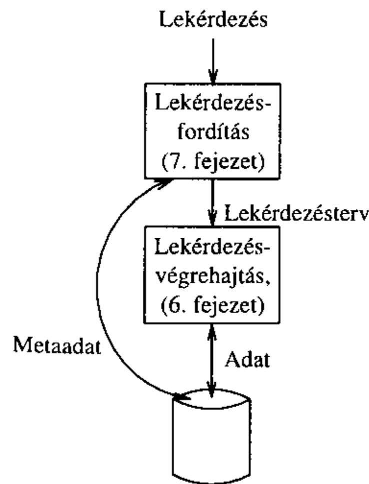
12. R. L. Rivest, „Partial match retrieval algorithms," *SIAM J. Computing* **5**:1 (1976), pp. 19–50.
13. J. T. Robinson, „The K-D-B-tree: a search structure for large multidimensional dynamic indexes," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1981), pp. 10–18.
14. J. B. Rothnie Jr. and 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.

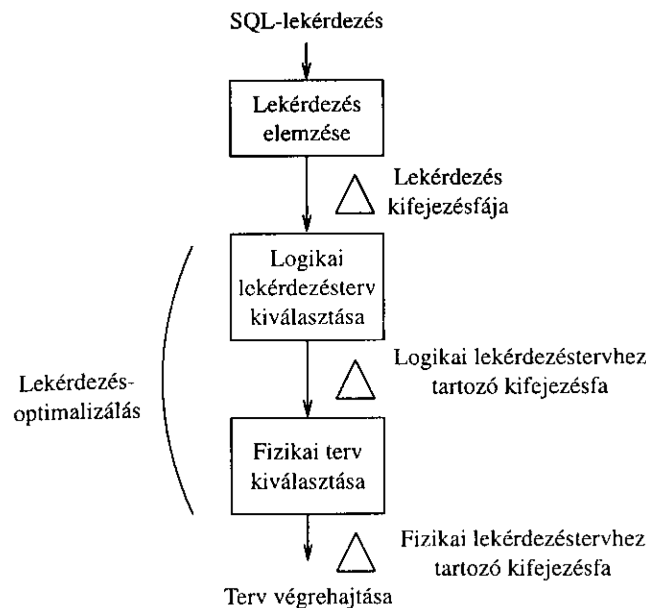
A relációs algebra használatának egyik előnye, hogy megkönnyíti a lekérdezések különböző lehetséges formáinak kifejezését. Az egy lekérdezéshez tartozó különböző algebrai kifejezéseket *logikai lekérdezésterveknek* (logical query plans) nevezzük. Ezeket a terveket gyakran kifejezésfákkal ábrázoljuk. 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 $x \to y$ kifejezést, ahol x és y attribútumnevek. Az $x \to y$ 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 $E \to z$ 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 $a + b \to x$, mint a lista egy eleme, az a és b attribútumok összegét reprezentálja x néven. A $c \parallel d \to e$ 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 $\pi_{a,\, b+c \to x}(R)$ 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 $\pi_{b-a \to x,\, c-b \to y}(R)$ 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:

$\gamma_{színészNév,\, MIN(év) \rightarrow minÉv,\, COUNT(cím) \rightarrow ctCím}$(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. □

$\pi_{színészNév,\, minÉv}$
|
$\sigma_{ctCím \geq 3}$
|
$\gamma_{színészNév,\, MIN(év) \rightarrow minÉv,\, COUNT(cím) \rightarrow ctCím}$
|
SzerepelBenne

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

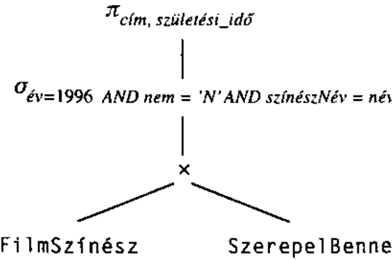
számítanak. Azok a sorok, amelyek megegyeznek az a_1 és a_2 értékeiken, az a_3 értékek szerint kerülnek rendezésre és így tovább. Azok a sorok, amelyek még az a_n attribútumon is megegyeznek, tetszőleges sorrendbe helyezhetők. Éppúgy, mint az SQL esetén, feltételezzük, hogy az alapértelmezett rendezési 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)
```

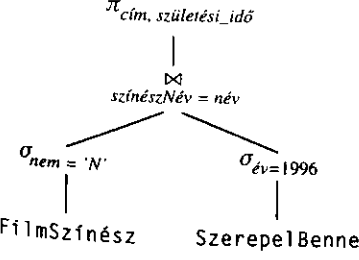
amelyekből szeretnénk kinyerni azokat a filmcímeket, a színésznők születésni idejével együtt, amelyekben az 1996-ban megjelent filmekben szereplő színésznők játszanak:

```
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-művelették, és ezeket a műveleteket lejjebb „csúsztatjuk" a fában, egészen a megfelelő relációkig. A $\sigma_{év = 1996}$ 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.

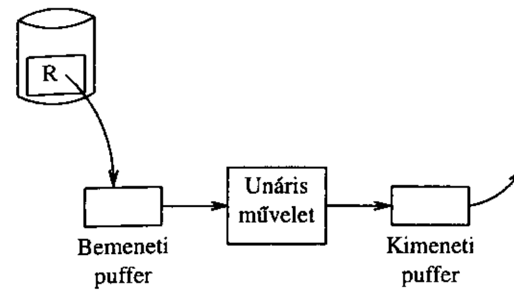
c) Végül vannak olyan módszerek, amelyek az adatok méretétől függetlenül működnek. Az ilyen algoritmusok három vagy annál is több menettel dolgoznak, és tulajdonképpen a kétmenetes algoritmusok természetes, rekurzív általánosításai. A többmenetes algoritmusokkal a 6.9. részben foglalkozunk.

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.

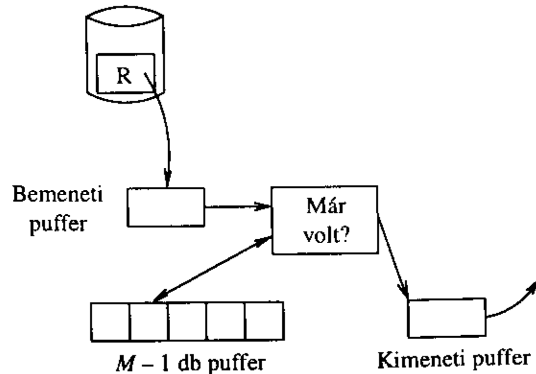
A döntés támogatására kell egy másolatot tartanunk a memóriában az összes általunk már látott sorról, amint azt a 6.11. ábra mutatja. Egy memóriapuffer tartalmazza R sorainak egy blokkját, míg a fennmaradó $M - 1$ puffert használhatjuk arra, hogy tartalmazzák a már előfordult minden egyes sor egy másolatát.

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 memória-struktúrára 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 nyalábolták. 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:

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

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) $R \overset{\circ}{\bowtie}_L S$, feltéve, hogy R befér a memóriába.

f) $R \overset{\circ}{\bowtie}_L S$, feltéve, hogy S befér a memóriába.
g) $R \overset{\circ}{\bowtie}_R S$, feltéve, hogy R befér a memóriába.
h) $R \overset{\circ}{\bowtie}_R S$, feltéve, hogy S befér a memóriába.
i) $R \overset{\circ}{\bowtie} S$, 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 öszekapcsolásnak* 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 cikluson 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

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

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 összekapcsolásét, 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.

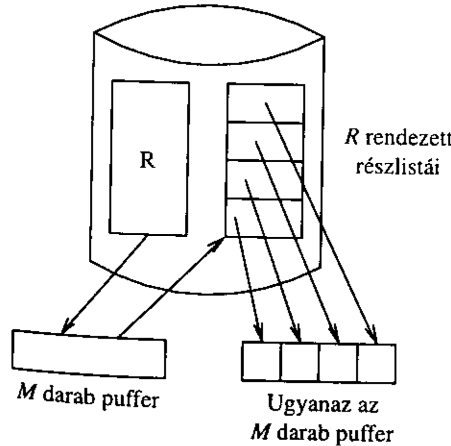
Ebben a részben a rendezést fogjuk vizsgálni, mint a relációs műveletek végrehajtására szolgáló eszközt. Az alapötlet a következő. Tegyük fel, hogy van egy nagy relációnk, R, amelyre $B(R)$ nagyobb M-nél, vagyis a rendelkezésre álló memóriapufferek számánál. Ekkor egymás után többször ismételve a következőket 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 $B(R) + B(S) \le M^2$ 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 beférnek 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 rendelkezik-e, 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 $\gamma_{a,AVG(b)}(R)$ 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 beférnek 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. Mint-hogy 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 $B(R) \leq M^2$ 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

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

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:

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

a teljes költség pedig

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

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 $B(S) \le B(R)$*

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.

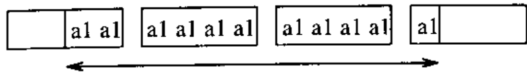
6.7.1. Nyalábolt és nem nyalábolt indexek

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 nyalábolтan 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 nyalábolтnak 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 keresésikulcs-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 lekérdezéstervet 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 mindösszesze $T(R)B(S)/V(S, Y)$ 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 $T(R)\big(\max(1, B(S)/V(S, Y))\big)$-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.

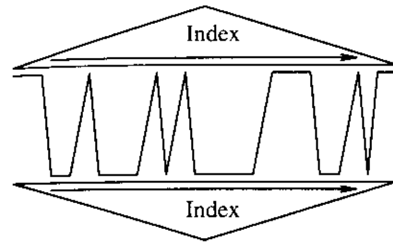
Szélsőséges esetben, amikor az R és az S egyaránt rendelkezik rendezett indexszel az Y attribútumo(ko)n, akkor a 6.5.5. rész egyszeru rendezésen alapuló összekapcsolásából csak a befejező lépést kell végrehajtanunk. Ezt a módszert néha *cikcakk-összekapcsolásnak* is szokták nevezni, mert oda-vissza ugrálunk 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 $\sigma_{C \leq a \text{ AND } a \leq D}(R)$ 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

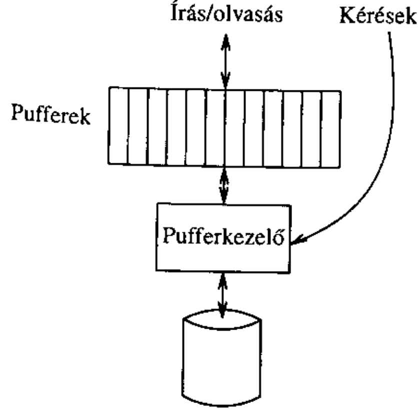
$\sigma_{title = \text{'King Kong'}}(\text{SzerepelBenne})$

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.)

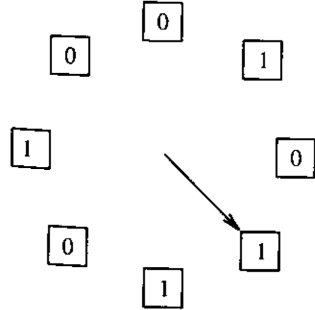
Bármelyik megoldást alkalmazza is az adatbázis-kezelő, mindkét esetben ugyanaz a probléma merül fel. Nevezetesen, a pufferkezelőnek úgy kell korlátoznia a használatban levő pufferek számát, hogy azok beleférjenek a rendelkezésre álló memóriába. Ha a pufferkezelő közvetlenül kezeli a memóriát, és a kérések 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 $R \bowtie S$ 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 $R_1, R_2, \ldots, R_M$-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 mentben 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).

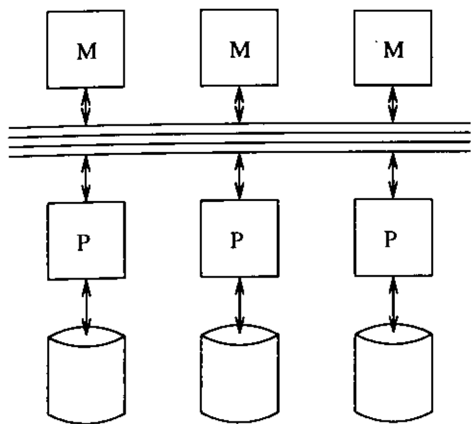
6.10. Párhuzamos algoritmusok relációs műveletekre

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

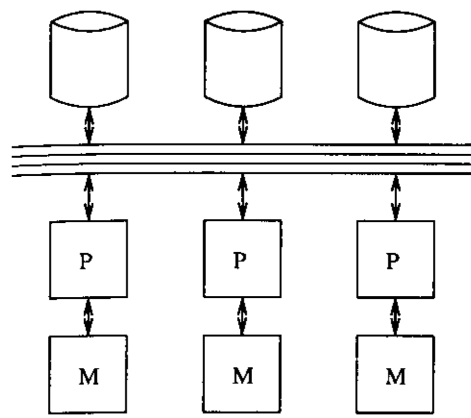


6.23. ábra. *Egy megosztott memóriájú gép*

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

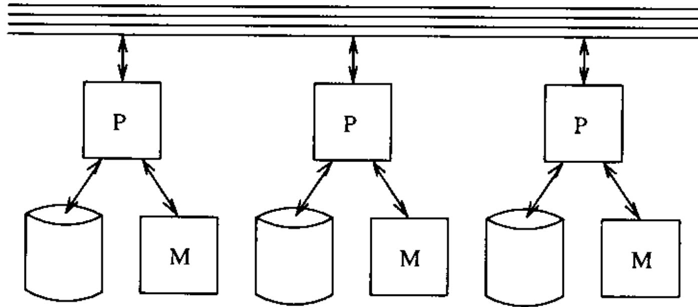
A párhuzamos számítógépek három legfontosabb osztálya a következő.

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 kiválasztáséval. 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 összeségében 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 kerül 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.