre mutat, és az óra járásával megegyező irányban fog elfordulni, ha szükség lesz egy pufferre egy lemezblokk beolvasásához. Minden pufferhez hozzárendelünk egy jelzőértéket, ami 0 vagy 1 lehet. A 0 értékű pufferek tartalmát visszaírhatjuk a lemezre, az 1 értékűekét nem. Amikor egy blokkot beolvasunk egy pufferbe, a puffer jelzőértékét 1-re állítjuk. Akkor is 1-re állítjuk a jelzőértéket, amikor a puffer tartalmára vonatkozóan adathozzáférés történik. Ha a pufferkezelőnek egy pufferre van szüksége egy új blokk beolvasásához, akkor az óra járásának megfelelő irány-



6.22. ábra. *Az óra algoritmus körbe-körbe haladva bejárja a puffereket, és az első 1-est 0-val helyettesíti*

További lehetőségek az óra algoritmussal kapcsolatban

A pufferek kiürítésére az óra algoritmust nem csak úgy alkalmazhatjuk, ahogyan azt a 6.8.2. részben leírtuk, amikor a pufferek értéke 0 vagy 1 értéket vehetett fel. Megtehetjük azt is, hogy egy fontos blokk esetén a puffer értékét 1-nél nagyobbra állítjuk, és minden alkalommal, amikor a mutató elhalad mellette, az értéket eggyel csökkentjük. Tulajdonképpen a blokkok rögzítését is megvalósíthatjuk ezzel a módszerrel úgy, hogy a rögzített blokknak végtelen értéket adunk, majd a rögzítést úgy engedjük el a megfelelő időben, hogy az értéket 0-ra állítjuk.

ban haladva megkeresi az első 0 értékű puffert. Ha 1 értékű pufferek mellett halad el, akkor azok értékét 0-ra változtatja. Ezzel a módszerrel egy blokkot csak akkor dobunk ki a pufferből, ha az alatt az idő alatt nem történik rá vonatkozóan adathozzáférés, amíg a mutató odaérve 0-ra állítja az értékét, majd még egy teljes kört megtéve még mindig a 0 értéket találja ott. Például a 6.22. ábrán a mutató 0-ra fogja állítani a bal oldalán levő puffert, majd továbbhaladva megtalálja a 0-s puffert, amelynek tartalmát kicseréli az új blokkal, majd ezután 1-re állítja a puffer értékét.

- *A rendszerfelügyelő.* A lekérdezésfeldolgozó vagy az adatbázis-kezelő más komponense további információt adhat a pufferkezelőnek, hogy az elkerülhessen néhány olyan jellegű hibát, amilyenek az LRU, a FIFO vagy az óra algoritmus szigorú alkalmazásával előfordulhatnak. Emlékeztetünk a 3.3.5. részre, amely szerint vannak olyan esetek, amikor technikai akadályai vannak annak, hogy egy memóriabeli blokkot lemezre írjunk anélkül, hogy előbb módosítanánk más blokkokat, amelyek az előbbire mutatnak. Az ilyen blokkokat „csatolt" blokkoknak nevezzük. A pufferkezelőnek módosítania kell a puffercserélési stratégiáját, nehogy a rögzített blokkokat kidobja. Ez lehetőséget ad számunkra, hogy olyan blokkok memóriában maradását is kikényszerítsük – „csatoltnak" deklarálva őket –, amelyek lemezre írásának egyébként nem lenne technikai akadálya. Például a korábban a FIFO-eljárásnál említett problémát a B-fa gyökerével kapcsolatosan megoldhatjuk oly módon, hogy a gyökeret „csatoljuk". és így biztosítjuk azt, hogy az véglegesen a memóriában maradjon. Hasonlóan, egy olyan algoritmusnál, mint az egymenetes tördelés alapú összekapcsolás, a lekérdezésfeldolgozó „csatolhatja" a kisebb reláció blokkjait, és ezzel eléri, hogy az a művelet teljes idejére a memóriában maradjon.

6.8.3. Kapcsolat a fizikai operátor kiválasztása és a pufferkezelés között

A lekérdezésoptimalizáló egy lekérdezés végrehajtásához néhány fizikai operátort választ ki. Az operátoroknak ez a kiválasztása feltételezheti, hogy mindegyikük végrehajtásához rendelkezésre áll adott M számú puffer. Azonban ahogyan azt már láttuk,

a pufferkezelő nem biztos, hogy képes garantálni ennek az M puffernek az elérhetőségét a lekérdezés végrehajtása alatt. Éppen ezért két, egymással szorosan összefüggő kérdés merül fel a fizikai műveletekkel kapcsolatosan:

1. Tud-e az algoritmus alkalmazkodni M-nek, az elérhető memóriapufferek számának a változásához?
2. Ha az előzetesen elvárt M puffer nem áll rendelkezésre, és ezért néhány blokkot a pufferkezelő lemezre ír, amelyekre pedig a memóriában számítottunk, akkor a puffercserélési stratégiánk hogyan befolyásolja az emiatt szükséges további I/O-műveletek számát?

6.26. példa: A fenti kérdések megvilágításához tekintsük a 6.13. ábrán szereplő, blokk alapú, egymásba ágyazott ciklusos összekapcsolást. Az alapalgoritmus nem függ M értékétől, a végrehajtás hatékonysága azonban igen. Így M értékét elég meghatározni közvetlenül a végrehajtás elkezdése előtt.

Az is előfordulhat, hogy M értéke változni fog a külső ciklus különböző iterációinál. Ez azt jelenti, hogy amikor betöltjük a memóriába S-nek, a külső ciklus relációjának egy részét, akkor egy kivételével az összes szabad puffert felhasználhatjuk. A fennmaradó egy puffert a belső ciklus relációjának, R-nek tartjuk fenn. Így a külső ciklusbeli iterációk száma attól függ, hogy átlagosan hány puffer szabad az egyes iterációk kezdetén. Mindaddig, amíg átlagosan M puffer szabad, addig a 6.4.4. részben kihozott költségelemzésünk érvényes marad. Szélsőséges esetben olyan szerencsénk is lehet, hogy az első iteráció alkalmával annyi szabad puffer áll rendelkezésre, hogy az egész S-et be tudjuk olvasni, és ebben az esetben a beágyazott cikluson alapuló összekapcsolás a 6.3.3. részbeli egymenetes összekapcsolássá egyszerűsödik.

Ha rögzítjük azt az $M-1$ blokkot, amikbe S részeit olvassuk be, akkor az iteráció alatt ezeket a puffereket biztosan nem fogjuk elveszíteni a memóriából. Az iteráció alatt emellett még további pufferek is szabaddá válhatnak. Ezek a pufferek lehetővé teszik, hogy R-nek több mint egy blokkját tartsuk egy időben a memóriában, de ha nem vagyunk elég körültekintőek, akkor ezek a további pufferek nem fogják javítani az összekapcsolás futási idejét.

Tegyük fel például, hogy az LRU puffercserélési stratégiát alkalmazzuk, és k puffer áll rendelkezésünkre az R blokkjainak tárolására. Ha sorban egymás után olvassuk be R blokkjait, akkor az iteráció végén a pufferekben R utolsó k blokkja fog maradni. Ezután betöltjük S következő $M-1$ blokkját, majd az iteráció következő lépésében ismét elkezdjük R blokkjait beolvasni. Ha azonban megint elölről kezdjük olvasni R blokkjait, akkor a k pufferben levő blokkot felül kell írnunk, és nem takarítunk meg egyetlen I/O-műveletet sem abból adódóan, hogy $k > 1$.

A beágyazott ciklusos összekapcsolásnak egy jobb megvalósítása az, amelyik váltakozó sorrendben olvassa be R blokkjait. Először az elsőtől az utolsóig, majd az utolsótól az elsőig. Ily módon, ha k puffer áll rendelkezésünkre R számára, akkor a külső ciklus minden iterációjakor k darab lemez I/O-műveletet takarítunk meg (kivéve az első iterációt). Vagyis a második és az azt követő iterációknak csak $B(R) - k$ lemezműveletre van szükségük R beolvasásához. Vegyük észre, hogy még $k = 1$ esetén is

(amikor nincsenek további puffereink R számára) megtakarítunk egy lemezműveletet minden iterációnál. □

A többi algoritmust szintén befolyásolja a pufferkezelő által választott puffercserélési stratégia, és az a tény, hogy M értéke változhat. Az alábbiakban néhány hasznos észrevételt közlünk:

- Ha valamelyik operátorhoz rendezésen alapuló algoritmust használunk, akkor alkalmazkodni tudunk M változásaihoz. Ha M értéke csökken, megváltoztathatjuk a részlisták méretét, mivel az általunk tárgyalt rendezés alapú algoritmusok számára nem volt lényeges, hogy a részlisták azonos méretűek legyenek. A legfontosabb korlátozás az lehet, hogy M csökkenésével olyan sok részlistát kell létrehoznunk, hogy nem fogunk tudni mindegyikük számára egy puffert lefoglalni az összefésülés fázisában.
- A részlisták memóriában történő rendezését különféle algoritmusokkal végezhetjük. Mivel az olyan algoritmusok, mint az összefésüléses rendezés és a gyorsrendezés rekurzívak, így az idő nagy részében viszonylag kis memóriaterületen dolgoznak. Ezért akár az LRU-, akár a FIFO-módszer jól fog működni az algoritmusnak ebben a rendezéssel foglalkozó részében.
- Ha az algoritmus tördelésen alapul, akkor M csökkenése esetén csökkenthetjük a kosarak számát egészen addig, amíg azok nem válnak olyan nagyméretűvé, hogy nem férnek el a lefoglalt memóriában. Itt azonban, a rendezésen alapuló algoritmusoktól eltérően, az algoritmus futása közben már nem tudunk reagálni M változásaira. Ha egyszer a kosarak számát meghatároztuk, akkor az rögzített marad egészen az első menet végéig. Így ha nincs több elérhető szabad puffer, akkor egyes kosarakhoz tartozó blokkokat kénytelenek vagyunk kiírni a lemezre.

6.8.4. Feladatok

6.8.1. feladat: Tegyük fel, hogy az R ⋈ S összekapcsolást szeretnénk végrehajtani, és ezalatt a rendelkezésre álló memória mérete M és $M/2$ között fog változni. Adjuk meg M, $B(R)$ és $B(S)$ segítségével kifejezve azokat a feltételeket, amelyek mellett a következő algoritmusok garantáltan végrehajthatók:

* a) Egymenetes összekapcsolás.
* b) Kétmenetes, tördelésen alapuló összekapcsolás.
c) Kétmenetes, rendezésen alapuló összekapcsolás.

! **6.8.2. feladat:** Mennyivel csökkenne az egymásba ágyazott cikluson alapuló összekapcsolás által elvégzett lemez I/O-műveletek száma, ha további pufferek állnának rendelkezésre és a puffercserélési módszer a következő lenne:

a) Elsőként bejövő-elsőként kimenő (FIFO).
b) Óra algoritmus.

!! **6.8.3. feladat:** A 6.26. példában azt javasoltuk, hogy az összekapcsolás alatt szabaddá váló további puffereket úgy használjuk fel, hogy az R-nek egynél több blokkját tartsuk a pufferben, és a külső ciklus minden páros számúadik iterációjakor R blokkjait fordított sorrendben vegyük. Másik lehetőségként megtehetjük, hogy az R számára továbbra is egy puffert tartunk fenn, és az S tárolására használt pufferek számát növeljük. Melyik stratégia esetén van szükség a legkevesebb lemez I/O-műveletre?

6.9. Több mint kétmenetes algoritmusok

Tudjuk, hogy két menet a műveletek számára elegendő, ha a relációk nem nagyon nagyok. Vegyük azonban észre, hogy a 6.5. és 6.6. részekben tárgyalt technikák olyan algoritmusokká általánosíthatók, amelyek tetszőleges méretű relációkra alkalmazhatók, szükség esetén több menetet használva. Ebben a szakaszban mind a rendezésen alapuló, mind a tördelésen alapuló módszerek általánosítását tárgyalni fogjuk.

6.9.1. Többmenetes, rendezésen alapuló algoritmusok

A 2.3.5. részben utaltunk rá, hogy a kétfázisos, összefésüléses rendezést hogyan lehet kiterjeszteni hárommenetes algoritmussá. Van egy egyszerű rekurzív megközelítése a módszernek, amellyel tetszőlegesen nagy relációt rendezni tudunk. A rendezést úgy is el tudjuk végezni, hogy teljesen rendezzük a relációt, vagy ha arra van szükségünk, akkor n darab rendezett részlistát is létre tudunk hozni vele, tetszőleges n-re.

Tegyük fel, hogy az R reláció rendezéséhez rendelkezésünkre áll M darab puffer. Azt is feltesszük még, hogy a relációt nyaláboltan tároltuk. Ekkor a következőket kell tennünk:

Alap: Ha R belefér az M darab blokkba (vagyis ha $B(R) \leq M$), akkor beolvassuk R-et a memóriába, rendezzük valamilyen rendezési algoritmussal, majd a rendezett relációt kiírjuk a lemezre.

Indukció: Ha R nem fér be a memóriába, akkor osszuk fel R blokkjait M csoportba. Az egyes csoportokat jelöljük R_1, R_2, ..., R_M-mel. Rendezzük rekurzívan R_i-t minden i-re ($i = 1, 2, \ldots, M$). Ezután fésüljük össze az M darab rendezett részlistát, ahogyan azt a 2.3.4. részben láttuk.

Ha nem csupán rendeznünk kell R-et, hanem valamilyen egyoperandusú (unáris) műveletet szeretnénk végrehajtani rajta, mint amilyen pl. a γ vagy a δ, akkor módosítsuk a fentieket oly módon, hogy az utolsó összefésüléskor a rendezett részlisták elején levő sorokra elvégezzük a megfelelő műveletet. Vagyis,

- δ esetén minden sornak egy példányát kiírjuk, a többi előfordulását pedig figyelmen kívül hagyjuk.

- γ esetén csak a csoportképző attribútumok szerint rendezünk, és azokat a sorokat, amelyek ezeken az attribútumokon megegyeznek, a szokásos módon összesítjük, ahogyan azt a 6.5.2. részben láttuk.

Ha egy kétoperandusú (bináris) műveletet szeretnénk elvégezni, pl. metszet vagy összekapcsolás, akkor tulajdonképpen ugyanezt az ötletet alkalmazzuk, azzal a különbséggel, hogy először a két relációt összesen M részlistába osztjuk szét. Ezután minden részlistát a fenti rekurzív módszerrel rendezünk. Végül beolvassuk az M darab részlistát a pufferekbe, és elvégezzük a megfelelő műveletet úgy, ahogyan azt a 6.5. rész megfelelő részében bemutattuk.

Az M darab puffert tetszőlegesen oszthatjuk szét az R és S relációk között. A menetek számát azonban úgy minimalizálhatjuk, ha a puffereket a relációk méretével arányosan osztjuk szét. Vagyis R kap $M \times B(R)/(B(R) + B(S))$ darab puffert, a többit pedig S-hez rendeljük.

6.9.2. Többmenetes, rendezésen alapuló algoritmusok műveletigénye

Az alábbiakban megvizsgáljuk, hogy milyen összefüggés van a szükséges lemez I/O-műveletek száma, a relációk mérete és a memória mérete között. Jelöljük $s(M, k)$-val annak a legnagyobb relációnak a méretét, amelyet M darab puffer segítségével k menetben rendezni tudunk. Ekkor $s(M, k)$-t a következőképpen számíthatjuk ki:

Alap: Ha $k = 1$, vagyis 1 menet elegendő, akkor szükségképpen $B(R) \leq M$. Másképpen megfogalmazva, $s(M, 1) = M$.

Indukció: Tegyük fel, hogy $k > 1$. Ekkor felosztjuk R-et M részre, ahol szükségképpen minden rész $k - 1$ menet alatt rendezhető. Ha $B(R) = s(M, k)$ akkor $s(M, k)/M$, ami az egyes részek mérete, nem lehet nagyobb mint $s(M, k - 1)$. Vagyis $s(M, k) = M\, s(M, k - 1)$.

Ha a fenti rekurziót tovább folytatjuk, a következőt kapjuk:

$$s(M, k) = M\, s(M, k - 1) = M^2\, s(M, k - 2) = \ldots = M^{k-1}\, s(M, 1)$$

Mivel $s(M, 1) = M$, ebből azt kapjuk, hogy $s(M, k) = M^k$. Ez azt jelenti, hogy k menet alatt akkor tudunk egy R relációt rendezni, ha $B(R) \leq s(M, k)$, ami azt jelenti, hogy $B(R) \leq M^k$. Másképpen megfogalmazva, ha egy R relációt k menet alatt akarunk rendezni, akkor ehhez a minimálisan szükséges memóriapufferek száma: $M = B(R)^{1/k}$.

Egy rendezési algoritmus minden menete beolvassa az összes adatot, majd ismét kiírja azokat lemezre. Így egy k menetes rendező algoritmusnak $2k\, B(R)$ lemez I/O-műveletre van szüksége.

Most vizsgáljuk meg egy többmenetes $R(X, Y) \bowtie S(Y, Z)$ összekapcsolás költségét, ami jó példája a kétoperandusú műveleteknek. Legyen $j(M, k)$ az a maximális blokkszám, amely mellett össze tudunk kapcsolni két relációt k menetben, M darab puffer használatával, ha a relációknak összesen legfeljebb $j(M, k)$ blokkja van. Ez azt jelenti, hogy az összekapcsolás elvégezhető ha $B(R) + B(S) \leq j(M, k)$.

Az utolsó menetben összefésüljük a két reláció M darab rendezett részlistáját. A részlisták mindegyikét $k - 1$ menetben rendeztük, így egyikük hossza sem lehet több, mint $s(M, k - 1)$, vagyis az összméret maximum $Ms(M, k) = M^k$. Eszerint $B(R) + B(S)$ nem lehet nagyobb, mint M^k, vagyis $j(M, k) = M^k$. A paraméterek szerepét felcserélve azt is kimondhatjuk, hogy a k menetes összekapcsoláshoz legalább $(B(R) + B(S))^{1/k}$ darab pufferre van szükség.

A lemez I/O-műveletek összeszámolásakor ne feledjük, hogy az összekapcsolásnál és más relációs műveleteknél a végeredmény lemezre írásának költségét nem számoljuk, ellentétben a rendezésnél alkalmazott gyakorlattal. Így a részlisták rendezéséhez $2(k - 1)(B(R) + B(S))$ lemez I/O-műveletet használunk, míg a végső rendezett részlisták beolvasásához további $B(R) + B(S)$ darabot. A végeredmény összesen $(2k - 1)(B(R) + B(S))$ lemez I/O-művelet.

6.9.3. Többmenetes, tördelésen alapuló algoritmusok

Van egy hasonló, rekurzív megközelítése a tördelésen alapuló műveleteknek is, ha azokat nagyon nagy relációkon végezzük. Ha M a rendelkezésre álló memóriapuffek száma, akkor osszuk fel a relációt a tördeléssel $M - 1$ kosárba. Ezután alkalmazzuk a műveletet az egyes kosarakra egyenként, amennyiben a művelet egyoperandusú. Ha a művelet kétoperandusú, mint pl. egy összekapcsolás, akkor a megfelelő kosarakból álló párokra alkalmazhatjuk azt, mintha azok maguk volnának a teljes relációk. Az eddig tárgyalt relációs műveletek esetén – ismétlődések eltüntetése, csoportosítás, egyesítés, metszet, különbség, természetes összekapcsolás, egyenlőséges összekapcsolás – a teljes relációkra alkalmazott művelet végeredménye meg fog egyezni a kosarakra alkalmazott műveletek egyesítésével. Ezt a megközelítést rekurzívan a következőképpen írhatjuk le:

Alap: Egyoperandusú művelet esetén, ha a reláció belefér az M pufferbe, akkor olvassuk be a memóriába, és végezzük el a műveletet. Kétoperandusú művelet esetén, ha bármelyik reláció befér $M - 1$ pufferbe, akkor végezzük el a műveletet úgy, hogy ezt a relációt beolvassuk a memóriába, majd a másik relációt blokkonként beolvassuk az M-edik pufferbe.

Indukció: Ha egyik reláció sem fér be a memóriába, akkor mindkettőt osszuk fel tördeléssel $M - 1$ kosárba úgy, ahogyan azt a 6.6.1. részben láttuk. Végezzük el a műveletet rekurzívan mindegyik kosárra, illetve a megfelelő kosarakból álló párokra, majd gyűjtsük össze a kosarakból, illetve a párokból készülő eredmény kimenetét.

6.9.4. Többmenetes, tördelésen alapuló algoritmusok műveletigénye

A továbbiakban azzal a feltételezéssel fogunk élni, hogy a reláció tördelésekor a sorok a lehető legegyenletesebben oszlanak meg a kosarak között. A gyakorlatban ezt a feltételezést megközelíthetjük, ha tényleg véletlenszerűen tördelőfüggvényt választunk, de valójában mindig lesz bizonyos egyenletlenség a sorok eloszlása tekintetében.

Először nézzük az egyoperandusú műveleteket, mint pl. a γ vagy a δ. A relációt továbbra is R-rel, a memóriapufferek számát pedig M-mel jelöljük. Legyen $u(M, k)$ annak a legnagyobb relációnak a blokkszáma, amelyet egy k menetes tördelő algoritmus kezelni tud. u-t a következőképpen határozhatjuk meg rekurzívan:

Alap: $u(M, 1) = M$, hiszen az R relációnak be kell férnie az M pufferbe, vagyis $B(R) \leq M$.

Indukció: Tegyük fel, hogy az első lépés az R relációt $M - 1$ egyenlő méretű kosárba osztja szét. Ekkor $u(M, k)$-t a következőképpen számolhatjuk ki. A kosaraknak a következő lépés előtt elég kicsinek kell lenniük ahhoz, hogy őket $k - 1$ lépésben kezelni lehessen, vagyis a kosarak mérete legfeljebb $u(M, k - 1)$. Mivel R-et $M - 1$ kosárba osztottuk, így azt kapjuk, hogy $u(M, k) = (M - 1)u(M, k - 1)$.

Ha tovább folytatjuk a fenti gondolatmenetet, akkor azt kapjuk, hogy $u(M, k) = M(M - 1)^{k-1}$, illetve feltételezve, hogy M elegendően nagy, u-ra közelítőleg a következő adódik: $u(M, k) = M^k$. Ez másképpen megfogalmazva azt jelenti, hogy akkor tudjuk az R reláción M puffer segítségével elvégezni az egyoperandusú műveleteket k menetben, ha $M \leq (B(R))^{1/k}$.

Hasonló elemzést végezhetünk a kétoperandusú műveletekre is. Most is az összekapcsolást fogjuk példaképpen venni, mint a 6.9.2. részben. Jelölje $j(M, k)$ a $R(X, Y) \bowtie S(Y, Z)$ összekapcsolásban résztvevő R és S relációk közül a kisebbiknek a méretére vonatkozó felső korlátot. Most is M jelöli a rendelkezésre álló memóriapufferek számát, k pedig a menetek számát.

Alap: $j(M, 1) = M - 1$, vagyis ha az egy menetes algoritmust használjuk az összekapcsolásra, akkor vagy R-nek vagy S-nek be kell férnie $M - 1$ pufferbe. Ezt már láttuk a 6.3.3. részben.

Indukció: $j(M, k) = (M - 1)j(M, k - 1)$, hiszen az első menetben mindkét relációt $M - 1$ kosárba osztjuk, és elvárásaink szerint az egyes kosarak mérete az eredeti relációnak $1/(M - 1)$-ed része lesz, továbbá azt is tudjuk, hogy a megfelelő kosarakból álló párokra a műveletet $k - 1$ menetben el kell tudnunk végezni.

A fenti gondolatmenetet folytatva azt kapjuk, hogy $j(M, k) = (M - 1)^k$. Ismét feltételezve, hogy M elegendően nagy, j-re a következő közelítés adódik: $j(M, k) = M^k$. Ez azt jelenti, hogy akkor tudjuk az $R(X, Y) \bowtie S(Y, Z)$ összekapcsolást elvégezni k menetben M puffer segítségével ha $M^k \geq \min(B(R), B(S))$.

6.9.5. Feladatok

6.9.1. feladat: Tegyük fel, hogy $B(R) = 20\,000$, $B(S) = 50\,000$ és $M = 101$. Elemezzük az alábbi algoritmusok viselkedését, amelyek segítségével $R \bowtie S$-et állítjuk elő.

* a) Hárommenetes, rendezésen alapuló algoritmus.
b) Hárommenetes, tördelésen alapuló algoritmus.

! **6.9.2. feladat:** A korábbiakban említettünk néhány „trükköt", amelyek segítségével a kétmenetes algoritmusok hatékonyságát lehet javítani. Az alábbi esetekre mondjuk meg, hogy a trükk alkalmazható-e többmenetes algoritmusra, és ha igen, hogyan?

a) A 6.6.6. részben említett hibrid tördeléses összekapcsolásnál szereplő trükk.
b) A rendezésen alapuló algoritmusok javítása oly módon, hogy a blokkokat közvetlenül egymás után tároljuk a lemezen (6.6.7. rész).
c) A tördelésen alapuló algoritmusok javítása oly módon, hogy a blokkokat közvetlenül egymás után tároljuk a lemezen (6.6.7. rész).

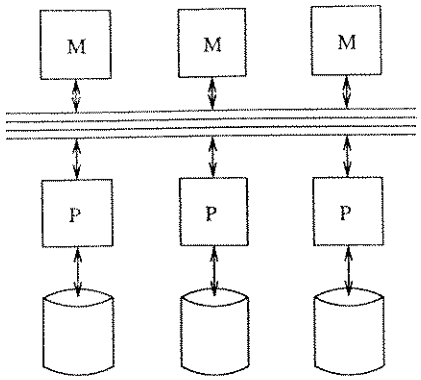
6.10. Párhuzamos algoritmusok relációs műveletekre

A gyakran időigényes és nagy adattömeggel dolgozó adatbázis-műveletek elvégzése során általában előnyt jelent a párhuzamos feldolgozás. Ebben a fejezetben áttekintjük a párhuzamos gépek főbb architektúráit. Ezt követően a „megosztás nélküli" architektúrával fogunk a legnagyobb terjedelemben foglalkozni, mert az adatbázis-műveletek terén ez tűnik a leghatékonyabbnak a költségeket tekintve, még akkor is, ha más párhuzamos alkalmazásokra ez nem feltétlenül a legjobb. A legtöbb relációs művelet szokványos algoritmusának léteznek olyan egyszerű módosításai, amelyek szinte tökéletesen kihasználják a párhuzamosság nyújtotta előnyöket. Ez alatt azt értjük, hogy egy p processzoros gépen egy művelet elvégzése nagyjából $1/p$-szer annyi ideig tart csupán, mintha ugyanezt egy egyprocesszoros gépen tennénk meg.

6.10.1. A párhuzamosság modelljei

A párhuzamos gépek működésének középpontjában processzorok egy halmaza áll. A processzorok p száma igen nagy, elérheti akár a százas vagy ezres nagyságrendet is. Azt fogjuk feltenni, hogy minden processzornak megvan a maga lokális gyorsítótára, amelyet ábráinkon explicit módon nem tüntetünk fel. A legtöbb elrendezésben minden egyes processzornak van lokális memóriája is, amit viszont az ábrákon is szerepeltetünk. Az adatbázis-feldolgozás szempontjából nagy fontossággal bír az a tény, hogy a processzorok mellett számos lemez is szerephez jut, processzoronként egy vagy akár még több is, illetve bizonyos architektúrákban lemezek egy nagyobb számú együttese az összes processzor számára közvetlenül elérhető.

A fentiek mellett a párhuzamos számítógépeknek minden esetben van még valamilyen kommunikációs eszközük is, amellyel a processzorok között információkat tudnak cserélni. Az itt szereplő ábrákon a kommunikációt úgy mutatjuk be, mintha a gép összes elemére létezne egy megosztott busz. A gyakorlatban azonban egy busz nem tud összekötni annyi processzort vagy egyéb elemet, mint amennyi a nagy gépekben ténylegesen megtalálható, így az összekötő rendszer sok architektúrában egy

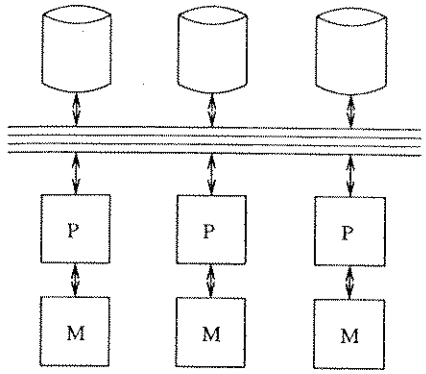


6.23. ábra. *Egy megosztott memóriájú gép*

nagy teljesítményű kapcsoló (switch), amelyet esetleg még kiegészítenek olyan buszok, amelyek a processzorokból képzett részhalmazokat lokális fürtökbe (cluster) kötik össze.

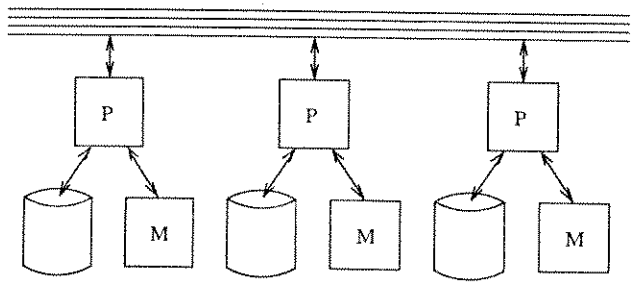
A párhuzamos számítógépek három legfontosabb osztálya a következő.

1. *Megosztott memória* (shared memory). Ebben az architektúrában, melynek sémáját a 6.23. ábra mutatja, minden processzor hozzáférhet minden processzor összes memóriájához. Ez azt jelenti, hogy a processzoronkénti egy címtartomány helyett az egész gépre egyetlen fizikai címtartomány létezik. A 6.23. ábrán látható rajz tulajdonképpen eltúlozza a valóságot, hiszen úgy tűnhet, mintha a processzoroknak nem is lenne saját memóriájuk. Valójában azonban minden processzornak van valamennyi lokális memóriája, amit ha csak lehet, ki is használ. Ugyanakkor szükség esetén adva van a többi processzor memóriájához való hozzáférés lehetősége is. Az ebbe az osztályba tartozó nagy gépek *NUMA* (nonuniform memory access) típusúak, ami azt jelenti, hogy egy processzor számára a „saját" (vagy a lokális clusterbe tartozó processzorok) memóriájában lévő adatok elérése gyorsabb, mint egy másik processzorhoz tartozó memóriabeli adatok elérése. Mindezekkel együtt a jelenleg használatos architektúrákban a memóriaelérési idők különbsége nem jelentős. Inkább arról van szó, hogy a memória elérése – bármilyen adatokról legyen is szó – sokkal hosszabb időt vesz igénybe, mint a gyorsítótár elérése, a kritikus pont tehát az, hogy a processzor számára szükséges adatok vajon saját gyorsítótárában vannak-e vagy sem.
2. *Megosztott lemez* (shared disk). Amint arra a 6.24. ábra is utal, ebben az architektúrában minden processzornak megvan a maga memóriája, amelyeket más processzorok közvetlenül nem érhetnek el. Ugyanakkor a lemezeket a kommunikációs hálózaton keresztül bármelyik processzor elérheti. A különböző processzorok esetleg egymással ütköző kéréseinek kezelése a lemezvezérlők feladata. A lemezeknek és a processzoroknak a száma nem feltétlenül kell hogy azonos legyen, noha a 6.24. ábra talán azt sugallhatja.



6.24. ábra. *Egy megosztott lemezes gép*

3. *Megosztás nélküli* (shared nothing). Ebben az esetben minden egyes processzornak megvan a maga saját memóriája és lemeze vagy lemezei, amint azt a 6.25. ábra mutatja. A processzorok közötti kommunikáció minden formája a kommunikációs hálózaton keresztül történik. Ha például egy *P* processzor egy másik *Q* processzor lemezéről akar sorokat olvasni, akkor *P* elküldi *Q* részére az adatkérést tartalmazó üzenetet. Ezt követően *Q* saját lemezéről megszerzi a sorokat, majd azokat a hálózaton keresztül egy másik üzenetben elküldi *P* részére, amely azt átveszi.



6.25. ábra. *Egy megosztás nélküli gép*

Amint azt a mostani rész bevezetésében már említettük, a megosztás nélküli architektúra a leggyakrabban használt modell az „adatbázisgépek"-nél, vagyis a speciálisan adatbázisok támogatására tervezett párhuzamos számítógépeknél. A megosztás nélküli gépek építése viszonylag olcsó, azonban amikor algoritmusokat tervezünk ezekre a gépekre, akkor tudatában kell lennünk annak, hogy egy processzortól egy másikhoz adatokat küldeni bizony költséges.

Normális esetben az adatokat a processzorok között üzenet formájában küldjük, ami jelentős többletköltséggel jár. Mindkét processzornak futtatnia kell egy, az üzenetek átvitelét támogató programot, sőt a kommunikációs hálózatban is lehetnek ütkö-

Algoritmusok más párhuzamos architektúrákra

A megosztott lemezes gép a hosszú üzeneteket részesíti előnyben, éppen úgy, mint a megosztás nélküli gépek. Ha az összes kommunikáció lemezen keresztül történik, akkor az adatokat blokkméretű darabokban kell mozgatnunk, és ha el tudjuk érni, hogy az ilyen formában mozgatni kívánt adatok egyetlen sávban vagy cilinderen legyenek, akkor a fejbeállási időből sokat le tudunk faragni, amint azt már a 2.4.1. részben is láttuk.

A megosztott memóriájú gép ezzel szemben lehetővé teszi, hogy bármely két processzor a memórián keresztül kommunikáljon egymással. Az üzenet küldéséhez nincs szükség bonyolult szoftverre, az elsődleges memória olvasásának vagy írásának költsége pedig arányos az érintett bájtok számával. A megosztott memóriájú gépek így előnyösen használhatják azokat az algoritmusokat, amelyek gyors, gyakori és rövid kommunikációt igényelnek a processzorok között. Érdekes megfigyelni, hogy noha más területeken ismertek ilyen algoritmusok, úgy tűnik az adatbázis-feldolgozásban mégsincs rájuk szükség.

zések vagy késedelmek. Egy üzenet költségét általában le lehet bontani egy nagyobb fix részre, és egy, az átküldött bájtok mennyiségével arányos kisebb részre. Ezért a párhuzamos algoritmusokat általában úgy célszerű megtervezni, hogy a processzorok közötti kommunikációkban egyszerre nagy mennyiségű adatot küldjünk át. Megtehetjük például, hogy néhány a Q processzornak szánt adatblokkot a P processzoron pufferelünk. Ha Q-nak nincs rögtön szüksége az adatokra, akkor sokkal hatékonyabb, ha várunk addig, amíg P-n összegyűlik egy hosszú üzenet, és csak ekkor küldjük el azt Q-nak.

6.10.2. Soronkénti műveletek párhuzamos megvalósítása

Vizsgálódásunkat kezdjük azzal, hogy szemügyre vesszük egy megosztás nélküli gép párhuzamos algoritmusait a kiválasztás műveletre vonatkozóan. Először nézzük meg azt, hogy az adatok tárolása milyen formában a legelőnyösebb. A 2.4.2. részben már megemlítettük, hogy hasznos, ha az adatokat a lehető legtöbb lemezen tudjuk szétosztani. Az egyszerűség kedvéért feltesszük, hogy processzoronként egy lemezünk van. Ekkor, ha összesen p számú processzor van, akkor bármely R reláció sorait a p darab processzor lemezei között egyenletesen osztjuk el.

Tegyük fel továbbá, hogy $\sigma_C(R)$-t akarjuk kiszámítani. Az egyes processzorokat használhatjuk arra, hogy mindegyik végignézze R-nek a saját lemezén található sorait. A processzorok megkeresik a C feltételnek eleget tevő sorokat, majd azokat a kimenetbe másolják. A processzorok közötti kommunikáció elkerülésére $\sigma_C(R)$-nek a t sorait ugyanazon processzornál tároljuk, amelynek a lemezén t megtalálható. Így az eredményként kapott $\sigma_C(R)$ reláció is szét van osztva a lemezek között, csakúgy mint maga az R reláció.

Mivel $\sigma_C(R)$ lehet éppenséggel egy másik művelet bemeneti relációja is, és mivel arra törekszünk, hogy az eltelt időt minél rövidebbre szorítsuk, illetve, hogy az összes processzort állandóan foglalkoztassuk, ezért azt szeretnénk, ha $\sigma_C(R)$ egyenlően lenne elosztva a processzorok között. Ha kiválasztás helyett vetítést végeznénk, akkor $\pi_L(R)$-nek az egyes processzoroknál levő sorainak száma ugyanannyi lenne, mint ahány sora R-nek volt az adott processzornál. Más szóval, ha R egyenletesen volt elosztva, akkor ez igaz marad a vetítésére is. A kiválasztás azonban alaposan megváltoztathatja a sorok eloszlását az eredményben, R eloszlásához képest.

6.27. példa: Vegyük a $\sigma_{a=10}(R)$ kiválasztást, vagyis R azon sorait keressük, amelyeknek az a attribútumon (R egyik attribútumán) vett értéke 10. Tegyük fel továbbá, hogy R-et az a attribútuma alapján osztottuk szét. Ekkor R összes $a = 10$ értékű sora a processzorok egyikénél van, és így a teljes $\sigma_{a=10}(R)$ reláció is egyetlen processzornál lesz. □

A 6.27. példában felmerült probléma elkerülése céljából gondoljuk át alaposan, hogy a tárolt relációinkat miképpen szeretnénk szétosztani a processzorok között. A legjobb megoldás talán egy olyan h tördelőfüggvény használata, amely egy sor összes komponensét figyelembe veszi, oly módon, hogy t egy komponensének a megváltozására $h(t)$ bármelyik lehetséges kosárszámot felveheti.[8] Ha például B kosarat szeretnénk, akkor megpróbálhatjuk valamilyen módon az egyes komponenseket 0 és $B-1$ közötti egész számmá konvertálni, összeadni az egyes komponensek egész értékeit, az eredményt elosztani B-vel, és a maradékot venni a kosár számaként. Ha a processzorok száma szintén B, akkor minden processzort egy kosárhoz rendelhetünk, a kosár tartalmát pedig a processzornak adhatjuk.

6.10.3. Teljes relációs műveletek párhuzamos algoritmusai

Foglalkozzunk először a $\delta(R)$ művelettel, amely némileg eltér a teljes relációs műveletek alaptípusától. Ha olyan tördelőfüggvényt használunk, amely R sorait a 6.10.2. részben ismertetett eljárás szerint osztja szét, akkor R sorainak másodpéldányait ugyanahhoz a processzorhoz tesszük. Ha így járunk el, akkor $\delta(R)$-et előállíthatjuk párhuzamosan egy szokványos, egyprocesszoros algoritmussal (mint pl. a 6.5.1. és a 6.6.2. részekben), amit R-nek az egyes processzoroknál található darabjaira alkalmazunk. Hasonlóan, ha R és S sorainak elosztására ugyanazokat a tördelőfüggvényeket használjuk, akkor R és S egyesítését, metszetét vagy különbségét megkaphatjuk úgy, hogy az egyes processzorok R és S darabjain párhuzamosan dolgoznak.

Tegyük fel most viszont, hogy R és S elosztása nem ugyanazzal a tördelőfüggvénnyel történt, és egyesítésüket ezt követően szeretnénk kiszámítani.[9] Ebben az eset-

[8] Pont arról van szó, hogy *nem* akarunk particionált tördelőfüggvényt használni (ezt az 5.2.5. részben tárgyaltuk), mert az az összes olyan sort, amelyek egy attribútumon azonos értékkel rendelkeznek – mondjuk az $a = 10$-zel – a kosaraknak csak egy kis részébe tenné.

[9] Elvben itt lehet szó akár halmaz, akár multihalmaz egyesítéséről. A 6.3.3. rész egyszerű multihalmaz-egyesítési technikája, ahol mindkét argumentum összes sorát másoltuk, párhuza-

ben először le kell másolnunk R és S összes sorát, majd azokat egyetlen h tördelőfüggvény szerint kell elosztani.[10]

Párhuzamosan dolgozva, R és S sorait minden egyes processzornál tördeljük a h tördelőfüggvény szerint. A tördelés a 6.6.1. részben leírtak szerint történik, azonban amikor a j jelű processzorban az i kosárnak megfelelő puffer megtelik, akkor ahelyett, hogy azt a j-nél levő lemezre vinnénk át, a tartalmát az i jelű processzorhoz visszük. Ha az elsődleges memóriában kosaranként több-blokknyi helyünk van, akkor esetleg megvárhatjuk, míg az i jelű kosár sorai megtöltenek néhány puffert, és csak akkor visszük azokat az i jelű processzorhoz.

Az i jelű processzor tehát megkapja R és S összes, az i jelű kosárba tartozó sorát. A második lépésben minden processzor elvégzi a kosarához tartozó R és S-beli sorok egyesítését. Az eredményként kapott $R \cup S$ reláció a processzorok között egyenletesen lesz elosztva. Ha a h tördelőfüggvény ténylegesen véletlenszerűen rakja a sorokat kosarakba, akkor azt várhatjuk, hogy az egyes processzoroknál $R \cup S$-nek nagyjából azonos számú sora lesz.

A metszet és a különbség műveletek az egyesítéshez hasonlóan végezhetők el. Az, hogy ezeknek a műveleteknek a halmaz- vagy a multihalmaz-változatáról van-e szó, tulajdonképpen lényegtelen. Az eddigieket kiegészíthetjük még a következőkkel:

- Az $R(X, Y) \bowtie S(Y, Z)$ összekapcsolás kiszámításánál R és S sorait a processzorok számával megegyező számú kosárba tördeljük. Az általunk használt h tördelőfüggvénynek azonban csak Y attribútumaitól szabad függenie, nem pedig az összes attribútumtól, hogy az összekapcsolt sorok mindig ugyanabba a kosárba kerüljenek. Az egyesítéshez hasonlóan az i jelű kosár sorait az i jelű processzorhoz küldjük. Ezt követően minden processzorral elvégezhetjük az összekapcsolást a jelen fejezetben leírt bármelyik egyprocesszoros összekapcsolási algoritmust használva.
- A csoportosítás vagy a $\gamma_L(R)$ összesítés elvégzéséhez R sorait egy olyan h tördelőfüggvény segítségével osztjuk el, amely csak az L listán lévő csoportosító attribútumoktól függ. Ha az egyes processzorokban a h egy kosarának megfelelő összes sor rendelkezésre áll, akkor a γ_L műveleteket ezeken a sorokon lokálisan is elvégezhetjük bármely egyprocesszoros γ algoritmussal.

6.10.4. A párhuzamos algoritmusok hatékonysága

Térjünk most rá annak vizsgálatára, hogy hogyan viszonyul egymáshoz egy p processzoros gépen végrehajtott párhuzamos algoritmus futási ideje, és az ugyanezeken az adatokon végrehajtott azonos művelet futási ideje egyprocesszoros gépen végrehajtva. A teljes munka – a lemez I/O-műveletek száma és a processzorciklusok száma

mosan is működik, ezért az itt bemutatott algoritmust nem különösebben érdemes multihalmaz-egyesítésre használni.

[10] Ha akár az R vagy az S reláció sorainak elosztására használt tördelőfüggvény ismert, akkor azt használhatjuk a másik relációra is, és akkor nem kell szétosztanunk mindkét relációt.

Naaagy hiba!

Ha műveletek elvégzésére vagy relációk processzorok közötti elosztására tördelésen alapuló algoritmusokat használunk – mint a 6.28. példában –, akkor ügyelnünk kell arra, hogy nehogy túlzásba vigyük egy tördelőfüggvény használatát. Tegyük fel ugyanis, hogy az R és az S relációk sorait a h tördelőfüggvénnyel osztottuk el a processzorok között, hogy vehessük az összekapcsolásukat. Kecsegtető lehetne, hogy ugyanezt a h függvényt használjuk lokálisan S sorainak kosarakba tördeléséhez akkor, amikor az egyes processzoroknál az egymenetes tördeléses összekapcsolást végezzük. Ha azonban így teszünk, akkor az összes ilyen sor ugyanabba a kosárba kerül, és a 6.28. példában javasolt memóriában történő összekapcsolás rendkívül rossz hatékonyságú lenne.

– a párhuzamos gépnél sem lehet kisebb, mint az egyprocesszorosnál. Ugyanakkor, mivel a p processzor p számú lemezzel dolgozik, a ténylegesen eltelt időt a multiprocesszoros esetben sokkal rövidebbnek várjuk, mint egy processzor esetén.

Egy unáris művelet – pl. $\sigma_C(R)$ – elvégzéséhez az egyprocesszoros végrehajtás idejének $1/p$-ed része elegendő, feltéve, hogy a reláció egyenletesen van elosztva, ahogyan azt a 6.10.2. részben feltettük. A lemez I/O-műveletek száma lényegében megegyezik az egyprocesszoros kiválasztásával. Az egyetlen különbség az, hogy átlagosan p darab félig telt blokkja lesz R-nek, minden egyes processzornál ahelyett, hogy egyetlen félig tele blokkja lenne. Ez utóbbi eset akkor állna elő, ha a teljes R-et egy processzornál tároltuk volna.

Nézzünk most egy bináris műveletet, mondjuk az összekapcsolást. Olyan tördelőfüggvényt használunk az összekapcsolási attribútumokon, amely az egyes sorokat a p darab kosár egyikébe teszi, ahol p a processzorok száma. Ahhoz, hogy az i jelű kosárba tartozó sorokat (minden i-re) az i jelű processzorhoz küldjük, az kell, hogy minden egyes sort a lemezről a memóriába olvassunk, kiszámoljuk a tördelőfüggvényt, majd minden sort a megfelelő helyre küldjünk. Átlagosan minden p darab sorból egy olyan lesz, amelyik a saját processzorának a kosarába kerül, és így nem kell elküldeni. Ha $R(X, Y) \bowtie S(Y, Z)$-t számoljuk, akkor R és S sorainak a beolvasásához és a kosarak meghatározásához $B(R) + B(S)$ lemez I/O-művelet szükséges.

Ha ez megvolt, akkor a hálózaton keresztül el kell küldenünk a $((p-1)/p)(B(R) + B(S))$ számú blokkot a megfelelő processzorhoz. Csupán a már amúgy is a jó processzornál lévő $(1/p)$-nyi sort nem kell mozgatni. A küldözgetés költsége lehet nagyobb vagy kisebb is az ugyanennyi számú lemez I/O-művelet költségénél, mindez a gép architektúrájától függ. Mi azt fogjuk feltenni, hogy a hálózaton keresztül történő mozgatás jóval olcsóbb, mint a lemez és a memória közötti adatmozgatás, hiszen az előbbihez nem kell semmilyen fizikai mozgás, a lemez I/O-művelethez viszont kell.

Elviekben feltételezhetjük, hogy a fogadó processzor az adatokat a saját lemezén tárolja, majd elvégzi a kapott sorok lokális összekapcsolását. Ha például minden processzornál kétmenetes rendezéses összekapcsolást végzünk, akkor egy naiv párhuza-

mos algoritmus minden processzornál $3(B(R) + B(S))/p$ számú lemez I/O-műveletet használna, hiszen a relációk mérete minden kosárban nagyjából $B(R)/p$ és $B(S)/p$ lenne, és ez a fajta összekapcsolás az argumentum relációk által elfoglalt minden blokkhoz három lemez I/O-műveletet használ. Ehhez még hozzá kell adnunk processzoronként újabb $2(B(R) + B(S))/p$ lemez I/O-műveletet, ami annak felel meg, amikor az egyes sorokat először beolvassuk, majd amikor a tördelés és az elosztás során a processzorokkal fogadjuk és tároljuk a sorokat. Ehhez jönne még az adatok mozgatásának költsége, de az előbbiekben már úgy döntöttünk, hogy ez az adatok lemez I/O-művelet költsége mellett elhanyagolható.

A fenti összevetés kiemeli a multiprocesszoros séma értékét. Noha összességében több lemez I/O-műveletet végzünk – konkrétan három helyett ötöt adatblokkonként – az egyes processzoroknál végrehajtott lemez I/O-műveletek számában mért eltelt idő azonban $3(B(R) + B(S))$-ről $5(B(R) + B(S))/p$-re csökkent, ami nagy p esetén jelentős nyereséggel jár.

Vannak emellett olyan javítási módszerek is, amelyek úgy növelik meg a párhuzamos algoritmus sebességét, hogy az összes szükséges lemez I/O-műveletek száma se haladja meg az egyprocesszoros algoritmusét. Valóban, mivel minden processzornál kisebb relációkkal dolgozunk, esetleg használhatunk olyan lokális összekapcsolási algoritmust, amely az adatblokkokra kevesebb lemez I/O-műveletet végez. Például, ha R és S olyan nagyok lennének is, hogy az egy processzoros elrendezésben kétmenetes algoritmust kellene használnunk, akkor is elképzelhető, hogy az adatok $(1/p)$-ed részére már az egymenetes algoritmus is elegendő lenne.

Blokkonként két lemez I/O-művelet megtakarítható, ha a kosár processzorához történő mozgatás során a szóban forgó processzor rögtön használni tudná a blokkot az összekapcsolási algoritmusának részeként. A legtöbb ismert algoritmus, ami az összekapcsolásra és a többi relációs műveletre vonatkozik, ezt lehetővé teszi, és ilyenkor a párhuzamos algoritmus úgy néz ki, mintha csak egy többmenetes algoritmusról lenne szó, ahol az első menet a 6.9.3. rész tördelési technikáját használja.

6.28. példa: Tekintsük ismét a korábbi példánkat a $R(X, Y) \bowtie S(Y, Z)$ műveletre vonatkozóan, ahol az R és az S relációk egyenként 1000, illetve 500 blokkot foglalnak el. Legyen továbbá egy 10 processzoros gépünk, minden processzoránál 101 pufferrel. Végül tegyük még fel, hogy R és S az említett 10 processzor között egyenletesen van elosztva.

Azzal kezdjük, hogy R és S minden egyes sorát 10 kosár valamelyikébe tördeljük egy olyan h tördelőfüggvénnyel, amely csak az Y összekapcsolási attribútumoktól függ. A 10 kosár a 10 processzort jelképezi, és a sorokat a kosárnak megfelelő processzorhoz küldjük. R és S sorainak beolvasásához összesen 1500 lemez I/O-művelet szükséges, azaz processzoronként 150. Minden processzornak mintegy 15 blokknyi adata lesz minden egyes másik processzor számára, így 135 blokkot kell elküldenie a többi kilenc processzornak. Az összes kommunikáció így 1350 blokkot érint.

Az algoritmust úgy fogjuk szervezni, hogy a processzorok S sorait R sorai előtt küldjék szét. Mivel az egyes processzorok körülbelül 50 darab S soraiból álló blokkot kapnak, így a szóban forgó sorokat a memóriában tárolhatják egy megfelelő adatstruktúrában, ezzel a 101 pufferből 50-et felhasználva. Így, amikor a processzorok elkezdik R sorainak a küldését, akkor ezen sorok mindegyikét összehasonlítjuk a lokális S sorokkal, az eredményként kapott összekapcsolt sorokat pedig a kimenetbe írjuk.

Ezzel a módszerrel az összekapcsolás költsége mindössze 1500 lemez I/O-művelet lesz, ami sokkal kevesebb, mint az ebben a fejezetben tárgyalt bármelyik másik módszeré. Sőt az eltelt idő elsősorban az egyes processzorok 150 lemez I/O-műveletére korlátozódik, amihez jön még a processzorok közötti sorküldések és a memóriabeli számítások ideje. Vegyük észre, hogy a 150 lemez I/O-művelethez szükséges idő kevesebb, mint 1/10-e annak az időnek, ami ugyanennek az algoritmusnak az egyprocesszoros végrehajtásához kell. Nem csak azáltal nyertünk tehát, hogy 10 processzorunk dolgozik egyszerre, hanem az a tény, hogy a 10 processzornak 1010 puffere van, még további hatékonyságnövekedéssel is jár.

Persze felmerülhet az az ellenvetés, hogy ha egyetlen processzornak lenne 1010 puffere, akkor a példánkban szereplő összekapcsolást egy menetben is elvégezhettük volna 1500 lemez I/O-művelet árán. Nem szabad azonban elfelejtenünk, hogy a multiprocesszoros gépek memóriája általában a processzorok számával arányos, és mi mindössze annyit tettünk, hogy a multiprocesszoros feldolgozás két előnyét egyszerre aknáztuk ki. Így két egymástól független sebességnövekedést értünk el egy időben: az egyik a processzorok számával volt arányos, míg a másik az volt, hogy a pluszmemória révén egy hatékonyabb algoritmust használhattunk.□

6.10.5. Feladatok

6.10.1. feladat: Tegyük fel, hogy egy lemez I/O-művelet 100 ezredmásodperc időt vesz igénybe. Legyen $B(R) = 100$, így a $\sigma_C(R)$ kiszámításához szükséges lemez I/O-műveletek egy egyprocesszoros gépen körülbelül 10 másodpercig tartanak. Mennyi időt takaríthatunk meg, ha a kiválasztást egy p processzoros gépen hajtjuk végre, ahol:

* a) $p = 8$.
b) $p = 100$.
c) $p = 1000$.

! **6.10.2. feladat:** A 6.28. példában megadtunk egy párhuzamos algoritmust, amelyik az $R \bowtie S$ összekapcsolást állítja elő oly módon, hogy először tördeléssel szétosztja a sorokat a processzorok között, majd egy egymenetes összekapcsolást hajt végre minden processzornál. Adjuk meg azt a feltételt, amely esetén ez az algoritmus végrehajtható, a következő paraméterekkel kifejezve: $B(R)$ és $B(S)$ a relációk mérete, p a processzorok száma, illetve M az egyes processzorok számára rendelkezésre álló memóriablokkok száma.

6.11. Összefoglalás

- *Lekérdezésfeldolgozás:* A lekérdezéseket először lefordítjuk, eközben sokrétű optimalizálást végzünk rajtuk, majd végrehajtjuk őket. A lekérdezés-végrehajtás területe olyan módszerek ismeretét foglalja magában, amelyekkel a relációs algebra műveleteit, illetve további kiegészítő műveleteket tudunk végrehajtani. A kiegészítő műveletekre azért van szükség, hogy az SQL lehetőségeit is ki tudjuk fejezni.
- *Lekérdezéstervek:* A lekérdezéseket először logikai lekérdezéstervekké alakítjuk, amelyek többnyire a relációs algebra kifejezéseihez hasonlítanak. Ezután ezekből fizikai lekérdezéstervet készítünk oly módon, hogy kiválasztjuk az egyes műveletek konkrét megvalósításának módját, az összekapcsolások sorrendjét, és további döntéseket hozunk. Ezekkel a kérdésekkel a 7. fejezetben fogunk foglalkozni.
- *Kibővített relációs algebra:* A relációs algebra szokásos műveleteit – amelyek az egyesítés, metszet, különbség, kiválasztás, vetítés, szorzat, és az összekapcsolás különböző formái – kissé módosított formában kell használnia a lekérdezésfeldolgozónak, mégpedig a műveletek halmazos formái helyett azok multihalmazos változatait véve. Ezenkívül további olyan műveleteket is hozzá kell még vennünk az algebrához, amelyek a következő SQL-beli műveleteknek felelnek meg: ismétlődések megszüntetése, csoportosítás és összesítés, rendezés.
- *Táblaátvizsgálás:* Egy reláció sorait többféle fizikai operátor segítségével érhetjük el. A táblaátvizsgálás-operátor egyszerűen beolvassa azokat a blokkokat, amelyben a reláció sorai találhatók. Az index alapú átvizsgálás egy index segítségével keresi meg a sorokat, míg a rendezéses átvizsgálás rendezett sorrendben állítja elő a sorokat.
- *Fizikai operátorok költsége:* Általában a fizikai operátorok végrehajtásakor a lemez I/O-műveletek száma jelenti a legfontosabb tényezőt a végrehajtási időben. Az általunk használt modellben csak a lemez I/O-műveletek számára szükséges időt vesszük figyelembe, továbbá az argumentumok beolvasásához szükséges időt és tárat számoljuk, a végeredmény kiírásához szükséges erőforrásokat azonban nem.
- *Iterátorok:* A lekérdezések végrehajtásában szereplő műveletek közül néhányat kényelmesebben elképzelhetünk úgy, hogy a végrehajtásukat egy iterátor végzi. Ez a módszer három függvényt feltételez, egyik végzi a reláció megnyitását, egy másik a reláció következő sorát adja vissza, és végül a harmadik függvény lezárja a relációt.
- *Egymenetes algoritmusok:* Amennyiben egy relációs algebrai művelet argumentumaiban szereplő relációk közül az egyik befér a memóriába, akkor a műveletet végrehajthatjuk oly módon, hogy a kisebb relációt beolvassuk a memóriába, és a másikat blokkonként olvassuk hozzá.
- *Egymásba ágyazott ciklusú összekapcsolás:* Ez az egyszerű összekapcsolási algoritmus akkor is működik, ha egyik reláció sem fér be a memóriába. Az algoritmus a kisebbik relációból beolvas a memóriába annyit, amennyi befér, és ezt hasonlítja össze a teljes másik relációval. A folyamat addig ismétlődik, amíg a kisebbik reláció minden sora be nem került a memóriába.
- *Kétmenetes algoritmusok:* A legtöbb algoritmus, amelynél a relációk nem férnek be a memóriába rendezés alapú, tördelés alapú vagy index alapú. Ez alól kivétel az egymásba ágyazott ciklusú összekapcsolás.

- *Rendezés alapú algoritmusok:* Ezek az algoritmusok az argumentumaikat a memória méretének megfelelő rendezett részlistákra osztják, majd a részlisták összefuttatásával állítják elő a kívánt eredményt.
- *Tördelés alapú algoritmusok:* Ezek az algoritmusok egy tördelőfüggvény segítségével kosarakba osztják szét az argumentumaikat. Ezután a műveletet az egyes kosarakra önállóan (egyoperandusú műveletek esetén), vagy a kosarakból álló párokra (kétoperandusú műveletek esetén) végzik el.
- *Tördelés vagy rendezés:* A tördelésen alapuló algoritmusok gyakran jobbnak bizonyulnak, mint a rendezésen alapulók, mert csupán azt követelik meg, hogy az egyik reláció „kicsi" legyen. A rendezésen alapuló algoritmusok viszont akkor bizonyulnak jónak, amikor van valami más ok is, ami miatt célszerű az adatok egy részét rendezetten tartani.
- *Index alapú algoritmusok:* Egy index használatával sokkal gyorsabban végrehajtható a kiválasztás művelet, ha annak feltételében az indexelt attribútumnak egy konstanssal való egyenlővé tétele szerepel. Az index alapú összekapcsolások is nagyon jól működnek, ha az egyik reláció kicsi, a másik relációnak pedig van indexe az összekapcsolás alapjául szolgáló attribútumokra.
- *A pufferkezelő:* A memóriablokkok elérhetőségét a pufferkezelő felügyeli. Ha egy új memóriapufferre van szükség, a pufferkezelő dönti el, hogy melyik puffert szabadítsa fel és írja vissza a lemezre a rendszer. Ehhez az ismert puffercserélési módszerek valamelyikét használja, mint amilyen például az LRU.
- *A pufferszám változásának problémája:* Gyakran előfordul, hogy egy művelet számára elérhető memóriapufferek számát nem lehet előre tudni. Ilyen esetekben a műveletet megvalósító algoritmusnak rugalmasan módosulnia kell, amikor az elérhető pufferek száma csökken.
- *Többmenetes algoritmusok:* A rendezésen és tördelésen alapuló kétmenetes algoritmusoknak vannak rekurzív módon kiterjesztett változatai, amelyek három vagy még több menetet használnak, és egészen nagy adatmennyiségre is működnek.
- *Párhuzamos számítógépek:* A mai párhuzamos számítógépeket általában a következő felépítések valamelyike jellemzi: megosztott memória, megosztott lemez, illetve az, hogy nincs megosztás. Az adatbázisrendszerek számára általában ez utóbbi felépítés (költség szempontjából) a leghatékonyabb.
- *Párhuzamos algoritmusok:* A relációs algebra műveleteit egy párhuzamos számítógépen általában közel annyiszorosára tudjuk felgyorsítani, amennyi a processzorok száma. A bemutatott algoritmusok először tördelés segítségével a processzoroknak megfelelő kosarakba osztják szét az adatokat, és a kosarakat a megfelelő processzoroknál helyezik el. Ezután a processzorok a lokális adatokon végzik el a műveletet.

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 lekérdezésfeldolgozó feladata három fő lépésből áll:

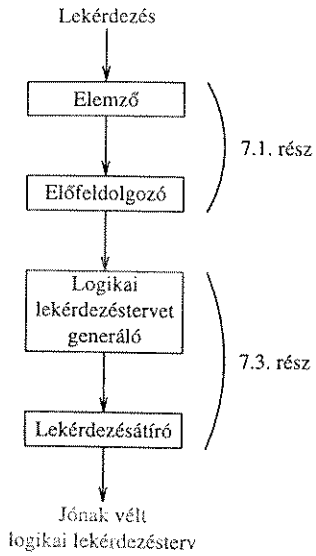
1. Az SQL-szerű nyelven megfogalmazott lekérdezés *elemzése,* azaz átalakítása egy elemzőfává, amely a lekérdezés szerkezetének egy jól használható reprezentációját adja.
2. Az elemzőfa átalakítása egy, a relációs algebrával (vagy más hasonló jelölésrendszerrel) megfogalmazott kifejezésfává, amit *logikai 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

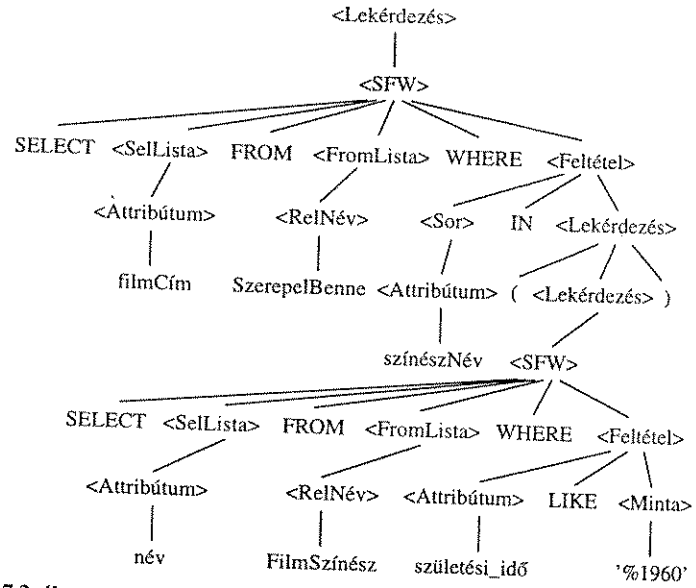
```
SELECT filmCím
FROM SzerepelBenne
WHERE színészNév IN (
   SELECT név
   FROM FilmSzínész
   WHERE születési_idő LIKE '%1960'
);
```

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. □



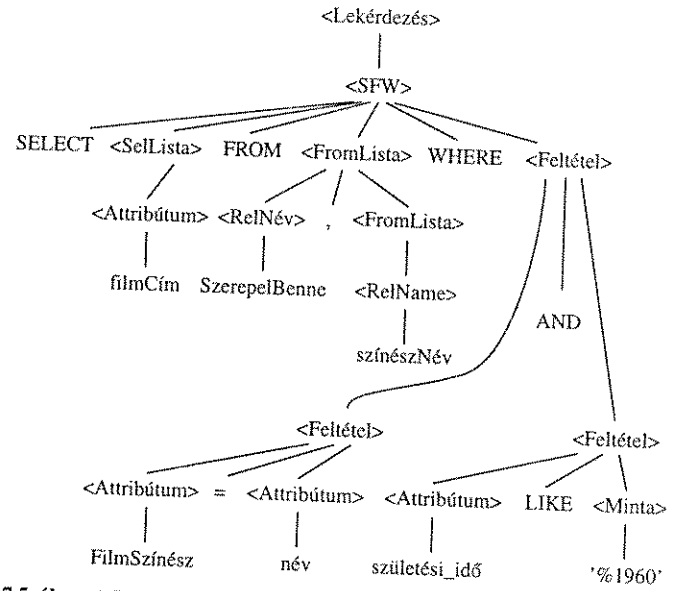
7.3. ábra. *A 7.2. ábrához tartozó elemzőfa*


```
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 karakterláncca 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 ⋈-ra vonatkozó kommutatív szabályt: $R \bowtie S = S \bowtie R$.

Először is tegyük fel, hogy egy t sor benne van az $R \bowtie S$ 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 ⋈-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:

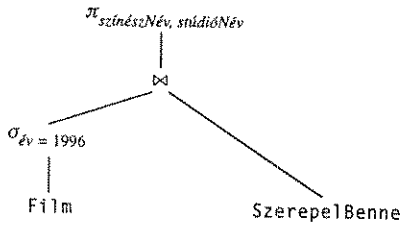
```
CREATE VIEW Filmek1996 AS
   SELECT *
   FROM Film
   WHERE év = 1996;
```

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:

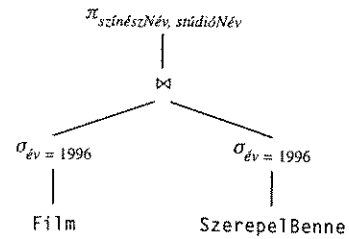
$\sigma_{év = 1996}$(`Film`)



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* 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 klaszszikus 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.

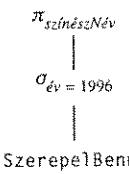
Csakúgy mint a 7.12. példában, lehetséges, hogy az L lista számításait inkább az M-ben végezzük el, feltéve, hogy a C feltétel nem igényli az L azon attribútumait, amelyek valamely számításban érintettek.

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.

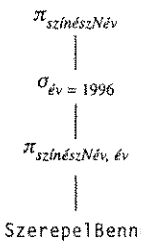


7.8. ábra. *Logikai lekérdezésterv a 7.13. példában szereplő lekérdezéshez*

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 vetítetni 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.

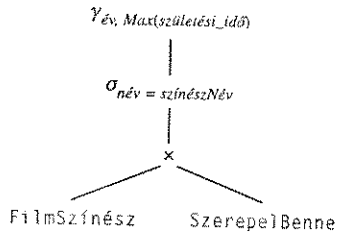
7.15. példa: Tegyük fel, hogy adottak a

```
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. ábrán 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.

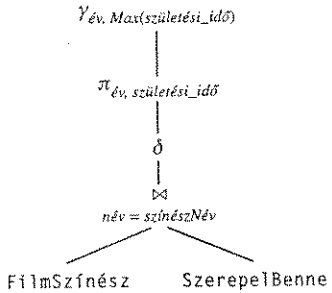


7.10. ábra. *Kiindulási logikai lekérdezésterv a 7.15. példa lekérdezéséhez*

A 7.10. ábrán elvégezhető több átalakítás is:

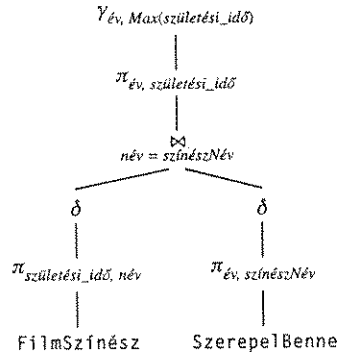
1. A kiválasztás és a szorzat összevonása egy egyenlőségen alapuló összekapcsolássá.
2. Egy δ beillesztése a γ alá, mivel γ ismétlődésérzéketlen.
3. Egy olyan π vetítés beillesztése a γ és az újonnan bevezetett δ közé, ami az `év`-re és a `születési_idő`-re, vagyis a γ szempontjából lényeges attribútumokra 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) $b + c \rightarrow x, c + d \rightarrow y$
b) $a, b, a + d \rightarrow z$

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

!! **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) $\sigma_C(R \overset{\cdot}{\bowtie} S) = \sigma_C(R) \overset{\cdot}{\bowtie} S$
  c) $\sigma_C(R \overset{\cdot}{\bowtie}_L S) = \sigma_C(R) \overset{\cdot}{\bowtie}_L S$ (ahol C-ben az R minden attribútuma szerepel)
  d) $\sigma_C(R \overset{\cdot}{\bowtie}_L S) = R \overset{\cdot}{\bowtie}_L \sigma_C(S)$ (ahol C-ben az R minden attribútuma szerepel)
  e) $\pi_L(R \overline{\ltimes} S) = \pi_L(R) \overline{\ltimes} S$
* f) $(R \overset{\cdot}{\bowtie} S) \overset{\cdot}{\bowtie} T = R \overset{\cdot}{\bowtie} (S \overset{\cdot}{\bowtie} T)$
  g) $R \overset{\cdot}{\bowtie} S = S \overset{\cdot}{\bowtie} R$
  h) $R \overset{\cdot}{\bowtie}_L S = S \overset{\cdot}{\bowtie}_L R$
  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.

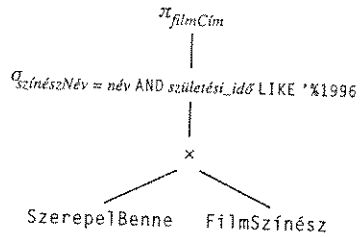
7.3.1. Átfordítás relációs algebrába

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.

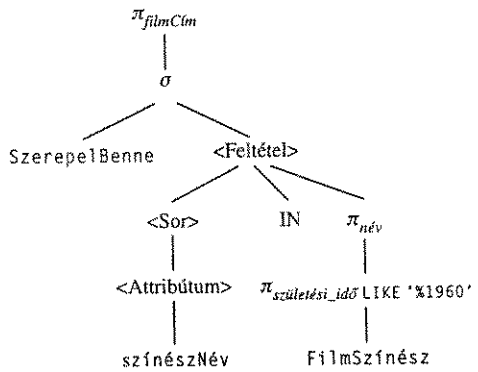
Alkalmazhatjuk azonban a „select-from-where" szabályt a 7.3. ábrán lévő alkérdésre. Az alkérdésből nyert relációs algebrai kifejezés: $\pi_{név}(\sigma_{születési\_idő\ \texttt{LIKE}\ \texttt{'\%1960'}}(\texttt{FilmSzínész}))$. □

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. □

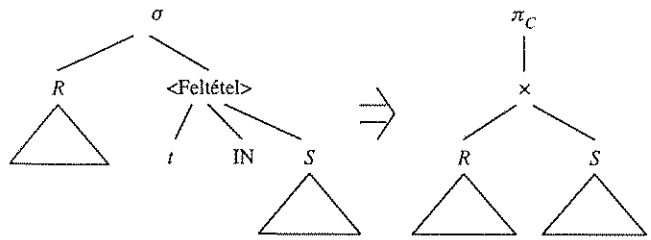
Szükségünk van szabályokra, amelyek lehetővé teszik, hogy egy kétargumentumú kiválasztást egy relációs algebrai egy argumentumú kiválasztással és egy másik relációs algebrai operátorral helyettesítsünk. A feltételek különböző formái külön szabályokat igényelhetnek. A legtöbb helyzetben a kétargumentumú 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.



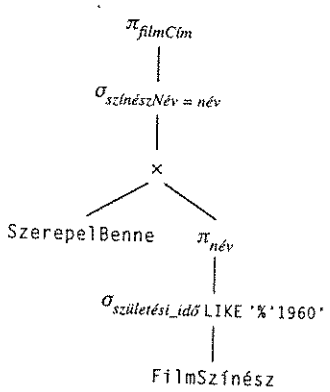
7.15. ábra. `IN`*-t tartalmazó feltétellel rendelkező kétargumentumú kiválasztást kezelő szabály*

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 × csomópont, és az × 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 ve-



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

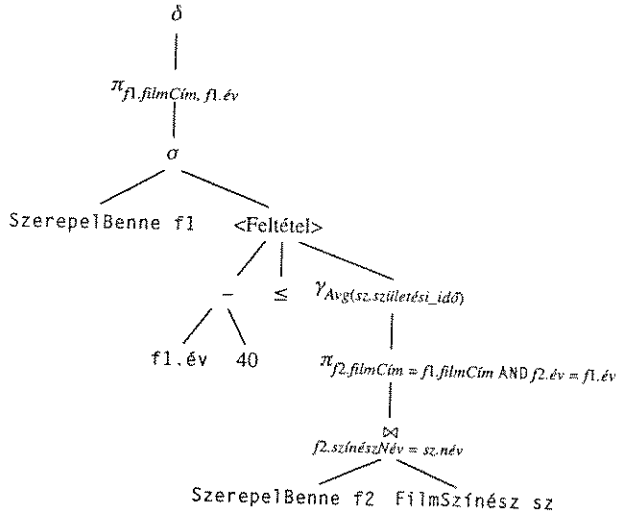
hetjük azok átlagát, amit aztán a `SzerepelBenne` reláció `év` attribútumával össze tudunk hasonlítani. A lekérdezést úgy fogalmaztuk meg, hogy mindhárom hivatkozott reláció rendelkezik a maga saját sorváltozójával, mutatva, hogy a különböző attribútumok honnan származnak.

```
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

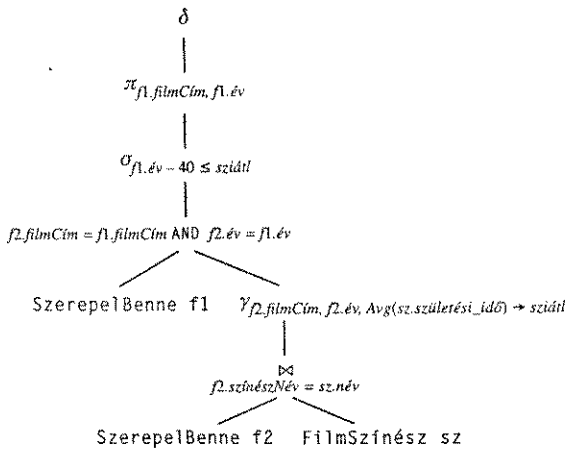


7.18. ábra. *Részlegesen transzformált elemzőfa a 7.17. ábra lekérdezéséhez*

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 $\sigma_{f2.filmCím = f1.filCím \text{ AND } f2.év = f1.év}$ 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

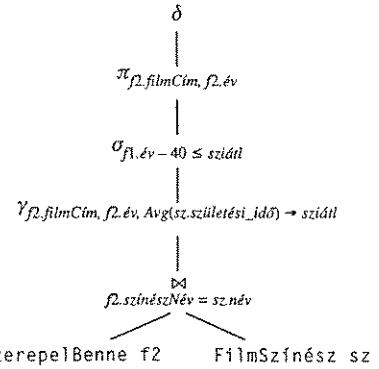


7.19. ábra. *A 7.18. ábra átalakítása egy logikai lekérdezéstervvé*

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.

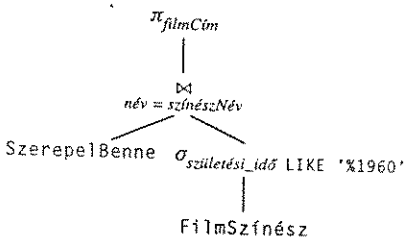
Nyitva hagyjuk viszont az „összekapcsolási sorrend" kérdését, így egy relációk összekapcsolását tartalmazó logikai lekérdezésterv úgy tekinthető, mint tervek egy családja, ami megfelel azoknak a különböző lehetőségeknek, ahogyan egy összekapcsolás sorba rendezhető és csoportosítható. Az összekapcsolási 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 különkülön. Ezeket a szabályokat a 7.2.5. részben tárgyaltuk.

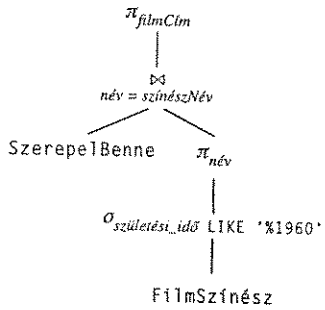
7.20. példa: Vegyük a 7.13. ábra lekérdezését. Először a kiválasztást vágjuk ketté a $\sigma_{színészNév = név}$ és a $\sigma_{születési\_idő\ \texttt{LIKE}\ \texttt{'1960'}}$ operátorokra. Az utóbbi levihető a fában, mivel az egyetlen érintett attribútum (`születési_idő`) a `FilmSzínész` relációból 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*

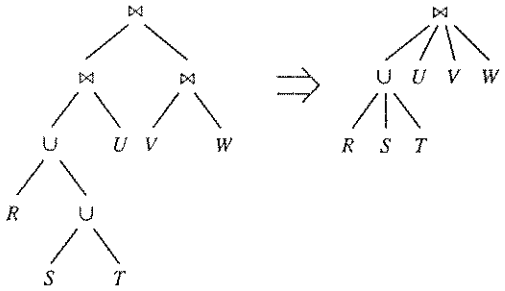
7.3.4. Asszociatív/kommutatív operátorok csoportosítása

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 – nyaláboltan vagy nem, indexelve vagy anélkül –, a lekérdezés végrehajtása közben kiszámított valamely reláció, amelyet lemezen tárolunk, tárolható nyaláboltan 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 atribútum 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 feltételt. 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 szorzatotot 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

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

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 $1000 \times 2000/50$, 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 $40\,000 \times 5000/\max(100, 500)$ 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 $1000 \times 20\,000/50$, 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 $v_1 \leq v_2 \leq \ldots \leq v_k$, 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 $1/v_2v_3\ldots v_k$. 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: $1000 \times 2000 \times 5000$. 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 $1000 \times 2000 \times 5000/(50 \times 200 \times 200)$, 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 fomulával 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 nyalábolatlan 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 értékre 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.

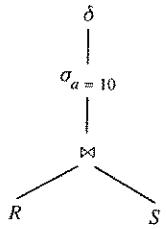
7.5.3. Logikai lekérdezéstervek költségének csökkentésére irányuló heurisztikák

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.

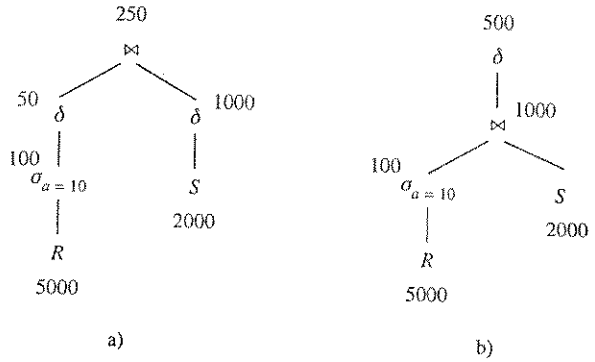
7.31. példa: Vegyük a 7.26. ábrán található kiindulási logikai tervet, valamint legyenek az R és S relációkhoz tartozó statisztikák a következők:

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

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 levelet 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 felfelé 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.

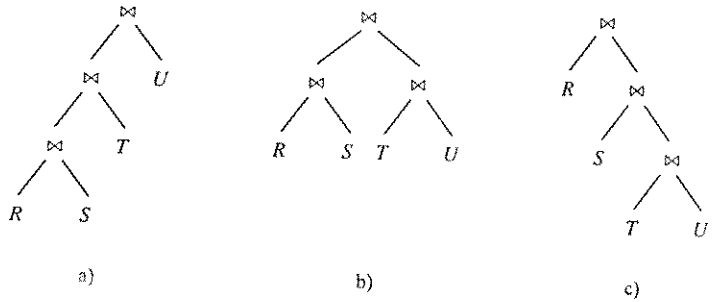
7.32. példa: Vegyük elő újra a 7.4. ábra következő lekérdezését:

```
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 ré-

szét állítja elő. Mivel általában több színész szerepel egy-egy filmben, a `SzerepelBenne` reláció várhatóan nagyobb, mint a `FilmSzínész` reláció, elmondható tehát, hogy $\sigma_{születési\_idő\ \texttt{LIKE}\ \texttt{'\%1960'}}(\texttt{FilmSzínész})$, 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 kereséstervek 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 összekapcsolás 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 `Filter`nek 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

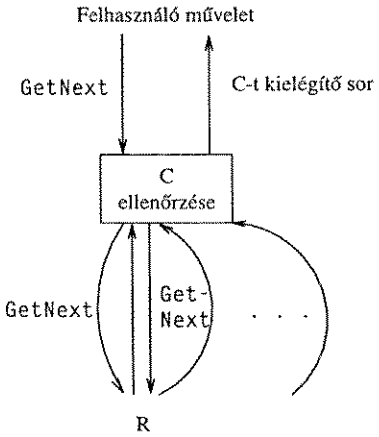
Bárki el tudja képzelni, hogy létezik egy közbülső megközelítés a futószalagosítás és a materializáció között, amikor egy művelet teljes eredményét a központi memória puffereiben (nem lemezen) tároljuk, mielőtt az azt felhasználó műveletnek átadódik. Az ilyen műveletmódot futószalagosításnak tekintjük, 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:

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

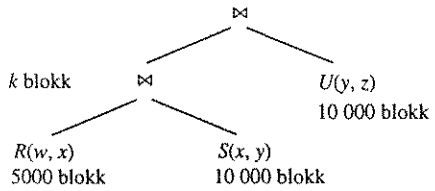
A következő feltételezésekkel élünk:

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 leveléből (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.

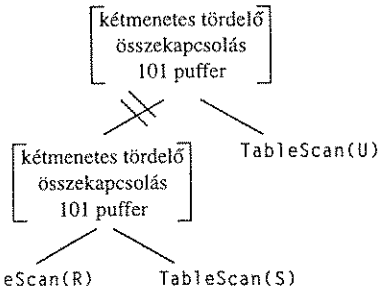
Fizikai operátorok a rendezéshez

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



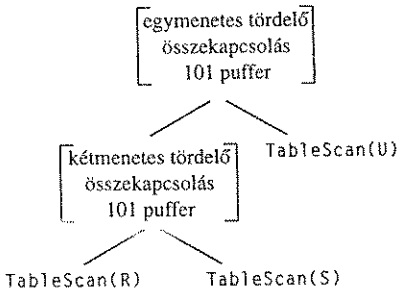
7.37. ábra. *Egy fizikai terv a 7.38. példából*

(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.

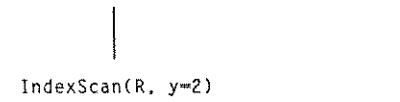
7.39. példa: A 7.38. példában a $k > 5000$ esetre kidolgozott fizikai tervet a 7.37. ábra mutatja. Ebben a tervben mindhárom relációt a `TableScan` operátor segítségével érjük el. Használunk egy kétmenetes tördelő összekapcsolást az első összekapcsoláshoz, materializáljuk az eredményt, majd használunk 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 $z < 5$. □



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.