Hector Garcia-Molina
Jeffrey D. Ullman
Jennifer Widom

Adatbázisrendszerek megvalósítása

Panem–John Wiley & Sons

Hector Garcia-Molina
Jeffrey D. Ullman
Jennifer Widom

Adatbázisrendszerek megvalósítása

A mű eredeti címe: Database System Implementation
First Edition by Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom, Copyright © 2000.
All rights reserved.
Published by arrangement with the original publisher Prentice Hall, Inc.,
a Pearson Education Company.

Translation copyright © 2001 by Panem Könyvkiadó Kft.

ISBN 963 545 280 2

A kiadásért felel a Panem Könyvkiadó Kft. ügyvezetője, 2001

Szerkesztette: dr. Benczúr András
Fordította: Búza Attila, Cserges Enikő, dr. Hajas Csilla, dr. Kiss Attila, Kovács György,
Limbek Réka, Nikovits Tibor, Vincellér Zoltán
Lektorálta: dr. Márkus Tibor
Műszaki szerkesztő: Érdi Júlia
Borítóterv: Érdi Júlia

A Panem könyvek megrendelhetők az (1) 340-1515 hívószámú telefonon, illetve
az 1385 Budapest, Pf. 809 levélcímen.
panem@panem.hu
www.panem.hu

Tartalomjegyzék

Előszó a magyar kiadáshoz

A Stanford Egyetem három neves számítástudósa újabb tankönyvvel jelentkezett az adatbázis-kezelő rendszerek témakörében. Az első, a magyar fordításban *Adatbázis-rendszerek. Alapvetés* címen megjelent könyvet most a szerzőknek az adatbázisrendszerek megvalósítási kérdéseiről írt könyve követi. Az előző könyv fordítása során szerzett igen jó véleményünk és a könyv sikere alapján figyelemmel kísértük az előre jelzett új könyv megjelenését, és ahogy megjelent, a Panem–Prentice-Hall Kiadóval azonnal felvettük a kapcsolatot a magyar fordítás kiadásának érdekében. Jóllehet most nem kaptunk támogatást az Oktatási Minisztérium felsőoktatási tankönyv pályázatán, mégis mi, mint a tárgy oktatói fontosnak tartottuk a könyv magyar megjelentetését, és a Kiadó legnagyobb örömünkre vállalkozott rá.

Ahogy az előző könyv is hiánypótló volt, ez is az, bár ez már elsősorban azoknak a szakembereknek szól, akik nagy adatbázisok működtetéséért felelősek, megvalósításán, hatékonyabbá tételén dolgoznak.

Folytatva az előző könyv magyar kiadásához írt előszó jéghegy hasonlatát, az adatbázis-kezelő rendszerekről a jéghegy csúcsa után most az alap bemutatására kerül sor. A jéghegy mélyén a megvalósítás technikái húzódnak. Amit a könyv bemutat, az közel 50 év fejlődése során a közvetlen elérésű háttértárolók megjelenését követő fejlődés folyamatában letisztult megoldások sora. Erőteljes hangsúlyt kap ebben a relációs adatbázis-kezelők tipikus lekérdezésoptimalizálási világa, de a legújabb, az internet lehetőségeire alapuló hálózati adatbázis-kezelés elemei is megjelennek már.

A tárolási adatszerkezetek bemutatása során a lemezelérés és adatelhelyezés részletei, a hibajavítás és a RAID (független lemezek redundáns tömbje) lemezegységek, az adatelemek tárolási módjai, az indexelési technikák a legfejlettebb többdimenziós indexszerkezetekig bezárólag kerülnek bemutatásra.

A lekérdezések feldolgozásának megvalósítását az elemi relációs algebrai műveletek kiértékelésének részletes elemzése vezeti be, majd az összetett lekérdezések kiértékelésének elemző, optimalizáló módszerei következnek. A kérdés átírása hatékonyabban kiértékelhető ekvivalens relációs algebrai kifejezéssé, a logikai lekérdezési terv összeállítása, a műveletek költségbecslése, majd a költségbecslésre alapuló kiértékelési tervválasztás módszere, az összekapcsolások optimális sorrendje és végül a fizikai kiértékelési terv megadása található a módszerek között.

A rendszerhibák és a kivédésükre szolgáló naplózási technikák bemutatását, a konkurencia ellenőrzését, a tranzakciók kezelését részletesen leíró két fejezet követi. Az ütemezések sorbarendezhetősége, ennek ellenőrzését és garantálását biztosító zárolási és időbélyegzős technikák bemutatása, az osztott adatbázisok tranzakciókezelési módszerei a legfontosabb témái ezeknek a fejezeteknek.

Az utolsó, 11. fejezet a legújabb, nagy teljesítményű rendszerek megvalósításairól ad ízelítőt. A nagy adatbázisok, a különböző forrású adatok közös kezelését biztosító információintegrációs módok sorában bemutatja a szövetséges adatbázisok, az adattárházak és a közvetítők (mediátorok) módszerét. A nagy erőforrás-igényű on-line döntéstámogató rendszerek (OLAP) működését elősegítő adatszerkezetek, például a csillag sémák és adatkockák bemutatása után a könyvet az adatbányászat elemei zárják.

A könyv igen jól példázza azt, ahogy az információs technológiák fejlődnek: sok apró, néha egyszerűnek tűnő részlet, ami közel 50 év fejlődése során tisztult le a háttérben tudományos elemzésekkel. A fejlődés során többszörös rétegben rakódnak egymásra a technológia elemei, az alkalmazásoknál közvetlenül látható réteg mögött ezekről gyakran megfeledkezünk. Az adatbázisrendszerek esetében ez igen veszélyes lehet, amire a könyv 2.1. részében, a memóriahierarchiák ismertetésénél idézett Moore törvénye hívja fel a figyelmet: míg a processzor sebessége, a memória és a háttértároló sűrűsége 18 hónaponként megkétszereződik, és az egységnyi teljesítmény ára feleződik, addig a memória elérésének és a lemezek forgásának sebessége alig növekszik! A relatív távolság a processzor, a memória és a háttértárolók között növekszik. A háttértárolókkal való gazdálkodás, ami az adatbázisrendszerek megvalósításának központi kérdésköre, továbbra is meghatározó tényezője lesz a rendszerfejlesztéseknek.

A könyv szakembereknek szólóan dolgozza fel a fentiekben vázolt témákat. Részletesen, példákon keresztül mutatja be az általában aprólékos lépésekből álló módszereket, eljárásokat. Széles spektrumát fogja át az ismereteknek és a technológiának. Az oktatásban alaposan kipróbált tárgyalásmód igen jól olvasható szakirodalmat jelent mind a hallgatóknak, mind a szakembereknek. Az adatbázis-tervezők, -felhasználók és -alkalmazási programozók számára a három neves szakember igen sok gyakorlati tanácsot nyújt a korszerű adatbázisrendszerek megvalósításához és használatához.

Azok számára, akik az alapozó általános ismereteken túl kívánnak adatbázisokkal foglalkozni, feltétlenül ajánlatos a könyv áttanulmányozása. A hazai felsőoktatásban a programtervező matematikus szakon az adatbázis-előadások eddig is sok mindent lefedtek a könyv témáiból, most végre magyar nyelvű tankönyvként is rendelkezésre áll egy igen kiforrott tananyag. Javasolható a könyv a műszaki informatikus képzés felsőbb évfolyamaiban is haladó adatbáziskurzus alapkönyveként.

A magyar nyelvű változat szinte egy éven belül követi a könyv megjelenését, amiért köszönet illeti a Panem Könyvkiadó gyors döntését a kiadás vállalásában, s leginkább a fordítók és a lektor áldozatvállalását, hogy nyári szabadságuk jelentős részének feláldozásával tartani tudták az igen rövid határidőt, és igen jó minőséggel végezték munkájukat. A fordítást végző csapat most is az Eötvös Loránd Tudományegyetem Információs Rendszerek Tanszékének oktatóiból, jelenlegi és volt doktorandus hallgatóiból alakult.

Dr. Benczúr András

Előszó

Ezt a könyvet a CS245 kurzus számára terveztük, amely a Stanford Egyetemen az adatbázisok sorozatban a második kurzus. Az első adatbáziskurzus itt a CS145, amely az adatbázis-tervezés és -programozás témaköröket fedi le, és Jeffrey D. Ullman és Jennifer Widom ehhez írta a Prentice-Hall Kiadónál 1997-ben megjelent *A First Course in Database Systems*[1] című könyvet. A CS245 kurzus ezt követően az ABKR-ek megvalósítását veszi át, nevezetesen a tárolási szerkezeteket, a kérdésfeldolgozást és a tranzakciókezelést.

A könyv használata

Stanfordban negyedéves, quarter rendszerünk van, ezért a könyvet használó alapkurzus, a CS245, mindössze 10 hétig tart. 1999 telén Hector Garcia-Molina a könyv „béta" verzióját használta, és a következő részeket adta le: a 2.1.–2.4. részeket, a teljes 3. és 4. fejezetet, az 5.1. és 5.2., a 6.1.–6.7. és a 7.1.–7.4. részeket, a teljes 8. és 9. fejezetet, kivéve a 9.8. részt, a 10.1.–10.3., a 11.1. és a 11.5. részeket.

A 6. és 7. fejezetek többi része (lekérdezések optimalizálása) egy haladó kurzusban, a CS346-ban szerepel, melynek során a hallgatók saját ABKR-t készítenek. A könyv további olyan részei, amelyek nem szerepelnek a CS245-ben, egyéb haladó kurzusban kaphatnak helyet, mint például a CS347, amely az osztott adatbázisokat és a fejlett tranzakciókezelést tárgyalja.

Olyan intézmények, amelyek féléves, szemeszterrendszerben oktatnak, élhetnek azzal a lehetőséggel, hogy kombinálják ezt a könyvet az elődjével: *A First Course in Database Systems* könyvvel. Azt javasoljuk, hogy azt a könyvet az első szemeszterben használják egy adatbázis-alkalmazási projekthez társítva. A második szemeszter lefedheti ennek a könyvnek legnagyobb részét. Annak előnye, hogy az adatbázisok tár-

[1] Szerkesztői megjegyzés: A könyv megjelent magyar fordításban, *Adatbázisrendszerek. Alapvetés* címen a Panem–Prentice-Hall kiadók gondozásában 1998-ban.

gyat két kurzusra bontjuk az, hogy azok a hallgatók, akik nem szándékoznak adatbázisokra specializálódni, beérhessék csak az első kurzus választásával, és képesek legyenek az adatbázisok használatára, bármilyen számítástudományi területre lépjenek is.

Előismeretek

Azt a kurzust, amely erre a könyve alapul, ritkán választják a negyedik (senior) év előtt, ezért elvárjuk, hogy a hallgató elég széles háttérrel rendelkezzen a számítástudomány tradicionális területein. Feltételezzük, hogy az olvasó már tanult valamennyit az adatbázis-programozásról, speciálisan az SQL-ről. Sokat segít a relációs algebra ismerete, valamint az alapvető adatstruktúrákkal való ismeretség. Hasonlóan a fájlrendszerek és az operációs rendszerek bizonyos mértékű ismerete is hasznos.

Feladatok

A könyv terjedelmes feladatrendszert tartalmaz, szinte minden szakaszhoz tartozik néhány feladat. A nehezebb feladatokat vagy feladatrészeket felkiáltójel jelöli. A legnehezebb feladatok két felkiáltójelet kaptak.

Néhány feladat vagy feladatrész csillag megjelölést kapott. Arra törekszünk, hogy ezeknek a megoldásait a könyv weblapján hozzáférhetővé tegyük. Ezek a megoldások nyilvánosan elérhetők, és segítenek az önellenőrzésben. Felhívjuk még a figyelmet arra, hogy vannak olyan esetek, amikor egy *B* feladat igényli az olvasó által egy korábbi *A* feladatra adott megoldás módosítását vagy felhasználását. Amennyiben *A* valamely részére weben publikált megoldás található, feltehető, hogy a *B* megfelelő részére is elérhető a megoldás.

Támogatás a World Wide Weben

A könyv honlapja:

`http://www-db.stanford.edu/~ullman/dbsi.html`

Itt megtalálhatók a csillaggal jelölt feladatok megoldásai, a hibajavítások, ahogy értesülünk róluk, valamint segédanyagok. Reményeink szerint elérhetővé tesszük minden meghirdetett CS245 kurzusunkhoz a kiosztott jegyzeteket, és az egyéb adatbáziskurzusok lényeges anyagait úgy, ahogy, tanítjuk őket, beleértve a házi feladatokat, zárthelyiket és megoldásokat.

Köszönetnyilvánítások

Köszönettel tartozunk Brad Adelberg, Karen Butler, Ed Chang, Surajit Chaudhuri, Rada Chirkova, Tom Dienstbier, Xavier Faz, Tracy Fujieda, Luis Gravano, Ben Holzman, Fabien Modoux, Peter Mork, Ken Ross, Mema Roussopolous és Jonathan Ullman segítségéért, amelyet az anyag összegyűjtésében és/vagy a mű korábbi vázlataiban lévő hibák felfedezésében nyújtottak. A megmaradt hibák természetesen a mieink.

Hector Garcia-Molina
Jeffrey D. Ullman
Jennifer Widom
Stanford, CA

1. fejezet

Bevezetés az adatbázis-kezelő rendszerek implementálásába

Az adatbázisok manapság az üzleti élet minden területén alapvető fontosságúak. Éppúgy használatosak saját feljegyzések kezelésére, mint a világhálós (World Wide Web) kereskedelemben, ahol az ügyfelek és vásárlók számára kell adatokat szolgáltatni. Emellett természetesen sok egyéb kereskedelmi, pénzügyi folyamatot is szoktak adatbázisokkal segíteni. Hasonló módon adatbázisokat találunk sok tudományos vizsgálat mélyén is. Ezek az adatbázisok olyan adatokat reprezentálnak, amelyeket sok tudós gyűjtött össze, például csillagászok, genetikusok vagy éppen a fehérjék gyógyhatású tulajdonságait kutató biokémikusok.

Az adatbázisok ereje a több évtizeden keresztül kifejlődött tudásanyagnak és technológiának köszönhető. Ez a tudás azokban a speciális szoftverekben ölt testet, amelyeket *adatbázis-kezelő rendszereknek* (*DBMS* – database management system) vagy röviden „adatbázisrendszernek" hívunk. Egy adatbázisrendszer hathatós eszköz arra, hogy hatékonyan készíthessünk, kezelhessünk nagy mennyiségű adatot, és lehetővé teszi azt is, hogy ezeket az adatokat hosszú ideig biztonságosan megőrizhessük. Ezek a rendszerek a jelenleg rendelkezésre álló legösszetettebb, legbonyolultabb programtermékek közé sorolhatók.

Nézzük meg, hogy milyen lehetőségeket nyújt egy adatbázisrendszer a felhasználónak:

1. *Maradandó tárolás* (persistent storage). Hasonlóan a fájlrendszerhez, egy adatbázisrendszer is támogatja a nagyon nagy mennyiségű adatok tárolását, és ez a tárolás nem csak az adatokat felhasználó folyamatok futása alatt tart, hanem ezektől függetlenül is létezik. Az adatbázisrendszer azonban tovább megy a fájlrendszernél abban a tekintetben, hogy olyan adatszerkezetekről is gondoskodik, amelyek segítségével ez a nagyon sok adat hatékonyan, gyorsan érhető el.
2. *Programozási felület* (programming interface). Az adatbázisrendszer megengedi a felhasználónak, hogy az adatokat egy olyan lekérdezőnyelven keresztül érje el, illetve módosíthassa, amelynek elég nagy a kifejezőereje. Az adatbázisrendszer előnye egy fájlrendszerrel szemben itt is a bővebb lehetőségekből adódik, ugyanis a tárolt adatok a fájlírás/olvasásnál sokkal összetettebb módon is kezelhetők.
3. *Tranzakciókezelés* (transaction management). Az adatbázisrendszer támogatja az adatok konkurens elérését, vagyis azt, hogy több különböző folyamat (tranzakció)

A legfontosabb szakkifejezések áttekintése

Ez a könyv azok számára készült, akik felhasználói (például SQL-programozás) szemszögből már ismerik az adatbázisrendszereket legalább az Ullman–Widom: *Adatbázisrendszerek. Alapvetés* (Panem Könyvkiadó Kft., Budapest, 1998.) könyv szintjén. A következő szakkifejezéseket ezért ismerteknek tételezzük fel.

- *Adat:* tetszőleges információ, amelyet érdemes valamilyen (leginkább elektronikus) formában megőrizni.
- *Adatbázis:* hosszú ideig megőrzendő adathalmaz, mely a hozzáféréshez, módosításhoz szükséges szervezettséggel is rendelkezik.
- *Lekérdezés:* olyan művelet, mely meghatározott adatokat gyűjt ki az adatbázisból.
- *Reláció:* az adatoknak olyan kétdimenziójú táblába történő szervezése, ahol a sorok (relációsorok[1]) valamilyen alaptényeket vagy alapegyedeket jelképeznek, és az oszlopok (attribútumok) pedig ezeknek az egyedeknek a tulajdonságait képviselik.
- *Séma:* az adatbázisban szereplő adatok szerkezetének leírása, melyet gyakran „metaadatoknak" is hívnak.

egyszerre tudja elérni az adatokat. Ahhoz, hogy az egyidejű hozzáférés nem kívánatos következményeit elkerüljük, az adatbázisrendszer támogatja az *elkülönítést* (isolation), az *atomosságot* (atomicity) és a *helyreállíthatóságot* (resiliency). Az elkülönítés azt jelenti, hogy a tranzakciók közül látszólag csak egyet hajt végre egy időben a rendszer. Az atomosság azt a követelményt takarja, hogy egy tranzakciót vagy teljesen végrehajtunk, vagy egyáltalán nem hajtunk végre.

Végül a helyreállíthatóságon azt értjük, hogy sokféle rendszerhiba vagy más hiba esetén is legyen meg a lehetőség a rendszer helyreállítására.

1.1. Bevezetés: a Megatron 2000 adatbázisrendszer

Ha valaki már használt adatbázisrendszert, például egy olyat, amely támogatja a megszokott SQL lekérdezőnyelvet, akkor lehet, hogy azt képzeli, hogy egy ilyen rendszer megvalósítása[2] (implementálása) nem is olyan nehéz. Képzeljük el például egy kita-

[1] A relációsor vagy másképpen *n*-es (tuple) egy *n* komponensű vektort jelent. *A fordító megjegyzése.*

[2] Az angol *implementation* – megvalósítás szó helyett a magyarban leggyakrabban az implementálás szót használják. *A fordító megjegyzése.*

lált rendszernek az implementálást, mondjuk a Megatron Rendszerek Rt. képzeletbeli kínálatából vegyük a Megatron 2000 adatbázis-kezelő rendszert. Tegyük fel, hogy ennek a rendszernek UNIX és más operációs rendszerek alatti verziója is kapható, továbbá támogatja a relációs megközelítést és az SQL lekérdezőnyelvet.

1.1.1. A Megatron 2000 implementálásának részletei

Kezdjük azzal, hogy a Megatron 2000 a fájlrendszert használja a relációinak eltárolásához, például a `Hallgatók(név, azonosító, tanszék)` relációt a `/usr/db/Hallgatók` fájlban tárolja. A `Hallgatók` fájlnak egy sora felel meg minden egyes relációsornak. Egy relációsor komponenseinek értékeit egymástól speciális karakterrel (például #) elválasztott karakterláncokként (string) tároljuk. Például a `/usr/db/Hallgatók` fájl kinézhet az alábbi módon:

```
Smith#123#IT
Johnson#522#VM
   ...
```

ahol IT az Informatika Tanszéket, VM a Villamosmérnök Tanszéket jelöli.[3]

Az adatbázissémát egy `/usr/db/séma` nevű speciális fájlban tároljuk. A `séma` nevű fájlnak minden relációhoz van egy olyan sora, amely a reláció nevével kezdődik, és amelyben attribútumnevek és típusok váltakozva követik egymást. Például a séma tartalmazhatja a következő sorokat:

```
Hallgatók#név#STR#azonosító#INT#tanszék#STR
Tanszékek#név#STR#iroda#STR
   ...
```

Ezzel leírtuk a `Hallgatók(név, azonosító, tanszék)` relációt; a `név` és `tanszék` attribútumok típusa karakterlánc (string), míg az `azonosító` egész (integer) típusú. A másik sor egy `Tanszékek(név, iroda)` sémájú relációhoz tartozik.

1.1. példa: Nézzünk meg egy példát a Megatron 2000 adatbázis-kezelő rendszer használatáról. Egy `dbhost` nevű gépen dolgozunk, amelyen az adatbázis-kezelő rendszert a `megatron2000` UNIX-szintű parancs segítségével hívjuk meg.

```
dbhost> megatron2000
```

[3] Az USA-ban az egyetemi hallgató választhat egy (esetleg több) tanszéket, amely által meghirdetett, egymásra épülő tárgyakat szeretné hallgatni. Így a hallgatók ehhez az egy (vagy több) tanszékhez tartoznak. Ez a tanszékekhez rendelés hasonlít ahhoz, amit a magyar egyetemeken a szakok jelentenek. *A fordító megjegyzése.*

Erre a következő választ kapjuk:

```
ÜDVÖZÖL A MEGATRON 2000!
```

Ezután a Megatron 2000 felhasználói felületével kezdünk beszélgetést, amely számára begépelhetünk SQL-lekérdezéseket[4], és ezzel válaszolunk a Megatron rendszer felhívására (prompt), melyet & jelöl. Egy lekérdezést a # jel zár le. Például az

```
& SELECT *
  FROM Hallgatók #
```

válaszul a következő táblát adja meg:

Név	*Azonosító*	*Tanszék*
Smith	123	IT
Johnson	522	VM

A Megatron 2000 azt is megengedi, hogy végrehajtsunk egy lekérdezést, és az eredményt egy új fájlban tároljuk el. Ehhez a lekérdezést egy függőleges vonallal és a fájlnévvel kell befejezni. Például az

```
& SELECT *
  FROM Hallgatók
  WHERE azonosító >= 500 | NagyAzonosító #
```

utasítás egy `/usr/db/NagyAzonosító` fájlt fog készíteni, amelybe csak a következő egyetlen sor kerül:

```
Johnson#522#VM □
```

1.1.2. Hogyan hajtja végre a Megatron 2000 a lekérdezéseket?

Tekintsük az alábbi általános formájú SQL-lekérdezést:

```
SELECT * FROM R WHERE <Feltétel>
```

A Megatron 2000 erre a következőket teszi:

1. Beolvassa a `séma` fájlt, hogy meghatározza az *R* reláció attribútumait és az attribútumokhoz tartozó típusokat.
2. Ellenőrzi, hogy a `<Feltétel>` szemantikusan érvényes-e az *R* relációra.

[4] Az SQL rövid áttekintését az 1.4.2. fejezetben találjuk.

3. Minden egyes attribútumot egy-egy oszlopnak a fejléceként jelenít meg, és húz egy vonalat.
4. Beolvassa az *R* nevű fájlt, és minden egyes sorra:

   a) ellenőrzi a feltételt, és
   b) megjeleníti a sort, ha a feltétel igaz.

Ahhoz, hogy a Megatron 2000 a

```
SELECT * FROM R WHERE <Feltétel> | T
```

utasítást végrehajtsa, a következőt fogja tenni:

1. Az előbb leírt módon végrehajtja a lekérdezést azzal a különbséggel, hogy kimarad a 3. lépés, amely az oszlopok fejléceit és a fejléceket a soroktól elválasztó vonalat generálja.
2. Kiírja az eredményt egy `/usr/db/T` nevű új fájlba.
3. Készít egy bejegyzést a `/usr/db/séma` fájlban a *T* számára, amely ugyanúgy néz ki, mint az *R*-hez tartozó bejegyzés, vagyis a *T* sémája ugyanolyan, mint az *R* sémája, azzal az eltéréssel, hogy a reláció neve nem *R*, hanem *T*.

1.2. példa: Most nézzünk egy bonyolultabb lekérdezést, nevezetesen egy olyat, amelyben a `Hallgatók` és `Tanszékek` mintarelációk összekapcsolására (join) van szükség:

```
SELECT iroda
FROM Hallgatók, Tanszékek
WHERE Hallgatók.név = 'Smith' AND
      Hallgatók.tanszék = Tanszékek.név #
```

Ez a lekérdezés azt igényli a Megatron 2000-től, hogy kapcsolja össze a `Hallgatók` és `Tanszékek` relációkat, vagyis a rendszernek egymás után vennie kell az összes sorpárt, ahol a pár egyik eleme az egyik, a másik eleme a másik relációnak sora, és meg kell határoznia, hogy vajon

a) a sorpár sorai ugyanazt a tanszéket reprezentálják-e, és
b) a hallgató neve Smith-e.

Az algoritmust – nem törekedve a teljes formalizálásra – a következőképpen írhatjuk le:

```
minden(Hallgatókhoz tartozó s sorra)
  minden(Tanszékekhez tartozó d sorra)
    ha(s és d kielégíti a WHERE-feltételt)
      jelenítse meg a Tanszékekből az iroda értékét;
  □
```


1.1.3. Mi a baj a Megatron 2000-rel?

Lehet, hogy nem meglepő, de egy adatbázisrendszert nem szokás úgy implementálni, mint a mi elképzelt Megatron 2000 rendszerünket. Számos oka van annak, hogy az itt leírt megvalósítás alkalmatlan azokra az alkalmazásokra, amelyek jelentős mennyiségű adattal dolgoznak, vagy többen használják az adatokat. A következőkben a teljesség igénye nélkül felsoroljuk a legfontosabb problémákat:

- A sorok elrendezése a lemezen nem megfelelő, mivel nem nyújtja azt a rugalmasságot, amire az adatbázis módosításakor szükség lenne. Például ha kicseréljük a `VM`-et `GAZD`-ra (Gazdaságtani Tanszék) egy `Hallgatók`hoz tartozó sorban, akkor az egész fájlt újra kell írni, mivel a `VM`-et követő összes karaktert két hellyel lejjebb kell mozgatni a fájlban.
- A keresés nagyon költséges. Mindig el kell olvasnunk a teljes relációt, még akkor is, ha a lekérdezés olyan értéket (vagy értékeket) használ, amely csak egy sort eredményezne, mint az 1.2. példában, ahol végig kellett néznünk a teljes `Hallgatók` relációt, annak ellenére, hogy egyedül a Smith nevű hallgatóra voltunk kíváncsiak.
- A lekérdezés végrehajtása úgynevezett „nyers erő" típusú, pedig az összekapcsoláshoz hasonló műveletek végrehajtására sokkal ügyesebb módszerek is léteznek. Például majd látni fogjuk, hogy az 1.2. példához hasonló lekérdezés esetén nem szükséges megnézni az összes sorpárt (ahol a pár egyik eleme az egyik, a másik eleme a másik relációnak a sora) még akkor sem, ha nem írtuk elő egy hallgató (Smith) nevét a lekérdezésben.
- Nincs mód arra, hogy a hasznos adatokat a központi memóriában eltároljuk (puffereljük); az összes adatot a lemezről szedjük le minden egyes esetben.
- Nincs konkurenciakezelés. Egyszerre több felhasználó is módosíthat egy fájt, és emiatt az eredményt nem lehet előre tudni.
- Nem beszélhetünk megbízhatóságról, ugyanis adatokat veszíthetünk el, ha összeomlik a rendszer, vagy műveleteket hagyunk félbe.
- Kicsi a biztonság. Lehet, hogy az alapul szolgáló operációs rendszer valamilyen durva módon felügyeli a hozzáférést, például a különböző felhasználóknak meg van engedve, vagy meg van tiltva, hogy hozzáférjenek egy adott relációt tartalmazó fájlhoz, de nem lehet, hogy valaki mondjuk egy reláció bizonyos attribútumaihoz férhessen csak hozzá, és máshoz nem.

Ezzel a könyvvel az a szándékunk, hogy bevezessük az olvasót abba, hogyan kell ügyesebben felépíteni egy adatbázis-kezelő rendszert. Reméljük, hogy ez a tananyag mindenkinek tetszeni fog.

1.2. Egy adatbázis-kezelő rendszer áttekintése

Az 1.1. ábrán egy teljes adatbázis-kezelő rendszer vázát látjuk. Az egyvonalas dobozok a rendszer alkotórészeit jelentik, míg a dupla dobozok memóriabeli adatszerkezeteket reprezentálnak. A folytonos vonalak jelölik az olyan vezérlésátadást, ahol adatok is áramlanak, a szaggatott vonalak pedig csak az adatmozgást jelölik. Mivel az ábra bonyolult, ezért a részleteket fokozatosan tekintjük át. Először is azt javasoljuk, hogy a legfelső részen az adatbázis-kezelőhöz intézett parancsoknak két forrását különböztessük meg:

1. Szokásos felhasználók és alkalmazói programok, melyek adatokat kérnek vagy módosítják az adatokat.
2. *Adatbázis-adminisztrátor* (database administrator – *DBA*) vagy másképpen *adatbázis-rendszergazda:* egy vagy több olyan személy, akik az adatbázissémáért, illetve -struktúráért felelősek.

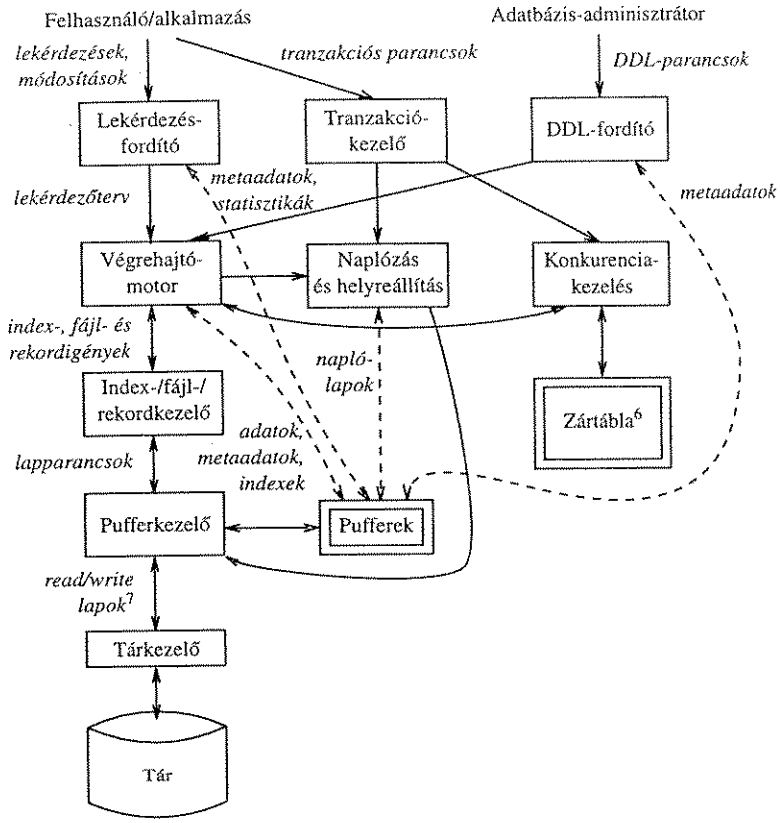
1.2.1. Az adatdefiníciós nyelv parancsai

A második fajta parancsot egyszerűbb feldolgozni. Megmutatjuk, hogyan követhető ez nyomon az 1.1. ábra jobb felső sarkából kiindulva. Például az adatbázis-rendszergazda elhatározza, hogy egy egyetemi nyilvántartás adatbázisában fel kellene venni egy olyan táblát vagy relációt, amelynek egyik oszlopa a hallgatót jelenti, másik oszlopa a tantárgyat, amit a hallgató felvett az indexébe, és egy harmadik oszlop pedig azt a jegyet tartalmazza, amilyen jegyet kapott a hallgató ebből a tárgyból. A DBA úgy dönt, hogy csak A, B, C, D és F jegyeket lehet megengedni.[5] Ez az információ a struktúráról és az értékekre vonatkozó megszorításról (constraint) mind az adatbázisséma része. Az 1.1. ábrán látható, ahogy megadja ezt az adatbázis-rendszergazda, akinek természetesen speciális felhatalmazásra van szüksége ahhoz, hogy végrehajthasson sémamódosító parancsokat, mivel ezek a parancsok alapvető hatással vannak az adatbázisra. Ezeket a sémamódosító *DDL-parancsokat* (ahol a „DDL" az adatdefiníciós nyelv angol megfelelőjének, a „data-definition language"-nek a rövidítése) a DDL-feldolgozó (DDL-processor) elemzi, majd továbbítja a végrehajtómotornak (execution engine), amely továbbadja az index-/fájl-/rekordkezelőnek, hogy az változtassa meg a metaadatokat, vagyis az adatbázis sémainformációját.

1.2.2. A lekérdezés feldolgozásának áttekintése

Az adatbázis-kezelő rendszerrel kapcsolatos kölcsönhatások döntő többsége az 1.1. ábra bal oldalán lévő útvonalat követi. A felhasználó vagy az alkalmazói program olyan működést indít el, amelynek nincs hatása az adatbázissémára, viszont hatással

[5] Az USA-ban általában betűkkel jelölik az osztályzatokat, + jellel a féljegyet, F jelenti az elégtelent az angol *failure* – bukás szó miatt, és A vagy A+ a legjobb jegy. *A fordító megjegyzése.*



1.1. ábra. *Az adatbázis-kezelő rendszer alkotórészei*

lehet az adatbázis tartalmára (módosító parancs esetében), illetve adatokat gyűjthet ki az adatbázisból (lekérdezés esetében). Két olyan útvonal van, amely mentén a felhasználó cselekménye hatást gyakorol az adatbázisra:

1. *A lekérdezés megválaszolása.* A lekérdezésfordító elemzi és optimalizálja a lekérdezést. Az eredményül kapott lekérdezés-végrehajtási tervet (röviden *lekérdezőtervet*), vagy a lekérdezés megválaszolásához szükséges tevékenységek sorozatát továbbítja a *végrehajtómotornak*. A végrehajtómotor kisebb adatdarabokra (tipikusan rekordokra vagy egy reláció soraira) vonatkozó kérések sorozatát adja át az erőforrás-kezelőnek (resource manager). Az erőforrás-kezelő ismeri a relációkat tar-

[6] A szakirodalomban igen elterjedt a lock table kifejezés is. *A fordító megjegyzése.*

[7] Azok az adategységek, amelyeket beolvasunk (*read*), illetve kiírunk (*write*). *A fordító megjegyzése.*

talmazó *adatfájlokat,* a fájlok rekordjainak formátumát, méretét és az *indexfájlokat* is. Az indexfájlok segítenek abban, hogy az adatfájlok elemeit gyorsan meg lehessen találni. Az adatkéréseket az erőforrás-kezelő lefordítja lapokra (page), és ezeket a kéréseket továbbítja a *pufferkezelőnek* (buffer manager). A pufferkezelő szerepét az 1.2.3. részben fogjuk megvizsgálni, most röviden csak annyit, hogy a másodlagos adattárolón, általában lemezen tartjuk folyamatosan az adatokat, és a pufferkezelő feladata, hogy innen az adatok megfelelő részét hozza be a központi memória puffereibe. A pufferek és a lemez közti adatátvitel egysége általában egy lap vagy egy lemezblokk. A pufferkezelő információt cserél a tárkezelővel (storage manager), hogy megkapja az adatokat a lemezről. Megtörténhet, hogy a tárkezelő az operációs rendszer parancsait is igénybe veszi, de tipikusabb, hogy az adatbázis-kezelő a parancsait közvetlenül a lemezvezérlőhöz intézi.

2. *A tranzakció feldolgozása.* A lekérdezéseket és más tevékenységeket tranzakciókba csoportosíthatjuk. A tranzakciók olyan egységek, amelyeket atomosan és elkülönítve kell végrehajtani, ahogy ezt a fejezet bevezetőjében megbeszéltük. Gyakran minden egyes lekérdezés vagy módosítás önmagában is egy tranzakció. Ezenkívül a tranzakció végrehajtásának *tartósnak* (durable) kell lennie, ami azt jelenti, hogy bármelyik befejezett tranzakció hatását még akkor is meg kell tudni őrizni, ha a rendszer összeomlik a tranzakció befejezése utáni pillanatban. A tranzakciófeldolgozót két fő részre osztjuk:

   a) Egy *konkurenciavezérlés-kezelő* vagy *ütemező* (scheduler) felelős a tranzakciók elkülönítésének és atomosságának biztosításáért.
   b) Egy *naplózás- és helyreállítás-kezelő* felelős a tranzakciók tartósságáért.

   Ezeket az alkotórészeket az 1.2.4. részben fogjuk tovább vizsgálni.

1.2.3. A központi memória pufferei és a pufferkezelő

Egy adatbázis adatai normális esetben másodlagos adattárolón helyezkednek el. A mai számítógépes rendszerek esetében a másodlagos adattárolón általában mágneses lemezt kell érteni. Igen ám, de ahhoz, hogy bármilyen hasznos műveletet végezzünk az adatokon, az adatoknak a központi memóriában kell lenniük. Így aztán az adatbázis-kezelő rendszer egyik komponensének, nevezetesen a *pufferkezelőnek* a feladata, hogy a rendelkezésre álló memóriát pufferekre ossza fel. A *pufferek* azok a lap méretű területek, ahová a lemezblokkokat lehet betölteni. Ebből következik, hogy ha az adatbázis-kezelő valamelyik komponensének lemezen lévő információra van szüksége, akkor vagy közvetlenül, vagy a végrehajtómotoron keresztül kapcsolatba kell lépnie a pufferekkel és a pufferkezelővel. A különböző komponenseknek a következő információkra lehet szükségük:

1. *Adatok:* magának az adatbázisnak a tartalma.
2. *Metaadatok:* az adatbázisséma, mely leírja az adatbázis struktúráját és megszorításait.

3. *Statisztikák:* az adatbázis-kezelő által az adatok tulajdonságairól (például az adatbázis különböző relációinak vagy más komponenseinek méreteiről, a bennük szereplő értékekről) összegyűjtött és tárolt információ.
4. *Indexek:* olyan adatszerkezetek, melyek támogatják az adatok hatékony elérését.

A pufferkezelő leírásának és szerepének sokkal teljesebb taglalását találhatjuk meg a 6.8. részben.

1.2.4. A tranzakció feldolgozása

Ahogy már említettük, természetes dolog, hogy egy vagy több adatbázis-műveletet egy *tranzakcióba* csoportosítunk, mely egy olyan munkaegység, amit atomosan és más tranzakcióktól látszólag elkülönítve kell végrehajtani. Ezenfelül az adatbázis-kezelő rendszer a tartósságot is garantálja: azaz egy befejezett tranzakció munkája sosem veszhet el. Így a *tranzakciókezelő* fogadja az alkalmazás *tranzakciós parancsait.* Az alkalmazás azt is megmondja a tranzakciókezelőnek, hogy mikor kezdődnek és végződnek a tranzakciók, és még egyéb információt is ad az alkalmazás elvárásairól (például lehet, hogy nem akarja megkövetelni az atomosságot). A tranzakciófeldolgozó a következő feladatokat hajtja végre:

1. *Naplózás:* annak érdekében, hogy a tartósságot biztosítani lehessen, az adatbázis minden változását külön feljegyezzük (naplózzuk) lemezen. A *naplókezelő* (log manager) többféle eljárásmód közül választja ki azt, amelyiket követni fog. Ezek az eljárásmódok biztosítják azt, hogy teljesen mindegy, mikor történik a rendszerhiba vagy a rendszer összeomlása, a *helyreállítás-kezelő* (recovery manager) meg fogja tudni vizsgálni a változások naplóját, és ez alapján vissza tudja állítani az adatbázist valamilyen konzisztens állapotába. A naplókezelő először a pufferekbe írja a naplót, és egyeztet a pufferkezelővel, hogy a pufferek alkalmas időpillanatokban garantáltan íródjanak ki lemezre, ahol már az adatok túlélhetik a rendszer összeomlását.
2. *Konkurenciavezérlés:* a tranzakcióknak úgy kell látszódniuk, mintha egymástól függetlenül, elkülönítve végeznénk el őket. A legtöbb rendszerben igazából sok tranzakciót kell egyszerre végrehajtani. Így aztán az ütemező (konkurenciavezérlés-kezelő) feladata, hogy meghatározza az összetett tranzakciók résztevékenységeinek egy olyan sorrendjét, amely biztosítja azt, hogy ha ebben a sorrendben hajtjuk végre a tranzakciók elemi tevékenységeit, akkor az összhatás megegyezik azzal, mintha a tranzakciókat tulajdonképpen egyenként és egységes egészként hajtottuk volna végre. A tipikus ütemező ezt a munkát azáltal látja el, hogy az adatbázis bizonyos részeire elhelyezett *zárakat* (lock) karbantartja. Ezek a zárak megakadályoznak két tranzakciót abban, hogy rossz kölcsönhatással használják ugyanazt az adatrészt. A zárakat rendszerint a központi memória *lock-táblájában* (lock table) tárolja a rendszer, ahogy ez az 1.1. ábrán is látható. Az ütemező azzal befolyásolja a lekérdezések és más adatbázis-műveletek végrehajtását, hogy megtiltja a végrehajtómotornak, hogy hozzányúljon az adatbázis zár alá helyezett részeihez.

A tranzakciók ACID-tulajdonságai

A helyesen implementált tranzakciókról rendszerint azt szokás mondani, hogy eleget tesznek az ACID-tulajdonságoknak, ahol a betűk a tulajdonságok angol megfelelőinek kezdőbetűit jelölik:

- Az „A" jelöli az atomosságot (atomicity), azaz a tranzakciók „mindent vagy semmit" jellegű végrehajtását.
- Az „I" jelenti az elkülönítést (isolation), vagyis azt a tényt, hogy minden tranzakciónak látszólag úgy kell lefutnia, mintha ez alatt az idő alatt semmilyen másik tranzakciót sem hajtanánk végre.
- A „D" a tartósságot (durability) jelöli, azaz azt a feltételt, hogyha egyszer egy tranzakció befejeződött, akkor már soha többé nem veszhet el a tranzakciónak az adatbázison kifejtett hatása.

A hiányzó „C" betű a konzisztenciát helyettesíti (consistency). Minden adatbázisnak vannak konzisztenciamegszorításai, vagy másképpen az adatelemek közti kapcsolatokra vonatkozó elvárásai. Ilyen például, hogy egy bizonyos attribútumkulcs vagy a hallgató nem vehet fel nyolcnál több tárgyat egyszerre stb. A tranzakcióktól elvárjuk, hogy megőrizzék az adatbázis konzisztenciáját. Ezt a témát a 9.1. részben fogjuk részletesebben tanulmányozni.

3. *Holtpont feloldása:* a tranzakciók az ütemező által engedélyezett zárak alapján versenyeznek az erőforrásokért. Így előfordulhat, hogy olyan helyzetbe kerülnek, amelyben egyiküket sem lehet folytatni, mert mindegyiknek szüksége lenne valamire, amit egy másik tranzakció birtokol. A tranzakciókezelő feladata, hogy ilyenkor közbeavatkozzon, és töröljön, abortáljon egy vagy több tranzakciót úgy, hogy a többit már folytatni lehessen.

1.2.5. A lekérdezésfeldolgozó

Az adatbázis-kezelő rendszer *lekérdezésfeldolgozó* részének van a legnagyobb hatása arra, amit a felhasználó is lát a működés hatékonyságából. Az 1.1. ábrán a lekérdezésfeldolgozó két részből áll:

1. A *lekérdezésfordító* a kérdést belső formátumra fordítja le. Ezt *lekérdezőtervnek* hívjuk. Ez valójában nem más, mint azoknak a műveleteknek a sorozata, amelyeket az adatokon el kell végezni. A lekérdezőterv műveletei gyakran a jól ismert relációs algebra műveleteinek implementálásai, ahogy ezt majd a 6.1. részben látni fogjuk. A lekérdezőfordító három fő részből áll:

a) *Lekérdezéselemző,* mely a kérdés szöveges formájából egy fastruktúrát hoz létre.
b) *Lekérdezés-előfeldolgozó,* mely egyrészt tartalmilag, azaz szemantikusan ellenőrzi a kérdést (például megbizonyosodik afelől, hogy a kérdésben szereplő összes reláció létezik), másrészt a nyelvi elemző fát (parse tree) átalakítja a kiindulási lekérdezőtervet reprezentáló, algebrai műveletekből álló fává.
c) *Lekérdezésoptimalizáló,* mely a kiindulási lekérdezőtervet átalakítja az aktuális adatok alapján a lehető legjobb műveletsorozattá.

A lekérdezésfordító a metaadatok és az adatokra vonatkozó statisztikák alapján tudja nagy valószínűséggel eldönteni, hogy melyik műveletsorozat lesz a leggyorsabb. Például egy index létezése az egyik tervet sokkal gyorsabbá teheti egy másik tervhez képest.

2. A *végrehajtómotor* felelős a választott lekérdezőterv minden egyes lépésének a végrehajtásáért. A lekérdezőmotor az adatbázis-kezelő rendszer egyéb komponensének legtöbbjével is kapcsolatba lép. Ez történhet közvetlenül vagy a puffereken keresztül. Ahhoz, hogy kezelhessük az adatokat, előbb az adatbázisból be kell hozni őket a pufferekbe. Ehhez kapcsolatba kell lépni az ütemezővel is, nehogy zár alá helyezett adathoz akarjunk hozzáférni. Ezenkívül a naplókezelővel is kapcsolatba kell lépni, hogy az adatbázis-változások biztosan megfelelő módon legyenek naplózva.

1.3. A könyv vázlatos felépítése

Az adatbázisrendszerek implementálásának témaköre nagy vonalakban három részre osztható fel:

1. *Tárkezelés:* hogyan használjuk hatékonyan a másodlagos tárat az adatok tárolására és a gyors elérésre.
2. *Lekérdezésfeldolgozás:* hogyan lehet hatékonyan végrehajtani a nagyon magas szintű nyelven (például SQL-ben) megfogalmazott lekérdezéseket.
3. *Tranzakciókezelés:* hogyan támogathatók az 1.2.4. részben leírt, ACID-tulajdonságú tranzakciók.

A fenti témák mindegyikét a könyv több fejezeten keresztül tárgyalja.

1.3.1. Előismeretek

A könyv ugyan nem tételez fel semmilyen előismeretet az adatbázis-kezelők implementálásáról, mégis olyan tankönyvnek szántuk, amely egy adatbázisokkal foglalkozó tantárgysorozat második tantárgyához vagy egy egyféléves, de átfogóbb tantárgyhoz nyújt segítséget. A könyv a Jeff Ullman és Jennifer Widom: *Adatbázisrendszerek.*

Alapvetés című könyv folytatásának is tekinthető. Ez a korábbi könyv a következőkkel foglalkozik:

1. *Adatbázis-tervezés:* az adatbázisséma közvetlen, magas szintű specifikálása, valamilyen jelölésrendszert alkalmazva, ami lehet az egyed/kapcsolat modell vagy akár az ODL (Object Description Language – Objektumleíró nyelv), valamint a tervek implementálása az SQL-nyelv adatdefiníciós részének segítségével.
2. *Adatbázis-programozás:* lekérdezések és adatbázis-módosító parancsok írása valamilyen megfelelő nyelven, például SQL-ben.

Az adatbázis-tervezés technológiájának nem sok köze van az adatbázis-kezelő rendszerek implementálásához, de szükséges, hogy az olvasó ismerje a relációs modellt és azt, hogy az adatokat hogyan reprezentáljuk a relációk segítségével. Erre azért van szükség, mivel a könyvben elmondottak jó része azzal foglalkozik, hogyan tároljuk a relációkat, hogyan optimalizáljuk a relációkra vonatkozó lekérdezéseket, és hogyan vezéreljük a relációkhoz történő hozzáféréseket valamilyen (például zárolási) módszerekkel. Emellett csak akkor látjuk tisztán a lekérdezések feldolgozása mögött rejlő technológiát, ha az SQL-programozást is ismerjük. Ezeknek a témáknak egy gyors áttekintését adja az 1.4. rész.

Továbbá ismertnek tételezzük fel a *fájlok* (azaz adatok tárolására használható nevesített tárterületek) fogalmát. Azt is elvárjuk, hogy az olvasónak legyenek előismeretei egy hagyományos fájlrendszer felépítéséről. A fájlrendszer az operációs rendszernek az a része, amely kezeli az operációs rendszer fájljait. Egy adatbázis-kezelő rendszer egészen másképp kezeli a fájlokat, de ennek a fontos témakörnek is átvesszük majd az alapjait.

1.3.2. A tárkezelés áttekintése

A könyv tárkezelésről szóló fejezetekkel kezdődik. A 2. fejezet bevezeti a memóriahierarchiát, de a legnagyobb részletességgel majd azt fogjuk inkább vizsgálni, hogy milyen módon kell tárolni és elérni a lemezen az adatokat, ugyanis a másodlagos adattárak, különös tekintettel a lemezekre, központi jelentőségűek abban, ahogy egy adatbázis-kezelő rendszer kezeli az adatokat. A lemezalapú adatok vonatkozásában bevezetjük a blokkmodellt, amely az adatbázisrendszerben történő majdnem minden tevékenységre kihat.

A 3. fejezet kapcsolatot teremt az adatelemek (relációk, sorok, attribútumértékek, illetve más adatmodell esetén az előbbieknek megfelelő fogalmak) tárolása és az adatok blokkmodelljére vonatkozó követelmények között. Ezután megnézzük azokat a fontos adatszerkezeteket, amelyeket indexek készítésére használunk. Már említettük, hogy az index egy olyan adatstruktúra, amely az adatok hatékony elérését támogatja. A 4. fejezet áttekinti a fontosabb egydimenziós indexstruktúrákat, indexszekvenciális fájlokat, B-fákat és tördelőtáblákat. Ezeket az indexeket használják hagyományosan az adatbázis-kezelő rendszerek az olyan lekérdezések támogatására, ahol egy attribútumértéket adunk meg, és azokat a sorokat keressük, amelyekben ez az érték szerepel. Az 5. fejezet a többdimenziós indexeket tárgyalja, amelyek olyan adatszerkezetek, amik speciális alkalmazásokhoz, például földrajzi adatbázisokhoz használhatók. Ezeknél a tipikus kérdések valamilyen tartomány, terület tartalmát kérdezik le.

A fenti indexstruktúrák az olyan összetett SQL-lekérdezéseket is támogatni tudják, amikor kettő vagy több attribútumértékre szól a korlátozás. Így nem csoda, hogy némelyikük feltűnik a kereskedelmi forgalomban kapható adatbázis-kezelő rendszerekben is.

1.3.3. A lekérdezésfeldolgozás áttekintése

A 6. fejezet bevezeti a relációs algebrát, amivel a lekérdezések végrehajtása is leírható. Ebben a fejezetben találjuk meg a lekérdezések végrehajtásával foglalkozó alapokat, beleértve számos olyan algoritmust, amelyek a kulcsfontosságú relációs műveletek (például relációk összekapcsolása) hatékony implementálására szolgálnak.

A 7. fejezetben áttekintjük a lekérdezésfordítónak és az optimalizálónak a felépítését. A vizsgálatot a lekérdezések elemzésével és a tartalmi, szemantikus ellenőrzésükkel kezdjük. A következő részben megnézzük, hogyan alakítható át egy lekérdezés SQL-ből relációs algebrába. Ezután a *logikai lekérdezőterv* kiválasztásával foglalkozunk. A logikai lekérdezőterv egy algebrai kifejezés, amely az adatokon elvégzendő speciális műveleteket reprezentálja a műveletek sorrendjére vonatkozó szükséges megszorításokkal együtt. Végül a *fizikai lekérdezőterv* kiválasztását nézzük meg. Ez utóbbi magában foglalja a műveletek speciális sorrendjének megadását és az egyes műveleteket implementáló algoritmust is.

1.3.4. A tranzakciófeldolgozó áttekintése

A 8. fejezetben látni fogjuk, miképp támogatja egy adatbázis-kezelő rendszer a tranzakciók tartósságát. Az alapötlet, hogy készítünk egy olyan naplót, amibe az adatbázis minden változtatása belekerül. Tudjuk, hogy bármi elveszhet, ami a központi memóriában van és nem lemezen egy összeomlás (például áramkimaradás) esetén, ezért körültekintőnek kell lennünk, hogy az adatbázis változását és a változást rögzítő naplót alkalmas sorrendben vigyük át a pufferből lemezre. Sok különböző naplózási módszer létezik, de mindegyik valamennyire korlátozza a további tevékenységeink szabadságát.

Ezután a 9. fejezetben rátérünk a konkurenciakezelésre, amely biztosítja az atomosságot és az elkülönítést. A tranzakciókat adatbáziselemek olvasási és írási műveleteiből álló sorozatnak tekintjük. Ebben a fejezetben az a fő kérdés, hogyan kezeljük az adatbáziselemekre elhelyezett zárakat, milyen különböző típusú zárak használhatók, hogyan lehet a tranzakcióknak megengedni, hogy zárakat tehessenek az elemekre, vagy elengedjék, feloldják a zárakat. Emellett azt is tanulmányozni fogjuk, hogyan biztosítható az atomosság és elkülönítés akkor, ha nem használunk zárakat.

A 10. fejezet összefoglalja a tranzakciófeldolgozásról tanultakat. Áttekintjük azt,

hogy hogyan egyeztethetők össze a naplózásra és a konkurenciára vonatkozó elvárásaink. Az ezekre vonatkozó követelményeket a 8. és 9. fejezetben fektettük le. A tranzakciókezelő másik fontos feladatát, a holtpontkezelést is itt nézzük meg alaposabban. A 10. fejezetben találjuk meg a konkurenciavezérlés kiterjesztését osztott környezetre. Végül bevezetjük annak a lehetőségét, hogy a tranzakciók hosszúak is lehetnek, azaz nem a másodperc ezredrészéig tartanak, hanem órákig vagy akár napokig is. Ha egy hosszú tranzakció zárolja az adatokat, akkor ezzel nagy káoszt idézhet elő azok között a lehetséges felhasználók között, akik éppen ezeket az adatokat használnák. Ez arra késztet bennünket, hogy újragondoljuk a konkurenciavezérlést azokra az alkalmazásokra, amelyekben hosszú tranzakciók is szerepelhetnek.

1.3.5. Az információintegráció áttekintése

Az adatbázisrendszerek fejlődését az utóbbi években nagyban meghatározta az a törekvés, hogy megteremtődjön a lehetősége annak, hogy különböző *adatforrások* (adatbázisok és/vagy nem adatbázisrendszerrel kezelt információforrások) úgy működjenek együtt, mintha egy nagy egésznek a részei lennének. A 11. fejezetet annak szenteljük, hogy áttekintsük ennek az *információintegrációnak* nevezett új technológiának a fontosabb szempontjait. Tárgyaljuk az integráció alapvető módszereit, amelyek sorában foglalkozunk az adatforrások lefordított és integrált másolataival, amit *adatraktárnak* vagy *adattárháznak* (data warehouse) hívunk, és az adatforrások halmazának virtuális nézeteivel, amit *közvetítőnek* (mediator) nevezünk.

1.4. Az adatmodellek és nyelvek áttekintése

Ebben a részben röviden áttekintjük az SQL-t és a relációs modellt. Ezenkívül ismertetjük az objektumok fogalmát, ahogy azt az objektumorientált adatbázisokban használjuk. A példákat az Ullman–Widom: *Adatbázisrendszerek. Alapvetés* című könyvből vesszük át.

1.4.1. A relációs modell áttekintése

A *reláció soroknak* a halmaza, a sorok pedig értékekből álló listák. Egy reláció minden sorának ugyanannyi számú komponense van, és a különböző sorokból vett megfelelő komponensek ugyanolyan típusúak. Egy relációt úgy jeleníthetünk meg, hogy minden egyes sorát felsoroljuk egy táblázat soraként. Az oszlopok fejléceit *attribútumoknak* hívjuk. Az attribútumok a sorok komponenseinek értelmét jelentik. A reláció neve, az attribútumok nevei és az attribútumokhoz tartozó típusok együtt alkotják a reláció *sémáját*.

1.3. példa: A példákban gyakran fogjuk idézni a `Film` relációt, mely tartalmazhatná a következő sorokat:

Filmcím	*Év*	*Hossz*
Csillagok háborúja	1977	124
Erős kacsák	1991	104
Wayne világa	1992	95

Ennek a relációnak a sémája az alábbi:

```
Film(filmcím, év, hossz)
```

Az attribútumai a `filmcím`, az `év` és a `hossz`, melyekről feltehetjük, hogy rendre karakterlánc, egész, egész típusúak. A vonal alatti három sor mindegyike egy-egy relációsor. Az első sor például azt mondja, hogy a „Csillagok háborúját" 1977-ben készítették és 124 perc hosszú. □

Az *adatbázisséma* relációsémáknak a halmaza. A filmekkel kapcsolatos állandó példánkban gyakran fogjuk használni a következő relációkat:

```
Film(filmcím, év, hossz, stúdiónév[8])
Filmszínész(név, cím, neme, születési_idő)
Szerepel(filmcím, év, színésznév)
Stúdió(név, cím)
```

Az első reláció majdnem ugyanaz, mint az 1.3. példában szereplő `Film` reláció, azzal a különbséggel, hogy a sémához még hozzávettük a filmet gyártó stúdió nevét is, azért hogy szükség esetén további kapcsolatokat tudjunk majd a példáinkban létrehozni. A második reláció a filmszínészekről nyújt információt, míg a harmadik összekapcsolja a filmeket a szereplőikkel. A negyedik reláció pedig a stúdiókról ad információt. A különböző attribútumok jelentése az attribútumok nevéből kézenfekvően következik.

1.4.2. Az SQL áttekintése

Az SQL nevű adatbázisnyelv nagyon sok lehetőséggel rendelkezik. Ezek között megtalálhatók azok az utasítások, amelyek az adatbázist lekérdezik, illetve módosítják. Az

[8] Az attribútum neve általában egy szó, ezért angolul a `studioName` jelölést használják, ami utal arra, hogy angolban a stúdiónév eredetileg két szó *(studio name)* lenne. A fordításban is ezt a konvenciót fogjuk használni, azaz két szóból szükség esetén úgy alkotunk egyet, hogy nagybetűvel kezdjük a második tagot. Mivel a magyar helyesírás szerint a stúdiónév eleve egybeírandó, ezért itt nem kell használnunk a megkülönböztető nagybetűt. *A fordító megjegyzése.*

adatbázis módosítása három parancson (INSERT, DELETE és UPDATE) keresztül történik, melyek formális megadását, szintaxisát most itt nem adjuk meg. A lekérdezéseket általában „select-from-where" utasításokkal fejezzük ki, melyek általános formáját ténylegesen az 1.2. ábra mutatja. Az utasításnak csak a SELECT és FROM szavakkal kezdődő első két sorát, vagy másképpen *záradékait*[9] (clause) kötelező megadni.

```
SELECT <attribútumok listája>
FROM <relációk listája>
WHERE <feltétel>
GROUP BY <attribútumok listája>
HAVING <feltétel>
ORDER BY <attribútumok listája>
```

1.2. ábra. *Egy SQL-lekérdezés általános formája*

Egy ilyen lekérdezés eredményét, még ha nem is a lehető legjobb módon, de kiszámolhatjuk az alábbiak szerint:

1. Vegyük a FROM záradékban szereplő relációkból a sorok összes lehetséges kombinációját.
2. Dobjuk el azokat a kombinációkat, amelyek nem elégítik ki a WHERE záradék feltételét.
3. A megmaradt sorkombinációkat csoportosítsuk a GROUP BY záradékban felsorolt attribútumokhoz (ha van ilyen egyáltalán) tartozó értékeik alapján.
4. Ellenőrizzük az összes csoportra a HAVING záradékban szereplő feltételt (ha egyáltalán megadtunk ilyet), és hagyjuk el az összes olyan csoportot, amely nem felel meg ennek a feltételnek.
5. Számítsuk ki a sorokat a SELECT záradékban megadott attribútumokból és attribútumok összesítéséből[10] (aggregation) (például a csoportokon belüli összegek képzéséből).
6. Rendezzük az eredményül kapott sorokat az ORDER BY záradékban megadott attribútumlistának megfelelő értékeik alapján.

1.4. példa: Az 1.3. ábra egy olyan egyszerű SQL-lekérdezést mutat be, amelynek csak az első három záradéka van megadva. Ez a lekérdezés a Paramount stúdió által gyártott filmek címeit és a bennük szereplő színészek nevét adja meg. Megjegyezzük, hogy a filmcím és év együtt a Film reláció kulcsa, mivel két filmnek lehetne ugyan megegyező címe, de reményeink szerint ekkor viszont nem ugyanabban az évben készültek. □

[9] Szokás még mondatrésznek, klauzulának vagy klóznak is hívni. *A fordító megjegyzése.*

[10] Ezeket a statisztikai, összesítő függvényeket aggregátoroknak is szokták fordítani. *A fordító megjegyzése.*


```
SELECT színészNév, Film.filmcím
FROM Film, Szerepel
WHERE Film.filmcím = Szerepel.filmcím AND
      Film.év = Szerepel.év AND
      stúdiónév = 'Paramount';
```

1.3. ábra. *A Paramount színészeinek megkeresése*

1.5. példa: Az 1.4. ábra egy bonyolultabb lekérdezést mutat be. Először is azt kell megkeresnünk, hogy kik azok a színészek, akik legalább három filmben szerepeltek. A lekérdezésnek ezt a részét úgy tudjuk megvalósítani, hogy a Szerepel sorait a GROUP BY záradékkal a színészek neve szerint csoportosítjuk, és aztán a HAVING záradékkal kiszűrjük azokat a csoportokat, amelyeknek kettő vagy kevesebb sora van.

```
SELECT színészNév, MIN(év) AS először
FROM Szerepel
GROUP BY színészNév
HAVING COUNT(*) >= 3
ORDER BY először;
```

1.4. ábra. *Azoknak a legkorábbi éveknek a megkeresése, amikor a legalább három filmben játszó színészek először szerepeltek filmben*

Ezután a SELECT záradék azt mondja, hogy a megmaradó csoportokból elő kell állítani a színészek nevét és a legkorábbi évet, amikor a színész először szerepelt egy filmben. A select-lista második tagját, vagyis a MIN(év)-et először-re nevezzük át. Végül az ORDER BY azt mondja, hogy az eredményül kapott sorokat az először értékei szerint növekvő sorrendben kell listázni, vagyis a színészek az első filmjük évének sorrendjében fognak következni. □

Alkérdések

Az SQL egyik legerőteljesebb sajátossága az, hogy a WHERE, FROM vagy HAVING záradékokon belül lehetőségünk van alkérdések (subquery) használatára. Az alkérdés egy olyan „select-from-where" utasítás, amelynek az értékét a fent említett záradékok ellenőrzik.

1.6. példa: Az 1.5. ábra egy olyan SQL-lekérdezést mutat, amely egy alkérdéssel is rendelkezik.

```
SELECT filmcím, év
FROM Film
WHERE stúdiónév IN (
   SELECT név
   FROM Stúdió
   WHERE cím NOT LIKE '%Hollywood%'
);
```

1.5. ábra. *A nem Hollywoodban készült filmek megkeresése*

A teljes lekérdezés a nem Hollywoodban gyártott filmek címét és gyártási évét keresi meg, míg a

```
SELECT név
FROM Stúdió
WHERE cím NOT LIKE '%Hollywood%'
```

alkérdés azt az egyoszlopos relációt adja vissza, amely azon stúdiók nevéből áll, melyek címében nem fordul elő a „Hollywood" szó. Ezután ezt az alkérdést használja a külső lekérdezés WHERE záradéka arra, hogy beazonosítsa azokat a filmeket, amelyek stúdiója nem szerepel az alkérdés által meghatározott stúdiónevek halmazában. □

Nézettáblák

Az SQL egy másik fontos lehetősége, hogy *nézettáblákat* vagy röviden *nézeteket* (view) lehet definiálni. A nézettáblák valójában relációk leírásai. Ezeket a relációkat nem tároljuk, de szükség esetén elő tudjuk állítani a tárolt relációkból.

1.7. példa. Az 1.6. ábra egy nézettábla definícióját mutatja. Ez a nézettábla a Paramount stúdiók által készített filmek címét és gyártási évét határozza meg. A ParamountFilm nézettábla definícióját az adatbázisséma részeként tárolja a rendszer, de a hozzá tartozó sorokat most még nem számolja ki. Csak akkor állítja elő a sorait, amikor egy lekérdezés a ParamountFilm relációt használja. Ha ennek a lekérdezésnek nincs szüksége a teljes relációra, akkor csak a sorainak egy szükséges részhalmazát állítja elő megfelelőképpen. Mindezt azzal éri el, hogy a nézettábla definícióját beépíti a lekérdezésbe. A nézettábla sorait így valójában sohasem tároljuk az adatbázisban. □

```
CREATE VIEW ParamountFilm AS
   SELECT filmcím, év
   FROM Film
   WHERE stúdiónév = 'Paramount';
```

1.6. ábra. *A Paramount filmjeit meghatározó nézettábla*

1.4.3. A relációs és objektumorientált adatok

A könyvben tárgyaltak többsége azt tételezi fel, hogy az adatbázis relációs adatbázis: vagyis az adatokat táblákkal modellezzük, az adatelemeket relációsorokként vagy a tábla soraiként. A soroknak rögzített számú komponense van, és ezek mindegyike a relációsémában rögzített típussal rendelkezik. Az adatoknak ezt a szemléltetését sugallta az 1.3. példa. Egy ettől különböző szinten úgy is gondolhatunk egy sorra, mint egy struktúrára (C nyelven „struct"-ra) vagy egy olyan rekordra, amelynek minden mezője egy attribútumértéknek felel meg.

Egyes adatbázisrendszerekben másmilyen adatmodellt használnak, nevezetesen az adatokat objektumoknak is lehet tekinteni. Ebben a modellben az elemi adatelem az *objektum*. Az objektumokat *osztályokba* csoportosítjuk, és minden osztálynak van egy sémája, ami az osztály *jellemzőinek, tulajdonságainak* listája.

1. A jellemzők között lehetnek attribútumok, melyeket a relációsorok attribútumaihoz hasonlóan lehet feltüntetni.
2. A *kapcsolatok* (relationship) is jellemzők. Ezek kötnek össze egy objektumot egy vagy több másik objektummal. Az implementációs szinten úgy gondolhatunk egy kapcsolatra, mint az ezekre az objektumokra mutató mutatók (pointerek) listájára.
3. Vannak még *metódusoknak* (módszerek, eljárások) hívott jellemzők is, melyek tulajdonképpen olyan függvények, amiket az osztály objektumaira lehet alkalmazni.

Attól eltekintve, hogy a metódusok kódja tipikusan az objektumokon kívül lesz tárolva, az adatok objektumorientált formalizálásának többi része jól illeszkedik az általános keretünkbe. Ugyanis általában úgy gondolunk a fájlokra, mint a legnagyobb adategységekre. A fájlokat tekinthetjük egyszerűen névvel ellátott adatgyűjteményeknek. A fájlok rendszerint kisebb egységekből állnak, amelyekre a következő elnevezéseket használjuk:

a) A legelső adatbázisokban a fájlok *rekordokból* álltak, a rekordok pedig *mezőkből*. A rekord a C nyelv és a leszármazott programozási nyelvek (C++, Java) „struct"-jával rokon fogalom.
b) A relációs adatbázisban a fájlok *relációk*, melyek relációsorokból állnak. A relációsorokat pedig *attribútumok* alkotják.
c) Az objektumorientált adatbázisokban a fájlok az osztályok *előfordulásai* (extent). Egy ilyen osztály-előfordulás az osztályhoz adott pillanatban tartozó objektumok halmazát jelenti. Az osztály-előfordulások *objektumokból* állnak, az objektumoknak pedig *mezői* vannak. A mezőket másképpen példányváltozóknak (instance variable) is nevezik, melyek értékei jelenthetik az objektum attribútumait vagy jelenthetik a kapcsolódó objektumok halmazát is valamilyen kapcsolaton keresztül.

Hasznos lehet, ha összegezzük a következő hasonlóságokat:

1. A fájl, a reláció és az osztály-előfordulás hasonló fogalmak. Mind olyan értéket jelentenek, melyek valamilyen kisebb, de közös sémával rendelkező elemekből (rekordok, relációsorok vagy objektumok) állnak.
2. Egy fájl vagy reláció sémája és egy osztály definíciója szintén hasonló fogalmat takar. Mind azt írja le, hogy milyenek egy fájlnak, relációnak vagy egy osztály-előfordulásnak az elemei.
3. A rekordok, relációsorok és objektumok is hasonló fogalmak. Mindegyiküket gyakran implementálhatjuk úgy, mint ha „struct"-ok lennének a C nyelvben.

1.5. Összefoglalás

- *Adatbázis-kezelő rendszerek:* Ezek a rendszerek azzal a képességükkel jellemezhetők, hogy támogatják a nagyon nagy mennyiségű adatok hatékony elérését, és ezek az adatok hosszú ideig megőrződnek. További jellemvonásuk még, hogy támogatják a nagy kifejezőerővel rendelkező lekérdezőnyelveket. Támogatják a tartós tranzakciók konkurens végrehajtását is oly módon, hogy a tranzakciók atomosnak és más tranzakcióktól függetlennek látszanak.
- *Összehasonlítás a fájlrendszerekkel:* A hagyományos fájlrendszer alkalmatlan adatbázisrendszernek, mert nem támogatja a hatékony keresést, a kisebb adatok hatékony módosítását, az összetett lekérdezéseket, a hasznos adatok vezérelt pufferelését a központi memóriában, és nem támogatja a tranzakciók atomos és független végrehajtását sem.
- *Az adatbázis-kezelő részei:* Az adatbázis-kezelő rendszer fő részei a tárkezelő, a lekérdezésfeldolgozó és a tranzakciókezelő.
- *A tárkezelő:* Ez a komponens felelős az adatok, metaadatok (adatszerkezeteket, sémát leíró információk), indexek (adatok gyors elérését biztosító adatszerkezetek) és naplók (az adatbázis változásáról szóló feljegyzések) lemezen történő tárolásáért. A tárkezelő fontos eleme a pufferkezelő, mely a lemez tartalmának egy részét a központi memóriában tartja.
- *A lekérdezésfeldolgozó:* Ez a komponens elemzi a lekérdezéseket, egy lekérdezőterv kiválasztásával optimalizálja őket, és végrehajtja a tervet a tárolt adatokon.
- *A tranzakciókezelő:* Ez a komponens felelős az adatbázis változásainak naplózásáért. Ezáltal támogatja, hogy egy rendszerösszeomlás után helyre lehessen állítani a rendszert. Emellett felelős a tranzakciók konkurens végrehajtásáért oly módon, amely biztosítja az atomosságot (egy tranzakciót vagy teljesen végrehajtunk, vagy egyáltalán nem hajtunk végre) és az elkülönítést (a tranzakciók végrehajtása olyan, mintha más konkurens tranzakciót nem hajtanánk végre).
- *SQL:* Ez egy fontos, szabványos, relációs modellen alapuló lekérdezőnyelv. Mind a nyelv, mind a relációs modell központi jelentőségű a könyvünk nagy részében.
- *Adatfogalmak:* A fájlrendszereknek, a C-hez hasonló hagyományos programozási nyelveknek, a relációs modellnek és az objektumorientált adatmodellnek sok közös fogalma van, bár ezekre gyakran különböző szakkifejezéseket használnak. Párhuzamot lehet vonni a „struct"-ok, relációsorok és objektumok vagy a fájlok, relációk és osztályok között.

1.6. Irodalomjegyzék

A ma már közvetlenül (on-line módon) kereshető bibliográfiákban megtalálható majdnem minden aktuális adatbázisrendszerekkel foglalkozó cikk. Emiatt könyvünkben nem szándékozunk teljes irodalomjegyzéket adni, inkább csak a történeti fontosságú cikkeket, a hasznos áttekintő tanulmányokat, és azokat a másodlagos lelőhelyeket adjuk meg, ahol további cikkeket lehet találni. Michael Ley [6] elkészítette az adatbázis-kutatással foglalkozó cikkeknek egy keresésre alkalmas tárgymutatóját. Lehet keresni Alf-Christian Achilles könyvtárában is, aki folyamatosan karbantartja az adatbázis témával kapcsolatos lényeges indexeket [1].

Az ehhez a könyvhöz szükséges háttérismereteket [8]-ból lehet elsajátítani. Az SQL2 és SQL3 szabványok letölthetők az [5] anonim FTP-szerverről. Akik egy SQL2 kézikönyvet szeretnének, azoknak [4]-et ajánljuk.

A témakör technológiájához sok prototípus adatbázisrendszer implementálása járult hozzá, ezek közül a két legismertebb az IBM Almaden Kutatási központjának System R rendszere [2], és az INGRES projekt, amit Berkeleyn fejlesztettek [7]. Mindkettő olyan korai relációs rendszer, melynek köszönhetően a relációs rendszer lett a meghatározó adatbázis-technológia.

Az 1998-as „Asilomar report" [3] a legfrissebb az adatbázisrendszerek kutatásáról és irányvonalairól szóló jelentések sorában. Ebben is találunk hivatkozásokat korábbi hozzá hasonló jelentésekre.

1. `http://www.ira.uka.de/bibliography/Database.`
2. M. M. Astrahan et al., „System R: a relational approach to database management", *ACM Trans. on Database Systems* **1**:2 (1976), pp. 97–137.
3. P. A. Bernstein et al., „The Asilomar report on database research", `http://s2k-ftp.cs.berkeley.edu:8000/postgres/papers/Asilomar_Final.htm.`
4. Date, C. J. and H. Darwen, *A Guide to the SQL Standard,* Fourth Edition, Addison-Wesley, Reading, MA, 1997.
5. `ftp://jerry.ece.umassd.edu/isowg3.`
6. `http://www.informatik.uni-trier.de/~ley/db/index.html.` Egy másolat található a következő helyen: `http://www.acm.org/sigmod/dblp/db/index.html.`
7. M. Stonebraker, E. Wong, P. Kreps, and G. Held, „The design and implementation of INGRES", *ACM Trans. on Database Systems* **1**:3 (1976), pp. 189–222.
8. J. D. Ullman and J. Widom, *A First Couse in Database Systems,* Prentice-Hall, Englewood Cliffs NJ, 1997. (Magyarul Ullman–Widom: *Adatbázisrendszerek. Alapvetés,* Panem Könyvkiadó Kft., Budapest, 1998.)

2. fejezet

Adattárolás

Az egyik legfontosabb különbség az adatbázis-kezelő rendszerek és más rendszerek között az, hogy az adatbázis-kezelő rendszerek nagyon sok adatot is hatékonyan tudnak kezelni. Ebben és a következő fejezetben megismerjük azokat az alapvető technikákat, amelyek arra vonatkoznak, hogyan kell kezelni az adatokat a számítógépben. A tanulmányozásunk két részre osztható:

1. Hogyan tárolja és kezeli egy számítógépes rendszer a nagyon nagy mennyiségű adatokat?
2. Milyen reprezentációk és adatszerkezetek támogatják a legjobban ezeknek az adatoknak kezelését?

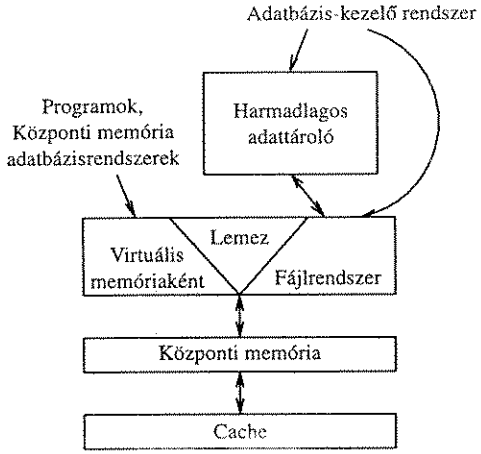
Az első kérdést ez a fejezet fedi le, míg a második a következő három fejezet témája lesz.

Ez a fejezet azzal kezdődik, hogy milyen technika használható a nagy mennyiségű adatok fizikai szintű tárolására. Megvizsgáljuk az információ tárolására használható eszközöket, elsősorban a forgó mozgást végző lemezeket. Bevezetjük a „memóriahierarchiát", és megnézzük, hogyan függ a nagyon nagy tömegű adatokra vonatkozó algoritmusok hatékonysága a különböző adatmozgatási sémáktól, ahol az adatok mozgatása a központi és a másodlagos adattároló eszközök (tipikusan lemezek) vagy esetleg „harmadlagos adattároló eszközök" között történik. Ez utóbbiak olyan robot-eszközök, melyek nagyszámú optikai lemezt vagy mágneses szalagot tároló kazetta (tape cartridge) tárolására és elérésére szolgálnak. A kétfázisú, többutas, összefésüléses rendezés nevű speciális algoritmust használjuk arra, hogy példát mutassunk olyan algoritmusra, amely hatékonyan használja a memóriahierarchiát.

A 2.4. részben számos technikát fogunk nézni arra is, hogyan lehet csökkenteni a lemezről történő adatolvasáshoz, illetve lemezre történő adatíráshoz szükséges időt. Az utolsó két rész olyan módszereket vizsgál meg, amelyekkel javítani lehet a lemezek megbízhatóságán. A felvetett problémák tartalmazzák az időszakos írási, olvasási hibákat és a lemez összeomlását is, vagyis amikor az adatok tartósan olvashatatlanná válnak.

2.1. A memóriahierarchia

Egy tipikus számítógépes rendszernek több különböző olyan része is van, amelyekben adatokat lehet tárolni. Ezeknek a részeknek az adatkapacitása akár hét nagyságrenddel is eltérhet egymástól, és a bennük tárolt adatok elérési sebessége is hét vagy még több nagyságrendben különbözhet. Az egy bájtra vonatkoztatott költség is különbözik ezeknél a komponenseknél, bár itt már sokkal kisebb az eltérés, talán három nagyságrend van a legolcsóbb és a legdrágább tárak ára között. Nem meglepő, hogy a legkisebb kapacitású eszközök kínálják a leggyorsabb elérési sebességet a legdrágább bájtra vetített költségért. A memóriahierarchia egy lehetséges sémája a 2.1. ábrán látható.



2.1. ábra. *A memóriahierarchia*

2.1.1. Cache[1]

A hierarchia legalsó szintjén találjuk a *cache*-t. A cache egy integrált áramkör („chip") vagy egy processzor chipjének a része. Alkalmas arra, hogy adatokat vagy gépi utasításokat tároljon. A cache-ben tárolt adatok, utasítások a központi memória (ami a memóriahierarchia következő szintje) egy részének, illetve bizonyos helyeinek a másolata. Időnként a cache-ben lévő értékek is megváltoznak, de a központi memória megfelelő módosítása általában későbbre halasztódik. Ennek ellenére a cache minden értéke bármelyik időpillanatban megfelel egy helynek a központi memóriában. A központi memória és a cache közti adatátvitel egysége általában csak néhány bájt. Ebből kifolyólag azt gondolhatjuk a cache-ről, hogy egyedi gépi utasításokat, egész vagy lebegő pontos számokat, vagy rövid karaktersorozatot tárol.

[1] A *cache* szó rejtett memóriát jelent, de a magyar szakirodalomban az angol szó terjedt el, bár a gyorsítótár kifejezés is használatos rá. *A fordító megjegyzése.*

A gép cache-je gyakran két szintre osztható. A *beépített cache* (on-board cache) ugyanazon a chipen található, mint maga a mikroprocesszor, míg a *második szintű cache* (level-2 cache) egy másik chipen helyezkedik el.

Amikor a gép utasításokat hajt végre, akkor mind az utasításokat, mind az utasítások által használt adatokat megpróbálja megkeresni a cache-ben. Ha nem találja ott őket, akkor a központi memóriából a cache-be másolja az utasításokat vagy az adatokat. Mivel a cache csak kevés adatot tud tárolni, ezért a cache-ből rendszerint el kell távolítanunk valamit, azért, hogy helyet biztosítsunk benne az új adatoknak. Ha az, amit el akarunk távolítani a cache-ből, nem változott meg, mióta a cache-be másoltuk, akkor semmi más teendőnk nincs, mint hogy kidobjuk a cache-ből. Ezzel szemben, ha a cache-ből kidobásra szánt értékek módosultak, akkor az új értéket a központi memória megfelelő helyére kell bemásolni.

Az egyszerű, egyprocesszoros számítógépnek nem szükséges aktualizálni a központi memória megfelelő helyét még abban az esetben sem, ha a cache-ben módosultak az adatok. Ezzel szemben az olyan multiprocesszoros rendszerben, ahol több processzor is hozzáférhet ugyanahhoz a központi memóriához és mindegyik processzor saját cache-t tart fenn, gyakran szükséges, hogy a cache módosulásait azonnal *átvezessék*, azaz azonnal módosítsák a központi memória megfelelő helyét.

Az ezredforduló idején használatos cache-ek legfeljebb egy megabájt kapacitásúak. A cache és a processzor közt az adatokat a processzor műveleti sebességével lehet olvasni vagy írni, ami rendszerint 10 nanoszekundumot (10^{-8} másodpercet) vagy kevesebbet jelent. Másrészt a cache és a központi memória közti adatelemek vagy utasítások mozgatása sokkal tovább tart, körülbelül 100 nanoszekundumot (10^{-7} másodpercet) vesz igénybe.

2.1.2. A központi memória

A tevékenységek középpontjában a számítógép *központi memóriája* áll. Mindenre, ami a számítógépben történik – utasítások végrehajtása, adatok kezelése – úgy gondolhatunk, mintha a központi memóriában jelen levő információval dolgoznánk (bár a gyakorlatban az is normális, ahogy a 2.1.1. részben tárgyalt módon a cache-be töltjük át az adatok, utasítások egy részét).

1999-ben egy tipikus számítógép körülbelül 100 megabájt (10^8 bájt) központi memóriával rendelkezett, bár lehetett kapni sokkal nagyobb, 10 vagy még több gigabájt (10^{10} bájt) központi memóriájú gépeket is.

A központi memóriák *véletlen hozzáférésűek* (random access), ami azt jelenti, hogy bármelyik bájtot ugyanannyi idő alatt lehet megkapni.[2] A központi memória adatainak tipikus elérési ideje 10 és 100 nanoszekundum között változik (azaz 10^{-8} és 10^{-7} másodperc között).

[2] Ezzel szemben egyes modern, párhuzamos működésű számítógépek központi memóriáján több processzor is osztozik. Ekkor a memória bizonyos részeinek elérési ideje eltérő lehet a különböző processzorokra vonatkozóan. Az egyik processzor akár háromszor olyan gyorsan is el tudja érni a memória egy részét, mint a másik processzor.

A számítógépes mennyiségek 2 hatványai

Általában úgy beszélünk a számítógép komponenseinek méretéről, kapacitásáról, mintha 10-nek lenne valamilyen hatványa, azaz megabájtot, gigabájtot és hasonlókat mondunk. A valóságban ezek a számok tulajdonképpen a legközelebbi 2-hatvány rövidítései. Emögött az húzódik meg, hogy úgy a legcélszerűbb megtervezni a komponenseket, például memóriachipeket, hogy 2-hatvány számú bitet tároljon. Mivel $2^{10} = 1024$ nagyon közel van ezerhez, ezért gyakran úgy teszünk, mintha 2^{10} egyenlő lenne ezerrel, és emiatt 2^{10} esetében a „kilo", 2^{20} esetében a „mega", 2^{30} esetében a „giga", 2^{40} esetében a „tera" és 2^{50} esetében a „peta" szócskákat használjuk előtagként, még akkor is, ha ezek az előtagok a tudományos beszédmódban valójában a 10^3, 10^6, 10^9, 10^{12}, 10^{15} számokra utalnak. Az eltérés annál nagyobb, minél nagyobb számokról beszélünk. Egy „gigabájt" valójában $1{,}074 \times 10^9$ bájt.

Ezekre a számokra a következő szabványos rövidítéseket használjuk: K felel meg a kilónak, M a megának, G a gigának, T a terának és végül P a petának. Így tehát 16 Gbájt 16 gigabájtot jelent, ami szigorúbb értelemben 2^{34} bájt. Mivel időnként olyan számokról is akarunk beszélni, amelyek 10-nek hagyományos értelemben vett hatványai, például ezer, millió stb., ezért ezeket az elnevezéseket továbbra is megtartjuk ezekre a hagyományos számokra, azaz ilyen esetben nem használjuk a „kilo", „mega" stb. előtagokat. Például „egymillió bájt" 1 000 000 bájtot, míg „egy megabájt" 1 048 576 bájtot jelent.

2.1.3. Virtuális memória

Program írásakor az általunk használt adatok – a program változói, a beolvasott fájlok stb. – egy *virtuális memória címterületet* foglalnak le. A program utasításai szintén a nekik megfelelő címterületet foglalják le. Sok gép használ 32 bites címeket, vagyis összesen 2^{32}, azaz körülbelül 4 milliárd különböző címet lehet megadni. Mivel minden bájtnak szüksége van saját címre, ezért a tipikus virtuális memóriát 4 gigabájtosnak tekinthetjük.

Abból kifolyólag, hogy a virtuális memória területe sokkal nagyobb, mint a szokásos központi memóriáé, az következik, hogy a teljesen kitöltött virtuális memória tartalmának legnagyobb részét valójában a lemezen tároljuk. A lemezműveletek közül a legtipikusabbakat a 2.2. részben fogjuk tárgyalni. Pillanatnyilag elég annyit tudnunk, hogy a lemez logikailag *blokkokra* van felosztva. A szokásos lemezek blokkmérete 4 K és 56 K, azaz 4 és 56 kilobájt között van. A virtuális memóriát a lemez és a központi memória között teljes blokkokban mozgatjuk. A központi memóriában a blokkokat *lapoknak* (page) hívjuk. A gép hardvere és az operációs rendszer megengedi, hogy a virtuális memória lapjait a központi memória tetszőleges részére lehessen behozni, miközben a blokkok minden bájtjára szabályosan lehet hivatkozni a virtuális memória címük alapján. A könyvünkben nem foglalkozunk azzal, hogy ezt milyen mechanizmussal lehet elérni.

Moore törvénye

Gordon Moore sok évvel ezelőtt megfigyelte, hogy az integrált áramkörök sok jellemzőjének fejlődése exponenciális görbét követ, méghozzá olyat, amely az értéket 18 hónaponként megduplázza. Az alábbiakban megadunk néhány paramétert, melyek Moore törvényének engedelmeskednek.

1. A processzorok sebessége, vagyis a másodpercenként végrehajtott utasítások száma és a processzor sebességének és árának aránya.
2. A központi memória egy bitre jutó ára és az egy chipbe tehető bitek száma.
3. A lemez egy bitre eső ára és a lemezen tárolható bájtok száma.

Másrészt vannak olyan fontos paraméterek is, melyek nem követik Moore törvényét, mivel lassabban vagy egyáltalán nem nőnek. Ezek között a lassan növekedő paraméterek között szerepel az a sebesség, hogy a központi memóriában milyen gyorsan lehet az adatokat elérni, vagy az a sebesség, amilyen gyorsan a lemez forog. Mivel ezek lassan nőnek, ezért a lemaradásuk egyre nagyobb. Emiatt az az idő, ami alatt az adatokat a memóriahierarchia szintjei között mozgatjuk, egyre tovább növekszik a számítási időhöz viszonyítva. Ennélfogva az elkövetkezendő években az várható, hogy a központi memória a cache-hez képest sokkal távolabb kerül a processzortól, és ugyanakkor a lemez adatai is még távolabb kerülnek a processzortól a fenti paraméterek tekintetében. Ennek az érzékelhető távolságnak 1999-ben már komoly hatásaival kellett számolni.

A virtuális memóriát is magában foglaló 2.1. ábrán az útvonal a hagyományos programok és alkalmazások kezelését reprezentálja. Ez nem egyezik meg egy adatbázis adatainak tipikus kezelésével. Ennek ellenére egyre nő az érdeklődés a *központi memória adatbázisrendszerek* iránt, melyek az adataikat ténylegesen a virtuális memórián keresztül kezelik, és ehhez az operációs rendszerre támaszkodnak, mivel ennek a lapozási mechanizmusával hozzák be a szükséges adatokat a központi memóriába. A központi memória adatbázisrendszerek a legtöbb alkalmazáshoz hasonlóan akkor a leghasznosabbak, ha az adatok mérete eléggé kicsi ahhoz, hogy a központi memóriában maradjanak anélkül, hogy az operációs rendszernek ki kelljen vinnie őket. Ha a gép 32 bites címterülettel rendelkezik, akkor a központi memória adatbázisrendszerek olyan alkalmazásokhoz jók, amelyeknél nem szükséges 4 gigabájtnál több adatot egyszerre a memóriában tartani. (A 4 gigabájtnál kisebb értéket kell venni, ha a gép tényleges központi memóriája 2^{32} bájtnál kisebb.) Ez a tármennyiség igen sok alkalmazás számára elegendő, de már nem elég az adatbázisrendszerek nagy és igényes alkalmazásaihoz.

A fentiek miatt a nagyméretű adatbázisrendszerek az adataikat közvetlenül a lemezen kezelik. Ezeknek a rendszereknek a méretét csak az korlátozza, hogy összesen mennyi adatot lehet tárolni az összes lemezen és a számítógépes rendszer által elérhető egyéb adattároló eszközön. Ezzel a működési móddal a következőkben fogunk foglalkozni.

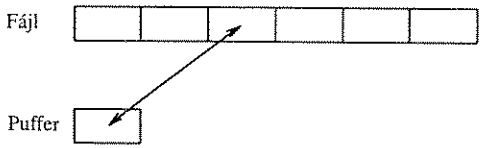
2.1.4. Másodlagos tárolás

Lényegében minden számítógépnek van valamilyen *másodlagos tárolója*, amely a tárolónak olyan formája, mely lényegesen lassabb, de lényegesen nagyobb kapacitású, mint a központi memória, ugyanakkor lényegében véletlen hozzáférésű, azaz különböző adatelemek eléréséhez szükséges időtartamok között viszonylag kicsi az eltérés. (Az eltérés okairól a 2.2. részben lesz majd szó.) A modern számítógépes rendszerek másodlagos memóriának valamilyen lemezt használnak. Ez a lemez rendszerint mágneses lemez, bár néha optikai vagy mágneses-optikai lemezeket is használnak. Ez utóbbi típusok olcsóbbak, de lehet, hogy nem támogatják azt, hogy könnyen lehessen adatokat írni rájuk, ha egyáltalában ez lehetséges, így ezeket általában csak olyan adatok archiválására szokták használni, melyek nem változnak.

A 2.1. ábrán észrevehetjük, hogy a lemezt úgy tekintjük, mint amely mind a virtuális memóriát, mind a fájlrendszert támogatja. Tehát, míg bizonyos lemezblokkok arra a célra szolgálnak, hogy egy alkalmazói program virtuális memóriájának lapjait tárolják, addig más lemezblokkok fájlokat vagy fájlok részeit tartalmazzák. A fájlokat a lemez és a központi memória között blokkokban mozgatjuk, és ezt a folyamatot az operációs rendszer vagy az adatbázisrendszer felügyeli. Egy blokk mozgatását a lemezről a központi memóriába *lemezolvasásnak* hívjuk, míg a központi memóriából a lemezre mozgatást *lemezírásnak* nevezzük. Mindkét műveletre *lemez I/O*[3]*-ként* fogunk hivatkozni. A központi memória bizonyos részei arra használatosak, hogy a fájlokat *puffereljék*, vagyis ezeknek a fájloknak blokkméretű darabjait tárolják.

Például, mikor olvasásra nyitunk meg egy fájlt, akkor az operációs rendszer valószínűleg lefoglal egy 4 KB-os blokkot a központi memóriában, ami egy puffer lesz ehhez a fájlhoz, feltéve, hogy a lemezblokkok 4 Kbájt méretűek. Kezdetben a fájl első blokkja másolódik a pufferbe. Mikor az alkalmazói program feldolgozta a fájlnak ezt a 4 Kbájtját, akkor a fájl következő blokkja másolódik a pufferbe, lecserélve annak régi tartalmát. Ez a 2.2. ábrán látható folyamat addig folytatódik, míg a teljes fájl beolvasásra nem kerül, vagy amíg a fájlt le nem zárjuk.

Az adatbázis-kezelő rendszer a lemezblokkokat saját maga kezeli, és nem hagyatkozik az operációs rendszer fájlkezelőjére, mikor blokkokat kell mozgatni a központi és a másodlagos memória között. Az is igaz viszont, hogy a kezelés alapvető pontjai lényegében megegyeznek, függetlenül attól, hogy egy fájlrendszert vagy egy adatbá-



2.2. ábra. *Egy fájl és a központi memóriában hozzá tartozó puffer*

[3] Az I az input (bemenet), az O az output (kimenet) szavak kezdőbetűjét jelölik. *A fordító megjegyzése.*

zisrendszert nézünk. Durván 10–30 milliszekundum (0,01–0,03 másodperc) ideig tart, hogy egy lemezblokkot olvassunk vagy írjunk. Ez alatt az idő alatt egy tipikus gép egymillió műveletet is végrehajthat. Ennek az a következménye, hogy általában egy lemezblokk olvasására vagy írására fordított idő a dominális feldolgozási idő, függetlenül attól, hogy a blokk tartalmával mit csinálunk. Emiatt lényeges fontosságú, hogy ha lehetséges, akkor az a lemezblokk, amiben a szükséges adatok szerepelnek, már a központi memóriának egy pufferében legyen jelen, mert ezután nem kell a lemez I/O-költségével számolnunk. Erre a problémára a 2.3. és a 2.4. részekben fogunk visszatérni, ahol példákat fogunk látni arra, hogyan foglalkozzunk azzal a nagy költséggel, amivel a memóriahierarchia szintjei közti adatmozgatás jár.

1999-ben a lemezegységek mérete általában 1-től 10-ig, vagy még ennél is több gigabájtig terjed. Ezenfelül a gépek több lemezegységet is használhatnak, így egy önálló gép reálisan akár 100 gigabájt kapacitású másodlagos memóriával is rendelkezhet. Végül is a másodlagos memória 10^5 nagyságrendben lassabb, de legalább 100-szor nagyobb kapacitású, mint a tipikus központi memória. A másodlagos memória jelentősen olcsóbb, mint a központi memória. 1999-ben a mágneses lemezegységek egy megabájtra jutó ára 5 és 10 cent közötti, míg a központi memóriánál ugyanez 1 és 2 dollár között mozog.

2.1.5. Harmadlagos tárolás

Bármilyen nagy is lehet több lemezegység együttes kapacitása, vannak olyan adatbázisok, amelyek sokkal nagyobbak annál, hogy egy vagy akár sok gép lemezein lehessen tárolni őket. Például áruházláncok terabájtnyi adatot őriznek a forgalmukkal kapcsolatban. A műholdas felvételek alapján összegyűjtött adatok is gyakran terabájtokban mérhetők, sőt a műholdak a közeljövőben petabájt (10^{15} bájt) információt fognak visszaadni évente.

Ezeknek az igényeknek a kielégítésére fejlesztették ki a *harmadlagos tárolókat*, melyek terabájtokban mérhető adatmennyiséget tudnak tárolni. A harmadlagos tárolókat azzal lehet jellemezni, hogy a másodlagos tárolókhoz képest sokkal lassabban olvassák vagy írják az adatokat, de ugyanakkor sokkal nagyobb a kapacitásuk, és ráadásul a mágneses lemezekhez viszonyítva kisebb az egy bájtra jutó költségük. Míg a központi memória egyforma idő alatt ér el tetszőleges adatot, a lemez esetében a különböző adatok elérési ideje csak kis tényezőben különbözik, addig a harmadlagos tárolóeszközök esetében az elérési idők széles sávban változhatnak attól függően, hogy milyen közel van az adat az olvasási/írási ponthoz. Az alapvető harmadlagos tárolóeszközök a következők:

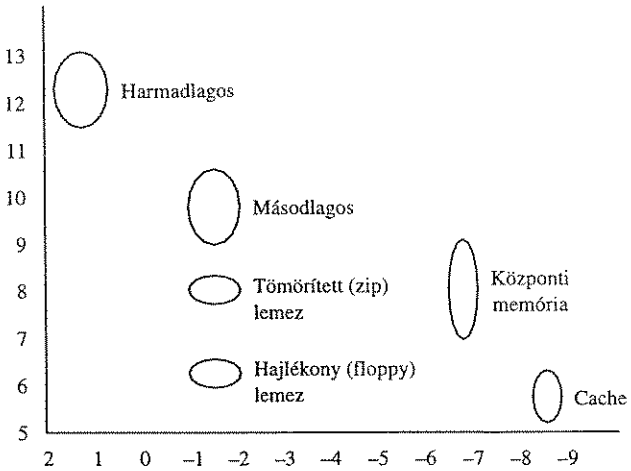
1. *Ad hoc szalagos tárolás.* A legegyszerűbb – és az elmúlt években sokáig az egyetlen – megvalósítása a harmadlagos tárolónak az, hogy az adatokat orsós vagy kazettás szalagokra mentjük ki, és ezeket utána tárolókba helyezzük. Amikor valamilyen információra van szükség a harmadlagos tárolóról, akkor a kiszolgáló személyzetből egy operátor megkeresi, és felteszi a szükséges szalagot a szalagolvasóra. Az információ megkeresése úgy történik, hogy a szalagot a megfelelő pozícióra tekerjük, és az információt a szalagról bemásoljuk a másodlagos vagy a központi memóriába. A harmadlagos tárolóra írás ennek a megfordítottja, azaz a megfelelő szalagon a megfelelő helyet keressük meg, és a lemezről kimásoljuk az információt a szalagra.
2. *Többlemezes optikai lemeztár* (juke box[4]). A lemeztár CD-ROM-tárolókból áll. (CD = kompaktlemez [compact disk]; ROM = csak olvasható memória [read-only memory]. Ezek azok az optikai lemezek, melyeken általában a szoftvereket forgalmazzák.) Az optikai lemezeken a biteket fekete vagy fehér kis területek reprezentálják, így a biteket úgy lehet elolvasni, hogy lézerrel megvilágítjuk az adott helyet, és megnézzük, hogy a fény visszaverődik-e. Egy robotkar is része a lemeztárnak, amely gyorsan ki tudja emelni bármelyik CD-ROM-ot, és be tudja helyezni egy CD-olvasóba. A CD tartalmát vagy annak egy részét így be lehet olvasni a másodlagos memóriába. Speciális eszközök nélkül általában nem lehet írni a CD-kre. Már kaphatók viszonylag olcsó CD-írók, és valószínűleg hamarosan gazdaságos lesz olyan harmadlagos tárolókat készíteni, amelyek írni és olvasni is tudják az optikai lemezeket.
3. *Szalagsilók.* A siló egy szoba nagyságú eszköz, mely szalagtárolókat tartalmaz. A szalagokat robotkarokkal lehet elérni, melyek aztán a kiválasztott szalagot a szalagolvasók valamelyikéhez tudják vinni. Ekképpen a siló a korábbi ad hoc szalagtárolásnak egy automatizált formája. Mivel a berendezés számítógéppel vezérelt és a szalag megkeresése is automatikus, ezért a működés legalább egy nagyságrenddel gyorsabb, mint az emberi személyzettel működtetett ad hoc rendszeré.

A mágneses szagot tartalmazó kazetta kapacitása 1999-ben körülbelül 50 gigabájt. Ennél fogva a szalagsilók sok terabájtot tudnak tárolni. A CD-k szabványosan 2/3 gigabájtot képesek tárolni, de a következő generációs szabvány szerint ez a kapacitás körülbelül 2,5 gigabájtra fog emelkedni. Már most is kaphatók olyan CD-ROM-lemeztárak, melyek sok terabájt adat tárolására képesek.

A harmadlagos tárolóeszközök esetében az adat-hozzáférési idő néhány másodperctől néhány percig is terjedhet. Egy lemeztár vagy szalagsiló robotkarja a kívánt CD-ROM-ot vagy kazettát másodpercek alatt megtalálja, ezzel szemben egy embernek valószínűleg perceket vesz igénybe, hogy megkeresse a szalagokat. Ha már egyszer betöltötték a CD-t az olvasóba, akkor a CD bármely része a másodperc törtrésze alatt elérhető, viszont ehhez képest sokkal több másodpercig is tarthat, míg egy szalag megfelelő részét a szalagolvasó eszköz olvasófeje alá tudjuk csévélni.

Végeredményben harmadlagos tárolás esetén az adatelérés körülbelül 1000-szer is lassabb lehet, mint a másodlagos memória esetében (az első esetében másodpercekben, a második esetben a másodperc ezredrészében mérhető ez az idő). Ezzel szemben a harmadlagos tárolóeszközök 1000-szer nagyobb kapacitásúak, mint a másodlagos eszközök (itt terabájtok állnak szemben gigabájtokkal). A 2.3. ábra egy log-log skálán mutatja az elérési idők és a kapacitások közti kapcsolatot a memóriahierarchia mind a négy általunk tárgyalt szintjén. Az ábrán szerepeltetjük a tömörített (zip) és a hajlé-

[4] A *juke box* szó szerint több lemezzel működő zenegépet jelent. *A fordító megjegyzése.*



2.3. ábra. *Az elérési idő és a kapacitás összevetése a memóriahierarchia különböző szintjein*

kony (floppy) lemezt is, mivel ezek szintén általánosan használt tárolóeszközök, habár adatbázisok esetén másodlagos tárolásra nem túl tipikus a használatuk. A vízszintes tengely beosztása tízhatvány értékű másodperceket jelöl, azaz például a –3 valójában 10^{-3} másodpercet, vagyis egy milliszekundumot jelöl. A függőleges tengely a bájtokat szintén 10 hatványaként jelöli, azaz például a 8-as 100 megabájtot jelent.

2.1.6. Felejtő és nem felejtő tárolás

Egy további megkülönböztetés az adattároló eszközök terén az, hogy vajon *felejtenek* vagy *nem felejtenek*. A felejtő eszköz, mint ahogy a neve is mondja, elfelejt minden benne tárolt adatot, ha áramszünet történik. Ezzel szemben a nem felejtő eszköz hosszú ideig épségben megőrzi a tartalmát még akkor is, ha kikapcsolják vagy ha áramkimaradás történik. A felejtés kérdése valóban fontos, mivel az adatbázis-kezelő rendszerek egyik jellemző vonása éppen az, hogy képesek megőrizni az adatokat áramszünet esetén is.

A mágneses anyagok megtartják a mágnesességüket áramhiány esetében is, így a mágneses lemezek és szalagok nem felejtő tárolók. Ugyanígy a CD-hez hasonló eszközök is megtartják a beléjük égetett fekete és fehér pöttyöket áram jelenléte nélkül is. Sok ilyen eszköz esetében valójában lehetetlen megváltoztatni azt, amit a felületükre írtak. Ebből kifolyólag az összes másodlagos és harmadlagos tárolóeszköz nem felejtő típusú.

A központi memória viszont általában felejtő típusú. Kiderült, hogy egy memóriachipet egyszerűbb áramkörökből lehet megtervezni, ha az is megengedett, hogy egy bit értéke bizonyos idő, például egy perc múlva megváltozzon. Ez az egyszerűsítés csökkenti a chip egy bitjére eső költségét. Tulajdonképpen az történik, hogy a bitet reprezentáló elektromos töltés lassan elfolyik abból a tartományból, mely ennek a bitnek volt kijelölve. Emiatt egy úgynevezett *dinamikus véletlen hozzáférésű memória*chipre, DRAM-ra (dynamic random-access memory) van szükség, amely periodikusan olvassa és újraírja a teljes tartalmát. Ha kimegy az áram, akkor ez a frissítés nem működik, és a chip hamarosan elveszti, amit tárolt.

Ha egy adatbázisrendszer felejtő központi memóriájú gépen fut, akkor minden változást ki kell menteni a lemezre, mielőtt a változást az adatbázis részének tekinthetnénk, mert különben azt kockáztatjuk, hogy áramkimaradás esetén elveszítjük az információt. Ennek következményeként a lekérdezések és az adatbázis-módosítások nagyszámú lemezírást fognak eredményezni, melyek bizonyos részére nem is lenne szükség, ha nem lennénk kénytelenek minden információt minden időben megőrizni. Egy másik lehetőség az, hogy olyan központi memóriát használjunk, amely nem felejtő. A nem felejtő memóriachipek egyre olcsóbbá váló, új típusa a *flash memória*. Egy további lehetőség, hogy a hagyományos memóriachipekből egy úgynevezett RAM-lemezt készítünk, melynek a tápegységét egy elemmel is kiegészítjük.

2.1.7. Feladatok

2.1.1. feladat: Tegyük fel, hogy 1999-ben egy tipikus számítógép 500 megahertz gyorsaságú processzorral rendelkezik, 10 gigabájt a lemeze és a központi memóriája 100 megabájtos. Tegyük fel továbbá, hogy Moore törvénye, miszerint ezek a tényezők 18 hónaponként megduplázódnak, az idők végtelenségéig igaz marad.

* a) Mikor lesznek tipikusak a terabájt kapacitású lemezek?
b) Mikor lesz tipikus a gigabájt központi memória?
c) Mikor lesz tipikus a terahertz gyorsaságú processzor?
d) Milyen lesz egy tipikus konfiguráció (processzor, lemez, memória) 2008-ban?

! **2.1.2. feladat:** Data parancsnok, aki a *Star Trek: The Next Generation* közismert amerikai sci-fi szappanopera sorozat egyik android szereplője a 24. századból, egyszer büszkén kijelentette, hogy az ő processzora 12 teraop gyorsaságú, azaz 12 tera műveletet hajt végre másodpercenként. Igaz, hogy a műveletek száma és a processzor órajelének frekvenciája általában nem egyezik meg, de most tegyük fel, hogy azonosak, azaz Data processzora 12 terahertz gyorsaságú. Tegyük fel, hogy Moore törvénye még 400 évig fennáll. Ekkor valójában milyen gyorsaságú lenne Data parancsnok processzora?

2.2. Lemezek

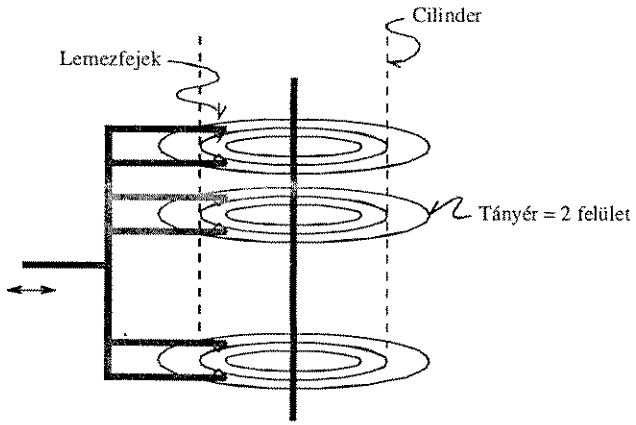
Az adatbázis-kezelő rendszerek egyik fontos jellemzője a másodlagos tárolók használata. A másodlagos tárolás szinte kizárólag mágneses lemezeken alapul. Így tehát ahhoz, hogy az adatbázis-kezelő rendszerek implementálásában használt ötleteket megindokolhassuk, először meg kell vizsgálnunk részletesen a lemezek működését.

2.2.1. A lemezek mechanikája

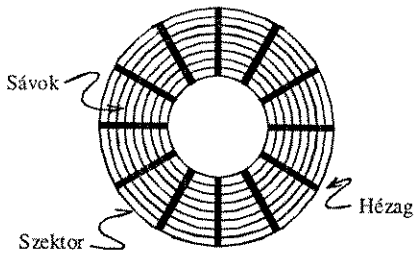
A 2.4. ábrán egy lemezmeghajtó két alapvető fontosságú mozgó részét láthatjuk, ezek a *lemezgyűjtemény* és a *fejszerelvény*. A lemezgyűjtemény egy vagy több kör alakú *tányérból* áll, melyek egy központi tengely körül forognak. A tányérok felső és alsó felülete is vékonyan be van vonva egy mágneses anyaggal, amelyek a biteket tárolják. A 0-t egy kis terület mágnesességének egyik iránya reprezentálja, míg az 1-et pont az ellenkező irányú mágnesesség jelenti. A lemeztányérok manapság szokásos átmérője 3,5 hüvelyk, bár készültek lemezek egy hüvelyktől több láb hosszig terjedő átmérővel is.

A bitek tárolására szolgáló helyet *sávokba* (track) szervezzük. A sávok koncentrikus köröket jelentenek egyetlen tányéron. A sávok a tányér felszínét szinte teljesen betöltik, kivéve a tengelyhez legközelebbi területet, ahogy az a 2.5. ábrán felülnézetben látható. Egy sáv sok pontból áll, melyek mindegyike egyetlen bitet reprezentál azáltal, hogy milyen irányú abban a pontban a mágnesesség iránya.

A sávok *szektorokba* vannak szervezve. A szektorok a kör olyan szeletei, melyek *hézaggal* (gap) vannak elválasztva. A hézag attól hézag, hogy semelyik irányba sincs



2.4. ábra. *Egy tipikus lemez*



2.5. ábra. *Egy lemez felszínének felülnézete*

Szektorok kontra blokkok

Ne feledjük, hogy a szektor a lemeznek egy fizikai egysége, míg a blokk egy logikai egység. A blokk a lemezt használó szoftverrendszer – operációs rendszer vagy például adatbázis-kezelő rendszer – alkotása. Ahogy már említettük, manapság a blokkok tipikusan legalább akkorák, mint a szektorok, és a blokkok egy vagy több szektorból állanak. Ennek ellenére nem világos, hogy a blokkok miért ne lehetnének egy szektornak a töredékei, miáltal több blokkot lehetne egy szektorba betenni. Tulajdonképpen léteztek olyan korábbi rendszerek, amelyek pont ezt a stratégiát követték.

mágnesezve.[5] A lemez olvasásának és írásának tekintetében a szektor képezi a felbonthatatlan egységet. Ugyanígy a hibák tekintetében is oszthatatlan egységet képez. Ha történetesen a mágneses felület egy kis részen valahogy elromlik úgy, hogy ez a rész nem tud információt tárolni, akkor az ezt a részt tartalmazó szektor teljes egészében használhatatlanná válik. A hézagok, melyek gyakran a teljes sáv 10%-át is kiteszik, arra használhatók, hogy segítséget nyújtsanak a szektorok elejének megtalálásához. A 2.1.3. részben említettük, hogy a blokkok olyan logikai adategységek, melyeket a lemez és a központi memória között mozgatunk. Egy ilyen blokk egy vagy több szektorból állhat.

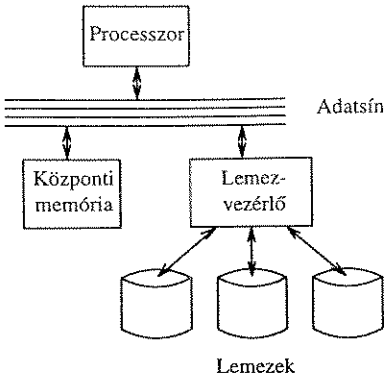
A 2.4. ábrán látható második mozgatható darab a fejszerelvény, amely a *lemezfejeket* tartja, áll. Mindegyik felülethez egy fej tartozik, mely igen-igen közel kerül a felülethez, de sohasem érintkezik vele (mert ha belezuhan a fej, akkor a lemez tönkremegy, és minden elvész, amit rajta tároltunk). A fej kiolvassa az alatta haladó mágnesességet, és arra is képes, hogy megváltoztassa ezt a mágnesességet, miáltal információt ír a lemezre. A fejek mindegyike egy-egy karhoz csatlakozik. A különböző felületekhez tartozó karok együttesen mozognak ki vagy be. Így a karok is részei a merev fejszerelvénynek.

2.2.2. A lemezvezérlő

Az egy- vagy többlemezes meghajtó vezérlését a lemezvezérlő végzi, mely a következő képességekkel jellemezhető kis processzor:

1. Vezérli azt a mechanikus szerkezetet, mely a fejszerelvényt mozgatja úgy, hogy a fejeket pozicionálja egy adott sugár mentén. Ezen a sugáron minden felülethez tartozó fej alatt a felület egy sávja fog elhelyezkedni, mely így írható és olvasható.

[5] A 2.5. ábrán minden sáv ugyanannyi szektorra van felosztva, bár a 2.1. példában látni fogunk olyan esetet, mikor a szektorok száma sávonként különbözhet, nevezetesen a külső sávokon több szektor van, mint a belsőkön.



2.6. ábra. *Egy egyszerű számítógépes rendszer vázlatos felépítése*

Az összes fej alatt egy időben elhelyezkedő összes sávot együttesen *cilindernek* nevezzük.

2. Kiválasztja azt a felületet, amelyről olvasni kell, vagy amelyre írni akarunk. Kiválasztja az ezen a felületen a fej alatti sáv egy szektorát. A vezérlő felelős azért is, hogy észrevegye, hogy a forgó tengely mikor jutott el abba a helyzetbe, hogy a kívánt szektor éppen a fej alatt kezdődik.
3. Átadja a kívánt szektorból kiolvasott biteket a számítógép központi memóriájába vagy fordítva, a központi memóriából vett biteket kiírja a kívánt szektorba.

A 2.6. ábra egy egyszerű egyprocesszoros számítógépet mutat. A processzor az adatsínen (data bus) keresztül tartja a kapcsolatot a központi memóriával és a lemezvezérlővel. A lemezvezérlő sok lemezt is irányíthat; az ábrán ennek a számítógépnek három lemezét tüntettük fel.

2.2.3. A lemeztárolók jellemzői

A lemeztechnológiák állandóan változnak, ahogy egyre csökken az egy bit tárolásához szükséges hely. 1999-ben a lemezekkel kapcsolatos tipikus mérőszámok a következők:

- *A lemezgyűjtemény forgási sebessége.* Általános az 5400 fordulat/perc (rpm), azaz egy körülfordulás 11 milliszekundum alatt történik, de találhatók ennél magasabb és alacsonyabb értékek is.
- *Az egységhez tartozó tányérok száma.* A tipikus meghajtónak körülbelül 5 tányérja és ennélfogva 10 felülete van, de 30 felülettel rendelkező lemezmeghajtókat is lehet találni. A szokásos hajlékony- és tömörített lemezeknek egyetlen tányérjuk van két felülettel. Az egyoldalas hajlékonylemez már elavult, de azért még ilyet is lehet találni. Ennek egy tányérja van, de csak egyetlen felületét lehet használni.

- *A felületen található sávok száma.* Egy felületen akár 10 000 sáv is lehet, bár a hajlékonylemezeknél ez a szám sokkal kisebb; lásd a 2.2. példát.
- *A sávokra jutó bájtok száma.* A szokásos lemezmeghajtóban 10^5 vagy több bájt jut egy sávra, de a hajlékonylemezek sávjai természetesen kevesebb bájtot tartalmaznak. Már említettük, hogy a sávokat szektorokra osztjuk. A 2.5. ábrán 12 szektor szerepel minden sávban, de a modern lemezek esetében 500 szektor is juthat egy sávba. A szektorok egyenként körülbelül 512 és 4096 közé eső számú bájtot tartalmazhatnak.

2.1. példa: A *Megatron 747* lemeznek a következő jellemző paraméterei vannak, melyek egyébként is tipikusak a középméretű vintage–1999 lemezmeghajtó esetén.

- Négy tányérja van nyolc felülettel.
- 2^{13}, azaz 8192 sáv van mindegyik felületen.
- Átlagosan $2^8 = 256$ szektor van minden sávban.
- $2^9 = 512$ bájt van minden szektorban.

A lemez kapacitását úgy kapjuk, hogy összeszorozzuk ezeket a számokat, azaz 8 felületszer 8192 sávszor 256 szektorszor 512 bájt, az annyi, mint 2^{33} bájt. Tehát a *Megatron 747* egy 8 gigabájtos lemez. Egy sáv 256-szor 512 bájtot, azaz 128 Kbájtot tartalmaz. Ha a blokkok 2^{12}, azaz 4096 bájtosak, akkor egy blokk 8 szektort használ fel, így 256/8 = 32 blokk jut egy sávra.

A *Megatron 747* felületének átmérője 3,5 hüvelyk. A sávok a felületek külső részére esnek, és a belső részen 0,75 hüvelyk nincs lefoglalva. A sugárirányban vett bitsűrűség így 8192 bit/hüvelyk, mert ennyi a sávok száma.

A sávokra számolt bitsűrűség sokkal nagyobb. Először tegyük fel, hogy minden sáv az átlagos számmal, 256-tal megegyező számú szektort tartalmaz. Tételezzük fel azt is, hogy a hézagok a sávok 10%-át foglalják el. Ekkor a sáv 128 Kbájtja (ami 1 Mbit) a sáv 90%-át foglalja el. A legkülső sáv hossza $3{,}5\pi$, azaz körülbelül 11 hüvelyk. Ennek a hossznak a kilencven százaléka, azaz körülbelül 9,9 hüvelyk tartalmaz 1 Mbitet. Ennél fogva a sávnak a bitek tárolására lefoglalt részén a bitsűrűség körülbelül 100 000 bit/hüvelyk.

Másrészt a legbelső sáv átmérője csak 1,5 hüvelyk. Így $0{,}9 \times 1{,}5 \times \pi$, azaz 4,2 hüvelyken kellene 1 Mbitet tárolni. A bitsűrűség a belső sávokon tehát 250 000 bit/hüvelyk körül lesz.

Mivel a sűrűség a belső és külső sávokon nagyon eltérne, ha a szektorok és bitek minden sávra egyformák lennének, ezért a *Megatron 747,* más modern meghajtókhoz hasonlóan a külső sávokon több szektort tárol, mint a belső sávokon. Például 256 szektort tárolhatunk sávonként a lemez középső harmadában, de csak 192 szektort a belső harmadában, ugyanakkor 320 szektort a külső harmadba eső sávokban. Ha így teszünk, akkor a sűrűség a legkülső és legbelső sávok bitsűrűsége között változna, azaz 114 000 bit/hüvelyk és 182 000 bit/hüvelyk közé esne. □

2.2. példa: A lemezek összehasonlításának az egyik végen szerepel a szabványos 3,5 hüvelykes hajlékonylemez. Ennek két felülete van, mindegyiken 40 sáv található, azaz

összesen 80 sáv. A lemez kapacitása körülbelül 1,5 Mbájt adat, függetlenül attól, hogy MAC-en vagy PC-n formáztuk meg. Ez azt jelenti, hogy 150 000 bit (18 750 bájt) jut minden sávra. A rendelkezésre álló területnek körülbelül a negyedét a hézagok és más lemezadminisztrációs részek töltik ki, mindkét típusú formázás esetén. □

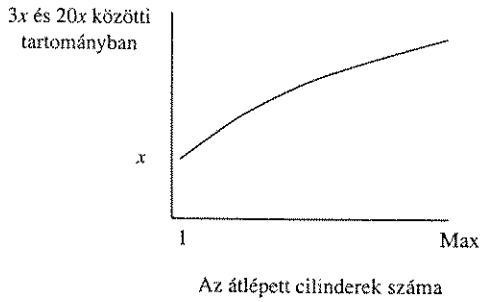
2.2.4. A lemezhozzáférés jellemzői

Az adatbázis-kezelő rendszerek tanulmányozása során nemcsak arra van szükségünk, hogy megértsük, miként tároljuk az adatokat a lemezeken, hanem arra is, hogy hogyan történik az adatok kezelése. Mivel minden számítás a központi vagy a cache memóriában történik, ezért az egyetlen lényeges kérdés a lemezekkel kapcsolatban az, hogyan mozgatjuk az adatblokkokat a lemez és a központi memória között. Már a 2.2.2. részben megemlítettük, hogy blokkokat (azaz azokat az egymás utáni szektorokat, melyek a blokkot tartalmazzák) akkor lehet írni vagy olvasni, mikor:

a) a fejek arra a cilinderre állnak, amelyik tartalmazza azt a sávot, melyen a blokk elhelyezkedik, és
b) a blokkot tartalmazó szektorok a lemezfej alá kerülnek a teljes lemezgyűjtemény forgása által.

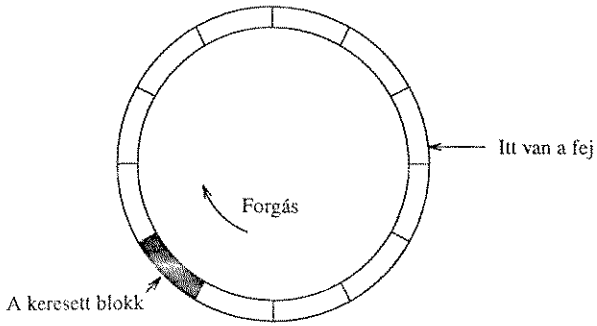
A blokkolvasási parancs kiadásának időpontja és a blokk tartalmának központi memóriába kerülésének időpontja közt eltelt időt a lemez *késésének* (latency) hívjuk. Ez a következő komponensekből áll össze:

1. A milliszekundum tört részével egyező idő telik el, míg a processzor és a lemezvezérlő feldolgozza az igényt. Ezt az időt a továbbiakban elhanyagoljuk. Mivel más folyamatok is olvashatják és írhatják ugyanakkor a lemezt, ezért a kiszolgálásért versenyeznek a folyamatok a lemezvezérlő és az adatsín esetében is, ami szintén késlekedést eredményezhet, de ezt az időt is elhanyagoljuk.
2. Időbe telik, míg a fejszerelvényt a megfelelő cilinderre állítjuk. Ezt hívjuk *keresési időnek,* ami akár 0 is lehet, ha a fej véletlenül pont a megfelelő cilinderen áll. Ha



2.7. ábra. *Keresési idő a megtett távolság függvényében*

nem áll a cilinderen, akkor a fejszerelvénynek minimális időbe telik, míg elkezd mozogni, az is időbe telik, míg megáll, és ehhez adódik még a megtett távolsággal nagyjából arányos idő. Az elinduláshoz, a következő sávra lépéshez, a megálláshoz szükséges minimális idő tipikusan néhány milliszekundum, míg az összes sávon keresztülhaladás maximuma 10 és 40 milliszekundum között változik. A 2.7. ábra érzékelteti, hogyan változik a keresési idő a távolság függvényében. Azt látni erről, hogy ha egycilindernyi távolság megtételéhez valamilyen x értékű keresési időre van szükség, akkor a maximális keresési idő $3x$ és $20x$ közé esik. Gyakran az átlagos keresési idővel jellemzik a lemez sebességét. A 2.3. példában látni fogjuk, hogy kell kiszámolni ezt az átlagot.
3. Ahhoz is idő kell, hogy a lemez úgy forduljon, hogy a blokkot tartalmazó szektorok közül az első kerüljön a fej alá. Ezt hívjuk *rotációs késésnek.* Egy tipikus lemez körülbelül egyszer fordul teljesen körbe 10 milliszekundum alatt. A kívánt szektor átlagosan félúton helyezkedik el a körön a fejekhez képest, így fél fordulatra van szükség, hogy elérjék a megfelelő cilindert. Az átlagos rotációs késés tehát körülbelül 5 milliszekundum. A 2.8. ábra mutatja be a rotációs késés problémáját.



2.8. ábra. *A rotációs késés oka*

4. *Átviteli időnek* nevezzük azt az időtartamot, ami alatt a blokk szektorai és a köztük levő hézagok forgás közben elhaladnak a fej alatt. Mivel a tipikus lemez körülbelül 100 000 bájtot tartalmaz sávonként, és nagyjából 10 milliszekundumonként fordul egyet, ezért ez az jelenti, hogy körülbelül 10 Mbájtot lehet olvasni a lemezről másodpercenként. Így az átviteli idő egy 4096 bájtos blokk esetén kevesebb, mint fél milliszekundum.

2.3. példa: Vizsgáljuk meg, hogy mennyi időbe telik egy 4096 bájtos blokkot beolvasni a *Megatron 747* lemezről. Először is ismernünk kell a lemez egyes időparamétereit:

- A lemez forgási sebessége 3840 fordulat/perc (rpm); vagyis egy teljes körfordulatot 1/64 másodpercenként tesz meg.
- A fejszerelvény mozgásánál az elindulás és megállás egy milliszekundumig tart. Minden 500 cilinderrel történő elmozdulás további egy milliszekundumot jelent.

Tehát a fejek egy sávot 1,002 milliszekundum alatt tesznek meg. Így ahhoz, hogy a legkülső sávból a legbelső sávig terjedő 8191 sáv távolságot megtegyék összesen körülbelül 17,4 milliszekundum szükséges.

Számoljuk ki a 4096 bájtos blokk olvasásához szükséges minimális, maximális és átlagos időt. A minimális idő éppen az átviteli idő, mivel a vezérlőre vonatkozó sorban állási időtől és az egyéb adminisztrációs időtől eltekintünk. Ez azt jelenti, hogy ekkor már a blokkot tartalmazó sáv felett tartózkodik a fej, és ráadásul éppen a blokk első szektora fog elhaladni a fej alatt.

Mivel a *Megatron 747* lemezen egy szektorban 512 bájt van (a 2.1. példában adtuk meg a lemez fizikai jellemzőit), így a blokk nyolc szektort foglal el. A fejnek ezért összesen nyolc szektor és a köztük levő hét hézag fölött kell elhaladnia. Emlékezzünk vissza arra, hogy a hézagok a kör 10%-át foglalják el, és a szektoroké a maradék 90%. Egy kör mentén 256 szektor és 256 hézag található. Így a hézagok a 360 fokos szögből összesen 36 fokot fednek le, míg a szektorok 324 fokot, azaz a 8 szektor és 7 hézag által lefedett szög összesen:

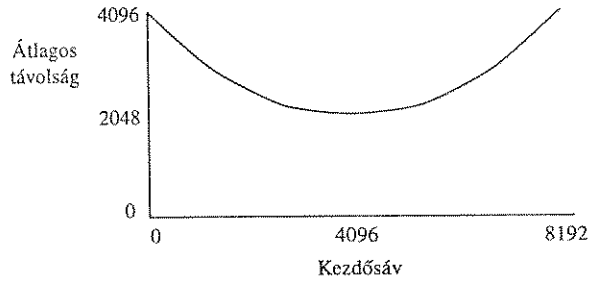
$$36 \times \frac{7}{256} + 324 \times \frac{8}{256} = 11{,}109.$$

Az átviteli idő emiatt (11,109/360)/64 másodperc; mivel osztani kell 360-nal, hogy megkapjuk, hogy a teljes körülfordulásnak milyen törtrészére van szükség, és aztán osztani kell még 64-gyel, mert a *Megatron 747* 64-szer fordul körbe egy másodperc alatt. Így az átviteli idő, ami egyben a minimális késés is körülbelül 0,5 milliszekundum.

Most nézzük meg, hogy egy blokk olvasásához maximálisan mennyi időre lehet szükség. Az a legrosszabb eset, ha a fejek a legbelső cilinderen állnak, míg a kívánt blokk a legkülső cilinderen helyezkedik el (vagy éppen fordítva). Először is a vezérlőnek el kell vinnie a fejeket a megfelelő helyre. Már az előbb észrevettük, hogy a *Megatron 747*-nek 17,4 milliszekundumra van szüksége ahhoz, hogy a fejeket az összes cilinderen keresztülvigye. Ez a mennyiség az olvasáshoz szükséges keresési idő.

Irányzatok a lemezvezérlők felépítésében

Mivel a digitális hardverek ára szemmel láthatóan csökken, ezért a lemezvezérlők is kezdenek egyre inkább számítógépre hasonlítani, általános célú processzorral és tekintélyes véletlen hozzáférésű memóriával is rendelkeznek. Sok mindenre lehet ezeket a kiegészítő hardvereket használni. A lemezvezérlők beolvashatják és tárolhatják egy lemeznek a teljes sávját még akkor is, ha csak a sáv egyetlen blokkjára van szükségük. Ezzel a lehetőséggel nagymértékben lehet csökkenteni az átlagos blokkhozzáférési időt, ha egy sáv valamennyi vagy a legtöbb blokkjára szükségünk van. A 2.4.1. részben mutatunk néhány alkalmazást a teljes sáv vagy teljes cilinder olvasására, írására.



2.9. ábra. *A megtett átlagos távolság a fej kezdeti helyének függvényében*

A legrosszabb esetben az történhet, hogy mikor a fejek a megfelelő cilinderhez érnek, éppen akkor haladt el a kívánt blokk eleje a fej alatt. Ha azt tételezzük fel, hogy a blokkot az elejétől kezdve kell olvasni, akkor lényegében egy teljes fordulatot kell várni, azaz 15,6 milliszekundumra (a másodperc 1/64 részére) van szükség ahhoz, hogy a blokk eleje ismét elérje a fejet. Amint ez megtörténik, már csak az átviteli időt, 0,5 milliszekundumot kell megvárni, hogy teljesen beolvassuk a blokkot. Tehát a legrosszabb esetben 17,4 + 15,6 + 0,5 = 33,5 milliszekundumra van szükség.

Végül számítsuk ki a blokkolvasáshoz szükséges átlagos időt. A késés két komponensét könnyű kiszámolni: az átviteli idő mindig 0,5 milliszekundum és az átlagos rotációs késés a lemez fél fordulatához szükséges idő, azaz 7,8 milliszekundum. Ismét feltehetjük, hogy az átlagos keresési idő megegyezik azzal, amennyi a sávok felén történő áthaladáshoz szükséges. Ez persze nem teljesen igaz, mivel az a tipikus, hogy a fejek kezdetben valahol középtájon helyezkednek el, ezért átlagosan a fél távolságnál kevesebbre van szükség, hogy a kívánt cilinderhez eljussanak.

Most egy részletesebb becslést adunk arra, hogy a fejnek átlagosan mennyi sávot kell elmozdulnia. Tegyük fel, hogy a fej kezdetben a 8192 cilinder bármelyikén egyforma valószínűséggel lehet. Ha az 1. vagy a 8192. cilinderen áll, akkor az átlépett sávok átlagos száma (1 + 2 + ... + 8191)/8192, azaz körülbelül 4096. Ha a 4096. cilinderen, azaz pont középen áll a fej, akkor egyforma valószínűséggel fog a lemezen kifelé vagy befelé mozdulni, így átlagosan a sávok negyedét, azaz 2048 sávot fog megtenni. Egy kis számolással belátható, hogyha a kezdő pozíció az első cilindertől a 4096. cilinderig változik, akkor a fej által szükségszerűen megtett átlagos távolság négyzetesen csökken 4096-tól 2048-ig. Hasonlóan látható, hogy ha a kezdő pozíció 4096-tól 8192-ig változik, akkor a megtett átlagos távolság 4096-ig fog négyzetesen növekedni. Mindezt a 2.9. ábra szemlélteti.

Ha a 2.9. ábrán szemléltetett mennyiségeket integráljuk az összes pozícióra, akkor azt kapjuk, hogy az átlagos megtett távolság pont a lemez egyharmadát, azaz 2730 cilindert tesz ki. Emiatt az átlagos keresési idő egy milliszekundum plusz az az idő, ami a 2730 cilinder megtételéhez kell, azaz 1 + 2730/500 = 6,5 milliszekundum.[6] Az átla-

[6] Vegyük észre, hogy a számítás nem veszi figyelembe azt a valószínűséget, mikor a fejet egyáltalán nem kell elmozdítani, de ez az eset mindössze egyszer fordul elő a 8192 esetből,

gos késésre vonatkozó becslésünk így 6,5 + 7,8 + 0,5 = 14,8 milliszekundum; ahol az egyes tagok rendre az átlagos keresési idő, az átlagos rotációs késés és az átlagos átviteli idő. □

2.2.5. Blokkok írása

Egy blokk írási folyamata a legegyszerűbb formában teljesen hasonló ahhoz, ahogy egy blokkot olvasunk. A lemez feje a megfelelő cilinderen áll, megvárjuk, hogy a megfelelő szektor vagy szektorok forgás közben a fej alá kerülnek, és most ahelyett, hogy olvasnánk a fej alatti adatot, arra használjuk a fejet, hogy új adatot írjon. Ekkor az íráshoz szükséges minimális, maximális és átlagos idő pontosan ugyanannyi, mint az olvasás esetében.

Bonyolódik a helyzet, ha azt is ellenőrizni akarjuk, hogy a blokkot helyesen írtuk-e ki. Ekkor meg kell várnunk még egy körülfordulást, és vissza kell minden kiírt szektort olvasnunk, hogy ellenőrizhessük, hogy azt tároljuk ott, amit ki akartunk írni. A helyesség ellenőrzésének egy egyszerűbb módja, mikor ellenőrző összegeket használunk. Erről a 2.5.2. részben lesz szó.

2.2.6. Blokkok módosítása

Egy blokkot nem lehet közvetlenül a lemezen módosítani. Még akkor is, ha csak néhány bájtot (például a blokkon tárolt néhány sor egyikének egy komponensét) kívánunk módosítani, akkor is a következőképpen kell eljárnunk:

1. Beolvassuk a blokkot a központi memóriába.
2. A központi memóriában a blokk másolatán elvégezzük a kívánt változtatást.
3. A blokk új tartalmát visszaírjuk a lemezre.
4. Ha szükséges, akkor ellenőrizzük, hogy az írás helyesen történt meg.

Ezek szerint egy blokk módosításához szükséges idő kiszámításához össze kell adni az olvasási időt, a központi memóriában a változtatás végrehajtásához szükséges időt (ez rendszerint elhanyagolható a lemez írási vagy olvasási idejéhez képest), az írási időt, és ha ellenőrzés is van, akkor a lemez még egy körülfordulásához szükséges időt.[7]

feltéve, hogy a kívánt blokkot véletlenszerűen adják meg. Másrészt az is igaz, hogy az a feltevés, miszerint a blokkot véletlenszerűen választják, valójában nem is mindig tekinthető helyesnek, ahogy ezt majd a 2.4. részben látni fogjuk.

[7] Első ránézésre meglepőnek tűnhet, hogy miért tart ugyanannyi ideig írni egy éppen beolvasott blokkot, mint egy olyan blokkot, amit véletlenszerűen jelöltek ki írásra. Ha a fejek maradnak ott, ahol voltak, akkor tudjuk, hogy az íráshoz egy teljes körülfordulást kell várnunk, viszont a keresési idő zéró. Ezzel szemben az igaz, hogy a lemezvezérlő nem tudja, hogy egy alkalmazás mikor fejeződik be a blokk új értékének visszaírásával, így megtörténhet, hogy a fejek közben elmozdulnak egy másik sávra, hogy végrehajtsanak valamilyen másik lemez I/O-műveletet, mielőtt a lemezvezérlő megkapja azt az igényt, hogy a blokk új értékét vissza kell írni.

2.2.7. Feladatok

2.2.1. feladat: A *Megatron 777* lemeznek a következő paraméterei ismertek:

1. Tíz felülete van, egyenként 10 000 sávval.
2. A sávok átlagosan 1000 darab egyaránt 512 bájt nagyságú szektort tartalmaznak.
3. Minden sáv 20%-át a hézagok töltik ki.
4. A lemez forgási sebessége 10 000 fordulat percenként.
5. Ahhoz, hogy a fej n sávot mozduljon el, $1 + 0{,}001n$ milliszekundumra van szükség.

Válaszoljuk meg a *Megatron 777*-re vonatkozó következő kérdéseket.

* a) Mekkora a lemez kapacitása?
b) Ha minden sáv ugyanannyi szektort tartalmaz, akkor mekkora egy sáv szektorában a bitsűrűség?
* c) Mekkora a maximális keresési idő?
* d) Mekkora a maximális rotációs késés?
e) Ha egy blokk 16 384 bájt nagyságú (32 szektor), mekkora egy blokk átviteli ideje?
! f) Mekkora az átlagos keresési idő?
g) Mekkora az átlagos rotációs késés?

! **2.2.2. feladat:** Tegyük fel, hogy a *Megatron 747* lemez feje a 1024. sávon helyezkedik el, vagyis a sávok 1/8 részénél áll. Tegyük fel, hogy a következő igény egy olyan blokkra vonatkozik, amely egy véletlenszerűen választott sávon helyezkedik el. Számítsuk ki ennek a blokknak az olvasásához szükséges átlagos időt.

*!! **2.2.3. feladat:** A 2.3. példa végén azt számoltuk ki, hogy mekkora az az átlagos távolság, amit a fejnek kell megtennie, ha egy véletlenszerűen választott sávról egy másik véletlenszerűen választott sávra kell eljutnia, és azt kaptuk, hogy ez a távolság a sávok 1/3 része. Tegyük fel, hogy egy sáv szektorainak száma fordítottan arányos a sáv hosszával (vagy a sugárral), így a bitsűrűség minden sávra megegyezik. Tegyük fel azt is, hogy a fejet egy véletlenszerűen választott *szektorról* egy másik véletlenszerűen választott szektorra kell mozgatni. Mivel a lemez külső része felé haladva a sávokban egyre több szektor gyűlik össze, ezért azt várhatjuk, hogy a fej átlagos mozgásánál a sávok kevesebb mint egy harmadát kell csak megtennie. Tegyük fel, hogy a *Megatron 747*-hez hasonlóan a sávok olyan körökön vannak, melyek sugara 0,75 és 1,75 hüvelyk közé esik. Számoljuk ki, hogy átlagosan mennyi sávot kell elmozdulnia a fejnek, ha két véletlenszerűen választott szektor közötti távolságot kell megtennie.

!! **2.2.4. feladat:** A 2.1. példa végén azt mondtuk, hogy a maximális sávsűrűség csökkenthető, ha a sávokat három tartományba soroljuk úgy, hogy a szektorok száma tartományonként eltérhet. Most ne követeljük meg, hogy egyformák legyenek a tartományok, azaz a három tartományt elválasztó két határoló kör tetszőleges sugarú lehet, továbbá a tartományokba eső szektorok száma is változhat azzal a megkötéssel, hogy egy felületen a 8192 sávhoz tartozó bájtok száma összesen 1 gigabájt. Ekkor az öt pa-

raméter (a tartományok közti két beosztás sugarai és a sávokra eső szektorok száma a három tartományban) milyen választása esetén lesz minimális egy tetszőleges sáv maximális sűrűsége?

2.3. A másodlagos tárolók hatékony használata

Az algoritmusokról szóló tanulmányok legtöbbjében azt szokták feltenni, hogy az adatok a központi memóriában helyezkednek el és bármely két adat eléréséhez ugyannyi idő szükséges. Ezt a számítási modellt gyakran „RAM-modellnek" vagy másképpen véletlen hozzáférésű számítási modellnek nevezik. Ezzel szemben mikor egy adatbázis-kezelő rendszert implementálunk, akkor azt kell feltennünk, hogy az adatok *nem* férnek el a központi memóriában. Emiatt a hatékony algoritmusok tervezésénél számításba kell venni a másodlagos, sőt esetleg a harmadlagos tárolók használatát is. Ebből következik, hogy a nagyon nagy mennyiségű adatokat feldolgozó legjobb algoritmusok gyakran különböznek az ugyanarra a problémára vonatkozó, de csak a központi memóriát használó legjobb algoritmusoktól.

Ebben a részben elsődlegesen a központi memória és a másodlagos tárolók közti kölcsönhatással fogunk foglalkozni. Különösképpen előnyös olyan algoritmusokat tervezni, melyek korlátozzák a lemezhozzáférések számát, még akkor is, ha az algoritmus során a központi memóriában az adatokon végzett műveletek nem a lehető legjobban használják ki a központi memóriát. Hasonló elvet alkalmazunk a memóriahierarchia minden szintjén. Egy központi memóriára vonatkozó algoritmuson is lehet javítani azzal, ha figyelembe vesszük a cache méretét, és az algoritmusunkat olyannak tervezzük, hogy a cache-be átmozgatott adatokat lehetőleg minél többször használjuk fel. Hasonlóan egy harmadlagos tárolót használó algoritmusnak is figyelembe kell vennie a másodlagos és harmadlagos memória között mozgatott adatmennyiséget, és érdemes ezt a mennyiséget minimalizálni még annak az árán is, hogy a hierarchia alsóbb szintjein többletmunkát kell végezni.

2.3.1. A számítás I/O-modellje

Képzeljünk el egy számítógépet, melyen fut egy adatbázis-kezelő rendszer. A számítógép megpróbál kiszolgálni bizonyos számú felhasználót. A felhasználók az adatbázist különböző módon akarják elérni: lekérdezések és adatbázis-módosítások révén. Pillanatnyilag tegyük fel, hogy a számítógépnek egy processzora, egy lemezvezérlője és egy lemeze van. Maga az adatbázis sokkal nagyobb annál, hogy beférjen a központi memóriába. Jóllehet az adatbázis lényeges részeit pufferelhetjük a központi memóriában, mégis az az általános, hogy az adatbázisnak minden egyes olyan darabját, amit egy felhasználó el akar érni, először vissza kell nyerni a lemezről.

Fel fogjuk tenni, hogy a lemez *Megatron 747* típusú, 4 Kbájt a blokkméret, és az időtényezők megegyeznek a 2.3. példában meghatározott értékekkel. Speciálisan az átlagos blokkírási vagy olvasási idő körülbelül 15 milliszekundum. Mivel sok felhasználó van, és minden felhasználó rendszeresen lemez I/O-igényeket ad ki, ezért a lemezvezérlőnek gyakorta az igények sorát kell kielégíteni. A kiszolgálásról kezdetben azt tesszük fel, hogy mindig az elsőnek beérkezett igényt szolgálja ki először a lemezvezérlő. Ennek a stratégiának az a következménye, hogy egy adott felhasználó minden igénye véletlenszerűnek fog tűnni (vagyis a lemez feje véletlenszerű helyen fog állni az igény előtt), még akkor is, ha ez a felhasználó csak egyetlen relációhoz tartozó blokkokat olvas, és ez a reláció ráadásul a lemez egyetlen cilinderén van eltárolva. Ebben a fejezetben azt is megvizsgáljuk, hogyan lehet különböző módszerekkel a rendszer működését javítani. A *számítás I/O-modelljére* vonatkozó következő szabályt azonban végig igaznak tételezzük fel:

Az I/O-költség dominanciája: Ha egy blokkot kell mozgatni a lemez és a központi memória között, akkor az íráshoz és olvasáshoz szükséges idő sokkal nagyobb annál, amennyi a központi memóriában az adatkezeléshez szükséges. Így az algoritmushoz szükséges idő értékére jó becslést ad a blokkhozzáférések (írások és olvasások) száma, vagyis ezt az értéket kell minimalizálni.

2.4. példa: Tegyük fel, hogy az adatbázisunkban van egy R reláció, és egy lekérdezés az R reláció bizonyos k kulcsértékű sorát keresi. Látni fogjuk majd, hogy igen célszerű az R táblán egy indexet létrehozni, és ennek a segítségével azonosítani azt a lemezblokkot, amelyen a k kulcsértékű sor van. Ezzel szemben általában az már nem fontos, hogy az index azt is megmondja nekünk, hogy ez a sor a blokkon hol helyezkedik el.

Ennek az az oka, hogy ezt a 4 Kbájt méretű blokkot mindössze körülbelül 15 milliszekundum alatt lehet teljesen beolvasni, és ez alatt az idő alatt egy modern mikroprocesszor akár utasítások millióit is végre tudja hajtani. Ha már egyszer a blokk a memóriában van, akkor a k kulcsérték kereséséhez még a legbutább lineáris keresési módszerrel is elegendő néhány ezer utasítás. Emiatt az a pluszidő, amivel ez a központi memóriában végrehajtott keresés jár, kevesebb, mint a blokkhozzáférési idő 1%-a, és így ezt nyugodtan el lehet hanyagolni. □

2.3.2. Adatok rendezése a másodlagos tárolóban

Lássunk egy bővebb példát arra, hogyan kell az algoritmusokat megváltoztatni a számítási költség I/O-modellje esetén. Tekintsük az adatok rendezését abban az esetben, amikor olyan nagyon sok az adat, hogy nem fér el a központi memóriában. Azzal kezdjük, hogy egy speciális rendezési problémát vezetünk be, és valamennyire részletezzük azt a gépet, amelyen a rendezés történik.

2.5. példa: Tegyük fel, hogy egy nagy R reláció 10 000 000 sort tartalmaz. Minden relációs sort egy rekord reprezentál, melynek több mezője is lehet, és a mezők között van egy *rendezési kulcs* mező. Ezt egyszerűen „kulcsmezőnek" fogjuk hívni, ha nem téveszthető össze másféle kulccsal. Egy rendezési algoritmus célja az, hogy rendezze a rekordokat a rendezési kulcsok értékeinek növekvő sorrendjében.

Egy rendezési kulcs lehet is, meg nem is „kulcs" az SQL *elsődleges kulcsának* szo-

kásos értelmében, ahol a rekordok garantáltan egyedi értékekkel rendelkeznek az elsődleges kulcsukban. Ha a rendezési kulcs értékei ismétlődhetnek, akkor az egyenlő rendezési kulccsal rendelkező rekordok bármelyik sorrendje elfogadható. Az egyszerűség kedvéért feltesszük, hogy a rendezési kulcsok egyediek. Szintén az egyszerűség végett azt is feltételezzük, hogy a rekordok állandó hosszúságúak, nevezetesen minden rekord 100 bájt hosszú. Így a teljes reláció egy gigabájtot foglal el.

Az a gép, amelyen rendezni szeretnénk, egy *Megatron 747* lemezzel rendelkezik, és olyan 50 megabájt méretű memóriája is van, amely alkalmas arra, hogy a reláció blokkjait pufferelje. A központi memória ténylegesen 64 Mbájt, de a központi memória többi részét a rendszer használja.

Feltesszük, hogy a lemez blokkjainak mérete 4096 bájt. Így 40 darab 100 bájt méretű sort vagy rekordot tudunk egy blokkba tenni, és még marad 96 bájt, ami vagy bizonyos adminisztrációs célra használható, vagy nem használjuk fel egyáltalán semmire. A reláció így 250 000 blokkot foglal el. Az 50 Mbájt (amely, mint tudjuk $50 \times \times 2^{20}$ bájt) méretű memóriában egyszerre $50 \times 2^{20} / 2^{12}$, azaz 12 800 blokk fér el. □

Ha az összes adat elfér a központi memóriában, akkor a számos jól ismert algoritmus bármelyike tökéletesen megfelel;[8] így például használhatjuk a „Gyorsrendezés" (Quicksort) algoritmusnak valamelyik variánsát, amit általában a leggyorsabbnak tartanak. Ezenfelül olyan stratégiát követhetünk, ahol csak a kulcsmezőket kell rendezni, pontosabban a kulcsmezőkhöz olyan mutatók is hozzá vannak csatolva, amelyek a megfelelő teljes rekordokra mutatnak. Csak ha a kulcs és mutató párok már sorrendben állnak, akkor használjuk fel a mutatókat arra, hogy behozzunk minden rekordot a neki megfelelő helyre.

Sajnos, ezek az elvek nem működnek túl jól, ha az adatok tárolásához másodlagos memóriára is szükség van. Amikor az adatok nagy része a másodlagos memóriában található, akkor jobban kedvelt a rendezésnek az a megközelítése, hogy valamilyen szabályos mintát követve minden blokkot csak néhányszor mozgatunk a központi és a másodlagos memória között. Ezek az algoritmusok gyakorta kisszámú *futamot* (pass) hajtanak végre; egy futam során minden rekordot egyszer olvasunk be a központi memóriába, és egyszer írunk ki a lemezre. A következő részben megnézünk egy ilyen algoritmust.

2.3.3. Az összefésülő rendezés (Merge-Sort)

Lehet, hogy az olvasó már találkozott az összefésülő rendezés nevű rendező algoritmussal, mely azon az elven működik, hogy rendezett listákat nagyobb rendezett listákká fésül össze. A rendezett listák *összefésülése* úgy történik, hogy ismételten összehasonlítjuk minden lista legkisebb megmaradt kulcsértékét, és a kisebb kulcsú rekordot áttesszük az eredménybe, és ezt ismételjük addig, amíg csak egy lista marad. Ekkor a választott rendezés szerint az eredmény mögé kell tenni a ki nem ürült lista maradék rekordjait, és ez adja az összes rekord kívánt sorrend szerint rendezett halmazát.

[8] Lásd D. E. Knuth, *The Art of Computer Programming, 3. kötet*, Addison-Wesley, Reading MA, 1998, 2. kiadás, *Rendezés és keresés* című fejezetét. (Magyarul *A számítógép programozásának művészete*, Műszaki Könyvkiadó, Budapest, 1987.)

2.6. példa: Tegyük fel, hogy két rendezett listánk van, négy-négy rekorddal. Ahhoz, hogy még egyszerűbbé tegyük a dolgunkat, a rekordokat egyedül a kulcsukkal reprezentáljuk, a többi adatra nincs szükségünk, továbbá a kulcsok legyenek egész számok. Az egyik rendezett lista az (1, 3, 4, 9) és a másik a (2, 5, 7, 8). A 2.10. ábrán az összefésülés folyamatának állapotait figyelhetjük meg.

Lépés	*1. lista*	*2. lista*	*Eredmény*
start	1, 3, 4, 9	2, 5, 7, 8	semmi
1)	3, 4, 9	2, 5, 7, 8	1
2)	3, 4, 9	5, 7, 8	1, 2
3)	4, 9	5, 7, 8	1, 2, 3
4)	9	5, 7, 8	1, 2, 3, 4
5)	9	7, 8	1, 2, 3, 4, 5
6)	9	8	1, 2, 3, 4, 5, 7
7)	9	semmi	1, 2, 3, 4, 5, 7, 8
8)	semmi	semmi	1, 2, 3, 4, 5, 7, 8, 9

2.10. ábra. *Két rendezett listának összefésülése egy rendezett listává*

Az első lépésben a két listának a sorban elsőként álló elemeit, azaz az 1-et és a 2-t hasonlítjuk össze. Mivel $1 < 2$, ezért az 1 értéket eltávolítjuk az első listából és beteszszük az eredménybe, ez lesz az eredmény első eleme. A 2. lépésben maradék listák legkisebb elemeit, azaz most a 3-at és a 2-t hasonlítjuk össze; a 2 nyer, így aztán őt tesszük be az eredménybe. Az összefésülés a 7. lépésig folytatódik, mikor is a második lista kiürül. Ekkor az első lista maradékát, ami most csak egyetlen elemből áll, hozzácsapjuk az eredményhez, és ezzel kész is az összefésülés. Vegyük észre, hogy az eredmény úgy van rendezve, ahogy lennie kell, mivel, minden lépésben a maradék elemek közül a legkisebbet választottuk. □

A központi memóriában végrehajtott összefésüléshez szükséges idő a listák hosszának összegében lineáris. Ennek az a magyarázata, hogy az adott listák rendezettek, így csak a két lista mindenkori első elemei között lehet a legkisebb ki nem választott elem, és az összehasonlítás konstans időt vesz igénybe. Az összefésülő rendezés klasszikus algoritmusa rekurzív módon rendez, és ha n elemet kell rendezni, akkor ezt $\log_2 n$ fázisban teszi, ahogy ez a következőkben látható:

Indukciós alap: Ha egy lista egy elemet tartalmaz, akkor semmit sem kell tenni, mivel ez már így is egy rendezett lista.

Indukció: Ha egy egynél több elemből álló listát kell rendezni, akkor tetszőleges módon osszuk fel a listát két egyenlő (vagy ha az eredeti lista páratlan hosszú, akkor majdnem egyenlő) hosszú részre. Rekurzívan rendezzük a két részlistát, majd az eredményül kapott rendezett listákat fésüljük össze egy rendezett listává.

Ennek az algoritmusnak a részletes elemzése meglehetősen közismert, és számunkra most nem is túl lényeges. Röviden összefoglalva, ha $T(n)$-nel jelöljük az n elem

rendezéséhez szükséges időt, akkor ez egy összegként írható fel. Az egyik tag az n időnek egy konstansszorosa (ami egyébként a lista szétvágásából és a rendezett listák összefésüléséből ered), a másik tag pedig az az idő, ami két $n/2$ méretű lista rendezéséhez szükséges. Így a következő egyenletet kapjuk: $T(n) = 2T(n/2) + an$, ahol a valamilyen konstans. Ennek a rekurzív függvényegyenletnek a megoldása $T(n) = O(n \log n)$, vagyis $n \log n$ kifejezéssel arányos.

2.3.4. Kétfázisú, többutas, összefésülő rendezés

Ahhoz, hogy a 2.5. példában megadott gépen egy relációt rendezzünk, nem az összefésülő algoritmust fogjuk használni, hanem annak egy változatát, melyet *kétfázisú, többutas, összefésülő rendezésnek* hívunk. Az adatbázis-alkalmazások legtöbbje ezt a rendező algoritmust szereti használni. Ez az algoritmus röviden a következőkből áll:

- *1. fázis:* Készítsünk az adatainkból központi memória méretű rendezett darabokat, vagyis minden rekord legyen része egy olyan rendezett listának, amely éppen befér a rendelkezésre álló központi memóriába. Így valahány, de már *rendezett részlistákat* kapunk, melyeket a következő fázisban összefésülünk.
- *2. fázis:* Az összes rendezett részlistát fésüljük össze egyetlen rendezett listává.

Az első megállapításunk ezzel kapcsolatban, hogy ha az adatok a másodlagos tárolón helyezkednek el, akkor nem akarjuk azzal indítani a rekurziót, mint az előbb, azaz nem egy, esetleg néhány rekordból indulunk ki. Ennek az az oka, ha a rendezésre váró rekordok kitöltik a memóriát, akkor az összefésülő rendezés nem olyan gyors, mint más algoritmusok. Tehát azzal kezdjük a rekurziót, hogy a teljes központi memóriát kitöltjük rekordokkal, és a gyorsrendezéssel vagy bármilyen alkalmas központi memóriás rendező algoritmussal rendezzük a rekordokat. Ezt aztán annyiszor ismételjük, amennyiszer szükséges:

1. Töltsük ki a teljes hozzáférhető központi memóriát a rendezésre szánt eredeti reláció blokkjaival.
2. Rendezzük a rekordokat a központi memóriában.
3. Írjuk ki a rendezett rekordokat a központi memóriából a másodlagos tároló új blokkjaiba. Ezzel egy rendezett részlistát kapunk.

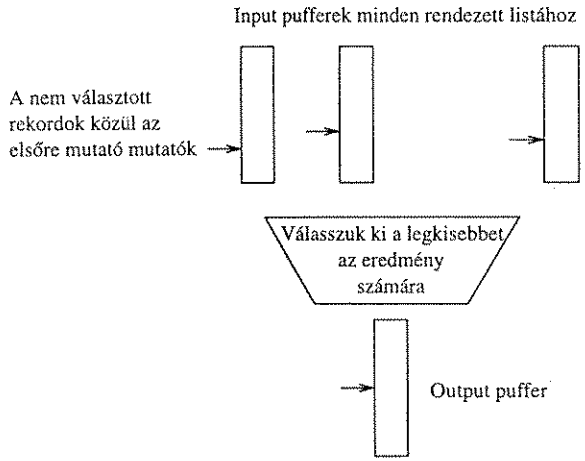
Az ily módon megadott *első fázis* végén az eredeti reláció összes rekordját egyszer olvastuk be a központi memóriába, minden rekord egy központi memória méretű rendezett részlistának lesz része, és ezeket a rendezett részlistákat kiírtuk a lemezre.

2.7. példa: Tekintsük a 2.5. példában leírt relációt. Már meghatároztuk, hogy a 250 000 blokkból egyszerre 12 800 fér el a memóriában. Tehát 20-szor töltjük ki a memóriát, rendezzük a rekordokat a központi memóriában, és írunk rendezett részlistákat a lemezre. A 20 részlista közül az utolsó rövidebb, mint a többi, mivel csak 6800 blokkot foglal el, míg a többi 19 részlista egyaránt 12 800 blokk méretű.

Mennyi időt vesz ez a fázis igénybe? A 250 000 blokk mindegyikét pontosan egyszer olvassuk el, és 250 000 új blokkot írunk összesen. Ez félmillió lemez I/O-műveletet eredményez. Pillanatnyilag tételezzük fel, hogy a blokkok véletlenszerűen helyezkednek el a lemezen. A 2.4. részben látni fogjuk, hogy ez egy olyan feltevés, amin jócskán lehet javítani. Mindenesetre a véletlenszerűségi feltevésünk mellett bármely blokk írása vagy olvasása egyformán 15 milliszekundumig tart. Ebből tehát az első fázisra vonatkozó I/O-idő összesen 7500 másodperc, ami 125 perc. Nem nehéz végiggondolni, hogy egy olyan processzorral, amely másodpercenként több tízmillió utasítást képes végrehajtani, a 10 000 000 rekord 20 rendezett részlistába sorolása az I/O-időnél sokkal kevesebb ideig tart. Tehát az első fázishoz szükséges összes idő 125 perc. □

Most térjünk rá arra, hogy hogyan fejezzük be a rendezést a rendezett részlisták összefésülésével. Megtehetnénk, hogy páronként fésüljük össze, ahogy a klasszikus összefésülő rendezés esetében, de ekkor n rendezett részlista esetén $2\log_2 n$-szer kellene a memóriából ki-be olvasni az összes adatot. Például a 2.7. példa 20 rendezett részlistáját be kellene olvasni a másodlagos tárolóról, majd kiírni ahhoz, hogy összefésüljük őket 10 rendezett részlistává, ezután újból egy teljes olvasást és írást kellene végrehajtani, hogy most már csak 5 rendezett lista maradjon, ebből aztán 4-nek az olvasása és írása után már csak 3 rendezett listánk marad és így tovább.

Jobban járunk, ha minden egyes rendezett részlista első blokkját egy központi memóriapufferbe olvassuk be. Nagyon nagy relációk esetén az is előfordulhat, hogy az első fázisban túl sok rendezett részlistát kapunk, így aztán nincs annyi hely a központi memóriában, hogy minden rendezett listából egy blokkot be tudjunk olvasni. Ezzel a problémával a 2.3.5. részben fogunk foglalkozni. A 2.5. példához hasonló adatok esetében viszonylag kevés listát kapunk, a példában pont húszat, így minden listából egy blokkot könnyen be tudunk egyszerre tölteni a központi memóriába.



2.11. ábra. *A központi memória szervezése a többutas összefésüléshez*

Kijelölünk még egy puffert az eredményblokk számára is, melyet output puffernek nevezünk. Ez a puffer a teljes rendezett listának annyi első elemét fogja tartalmazni, amennyi csak belefér. Kezdetben az output puffer üres. A pufferek elrendezését a 2.11. ábrán láthatjuk. A rendezett részlistákat a következő módon tudjuk egy olyan rendezett listába fésülni, amely már az összes rekordot tartalmazza.

1. Az összes lista megmaradt elemei közül válasszuk ki a legkisebb kulccsal rendelkező elemet. Mivel az összehasonlítás a központi memóriában történik, ezért elég egy lineáris keresést használnunk, amely a részlisták számával arányos gépi utasítást hajt végre. Természetesen használhatunk a legkisebb elem megtalálására jobb módszert is, például azt az eljárást, mely a „prioritásos sorban álláson"[9] alapul. Ez utóbbi esetén a legkisebb elemet a részlisták számának logaritmusával arányos idő alatt lehet megtalálni.
2. Tegyük a legkisebb elemet az output blokk első elérhető helyére.
3. Ha az output blokk már megtelt, akkor írjuk ki a lemezre a tartalmát, és inicializáljuk újra a központi memóriának ugyanezt a pufferét a következő output blokk tárolásához.
4. Ha az a blokk, amelyben a legkisebb elemet vettük, ezáltal kiürül, akkor ugyanezen rendezett részlistából vegyük a következő blokkot, és olvassuk be ugyanebbe a pufferbe. Ha pedig már nincs több blokk, akkor hagyjuk ezt a puffert üresen, és ennek a listának az elemeit már nem kell figyelembe vennünk a továbbiakban, mikor is a maradék elemek közül a legkisebbet keressük.

Az első fázissal ellentétben a második fázisban nem lehet előre megmondani, hogy a blokkokat milyen sorrendben fogjuk beolvasni, mivel nem tudjuk megmondani, hogy az input blokk mikor ürül ki. Azt azonban megfigyelhetjük, hogy bármelyik rendezett lista rekordjait tartalmazó blokkot pontosan egyszer olvasunk be a lemezről. Emiatt a második fázisban összesen 250 000 blokkolvasást hajtunk végre, ugyananynyit, mint az első fázisban. Hasonlóan minden rekordot egyszer beteszünk egy output blokkba, és ezeket a blokkokat a lemezre kiírjuk. Tehát a második fázisban a blokkírások száma szintén 250 000. Mivel a második fázisban a központi memóriában végzett számítást ismét csak el lehet hanyagolni az I/O-költséghez viszonyítva, így arra következtethetünk, hogy a második fázis költsége szintén 125 perc, vagyis a teljes rendezés költsége 250 perc.

2.3.5. A többutas összefésülés kiterjesztése nagyobb relációkra

Az előbb leírt kétfázisú, többutas, összefésülő rendezést nagyon nagy rekordhalmazok rendezésére is használhatjuk. Ahhoz, hogy lássuk, mekkora lehet ez a „nagyon nagy", tételezzük fel a következőket:

[9] Lásd Aho, A. V., J. D. Ullman *Foundations of Computer Science,* Computer Science Press, 1992.

Milyen nagynak kell egy blokknak lennie?

A *Megatron 747* lemezt használó algoritmusaink elemzése során azt tételeztük fel, hogy 4 Kbájt méretű minden blokk. Ezzel szemben indokolható az is, hogy ennél nagyobb blokkméret előnyösebb lenne. Emlékezzünk vissza a 2.3. példára, ahol kiszámoltuk, hogy körülbelül fél milliszekundum a 4 K méretű blokk átviteli ideje, és körülbelül 14 milliszekundum az átlagos keresési idő és rotációs késés együttesen. Ha megdupláznánk a blokkok méretét, akkor feleannyi lemez I/O-műveletre lenne szükség egy olyan algoritmus során, mint amilyen az itt leírt többutas, összefésülő rendezés. Ezzel szemben a blokkhozzáférési időben az egyetlen változás az lenne, hogy az átviteli idő 1 milliszekundumra növekedne. Ezzel tehát a rendezést hozzávetőleg feleakkora idő alatt tudnánk végrehajtani, mint az eredeti blokkméret esetén.

Ha most ismét megdupláznánk a blokkméretet 16 Kbájtra, akkor az átviteli idő 2 milliszekundumra emelkedne, míg 64 K blokkméret esetén 8 milliszekundum lenne. Ekkor már az átlagos blokkhozzáférési idő 22 milliszekundum lenne, de csak 62 500 blokkhozzáférésre lenne szükségünk ahhoz, hogy a rendezést 10-szeresére gyorsítsuk.

Különböző okai vannak annak, hogy a fentiek ellenére a blokkméret általában meglehetősen kicsi. Először is nem tudjuk hatékonyan használni az olyan blokkokat, amelyek több sávon helyezkednek el. Másodszor a kis relációk egy blokknak csak a töredékét foglalnák el, így sok elpazarolt hely lenne a lemezen. Aztán a másodlagos adattároló szervezésére léteznek olyan adatstruktúrák is, melyek jobban szeretik, ha az adatok sok blokkba vannak szétosztva, és ezért ezek kevésbé jól működnek, ha a blokkméret túl nagy. Tulajdonképpen a 2.3.5. részben látni fogjuk, hogy minél nagyobb a blokkméret, annál kevesebb rekordot tudunk az itt leírt kétfázisú, többutas módszerrel rendezni. Mindazonáltal a gépek sebességének és a lemezek kapacitásának növekedésével megfigyelhető a blokkok méretének megnövelésére irányuló tendencia.

1. A blokkok mérete B bájt.
2. M bájt használható a központi memóriában a blokkok pufferelésére.
3. A rekordok mérete R bájt.

A központi memóriában ezért összesen M/B számú puffer képezhető. A második fázisban ezek közül a pufferek közül egy kivételével mindegyik hozzá van rendelve valamelyik rendezett részlistához. A fennmaradó puffer pedig az output blokkhoz kell. Emiatt $(M/B) - 1$ rendezett részlistát lehet készíteni ebben a fázisban. Ez a szám egyezik azzal a számmal, mely megmondja, hogy hányszor kell feltölteni a központi memóriát a rendezni kívánt rekordokkal. A központi memória minden feltöltése során összesen M/R rekordot rendezünk. Így az összes rekord, amit rendezni tudunk $(M/R)((M/B) - 1)$, azaz körülbelül M^2/RB rekord.

2.8. példa: Ha a 2.5. példában vázolt paramétereket használjuk, akkor M = = 50 000 000, $B = 4096$ és $R = 100$. Tehát összesen $M^2/RB = 6{,}1$ milliárd rekordot tudunk rendezni, amely összesen egy terabájt hattizedét foglalja el.

Vegyük észre, hogy egy ekkora méretű reláció nem is fér el egy *Megatron 747* lemezen, sőt még gyakorlati szempontból elfogadható számú lemezen sem. Valószínűleg a rekordok tárolására egy harmadlagos tárolóeszközt kellene használni, és a harmadlagos tárolóról kellene a rekordokat a lemezre vagy lemezekre átmozgatnunk egy többutas, összefésülő rendezéshez hasonló stratégiával, csak most a harmadlagos és másodlagos tároló játssza azt a szerepet, amit előzőleg a másodlagos tároló és a központi memória játszott. □

Ha még több rekordot kell rendeznünk, akkor kiegészítjük egy harmadik menettel. A kétfázisú, többutas, összefésülő rendezéssel M^2/RB rekordcsoportokat rendezünk rendezett részlistákká. Ezután egy harmadik fázisban ezek közül a listák közül legfeljebb $(M/B) - 1$ listát összefésülünk egy többutas összefésüléssel.

A harmadik fázisban nagyjából M^3/RB^2 rekord rendezését teszi lehetővé, melyek M^3/B^2 blokkot foglalnak el. A 2.5. példa paramétereit használva körülbelül 75 trillió rekordot kapunk, melyek összesen 7500 petabájtot foglalnak el. Ez akkora mennyiség, amiről ma még hallani sem lehet. Mivel még a kétfázisú, többutas, összefésülő rendezésre adott 0,61 terabájt limit is valószínűtlenül nagyobb, mint amekkora mennyiséget a másodlagos tárolóval kell kezelnünk, ezért azt mondhatjuk, hogy a többutas, összefésülő rendezésnek a kétfázisú változata valószínűleg minden gyakorlati célnak megfelel.

2.3.6. Feladatok

2.3.1. feladat: A 2.5. példa relációját mennyi idő alatt lehet a kétfázisú, többutas, összefésülő rendezéssel rendezni, ha a *Megatron 747* lemezt lecseréljük a 2.2.1. feladatban leírt *Megatron 777* lemezre, de egyébként a gépnek és az adatoknak minden más jellemzője változatlanul marad?

2.3.2. feladat: Tegyük fel, hogy a kétfázisú, többutas, összefésülő rendezést akarjuk használni a 2.5. példában megadott gépre és R relációra, de a paramétereken változtatunk. Számoljuk ki, hogy a rendezéshez mennyi lemez I/O-műveletre van szükség, ha az R reláció és/vagy a gép jellemzőit a következőkre változtatjuk:

* a) Az R relációban a sorok számát megduplázzuk (minden más változatlan marad).
b) A sorok hosszát megduplázzuk, azaz 200 bájtra változtatjuk (minden más megegyezik a 2.5. példában szereplő értékekkel).
* c) A blokkok méretét duplázzuk meg, azaz 8192 bájtra növeljük (minden más paraméter, mint ebben a feladatban végig, változatlan marad).
d) A hozzáférhető memória méretét duplázzuk meg, azaz 100 megabájtra növeljük.

! **2.3.3. feladat:** Tegyük fel, hogy a 2.5. példa R relációja olyan nagyra nő, hogy már annyi sorral rendelkezik, amennyit maximálisan rendezni lehet a kétfázisú, többutas, összefésülő rendezéssel a példában szereplő gépen. Azt is tegyük fel, hogy a lemez is akkora, hogy az R relációt be tudja fogadni. Mennyi ideig tart az R rendezése akkor, ha a lemez, gép és az R reláció összes többi jellemzőjét változatlanul hagyjuk?

* **2.3.4. feladat:** Tekintsük újra a 2.5. példa R relációját, de most azt tételezzük fel, hogy a rendezési kulcs (ami szokásos értelemben egy kulcs, azaz egyértelműen azonosítja a rekordokat) alapján rendezve tároljuk. Továbbá még azt is tegyük fel, hogy az R relációt olyan blokkok sorozatán tároljuk, melyek elhelyezkedését pontosan ismerjük, azaz bármilyen i esetén az R i. blokkját egyetlen lemez I/O-művelettel lehet elérni. Ha adott egy K kulcsérték, akkor az ezzel a kulccsal rendelkező sort a szabványos bináris kereséssel találhatjuk meg. Maximálisan hány lemez I/O-műveletre van szükség ahhoz, hogy megtaláljuk a K kulccsal rendelkező sort?

!! **2.3.5. feladat:** Tegyük fel, hogy ugyanaz a helyzet, mint a 2.3.4. feladatban, de most 10 előre adott kulcsértéket keresünk. Maximálisan hány lemez I/O-művelet szükséges ahhoz, hogy megtaláljuk mind a 10 sort?

* **2.3.6. feladat:** Tegyük fel, hogy van egy relációnk n sorral és mindegyik sor R bájt hosszú. Adott továbbá egy gép olyan M méretű központi memóriával és B méretű lemezblokkokkal, amely éppen elég ahhoz, hogy az n sort a kétfázisú, többutas, összefésülő rendezéssel rendezni lehessen. Hogyan változik a maximális n, ha a paramétereken a következő változtatásokat tesszük?

a) Megduplázzuk B-t.
b) Megduplázzuk R-et.
c) Megduplázzuk M-et.

! **2.3.7. feladat:** Ismételjük meg a 2.3.6. feladatot, de most olyan paraméterekkel, amelyek mellett még lehetséges a háromfázisú, többutas, összefésülő rendezés.

*! **2.3.8. feladat:** Határozzuk meg, hogy k fázisú (k egy egész szám), többutas, összefésülő rendezés esetén hány rekordot lehet maximálisan rendezni. Az eredményt a 2.3.6. feladatban használt R, M, B és k függvényében adjuk meg.

2.4. A másodlagos tároló hozzáférési idejének javítása

A 2.3.4. rész elemzései során feltettük, hogy az adatokat egyetlen lemezen tároljuk, és a blokkokat véletlenszerűen választjuk a lemez lehetséges helyei közül. Ez a feltevésünk egy olyan rendszer esetében alkalmazható, amely nagyon sok, de kis lekérdezést hajt végre szimultán módon. Ezzel szemben, ha a rendszernek nincs más feladata,

mint hogy egy nagy relációt rendezzen, akkor jelentős időt meg tudunk takarítani, ha jobban megfontoljuk, hogy a rendezésbe bevont blokkokat hová helyezzük. Ezáltal ki tudjuk használni azt is, hogy hogyan működik a lemez. Tulajdonképpen még az előző esetben is, vagyis mikor a rendszeren a legnagyobb terhet sok egymással kapcsolatban nem álló lekérdezés okozza azáltal, hogy „véletlen" blokkokat kell elérni a lemezen, még ekkor is sok mindent tehetünk annak érdekében, hogy a lekérdezések sokkal gyorsabban fussanak, és/vagy a rendszer megengedje, hogy egyszerre több lekérdezést is végre lehessen hajtani (azaz növeljük az *„teljesítményt"*). Ezek közül a stratégiák közül ebben a részben a következőket tekintjük át:

- Helyezzük ugyanarra a cilinderre azokat a blokkokat, amelyeket egyszerre kell elérni. Ezzel gyakran megússzuk a keresési időt és lehet, hogy még a rotációs késést is.
- Ahelyett, hogy egy nagy lemezt használnánk, inkább osszuk szét az adatainkat több kisebb lemezre. Mivel a több fejszerelvény a blokkokat egymástól függetlenül keresheti, ezért az egységnyi időre eső blokkhozzáférések száma ezzel megnövekedhet.
- „Tükrözzük" a lemezt: Készítsünk kettő vagy több másolatot a lemez adatairól. Amellett, hogy lemezhiba esetén mentésünk marad az adatainkról, ez a stratégia arra is jó, hogy egyszerre több blokkhoz tudunk hozzáférni, ahogy az előző pontban, azaz mikor az adatokat több lemezre osztottuk szét.
- Használjunk valamilyen lemezütemező algoritmust. Ez lehet az operációs rendszerben, az adatbázis-kezelő rendszerben vagy a lemezvezérlőben, és ez adja meg, hogy ha több blokkolvasási vagy írási igény érkezik be, akkor ezeket milyen sorrendben kell végrehajtani.
- Hozzuk be előre azokat a blokkokat a központi memóriába, amelyekről előre látható, hogy a későbbiekben használni fogjuk őket.

A vizsgálataink során hangsúlyozni fogjuk, hogy javulás akkor várható, ha a rendszer legalább egy bizonyos ideig egy speciális feladattal foglalkozik, amely lehet például a 2.5. példában bevezetett rendezési művelet. Van még legalább két másik nézőpont is, amelynek segítségével mérhetjük a rendszer működését, illetve a másodlagos memória használatát:

1. Mi történik abban az esetben, mikor nagyon sok folyamatot kell a rendszernek egy időben támogatnia? Például egy repülőgépes helyfoglalási rendszernek egyszerre nagyon sok ügynököt kell kiszolgálnia, akik a járatokról lekérdezéseket tesznek fel, vagy helyet foglalnak le.
2. Mit csináljunk, ha előre megadott fix költségből kell kiépítenünk egy számítógépes rendszert, illetve mit csináljunk, ha mindenféle lekérdezést kell végrehajtanunk egy olyan rendszeren, amely már adott és nem könnyen változtatható meg?

Ezeket a kérdéseket a 2.4.6. részben nézzük majd meg, miután megvizsgáltuk a fenti lehetőségeket.

2.4.1. Az adatok cilinderes szervezése

Mivel a keresési idő általában az átlagos blokkhozzáférési időnek a felét teszi ki, ezért sok alkalmazásban lehet értelme annak, hogy azokat az adatokat, amelyeket egyszerre kell majd elérni, például a relációkat, egy cilinderen tároljuk. Ha nem lenne elég hely, akkor néhány szomszédos cilindert használhatunk fel erre a célra.

Valójában, ha egyszerre beolvassuk a sávon vagy a cilinderen tárolt összes blokkot, akkor lehet, hogy csak az első keresési idő (ami a cilinderre álláshoz szükséges) és az első rotációs idő (ami az első blokknak a fej alá kerüléséhez kell) marad meg, minden más késleltetéstől el lehet tekintenünk. Ebben az esetben az adatok lemezről olvasásánál, írásánál meg tudjuk közelíteni az elméletileg elérhető átviteli gyorsaságot.

2.9. példa: Nézzük meg újra a 2.3.4. részben leírt kétfázisú, többutas, összefésülő rendezés működését. A 2.3. példában meghatároztuk az átlagos blokkátviteli időt, keresési időt és rotációs késést. Ezekre rendre a 0,5 milliszekundum, 6,5 milliszekundum és 7,8 milliszekundum értékeket kaptuk *Megatron 747* lemez esetén. Azt is megállapítottuk, hogy az egy gigabájtot elfoglaló 10 000 000 rekord rendezése 250 percet vesz igénybe. Ezt az időt négy nagy műveletre osztottuk: egy olvasás és egy írás tartozott az algoritmus két fázisának mindegyikéhez.

Nézzük meg, hogy az adatok cilinderes szervezése tud-e javítani ezeken a műveleti időkön. Az első művelet az volt, hogy az eredeti rekordokat beolvastuk a központi memóriába. A 2.7. példa szerint 20 alkalommal töltöttük fel a központi memóriát, méghozzá minden esetben 12 800 blokkal.

Az eredeti adatokat egymás utáni cilindereken is tárolhatjuk. A *Megatron 747* lemez 8192 cilinderének mindegyike körülbelül egy megabájtot tud tárolni; valójában ez az érték csak egy átlag, mivel a belső sávok kevesebbet, a külső sávok többet tudnak tárolni, de az egyszerűség kedvéért feltesszük, hogy minden sáv és minden cilinder kapacitása az átlaggal egyezik meg. Így tehát a kezdeti adatainkat 1000 cilinderen tudjuk tárolni, amiből 50 cilindert olvasunk be a központi memóriába. Emiatt egy cilindert egy keresési idővel olvashatunk be. Még azt sem kell megvárnunk, hogy a cilindernek egy speciális blokkja kerüljön a fej alá, mivel ebben a fázisban a rekordok sorrendje nem játszik szerepet. Összesen 49 alkalommal kell a fejet a szomszédos cilinderre mozgatnunk. A 2.3. példa paraméterei mellett kiszámoltuk, hogy egysávnyi elmozduláshoz csak egy milliszekundum szükséges. Ekkor tehát a központi memória kitöltéséhez szükséges összes idő:

1. Az átlagos kereséshez 6,5 milliszekundum kell.
2. A 49 egycilinderes mozgáshoz 49 milliszekundum szükséges.
3. A 12 800 blokk beolvasásához 6,4 másodperc kell.

Az utolsó érték kivételével a többit el lehet hanyagolni. Mivel 20-szor töltjük ki a memóriát, így az első fázishoz tartozó teljes beolvasás idő körülbelül 2,15 perc. Hasonlítsuk össze ezt az időt azzal az egy órával, amit a 2.7. példában kaptunk az első fázis olvasási részére, igaz, akkor azt tételeztük fel, hogy a blokkok eloszlása a lemezen véletlenszerű. Az első fázis írási részénél a rekordok 20 rendezett részlistájának

tárolására is hasonlóan használhatunk szomszédos cilindereket. Ezeket a rendezett listákat szintén ki lehet írni másik 1000 cilinderre. Ehhez ugyanolyan fejmozgatásokra van szükség, mint az olvasás esetén: egy véletlen keresés és 49 egycilinderes keresés szükséges a 20 lista mindegyikénél. Így az első fázis írási ideje szintén körülbelül 2,15 perc, azaz 4,3 percet kaptunk a teljes első fázisra, szemben a véletlen eloszlású blokkok esetén kiszámolt 125 perccel.

Másrészt a cilinderes tárolás nem segít a rendezés második fázisában. Emlékezzünk vissza arra, hogy a második fázisban a 20 rendezett részlista elejéről olvassuk be a blokkokat, méghozzá olyan sorrendben, amit az adatok határoznak meg, illetve az, hogy melyik lista aktuális blokkja ürül ki legközelebb. Hasonlóan a végleges rendezett listát tartalmazó output blokkokból is időnként egyet ki kell írnunk, így időnként egy blokkírás tarkítja a blokkolvasások sorozatát. Tehát a második fázis változatlanul körülbelül 125 percig tart. Következésképpen a rendezési időt majdnem a felére tudtuk csökkenteni, de csupán a cilinderek átgondolt használatával nem lehet jobb eredmény elérni. □

2.4.2. Több lemez használata

A rendszerünk sebességén gyakran javítani tudunk azzal, hogy ahelyett, hogy egy lemezt használnánk, sok egymáshoz kapcsolódó fejjel, inkább több lemezt használunk független fejekkel. Egy ilyen elrendezést mutatott be a 2.6. ábra, ahol három lemez kapcsolódott egyetlen vezérlőhöz. Ha a lemezvezérlő, az adatsín és a központi memória az átvitt adatokat nagyon gyorsan tudja kezelni, akkor nagyjából az lesz a hatás, mintha a lemezolvasási és -írási sebességeket osztanánk a lemezek számával. Egy példán mutatjuk be az így keletkezett változást.

2.10. példa: A *Megatron 737* lemez összes jellemzője megegyezik a 2.1. és 2.3. példában megadott *Megatron 747* lemezével, de a 737 esetében csak egy tányér van két felülettel. Ily módon egy *Megatron 737* lemez 2 gigabájtot képes tárolni. Tegyük fel, hogy egy *Megatron 747* lemezünket lecseréljük négy darab *Megatron 737* lemezre. Tekintsük át, hogy a kétfázisú, többutas, összefésülő rendezést hogyan lehet végrehajtani.

Először szétosztjuk az adott rekordokat a négy lemez között. Az adatok minden egyes lemezen 1000 szomszédos cilindert fognak elfoglalni. Mikor az első fázis során meg akarjuk tölteni a lemezről a központi memóriát, akkor minden lemezről a központi memória 1/4 részét töltjük ki. A 2.9. példa alapján megint kihasználhatjuk, hogy a keresési idő és a rotációs késés lényegében tart a nullához. Viszont ahhoz, hogy a memória negyedét elég blokkal töltsük ki, be kell olvasnunk 3200 blokkot a lemezről, amit körülbelül 1600 milliszekundum, azaz 1,6 másodperc alatt tehetünk meg. Ha rendszer ilyen sebességgel tudja a négy lemezről egyszerre jövő adatokat is kezelni, akkor a központi memória 50 megabájt méretű részét 1,6 másodperc alatt tudjuk megtölteni, szemben az egy lemez használatánál kiszámolt 6,4 másodperccel.

Hasonlóan, mikor az első fázisban sor kerül arra, hogy az adatokat ki kell írni a központi memóriából, akkor a rendezett részlistákat is szétoszthatjuk a négy lemez között, amihez minden lemezen körülbelül 50 szomszédos cilindert foglalunk le. Tehát az első fázis írási részének sebességét is négyszeresére tudjuk gyorsítani. Így a

teljes első fázis egy percig tart szemben a 2.4.1. részben leírt, csak cilinderes javítással elért 4 perccel, nem beszélve az eredeti, véletlenszerűségi feltevésnél kapott 125 percről.

Most vegyük szemügyre a kétfázisú, többutas, összefésülő rendezés második fázisát. A különböző listák elejéről látszólag véletlenszerű, adatfüggő sorrendben kell még beolvasni a blokkokat. A második fázis algoritmusának magja azt követeli meg, hogy mind a 20 listának megfelelő blokk teljesen be legyen töltve a központi memóriába, ahhoz, hogy a 20 részlista megmaradó elemei közül a legkisebbet kiválaszthassuk. Emiatt nem tudjuk kihasználni, hogy 4 lemezünk van. Minden esetben, mikor kimerül egy blokk, várnunk kell, amíg egy új blokkot teljesen beolvasunk ugyanarról a listáról, hogy ezzel helyettesítsük az előzőt. Így egyszerre mindig csak egy lemezt használunk.

Ha azonban ügyesebben írjuk meg az algoritmus kódját, akkor a 20 legkisebb elem összehasonlítását már abban a pillanatban folytathatjuk, mikor az új blokk első eleme megjelenik a központi memóriában.[10] Ha így teszünk, akkor egyszerre több lista is betöltetheti a blokkjait a központi memóriába. Amikor ezek a blokkok különböző lemezeken helyezkednek el, akkor minden blokkolvasást egy időben tudunk elvégezni, és ezáltal a 2. fázis olvasási részének sebességét egy potenciális 4-szeres faktorral tudjuk növelni. A blokkolvasások véletlen sorrendje azért továbbra is korlátoz bennünket; ugyanis, ha a következő két blokk történetesen ugyanazon a lemezen helyezkedik el, akkor az egyiknek meg kell várni a másikat, és a teljes központi memória áll addig, amíg legalább a második blokk eleje meg nem érkezik a központi memóriába.

A 2. fázis írási részét könnyebb felgyorsítani, ugyanis használhatunk négy output puffert, és sorba mindegyiket kitöltjük. Ha valamelyik puffer megtelik, akkor rögtön kiírjuk az egyik meghatározott lemezre úgy, hogy a cilindereket sorba töltjük fel. Ezáltal a pufferek közül egyet mindig fel lehet tölteni, amíg a többi hármat éppen kiírjuk.

Mindazonáltal nem lehet gyorsabban kiírni a teljes rendezett listát annál, ahogy a 20 köztes listáról az adatokat beolvastuk. A fentiekben láttuk, hogy nem lehet azt elérni, hogy mind a négy lemez egyszerre és folyamatosan hasznos munkát végezzen, ezért a második fázis valószínűleg csak 2-3-szorosára gyorsítható, de még a kétszeres tényező is egy órát takarít meg nekünk. Összegezve: a cilinderes szervezés a 4 lemezes adattárolással együtt a rendezéses példánk idejét mindkét fázisban csökkenteni tudja 125 percről 1 percre az első fázis esetében, és 1 órára, a második fázis esetében. □

2.4.3. Lemezek tükrözése

Előfordulhatnak olyan helyzetek, amikor értelmesnek tűnik, hogy két vagy több lemez ugyanazoknak az adatoknak a másolatait tartalmazza. Ekkor azt mondjuk, hogy ezek a lemezek egymás *tükrözései*. Egyik fontos késztetésünk lehet, hogy ilyen módon bár-

[10] Azt azonban hangsúlyozni kell, hogy ez a megközelítés rendkívül finom implementálást követel meg, és csak akkor kell ezzel megpróbálkozni, ha fontos előny származik belőle. Jelentős ugyanis a kockázata annak, hogy ha nem vagyunk elég előrelátóak, akkor egy rekordot már azelőtt próbálunk elolvasni, hogy az megérkezett volna a központi memóriába.

melyik lemez fejének meghibásodását az adataink túlélik, mivel a meghibásodott lemez egyik tükrözéséről még mindig beolvashatók. Azoknál a rendszereknél, amelyeket úgy terveztek, hogy támogassák a megbízható működést, gyakran lemezpárokat használnak, melyek egymás tükörképei.

Ettől túlmenően a lemezek tükrözése is felgyorsíthatja az adatelérést. Emlékezzünk vissza, hogy a 2.10. példában a kétfázisú, összefésülő rendezés 2. fázisának elemzése során észrevettük, hogyha nagyon előrelátóak vagyunk az időzítésekkel, akkor azt is el tudnánk érni, hogy a négy különböző rendezett listáról négy blokkot töltünk fel egyszerre, ha az előző blokkjuk már kimerült. Azt viszont nem tudjuk előre kiválasztani, hogy melyik négy listának lesz szüksége új blokkra. Ha nagyon szerencsétlenek vagyunk, akkor például azt találjuk, hogy az első két lista ugyanazon a lemezen van, vagy az első három listából kettő van ugyanazon a lemezen.

Ha hajlandóak vagyunk arra, hogy egy nagy lemezről 4 másolatot készítsünk, és ezáltal pazaroljuk a lemezterületet, akkor cserébe a rendszer mindig garantáltan visszanyerhet négy blokkot egy időben. Vagyis, ha mindegy, hogy melyik négy blokkra van szükség, akkor mindegyiket hozzárendeljük a négy lemez valamelyikéhez, és arról a lemezről olvastatjuk be a blokkot.

Általánosítva, ha n másolatot készítünk egy lemezről, akkor tetszőleges n blokkot olvashatunk párhuzamosan. Ha n blokknál kevesebbet kell egyszerre olvasni, akkor gyakran növelni tudjuk a sebességet azáltal, hogy megfontoltan választjuk ki azt a lemezt, amelyről olvasunk. Ugyanis vehetjük a rendelkezésre álló lemezek közül azt, amelyiknek a feje a legközelebb esik ahhoz a cilinderhez, amelyet olvasni akarunk.

A tükrözött lemezek használata az egylemezes használathoz képest nem gyorsítja fel az írást, de szerencsére nem is lassítja le. Ugyanis mikor egy blokkot kell írnunk, akkor igaz, hogy minden olyan lemezre ki kell ezt írni, amelyiken másolat található, de mivel az írás párhuzamosan történik, így az eltelt idő körülbelül ugyanakkora, mintha egy lemezt írtunk volna. Valójában a különböző tükrözött lemezek esetében picit eltérhet az íráshoz szükséges idő, mert nem lehetünk biztosak abban, hogy a forgásuk pontosan szinkronizáltan történik. Így lehet, hogy az egyik lemez feje éppen lekésik egy blokkot, míg a másik lemez feje lehet, hogy éppen most készül elhaladni ugyanazon blokk fölött. Azonban ezek a rotációs késési időkre vonatkozó eltérések átlagosan kiegyenlítődnek, és ha a 2.4.1. rész cilinderalapú stratégiáját használjuk, akkor a rotációs késés teljesen elhanyagolható.

2.4.4. A lemez ütemezése és a lift algoritmus

Bizonyos esetekben a lemezhozzáférést más hatékony módon is lehet gyorsítani, például azzal, hogy a lemezvezérlővel választatjuk ki, hogy több igény közül melyiket hajtsa végre először. Igaz, hogy ezt a lehetőséget nem nagyon tudjuk kihasználni abban az esetben, mikor a rendszernek bizonyos adott sorrendben kell a lemezblokkokat olvasnia vagy írnia, mint például az összefésülő rendezésünk egyes részeiben. Abban az esetben viszont, amikor a rendszer sok kis folyamatot támogat, melyek mindegyike néhány blokkot akar csak elérni, akkor növelni lehet a teljesítményt azzal, hogy kiválasztjuk, hogy mely folyamatigény kapja meg az elsőséget.

Egy egyszerű és hatékony ütemező módszer nagyon sok blokkhozzáférés esetére az úgynevezett *lift algoritmus*. Képzeljük el a lemez fejét, amint pásztázza a lemezt, a legbelső cilindertől a legkülsőig, aztán újból vissza. Úgy is gondolhatunk a lemez fejére, mint egy liftre, amely függőlegesen mozog egy épület aljától a tetejéig, aztán vissza. Amint a fej egy cilinderen halad keresztül, megáll, ha egy vagy több igény vonatkozik olyan blokkokra, melyek ezen a cilinderen találhatók. Minden ilyen blokkot az igény szerint olvasunk vagy írunk. A fejek ezután ugyanabban az irányban haladnak tovább, mint eddig. Egészen addig haladnak, amíg el nem érnek a következő olyan cilinderhez, amelyen olyan blokkok vannak, amiket el akarunk érni. Amikor a fejek olyan helyre érnek, hogy az eddig mozgási irányukban már nincs több keresett blokk előttük, akkor irányt váltanak, és az ellenkező irányba folytatják a keresést.

2.11. példa: Tegyük fel, hogy a *Megatron 747* lemezt akarjuk ütemezni. Emlékezzünk vissza, hogy a lemez átlagos keresési ideje, rotációs késése és átviteli ideje rendre 6,5, 7,8 és 0,5. Ebben a példában minden idő milliszekundumban értendő. Tegyük fel, hogy valamikor olyan blokkokat akarunk elérni, melyek a 1000., 3000. és a 7000. cilinderen találhatók. A fejek az 1000. cilinderen helyezkednek el. Továbbá később még befut három blokkhozzáférési igény, ahogy ezt a 2.12. ábra szemlélteti. Például egy 2000. cilinderen található blokk elérését igényeljük a 20. milliszekundumban.

Feltesszük azt is, hogy minden blokkhozzáférés esetén 0,5 jut az átvitelre és 7,8 az átlagos rotációs késésre, vagyis a blokkhozzáféréshez összes 8,3 plusz annyi milliszekundum kell, amennyi a keresési idő. Ezt a keresési időt a *Megatron 747* lemezre a 2.3. példában megadott szabállyal lehet kiszámolni, azaz a sávok számához hozzáadunk egyet, és osztjuk 500-zal. Nézzük meg, hogy mi történik, ha a lift algoritmussal ütemezzük a végrehajtást. Az 1000. cilinderre vonatkozó első igényhez nem kell keresési idő, mivel már ott vannak a fejek. Tehát az első igény kielégítéséhez szükséges idő 8,3. Mivel a 2000. cilinderre vonatkozó igény ekkor még nem jött be, így a fejeket továbbvisszük a 3000. cilinderre, amely a következő igényelt megállás a magasabb sorszámú sávok irányába. Az 1000.-től a 3000. cilinderig tartó mozgás keresési ideje 5 milliszekundum, így 13,3-kor érkezünk oda, és a blokkelérést 8,3 milliszekundum múlva fejezzük be. Tehát a második eléréssel is végzünk 21,6-kor. Ekkorra befut az igény a 2000. cilinderre, de mi már túlhaladtunk ezen a cilinderen 11,3-kor, és nem is jövünk vissza a következő menetig.

Tehát továbbmegyünk a 7000. cilinderre. A keresési idő 9, a rotációs és átviteli idő 8,3, tehát a harmadik eléréssel 38,9-kor végzünk. Most már megérkezett a 8000. cilinderre vonatkozó igény is, tehát továbbmegyünk ugyanebbe az irányba. A keresési idő most 3 milliszekundum, így az elérést 38,9 + 3 + 8,3 = 50,2-kor fejezzük be. Ekkor már az 5000. cilinderre vonatkozó igényt is megtették, így ez és a 2000. cilinder maradt még hátra. Így visszafele, azaz a lemez belseje felé indulunk, hogy ezt a két igényt is kielégítsük.

Hasonlítsuk össze a lift algoritmus végrehajtását egy sokkal naivabb megközelítéssel, például azzal, hogy mindig az elsőre bejövő igényt szolgáljuk ki (first-come-first-served). Az első három igényt pontosan ugyanúgy hajtjuk végre, mint az előbb, feltéve, hogy az első három igény beérkezési sorrendje 1000, 3000, 7000. Ennél a pontnál

viszont vissza kell menni a 2000. cilinderre, mivel ez volt a negyediknek beérkező igény. Az ehhez az igényhez tartozó keresési idő most 11,0, mivel majdnem a fél lemezt megtesszük, míg a 7000.-ről a 2000. cilinderhez jutunk. A 8000. cilinderre vonatkozó ötödik igény 13 milliszekundum keresési időt jelent, az utolsó, azaz az 5000. cilinderhez tartozó keresési idő pedig 7. A 2.14. ábra összegzi, hogy ez a megközelítés milyen tevékenységekkel járt. A két algoritmus között 14 milliszekundum az eltérés, amely ugyan nem tűnik jelentősnek, de ne feledjük, hogy ebben az egyszerű példában az igények száma kicsi volt, és az algoritmusok a hat igény közül a negyedikig nem is térnek el. □

Igényelt cilinder	*Az igénylés ideje*
1000	0
3000	0
7000	0
2000	20
8000	30
5000	40

2.12. ábra. *Hat blokkhozzáférési igény érkezési sorrendje*

Igényelt cilinder	*A befejezés ideje*
1000	8,3
3000	21,6
7000	38,9
8000	50,2
5000	65,5
2000	80,8

2.13. ábra. *A blokkhozzáférések befejezési időpontjai a lift algoritmus használata esetén*

Igényelt cilinder	*A befejezés ideje*
1000	8,3
3000	21,6
7000	38,9
2000	58,2
8000	79,5
5000	94,8

2.14. ábra. *A blokkhozzáférések befejezési időpontjai az „első érkezés első kiszolgálás" algoritmus használata esetén*

Ha a lemezre váró igények átlagos száma növekszik, akkor a lift algoritmus tovább javítja a teljesítményt. Például, ha a várakozási igények száma megegyezik a cilinderek számával, akkor néhány cilinder kivételével mindegyiket meg kell keresni, és ekkor az átlagos keresési idő megközelíti a minimumot. Ha több lekérdezésünk van, mint amennyi cilinder, akkor tipikusan egynél több igény jut egy cilinderre. Ekkor a

A lift algoritmus tényleges késése

Bár a 2.11. példában azt láttuk, hogy a lemezeléréshez szükséges átlagos idő csökkenthető, de a nyereség nem egyforma minden igényre. Például a 2.13. és 2.14. ábrákat megvizsgálva észrevehetjük, hogy a 2000. cilinderre vonatkozó igényt az első érkezés első kiszolgálás algoritmus 58,2-kor elégíti ki, ezzel szemben a lift algoritmus 80,8-kor. Mivel az igényt 20-kor adták ki, ezért a lemez látszólagos késése az igénylés folyamatára vonatkozóan 38,2-ről 60,8 milliszekundumra változik.

Ha sokkal több lemezelérési igény várakozik, akkor a lift algoritmus alatti fejpásztázások nagyon hosszú ideig fognak tartani. Ha egy igény éppen lekéste a liftet, akkor a látszólagos késés ebben az esetben valójában rendkívül magas lesz. Viszonzásképpen viszont, ha nem használjuk a lift algoritmust vagy egy másik jó ütemező módszert, akkor a teljesítmény csökken, és a lemez nem tudja olyan sebességgel kielégíteni az igényeket, amilyen gyorsan azok generálódnak. A rendszerben végül tetszőleges hosszú késéseket tapasztalnánk, vagy egy másodperc alatt csak kevesebb folyamatot lehetne kiszolgálni.

lemezvezérlő rendezheti az egy cilinderhez tartozó igényeket, és ezzel csökkenteni tudja az átlagos rotációs késést, és ezzel együtt az átlagos keresési időt is. Vigyázzunk arra, hogy ha az igények száma nagyon nagyra nő, akkor bármelyik igény kiszolgálásához szükséges idő különösen nagy lesz. A következő példa mutatja be ezt az esetet.

2.12. példa: Tegyük fel, hogy megint a *Megatron 747* lemezzel dolgozunk, melynek 8192 cilindere van. Képzeljük el, hogy 1000 lemezelérési igény várakozik. Az egyszerűség kedvéért tegyük fel, hogy minden igényelt blokk különböző cilinderen van, 8 cilinderenként. Ha a lemez egyik végéről indulunk és végighaladunk a lemezen, akkor az 1000 igény mindegyikéhez valamivel több, mint 1 milliszekundum keresési idő, 7,8 milliszekundum rotációs késés és 0,5 milliszekundum átviteli idő tartozik. Így minden 9,3 milliszekundumban ki tudunk elégíteni egy igényt, ami körülbelül a 60%-a a véletlen blokkeléréshez tartozó átlagos 14,4 milliszekundumnak. A teljes ezer igény kielégítése így 9,3 másodpercig tart. Emiatt az egy igény kielégítéséhez szükséges átlagos késés ennek a fele, azaz 4,65 másodperc, ami már számottevő késést jelent.

Most tegyük fel, hogy nagyon sok igényt kell kielégíteni, mondjuk 16 384-et, és az egyszerűség kedvéért feltesszük, hogy minden cilinderre pontosan két elérési igény esik. Ebben az esetben minden keresési idő egy milliszekundum, és az átviteli idő igényenként természetesen fél milliszekundum. Mivel minden cilinderen két blokkot kell elérni, ezért a 2 blokk közül a távolabbi 2/3 útnyira helyezkedik a lemezen, mikor a fejek ehhez a sávhoz érkeznek. Ennek a becslésnek a bizonyítása trükkös, ezt fogjuk elmagyarázni a „Várakozás két blokk közül az utolsóra" keretes részben.

Tehát ennek a két blokknak az átlagos késése a 2/3 körülfordulási időnek a fele

Várakozás két blokk közül az utolsóra

Tegyük fel, hogy egy cilinderen véletlenszerűen van két blokkunk. Legyen x_1 és x_2 a két pozíció a teljes kör törtrészeként megadva, azaz mindkét szám 0 és 1 közé esik. A nagyobbik érték várható értékére vagyunk kíváncsiak. Annak a valószínűsége, hogy a nagyobbik szám kisebb, mint egy 0 és 1 közé eső y szám, megegyezik azzal, hogy mindkét szám, egymástól függetlenül kisebb ennél az y-nál, ami y^2-tel egyenlő. Így a nagyobbik számhoz tartozó sűrűségfüggvény az y^2 deriváltja, azaz $2y$, ami egy lineáris függvény. A nagyobbik szám várható értékét úgy kapjuk, hogy a sűrűségfüggvénynek és y-nak a szorzatát integráljuk 0 és 1 között, és $\int_0^1 2y^2 dy = 2/3$. Azaz a távolabbi blokk átlagosan a 2/3 lemezkörbefordulásra helyezkedik el.

lesz, azaz $0{,}5 \times \frac{2}{3} \times 15{,}6 = 5{,}2$ milliszekundum. Ezzel az egy blokk eléréséhez szükséges átlagos időt lecsökkentettük 1 + 0,5 + 5,2 = 6,7 milliszekundumra, ami kevesebb, mint a fele az első érkezés első kiszolgálás ütemezésnél kapott átlagos időnek. Másrészt viszont a 16 384 elérés összesen 102 másodpercig tart, így egy igény átlagos késése 51 másodperc. □

2.4.5. Korai beolvasás és nagy léptékű pufferezés

Az utolsó javaslatunk egy másodlagos memória algoritmus felgyorsítására az úgynevezett *korai beolvasás* (prefetching) vagy másképpen *dupla pufferezés*. Bizonyos alkalmazásokban előre meg lehet mondani, hogy a lemezről milyen sorrendben igényeljük a blokkokat. Ilyen esetben betölthetjük ezeket a blokkokat a központi memória puffereibe, mielőtt szükségünk lenne rájuk. Az ebből származó egyik előny az, hogy csökkenteni tudjuk a blokkeléréshez szükséges átlagos időt úgy, hogy jobb lemeztemezést használunk, például ahogy a lift algoritmus esetében tettük. A 2.12. példában látott blokkhozzáférések felgyorsítását is elérhetjük anélkül, hogy az igények kielégítése során a példában is bemutatott nagy késés lépne fel.

2.13. példa: Ahhoz, hogy egy példát lássunk a dupla pufferezés használatára, nézzük meg újra a kétfázisú, többutas, összefésülő rendezés második fázisát, amit a 2.3.4. részben vázoltunk. Emlékezzünk vissza arra, hogy úgy fésültünk össze 20 rendezett részlistát, hogy minden listáról egy blokkot hoztunk be a központi memóriába. Ha annyi rendezett részlistát kell összefésülni, hogy a listákról behozott blokkok teljesen kitöltik a központi memóriát, akkor nem tudunk semmit sem jobbá tenni. Igen ám, de példánkban bőséges mennyiségű memória maradt meg. Például megtehetjük, hogy

minden listához kétblokkos puffert rendelünk, és az egyik puffert feltöltjük, amíg a másikból az összefésüléshez válogatjuk a rekordokat. Ha kimerül az egyik puffer, akkor késlekedés nélkül átkapcsolunk ugyannak a listának a másik pufferére. □

Ennek ellenére a 2.13. példa sémája még mindig annyi időt vesz igénybe, amennyi ahhoz kell, hogy a rendezett listák összes (250 000) blokkját beolvassuk. A 2.4.1. rész cilinder alapú stratégiáját és a korai beolvasást kombinálhatjuk is:

1. Ha a rendezett listákat teljes egészében egymás utáni cilindereken tároljuk, méghozzá úgy, hogy minden sávon a blokkok a rendezett lista egymás utáni blokkjai.
2. Ha egy adott listáról bizonyos rekordokra van szükségünk, akkor az egész sávot vagy cilindert beolvassuk.

2.14. példa: Ahhoz, hogy megértsük, miért előnyösek a sáv vagy cilinder méretű olvasások, vegyük elő megint a kétfázisú, többutas, összefésülő rendezés második fázisát. A központi memóriában annyi hely van, hogy mind a 20 listához két sáv méretű pufferek tartozhatnak. Emlékezzünk vissza arra, hogy a *Megatron 747* lemezen egy sáv 128 Kbájtot tartalmaz, így a teljes pufferterülethez körülbelül 5 megabájt központi memória szükséges. Egy sáv olvasását tetszőleges szektorától kezdhetjük, így egy sáv olvasásához szükséges idő lényegében az átlagos keresési időnek és a lemez egyszeri körülfordulási idejének az összegével egyezik meg, azaz 6,5 + 15,6 = 22,1 milliszekundum. Mivel az 1000 cilinder, azaz 8000 sáv összes blokkját be kell olvasni ahhoz, hogy mind a 20 rendezett részlistát elolvassuk, így az összes adat elolvasásához szükséges teljes idő körülbelül 2,95 perc.

Még jobban járunk, ha két cilinder méretű puffereket használunk minden rendezett listához, és az egyiket feltöltjük, amíg a másikat használjuk. Mivel a *Megatron 747* lemeznek 8 sávja van minden cilinderen, így összesen 40 egy megabájt méretű puffert fogunk használni. Amennyiben 50 megabájt használható a rendezéshez, akkor van elég hely a központi memóriában ehhez a módszerhez. Cilinder méretű pufferek esetén cilinderenként csak egy keresést kell elvégezni. A keresési idő és az egy cilinderhez tartozó 8 sáv olvasási ideje így 6,5 + 8 × 15,6 = 131,3 milliszekundum. Ahhoz, hogy mind az 1000 cilindert elolvassuk, 1000-szer több időre van szükség, azaz körülbelül 2,19 percre. □

A most bemutatott ötlet nem csak az olvasáshoz, hanem analóg módon az íráshoz is használható. A korai beolvasás szellemében a pufferelt blokkok kiíratását késleltetjük addig, amíg a közeljövőben már nem kell újból felhasználni a blokkot. Ez a stratégia megóv bennünket a késéstől, noha várakozunk addig, amíg egy blokkot ki lehet írnunk.

Még jobb az a stratégia, amely nagy (sáv vagy cilinder méretű) output puffereket használ. Ha az alkalmazás megengedi, hogy ilyen nagy tömbökben írjunk, akkor a keresési idővel és a rotációs késéssel nem kell számolnunk, így a lemezre írás a maximális lemez átviteli sebességgel végezhető el. Például, ha úgy módosítjuk a rendező algoritmusunk második fázisát, hogy két egy megabájtos output puffert használunk,

akkor feltölthetjük rendezett rekordokkal az egyik puffert, és kiírjuk az egyik cilinderre, miközben feltöltjük a másik output puffert a következő rendezett rekordokkal. Így az íráshoz szükséges idő 2,15 perc lenne, hasonlóan a 2.14. példa olvasási idejéhez, és a teljes 2. fázis 4,3 percig tartana, éppen annyi, amint a 2.9. példa javított 1. fázisában. Összességében a cilinderes stratégia, a cilinder méretű pufferezés és a korai beolvasás trükkök kombinációjával a rendezést 8,6 perc alatt is el lehet végezni, szemben a naiv lemezkezelő stratégiával kapott 4 órával.

2.4.6. A stratégiák előnyeinek és hátrányainak összegzése

Az előzőekben öt különböző trükköt láttunk, melyekkel néha javítani lehet egy lemezes rendszer működésén. Ezek a következők:

1. Az adatokat cilinderenként szervezzük.
2. Egy helyett több lemezt használunk.
3. Tükrözzük a lemezeket.
4. Az igényeket a lift algoritmussal ütemezzük.
5. Sáv vagy cilinder méretű tömbökben előre behozzuk az adatokat.

Megnéztük a fentiek hatását két esetben, amelyek a lemezelérési igényeknek két szélsőséges esetét reprezentálják:

a) A legszabályosabb esetben, amit a kétfázisú, többutas, összefésülő rendezés első fázisával szemléltettünk, a blokkokat előre megjósolt sorrendben lehet előre beolvasni vagy kiírni, és egyszerre egy folyamat használja a lemezt.
b) Kisebb folyamatok gyűjteménye esetén, mint amilyenek a repülőjegy-foglalások, vagy a bankszámlák változtatásai, a folyamatokat párhuzamosan lehet végrehajtani, megosztozhatnak ugyanazon a lemezen vagy lemezeken, és előre nem tudunk semmit megjósolni. A kétfázisú, többutas, összefésülő rendezés második fázisa rendelkezik ezek közül bizonyos jellemzőkkel.

Az alábbiakban ezeknek a módszereknek az előnyeit és hátrányait összegezzük a fenti kétféle alkalmazásra és azokra, amelyek ezek közé esnek.

Cilinder alapú szervezés

- Előny: Kiváló az a) típusú alkalmazásokra, ahol a hozzáféréseket előre meg lehet mondani, és csak egy folyamat használja a lemezt.
- Hátrány: Nem segít a b) típusú alkalmazásoknál, ahol a hozzáféréseket nem lehet előre megjósolni.

Több lemez használata

- Előny: Mindkét típusú alkalmazásra növeli az írási/olvasási igények kielégítési sebességét.
- Probléma: Ugyanarra a lemezre vonatkozólag egy időben nem lehet egyszerre több olvasási vagy írási igényt kielégíteni, így a gyorsulási tényező kisebb, mint az a tényező, amennyivel a lemezek számát növeltük.
- Hátrány: Több kis lemez költsége meghaladja azt a költséget, amennyibe egy ugyanolyan összkapacitású lemez kerül.

Tükrözés

- Előny: Mindkét típusú alkalmazásra növeli az írási/olvasási igények kielégítési sebességét. A több lemez használatánál említett összeütköző elérések problémája nem fordul elő.
- Előny: Minden alkalmazásra növeli a hibatűrést.
- Hátrány: Két vagy több lemez árát kell megfizetni, de csak egynek megfelelő tárolási kapacitást kapunk érte.

Lift algoritmus

- Előny: Csökkenti a blokkok átlagos írási/olvasási idejét abban az esetben, mikor a blokkelérésekről előre nem tudunk semmit mondani.
- Probléma: Az algoritmus akkor a leghatékonyabb, mikor sok lemezelérési igény várakozik, vagyis mikor az igénylési folyamat átlagos késése magas.

Korai beolvasás/Dupla pufferezés

- Előny: Felgyorsítja az elérést, mikor a szükséges blokkokat ismerjük, de az igények időzítése adatfüggő, mint a többutas, összefésülő rendezés 2. fázisában.
- Hátrány: Újabb puffereket igényel a központi memóriában. Nem segít abban az esetben, mikor az elérések véletlenszerűek.

2.4.7. Feladatok

2.4.1. feladat: Tegyük fel, hogy egy *Megatron 747* lemez I/O-igényeit akarjuk ütemezni. Az igények a 2.15. ábrán láthatók. Kezdetben a fej a 4000. sávon tartózkodik. Mikorra fogjuk mindegyik igényt teljesen kielégíteni, ha:

Igényelt cilinder	*Az igény érkezési ideje*
1000	0
6000	1
500	10
5000	20

2.15. ábra. *Négy blokkelérési igény beérkezési ideje*

a) a lift algoritmust használjuk (kezdetben bármelyik irányba megengedett az indulás),
b) az „első érkezés, első kiszolgálás" algoritmust használjuk.

*! **2.4.2. feladat:** Tegyük fel, hogy két *Megatron 747* lemezt használunk, melyek egymásnak tükörképei. Most ne engedjük meg a két lemez bármelyik blokkjának olvasását. Tegyük fel, hogy az első lemez fejét mindig a lemez belső felén található cilindereken tartjuk, míg a második lemez fejét a külső fél cilindereire korlátozzuk. Tegyük még azt is fel, hogy az olvasási igények véletlenszerűen választott sávokra vonatkoznak, és soha semmit sem kell írnunk.

a) Milyen átlagos sebességgel tudja ez a rendszer olvasni a blokkokat?
b) Hogy viszonyul ez a sebesség a fenti megszorítás nélküli, tükrözött *Megatron 747* lemezekre vonatkozó átlagos sebességhez?
c) Milyen hátránya látszik előre ennek a rendszernek?

! **2.4.3. feladat:** Vizsgáljuk meg a kapcsolatot az igények átlagos beérkezési sebessége, a lift algoritmus teljesítménye és az igények átlagos késése között. A probléma egyszerűsítése érdekében tegyük fel a következőket:

1. A lift algoritmusban egy menet mindig a legbelső sávtól a legkülső sávig tart, illetve fordítva, még akkor is, ha a legszélső cilinderekre nincs is igény.
2. Amikor egy menet elkezdődik, akkor csak azokat az igényeket kell figyelembe venni, amelyek a kezdéskor éppen várakoztak, és semmilyen olyan igénnyel nem kell foglalkozni, amely a menet indulása után érkezett be, még akkor sem, ha a fej éppen egy ilyen cilinderhez érkezik.[11]
3. Egy menet alatt egy cilinderre legfeljebb egy blokkigény vonatkozhat.

Jelölje A a beérkezési sebességet, vagyis azt az időt, amennyi két blokkigény beérkezése között telik el. Tegyük fel, hogy a rendszer stabil, kiegyensúlyozott állapotban van, azaz már hosszú idő óta fogadja és megválaszolja az igényeket. Az A függvényében számoljuk ki egy *Megatron 747* lemezre az alábbiakat:

[11] Ennek a feltevésnek az a célja, hogy ne kelljen foglalkozni azzal a ténnyel, hogy a lift algoritmusban egy tipikus menet először gyorsan halad, mivel kevés várakozó igény vonatkozik arra a helyre, ahol a fej éppen tartózkodott, és felgyorsul, amikor olyan helyre érkezik a lemezen, ahol mostanáig még nem járt. Önmagában érdekes feladat annak az elemzése, hogy egy menet alatt hogyan változik az igénysűrűség.

* a) Egy menet végrehajtása átlagosan mennyi ideig tart?
b) Egy menet alatt mennyi igényt fogunk kielégíteni?
c) Egy igénynek mennyit kell átlagosan várnia a kiszolgálásra?

*!! **2.4.4. feladat:** A 2.10. példában láttuk, hogy ha a rendezni kívánt adatokat szétosztjuk négy lemez között, akkor egy időben egynél több blokkot is lehet olvasni. Tegyük fel, hogy az összefésülő fázis során olyan blokkokat kell olvasni, melyek véletlenszerűen választott lemezeken helyezkednek el. Azt is tételezzük még fel, hogy minden olvasási igényt csak akkor veszünk figyelembe, ha nem olyan lemezre vonatkozik, amely éppen egy másik igényt szolgál ki. Határozzuk meg, hogy az igények kiszolgálását egyszerre átlagosan hány lemez végzi. Megjegyzés: a következő két észrevétel egyszerűsíti a problémát:

1. Amint egy blokkolvasási igényt nem lehet végrehajtani, akkor az összefésülésnek meg kell állnia, és nem generál több igényt, mivel a kimerült listában már nincs olyan adat, amely a kielégítetlen olvasási igényt generálta.
2. Amint az összefésülést folytatni lehet, akkor egy olvasási igényt fog generálni, mivel a központi memóriában az összefésülés elhanyagolható ideig tart összevetve az olvasási igény kielégítéséhez szükséges idővel.

! **2.4.5. feladat:** Ha egy cilinderről k darab véletlenül választott blokkot akarunk beolvasni, akkor átlagosan mekkora körülfordulást kell tennünk a cilinderen ahhoz, hogy túljussunk mind a hat blokkon?

2.5. Lemezhibák

Ebben és a következő fejezetben megnézzük, hogyan hibázhatnak a lemezek, és miképpen lehet mérsékelni az ilyen meghibásodásokat.

1. A legszokásosabb formája a meghibásodásnak az *ideiglenes meghibásodás*. Ez azt jelenti, hogy amikor megpróbálunk írni vagy olvasni egy szektort, először nem sikerül, de ismételt próbálkozásokkal végül sikerül írni vagy olvasni.
2. Súlyosabb formájú az a meghibásodás, mikor egy vagy több bit végérvényesen elromlik, és emiatt lehetetlenné válik egy szektor olvasása, mindegy hányszor próbáljuk újra. Ezt a meghibásodási formát *eszközhibának* hívjuk.
3. Ezzel rokon hibatípus az *írási hiba,* ami azt jelenti, hogy megpróbálunk írni egy szektort, de az írás nem sikerül, és a szektor korábban írt tartalmát sem lehet már visszanyerni. Ennek egy lehetséges oka, hogy áramkimaradás történt, mialatt a szektort írtuk.
4. A legkomolyabb formája a lemez meghibásodásának a *lemezhiba,* amikor hirtelen a teljes lemez végérvényesen olvashatatlanná válik.

Ebben a fejezetben a lemezmeghibásodásnak egy egyszerű modelljét vesszük figyelembe. Áttekintjük a paritás-ellenőrzést, ami egy lehetséges módszer arra, hogy felderítsük az ideiglenes meghibásodásokat. Megvizsgáljuk még a „stabil tárolást" is. Ez egy olyan lemezszervezési technika, amely arra jó, hogy eszközhiba vagy hibás írások sem eredményeznek végleges adatvesztést. A 2.6. részben megnézzük azokat a technikákat, melyek „RAID" gyűjtőnéven ismeretesek. Ezek segítenek abban, hogy megbirkózzunk a lemezhibákkal.

2.5.1. Ideiglenes meghibásodás

A lemezszektorokat általában redundáns bitekkel együtt szokták tárolni. Erről a 2.5.2. részben lesz majd részletesebben szó. Ezeknek a biteknek az a célja, hogy segítségükkel meg tudjuk mondani, hogy amit beolvasunk egy szektorból jó-e vagy hibás, vagy ha kiírtunk egy szektort, akkor az írás helyesen történt-e meg.

Egy jól használható modell a lemezolvasásokra a következő: Az olvasási függvény egy (w, s) párt ad vissza, ahol w a szektorból beolvasott adat, és s egy olyan *státusbit*, amely azt mondja meg, hogy az olvasás sikeres volt-e vagy nem, vagyis megbízhatunk-e abban, hogy w a szektor igazi tartalma. Egy ideiglenes meghibásodás esetén többször is a „rossz" státust kaphatjuk, de ha elégszer (tipikusan legfeljebb 100-szor) ismételjük az olvasási függvényt, akkor végül meg fogjuk kapni a „jó" státust, és biztosak lehetünk abban, hogy azok az adatok, amiket ezzel a státusszal kaptunk vissza, a lemezszektor valódi tartalmával egyeznek meg. A 2.5.2. részben látni fogjuk, hogy előfordulhat, hogy mégis be leszünk csapva, azaz a státus „jó", de a visszakapott adatok valójában rosszak. Azonban az ilyen eset előfordulásának valószínűségét tetszőlegesen kicsivé tehetjük, ha még több redundanciát adunk a szektorokhoz.

A szektorok írásánál is előnyünkre szolgálhat, ha megnézzük annak a státusát, amit írunk. Ahogy a 2.2.5. részben említettük, megtehetjük, hogy minden szektort a kiírás után megpróbálunk beolvasni, hogy meghatározzuk, vajon sikeres volt-e az írás. Egy nyilvánvaló módszer az ellenőrzésre, hogy beolvassuk a szektort, és összehasonlítjuk azzal a szektorral, amit ki akartunk írni. Azonban ahelyett, hogy a teljes összehasonlítást a lemezvezérlővel végeztetnénk el, egyszerűbb, ha megpróbáljuk beolvasni a szektort, és megnézzük, hogy a státusa „jó". Ha igen, akkor feltesszük, hogy helyes volt az írás, ha a státus „rossz", akkor az írás láthatólag sikertelen volt, és meg kell ismételni. Vegyük észre, hogy az olvasáshoz hasonlóan itt is előfordulhat, hogy be leszünk csapva, azaz a státus „jó", de az írás valójában sikertelen volt. Az olvasásnál említett lehetőség most is rendelkezésre áll, azaz, ha akarjuk, akkor az ilyen tévesztés valószínűségét tetszőlegesen kicsivé tehetjük.

2.5.2. Ellenőrző összegek

Első pillanatra rejtélyesnek tűnhet, hogyan tudja meghatározni az olvasási művelet egy szektor jó/rossz státusát. Ennek ellenére a modern lemezmeghajtókban használt technikák teljesen egyszerűek: minden szektornak vannak még további bitjei, ezeket

hívjuk *ellenőrző összegeknek*, és ezeknek a beállított értéke az ebben a szektorban tárolt adatok értékeitől függ. Ha olvasáskor azt találjuk, hogy az ellenőrző összeg nem helyes az adatbitekre, akkor „rossz" státust adunk vissza, különben pedig „jó"-t. Annak is van egy kis valószínűsége, hogy az adatbiteket rosszul olvastuk be, de a téves bitekkel is ugyanazt az ellenőrző összeget kapjuk, mint a helyes bitekkel (és emiatt a rossz bitek is „jó" státust fognak kapni), de elég sok ellenőrző bit használatával ezt a valószínűséget tetszőlegesen kicsivé tehetjük.

Az ellenőrző összegek egyik egyszerű formája a szektor összes bitjének *paritásán* alapul. Ha egy bitekből álló halmazban páratlan sok egyes található, akkor azt mondjuk, hogy a bitek paritása *páratlan*, vagyis a bitekhez tartozó paritás bit értéke 1. Hasonlóan, ha egy bitekből álló halmazban páros sok egyes található, akkor azt mondjuk, hogy a bitek paritása *páros*, és a bitekhez tartozó paritás bit értéke 0. Ebből következik, hogy:

- Ha egy bitekből álló halmazhoz hozzávesszük a nekik megfelelő paritás bitet, akkor az így kapott számok között mindig páros sok egyes szerepel.

Amikor egy szektort írunk, akkor a lemezvezérlő ki tudja számolni a paritás bitet, és hozzáteszi ahhoz a bitsorozathoz, amit a szektorba írunk. Emiatt minden szektornak páros a paritása.

2.15. példa: Ha egy szektorban 01101000 volt a bitek sorozata, akkor páratlan sok 1 szerepel, ezért a paritás bit értéke 1. Ha a sorozat végére tesszük a paritás bitet, akkor 011010001 sorozatot kapjuk. Ha a sorozatunk az 11101110 volt, akkor páros sok 1 szerepel, így a paritás bit értéke 0. Ha a sorozat végére tesszük a paritás bitet, akkor az 111011100 sorozatot kapjuk. Vegyük észre, hogy a paritás bit hozzáadásával keletkezett kilenc bit hosszú sorozat mindegyikének páros a paritása. □

Ha a paritás bittel kiegészített bitsorozat olvasása vagy írása során pontosan egy bit hiba keletkezik, akkor páratlan lenne a paritás, vagyis páratlan sok 1 szerepelne. A lemezvezérlő könnyen össze tudja számolni az egyesek számát, és meg tudja határozni a hibát azáltal, ha a szektor paritására páratlant kap.

Természetesen a szektorban egynél több bit is elromolhat. Ha ez történt, akkor 50% a valószínűsége annak, hogy az 1 értékű bitek száma páros, és ekkor nem fogjuk a hibát észrevenni. Ha több paritás bitet használunk, akkor nagyobb lesz az esélyünk arra, hogy sok hibát fel tudunk ismerni. Például használhatunk 8 paritás bitet, egyet minden bájt első bitjeire, egyet minden bájt második bitjeire és így tovább, egészen a nyolcadikig, amely a bájtok utolsó bitjeire vonatkozik. Nagy tömegű hiba esetén 50% annak a valószínűsége bármelyik paritás bit esetére, hogy hibát jelez, és annak az esélye, hogy a nyolcból egyik sem jelez hibát, csak egy a 2^8-hoz, azaz 1/256. Általában, ha n független bitet használunk ellenőrző összegként, akkor annak az esélye, hogy nem veszünk észre egy hibát, mindössze $1/2^n$. Például, ha 4 bájtot szánunk egy ellenőrző összegre, akkor annak az esélye, hogy nem fogunk felismerni egy hibát, csak 1 a 4 milliárdhoz.

2.5.3. Stabil tárolás

Igaz, hogy az ellenőrző összegek majdnem biztosan felismerik, ha az eszköz elromlott, vagy az olvasás vagy írás nem sikerült hibátlanul, de nem segítenek kijavítani a hibát. Továbbá, írás esetén abba a helyzetbe kerülhetünk, hogy már átírtuk egy szektor korábbi tartalmát, de mégsem tudjuk elolvasni az új tartalmat. Ez a helyzet nagyon komoly is lehet. Képzeljük el például, hogy a folyószámlához egy kis összeget akartunk adni, és most elvesztettük a folyószámla eredeti egyenlegét és az újat is. Ha biztosak lehetnénk abban, hogy a szektor tartalma vagy az új vagy a régi egyenleg, akkor csak azt kellene meghatározni, hogy az írás sikerült-e vagy sem.

Ahhoz, hogy ezt a problémát kezelni tudjuk egy vagy több lemezen, egy olyan elv megvalósítását használjuk, amit *stabil tárolásnak* hívnak. Az alapötlet, hogy a szektorokból párokat képezünk, és minden pár egy X szektortartalmat reprezentál. Az X-et reprezentáló szektorhoz tartozó pár tagjaira bal (X_L) és jobb (X_R) másolatként hivatkozunk. Feltesszük, hogy a másolatok írásakor elég sok paritás-ellenőrző bitet használtunk, így kizárhatjuk annak az esélyét, hogy egy rossz szektort jónak látunk a paritás-ellenőrzéskor. Tehát feltesszük, hogy ha az olvasási függvény a (w, jó) értéket adja vissza az X_L vagy az X_R esetében, akkor a w az X valódi értéke. A stabil tárolás írási elve a következő:

1. Írjuk az X értékét X_L-be. Ellenőrizzük, hogy az érték státusa „jó"; vagyis a paritás-ellenőrző bitek helyesek a kiírt másolatban. Ha nem, akkor ismételjük meg az írást. Ha egy meghatározott számú próbálkozás után sem sikerül X értékét X_L-be írni, akkor azt tesszük fel, hogy ebben a szektorban megsérült az eszköz. Ekkor valamilyen javítási elvet kell alkalmazni, például egy másik szabad szektorral helyettesítjük X_L-t.
2. Ismételjük meg 1.-t az X_R-re.

A stabil tárolás olvasási elve a következő:

1. Ahhoz, hogy X értékét visszanyerjük, olvassuk be X_L-t. Ha a „rossz" státust kapjuk vissza, akkor ismételjük meg az olvasást valamilyen előre megadott számszor. Ha végül kapunk egy értéket, amelynek „jó" a státusa, akkor ezt az értéket vesszük X-nek.
2. Ha nem tudjuk X_L-t elolvasni, akkor 1.-t ismételjük X_R-rel.

2.5.4. A stabil tárolás hibakezelő képessége

A 2.5.3. részben leírt elv több különféle hibatípus ellen használható, melyeket az alábbiakban sorolunk fel.

1. *Eszközhibák.* Tegyük fel, hogy az X-et már letároltuk az X_L és X_R szektorokban. Ekkor, ha valamelyik a kettő közül eszközhiba miatt állandó jelleggel olvashatatlanná válik, akkor az X-et a másikból tudjuk kiolvasni. Ha az X_R hibás, de az X_L nem, akkor az olvasási elv alapján, az X_R-re ügyet sem vetve az X_L-t helyesen fogjuk elolvasni. Akkor fogjuk észrevenni, hogy az X_R hibás, mikor legközelebb megpróbálunk új X értéket írni. Ha csak az X_L sérült meg, akkor nem sikerül „jó" státust kapni az X-re, bárhogy is próbáljuk az X_L-t beolvasni. (Emlékezzünk vissza arra, hogy a feltételezésünk szerint egy rossz szektor mindig „rossz" státust ad vissza, még ha a valóságban van is egy csöpp esély arra, hogy „jó" lesz a státus, ha véletlenül az összes paritás-ellenőrző bit megfelelő értékű.) Tehát végrehajtjuk az olvasási algoritmus 2. lépését és helyesen beolvassuk X-et az X_R-ből. Vegyük észre, hogy ha az X_L és X_R mindegyike hibás, akkor az X értékét nem lehet elolvasni, de az egyidejű két meghibásodás valószínűsége rendkívül kicsi.
2. *Íráshiba.* Tegyük fel, hogy az X írása közben rendszerhiba, például áramkimaradás történik. Lehet, hogy el fog veszni az X a központi memóriából, és ráadásul az X-nek az a másolata, amelyet éppen kiírunk, összezavarodik. Például az X új értékének egy részével már megírtuk a szektor felét, de a szektor másik fele még változatlan. Mikor a rendszer újra működőképessé válik, megvizsgáljuk az X_L-t és X_R-t, és biztonsággal meg tudjuk határozni az X régi vagy új értékét. A lehetséges esetek a következők:

   a) Akkor történt a hiba, mikor az X_L-t írtuk. Ekkor azt fogjuk kapni, hogy az X_L státusa „rossz", viszont mivel az X_R-t nem kellett írni, ezért az X_R-nek „jó" a státusa (kivéve, ha éppen egy X_R-re vonatkozó eszközhiba is történt, aminek olyan kicsi az esélye, hogy nyugodtan elvethetjük). Így megkaphatjuk az X régi értékét. Az X_L sérülésének kijavításához megtehetjük, hogy X_R-t X_L-be másoljuk.
   b) A hiba az X_L írása után történt. Ekkor várhatóan az X_L-nek „jó" lesz a státusa, és így kiolvashatjuk az X_L új értékét az X_L-ből. Megjegyezzük, hogy az X_R-nek lehet, hogy „rossz" a státusa, és ekkor az X_L-t X_R-be kell másolni.

2.5.5. Feladatok

2.5.1. feladat: Számoljuk ki a következő bitsorozatok paritás bitjét:

* a) 00111011.
b) 00000000.
c) 10101101.

2.5.2. feladat: Egy bináris sorozat végére kétféle paritás bitet teszünk, az első a páratlan pozíciókhoz tartozó paritás bit, a második pedig a páros pozíciókhoz tartozó. A 2.5.1. feladatban szereplő összes sorozatra határozzuk meg ezeket a paritás biteket.

2.6. Lemezhiba helyreállítása

Ebben a fejezetben a lemezhibák közül a legsúlyosabbat fogjuk tárgyalni, az úgynevezett „fejsérülést" (head crash), amikor az adatok végérvényesen tönkrementek. Egy ilyen esemény bekövetkezésekor az adatokból semmit sem lehet helyreállítani, csak abban az esetben, ha az adatokat egy másik eszközre, például szalagos archiváló eszközre vagy a 2.4.3. részben vizsgált tükrözött lemezre előtte lementettük. Ha nem lenne mentésünk, akkor egy ilyen helyzet katasztrofális lenne az adatbázis-kezelő rendszerre épülő legfőbb alkalmazásokra (banki, pénzügyi alkalmazásokra, repülőjegy- vagy másmilyen helyfoglalási rendszerekre, raktárnyilvántartó rendszerekre és egyebekre) nézve.

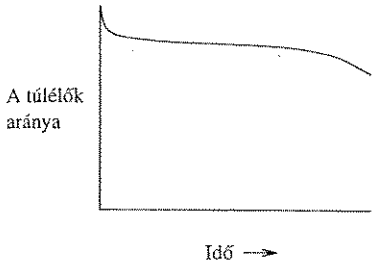
Sok különböző sémát fejlesztettek ki, hogy csökkentsék a lemezhibából származó adatveszteséget. Ezek általában redundanciát okoznak, például a 2.5.2. részben tárgyalt paritás-ellenőrzés ötletének kiterjesztésével, vagy a 2.5.3. részben leírtakhoz hasonló megduplázott szektorok használatával. Ezeknek a stratégiáknak az osztályát közös kifejezéssel RAID[12]-nek, *független lemezek redundáns tömbjének* (Redundant Arrays of Independent Disks) hívjuk. Elsődlegesen három sémát fogunk megnézni közelebbről, melyeket 4, 5 és 6 szintű RAID-nek hívunk. Ezek a RAID-sémák arra is jók, hogy kezeljék a 2.5. részben tárgyalt meghibásodásokat, azaz az eszközhibát és egy szektor adatainak ideiglenes rendszerhibából származó sérüléseit.

2.6.1. A lemezek meghibásodási modelljei

Ahhoz, hogy elkezdjük a lemezsérülések tárgyalását, először az ilyen hibák statisztikáit kell megnéznünk. A legegyszerűbben úgy írhatjuk le az ilyen hibák viselkedését, hogy megadunk egy mértéket, a *meghibásodás várható idejét*. Ez a szám annak az időnek a hosszát jelenti, amennyi idő elteltével a lemezek egy előre adott sokaságának az 50%-a katasztrofálisan tönkremegy, vagyis olyan lemezhiba történik, hogy soha többé nem lesz már olvasható a lemez. A modern lemezek esetében a várható meghibásodási idő körülbelül 10 év.

A legegyszerűbben úgy használhatjuk fel ezt a számot, hogy feltesszük, hogy a meghibásodások lineárisan történnek, vagyis ha 50% 10 év alatt megy tönkre, akkor 5% megy tönkre az első, a második stb. évben 20 éven keresztül. A valóságban a lemezek túlélési százaléka inkább a 2.16. ábrán megadott grafikont követi. Az elektromos berendezések legtöbb típusára igaz, hogy az életciklus korai szakaszában sok lemezhiba jelentkezik, amelyek főleg a lemez gyártása közben keletkezett apró hibákból származnak. Ezeknek a gyártási hibáknak a legtöbbjét remélhetőleg észreveszik, mielőtt a lemez elhagyja a gyárat, de néhánynál még hónapok elteltével sem jelentkezik. Egy olyan lemez, amelynél nem jelentkezik hamar semmilyen hiba, valószínűleg sok évig fog jól működni. Az életciklus későbbi szakaszában több tényező (a használatból eredő kopás, a parányi porszemcsék összegzett hatása) növeli a meghibásodás esélyét.

[12] Korábban a RAID betűszóban az I betű az olcsó (Inexpensive) szónak a kezdőbetűjét jelentette, és ezzel az értelmezéssel még mindig találkozhatunk a szakirodalomban.



2.16. ábra. *Lemezekre vonatkozó meghibásodási ráta görbéje*

Valójában egy lemez meghibásodásának várható ideje nem kell, hogy megegyezzen az adatvesztés várható idejével. Ennek az az oka, hogy sok olyan séma áll a rendelkezésünkre, melyek biztosítják, hogyha egy lemez meghibásodik, akkor más lemezek segítenek helyreállítani a sérült lemez adatait. Ebben a fejezetben át fogjuk tekinteni a legáltalánosabb sémákat.

Ezeknek a sémáknak mindegyike azzal kezdődik, hogy egy vagy több lemez adatokat tartalmaz (ezeket hívjuk majd *adatlemezeknek*), és ezekhez hozzáveszünk egy vagy több olyan lemezt, amelyeken a tárolt információt teljesen az adatlemezek tartalma határozza meg. Ez utóbbiakat nevezzük *redundáns lemezeknek*. Amikor egy adatlemez vagy egy redundáns lemez megsérül, akkor a többi lemez használható a sérült lemez visszaállítására, és emiatt nem lesz maradandó információvesztés.

2.6.2. A tükrözés mint redundanciatechnika

A legegyszerűbb séma, ha minden lemezt úgy tükrözünk, ahogy a 2.4.3. részben ezt leírtuk. Az egyik lemezt *adatlemeznek*, a másik lemezt *redundáns lemeznek* hívjuk, hogy melyik melyik, nem játszik szerepet ebben a sémában. A tükrözést mint az adatvesztés elleni egyik lehetséges védekezést, gyakran *1. szintű RAID-nek* hívják. Ezáltal a memóriavesztés várható ideje sokkal nagyobb, mint a lemezhiba várható ideje, ahogy ezt a következő példa szemlélteti. Lényegében a tükrözés és más redundanciasémák esetén adatvesztés csak akkor történhet, ha a második lemez is tönkremegy, mialatt az első sérülését javítják.

2.16. példa: Tegyük fel, hogy mindegyik lemeznek 10 év a várható meghibásodási ideje. A meghibásodásokra a 2.6.1. részben leírt lineáris modellt használjuk, amely azt jelenti, hogy annak az esélye, hogy egy lemez tönkremegy, 5% évente. Ha a lemezeket tükrözzük, akkor egy lemezsérülés esetén csak ki kell cserélnünk egy jó lemezzel, és a (tükörképet tartalmazó) tükörlemezt az új lemezre kell másolni. Végül újra két lemezünk lesz, melyek tükörképei egymásnak, és ezzel a rendszer visszaállt az előző állapotára.

Az egyetlen baj, ami történhet, hogy a másolása közben a tükörlemez is tönkremegy. Mivel most legalább az adatok egy részének mindkét másolata elveszett, ezért nincs mód a helyreállításra.

De vajon milyen gyakran fordul elő az eseményeknek ez a láncolata? Tegyük fel, hogy a megsérült lemez cserélésének folyamata 3 órát vesz igénybe, amely 1/8 nap, azaz 1/2920 része az évnek. Mivel 5% éves meghibásodási rátát tételeztünk fel, ezért annak a valószínűsége, hogy a tükörlemez megsérül a másolás alatt (1/20) × (1/2920), azaz 1 az 58 400-hoz. Ha egy lemez 10 évenként megy tönkre, akkor két lemezből egy átlagosan 5 évente hibásodik meg. Minden 58 400 ilyen hiba közül egy fog adatvesztést eredményezni, azaz másképpen fogalmazva az adatvesztést eredményező meghibásodás várható ideje 5 × 58 400 = 292 000 év. □

2.6.3. Paritásblokkok

Bár a lemezek tükrözése hatékonyan képes csökkenteni az adatvesztéssel járó lemezhiba valószínűségét, de ehhez annyi redundáns lemezre van szükség, mint amennyi az adatlemezek száma. Egy másik lehetséges megközelítés, hogy az adatlemezek számától függetlenül csak egyetlenegy redundáns lemezt használunk. Ezt gyakran *4. szintű RAID-nek* nevezik. Feltesszük, hogy a lemezek egyformák, azaz minden lemezen 1-től *n*-ig számozhatjuk meg a blokkokat. Természetesen minden lemez minden blokkjában ugyanannyi bit van, például az alappéldánkban szereplő *Megatron 747* lemezünkön minden blokk 4096 bájt méretű, azaz bármelyik blokkban 8 × 4096 = = 32 768 bit van. A redundáns lemezen az *i*. blokk paritás-ellenőrzéseket tartalmaz az összes adatlemez *i*. blokkjaira nézve. Vagyis az adatlemezek és a redundáns lemez *i*. blokkjainak *j*. bitjei között páros sok 1-nek kell szerepelnie, és a redundáns lemez bitjeit mindig úgy választjuk meg, hogy ez a feltétel igaz legyen.

A 2.15. példában láttuk, hogy lehet elérni, hogy igazzá váljon ez a feltétel. A redundáns lemezen a *j*. bitet 1-nek választjuk, ha páratlan sok adatlemezen szerepel 1 ezen a biten. A redundáns lemezen a *j*. bitet 0-nak választjuk, ha páros sok adatlemezen szerepel 1 ezen a biten. Ennek a számításnak a neve *modulo-2 összegzés*. Tehát adott bitek modulo-2 összege 0, ha páros sok 1 szerepel a bitek között, és 1, ha páratlan sok 1 szerepel.

2.17. példa: Vegyük a lehető legegyszerűbb esetet, vagyis mikor a blokkok csak egyetlenegy bájtot, azaz 8 bitet tartalmaznak. Legyen három adatlemezünk, nevezzük őket 1., 2. és 3. lemeznek, és a 4. lemez legyen a redundáns lemez. Vegyük szemügyre, mondjuk, az összes lemez első blokkját. Ha az adatlemezek első blokkjaiban a következő bitsorozatok találhatók:

1. lemez: 11110000
2. lemez: 10101010
3. lemez: 00111000

akkor a redundáns lemez 1. blokkjában a paritás-ellenőrző bitek a következők:

4. lemez: 01100010

Vegyük észre, hogy a négy darab 8 bit hosszú sorozat közül minden pozícióban páros számúnál látunk 1 értéket. Két 1 érték szerepel az 1., 2., 4., 5. és 7. helyen, négy 1 van a 3. helyen, és zéró darab 1-et találunk a 6. és 8. helyen. □

Olvasás

Egy adatlemezről ugyanúgy olvasunk be blokkokat, mint ahogy bármilyen más lemezről. Általában semmi okunk, hogy a redundáns lemezt olvassuk, de ha akarjuk, megtehetjük. Bizonyos körülmények között a redundáns lemezt arra is használhatjuk, hogy a segítségével az adatlemezek egyikének két különböző blokkját tudjuk egy kis trükkel egy időben beolvasni, bár várhatóan ritkán teljesülnek az ehhez szükséges feltételek.

2.18. példa: Tegyük fel, hogy az első adatlemezen éppen olvasunk egy blokkot, mikor ugyanennek a lemeznek egy másik blokkjára, mondjuk az elsőre beérkezik egy másik olvasási igény. Közönséges esetben meg kellene várnunk, hogy az első igény befejeződik. Viszont, ha éppen ekkor a többi lemez egyikét sem használjuk, akkor ezalatt beolvashatjuk ezekről az első blokkot, és a modulo-2 összeadás segítségével kiszámolhatjuk az első lemez első blokkját.

Speciálisan legyenek a lemezek és az első blokkjaik ugyanazok, mint a 2.17. példában, azaz a 2., 3. és a redundáns lemezt beolvasva a következő blokkokat kapjuk:

2. lemez: 10101010
3. lemez: 00111000
4. lemez: 01100010

Ha most minden oszlopban vesszük a bitek modulo-2 összegét, akkor az alábbit kapjuk:

1. lemez: 11110000

ami pontosan megegyezik az első lemez első blokkjával. □

Írás

Amikor egy adatlemezen egy új blokkot akarunk írni, akkor nem csak ezt a blokkot kell megváltoztatni, hanem a redundáns lemez megfelelő blokkját is, hogy továbbra is az összes adatlemez megfelelő blokkjainak paritás-ellenőrzéseit tartalmazza. Egy naiv megoldás lenne, ha az *n* adatlemez megfelelő blokkjait beolvasnánk, vennénk a modu-

lo-2 összegzést, és a redundáns lemez blokkját újraírnánk. Ez a módszer $n - 1$ olyan adatblokkot olvas be, amit nem is írtunk át, kiírja az újraírt adatblokkot, és egy blokkot ír újra a redundáns lemezen. Ez összesen $n + 1$ lemez I/O-műveletet jelent.

Jobb az a megközelítés, mely szerint csak az újraírt *i.* adatblokknak nézzük meg a régi és az új értékét. Ha vesszük ezek modulo-2 összegét, akkor tudni fogjuk, hogy az összes lemez *i.* blokkjaiban melyik pozícióban változott meg az 1 értékek száma. Mivel egy ilyen változás csak eggyel különbözhet az előző értéktől, ezért páros számú egyesből páratlan számú egyes lesz. Ha most a redundáns lemezen is megváltoztatjuk az értéket ugyanezen a pozíción, akkor az egyesek száma újból minden helyen páros lesz. Ezekhez a számításokhoz négy lemez I/O-műveletre van szükség:

1. Beolvassuk a változtatni kívánt adatblokk régi értékét.
2. Beolvassuk a redundáns lemezről a megfelelő blokkot.
3. Kiírjuk az új adatblokkot.
4. Újraszámoljuk és kiírjuk a redundáns lemez blokkját.

2.19. példa: Tegyük fel, hogy három adatlemezen az első blokkok ugyanazok, mint a 2.17. példában:

1. lemez: 11110000
2. lemez: 10101010
3. lemez: 00111000

Tegyük most fel, hogy a második lemezen a blokkot 10101010-ről 11001100-re változtatjuk. Ha most a 2. lemez blokkjának régi és új értékének vesszük a modulo-2 összegét, akkor a 01100110 sorozatot kapjuk. Ebből kiolvashatjuk, hogy a redundáns lemez első blokkjában a 2., 3., 6. és 7. helyen kell változtatni. Beolvassuk ezt a blokkot: 01100010. Ezt a blokkot azzal az új blokkal helyettesítjük, amelyet úgy kapunk, hogy a megfelelő pozíciókban változtatunk, ami hatását tekintve megegyezik azzal, mintha vennénk a redundáns lemezről beolvasott blokknak és a 01100110-nak a modulo-2 összegét, azaz 00000100-t. Egy másik mód arra, hogy kifejezzük az új redundáns blokkot az, hogy modulo-2 összeadjuk az újraírt blokk régi és új értékét és a redundáns blokk régi értékét. Ezzel a példánkban a 4 lemez (három adatlemez és egy redundáns lemez) első blokkja a második lemez blokkjának és a redundáns blokkon végrehajtott szükséges újraszámolás-kiírása után a következő lesz:

1. lemez: 11110000
2. lemez: 11001100
3. lemez: 00111000
4. lemez: 00000100

Vegyük észre, hogy a fenti blokkok esetében minden oszlopban továbbra is páros sok egyes szerepel.

Vegyük észre, hogy példánkban egy adatblokk újraírásához 4 lemez I/O-műveletre volt szükség, annyira, amennyit a fenti séma alapján az adatblokkok írásához kell használni. Ez a műveletszám most történetesen megegyezik a naiv módszer műveletigényével. A naiv séma esetén az újraírt blokk kivételével beolvassuk a többi blokkot, és közvetlenül újraszámoljuk a redundáns blokkot, azaz beolvassuk az adatokat az első és a harmadik lemezről; ez két művelet, és két művelet kell még a második lemez és a redundáns lemez írásához, azaz összesen négy műveletre van szükség. Természetesen, ha háromnál több adatlemezünk lenne, akkor a naiv séma I/O-száma az adatlemezek számával lineárisan nőne, míg az itt javasolt séma költsége továbbra is négy művelet maradna. □

A modulo-2 összegzések algebrája

Ahhoz, hogy jobban megértsük a paritás-ellenőrzésekhez használt trükköket, hasznos lehet, ha ismerjük, hogy milyen algebrai szabályok vonatkoznak a bitvektorok modulo-2 összeadási műveletére. Ezt a műveletet $\oplus$ jellel jelöljük. Például $1100 \oplus 1010 = 0110$. Az alábbiakban megadunk néhány hasznos szabályt a $\oplus$ műveletre:

- A *kommutatív törvény:* $x \oplus y = y \oplus x$.
- Az *asszociatív törvény:* $x \oplus (y \oplus z) = (x \oplus y) \oplus z$.
- A megfelelő hosszú csupa 0 vektor, melyet $\overline{0}$-val jelölünk, a $\oplus$ művelet *egységeleme,* vagyis $x \oplus \overline{0} = \overline{0} \oplus x = x$.
- A $\oplus$ műveletnek saját maga az inverze, azaz $x \oplus x = \overline{0}$. Ennek fontos következménye, hogyha $x \oplus y = z$, akkor hozzáadhatjuk x-et mindkét oldalhoz, amivel azt kapjuk, hogy $y = x \oplus z$.

Hibajavítás

Most nézzük meg mit tennénk, ha az egyik lemez megsérülne. Ha ez a redundáns lemez, akkor kicseréljük egy új lemezre, és újraszámoljuk a redundáns blokkokat. Ha a tönkrement lemez egy adatlemez, akkor ezt kell kicserélni egy jó lemezre, és a többi lemez adataiból újra kell számolni az adatait. Bármely hiányzó adat újraszámolásához használhatunk egy tulajdonképpen egyszerű szabályt, amely nem függ attól, hogy melyik lemez, adat vagy redundáns lemez ment tönkre. Mivel tudjuk, hogy az összes lemezen a megfelelő bitek között páros számú egyes szerepel, ebből az következik, hogy:

- Bármelyik bit az összes többi lemez megfelelő helyén álló bitek modulo-2 összege.

Aki kételkedne ebben a szabályban, annak csak két esetet kell végiggondolnia. Ha a kérdéses bit 1, akkor a megfelelő bitek között páratlan sok egyesnek kell szerepelnie, azaz a modulo-2 összegük 1. Ha a kérdéses bit 0, akkor a megfelelő bitek között páros sok egyesnek kell szerepelnie, azaz a modulo-2 összegük 0.

2.20. példa: Tegyük fel, hogy a második lemez megy tönkre. Ki kell számolnunk a lecserélt lemez minden blokkját. Követve a 2.17. példát, nézzük meg, hogy lehet újraszámolni a második lemez első blokkját. Adottak az első, harmadik és a redundáns lemez megfelelő blokkjai, azaz a következő a helyzet:

1. lemez: 11110000
2. lemez: ????????
3. lemez: 00111000
4. lemez: 01100010

Ha most minden oszlopban vesszük a modul-2 összeget, akkor arra következtethetünk, hogy a hiányzó blokk 10101010, ahogy azt a 2.17. példa kiindulásánál láttuk. □

2.6.4. Egy továbbfejlesztés: az 5. szintű RAID

A 2.6.3.-ban leírt 4. szintű RAID eredményesen használható adatok megőrzésére, kivéve, ha majdnem egyszerre két lemez megy tönkre. Hogy hol van a legnagyobb baj ezzel a módszerrel, az rögtön látható, ha újra megvizsgáljuk, hogy mi történik egy új adatblokk írásánál. Bármilyen sémát is használunk a lemezek módosítására, a redundáns lemez blokkját olvasni és írni is kell. Ha n darab adatlemezünk van, akkor a redundáns lemezre írások száma az n-szerese annak, mint ahányszor átlagosan írunk egy tetszőleges adatlemezt.

Viszont a 2.20. példában megfigyeltük, hogy a javítási szabály ugyanaz az adatlemezekre és a redundáns lemezre, azaz venni kell a többi lemez megfelelő bitjeinek modulo-2 összegét. Emiatt nem kell külön foglalkoznunk azzal, hogy melyik a redundáns lemez és melyik az adatlemez. Sőt minden lemezt úgy tekinthetünk, mintha bizonyos blokkokhoz tartozó redundáns lemez lenne. Ezt a továbbfejlesztést gyakran *5. szintű RAID-nek* hívják.

Például ha $n + 1$ lemezünk van, melyeket 0-tól n-ig számozunk, akkor a j. lemez i. cilinderét redundánsnak tekintjük, ha az i-t $n + 1$-gyel osztva j-t kapunk maradékul.

2.21. példa: Vegyük az alappéldánkat, azaz $n = 3$, így 4 lemezünk van. Az első, azaz a 0 sorszámú lemeznek a 4, 8, 12 stb. sorszámú cilinderei lesznek redundánsak, mert ezek a számok adnak 0-t, ha néggyel osztjuk őket. Az első lemezen az 1, 5, 9 stb. blokkok lesznek redundánsak, a 2. lemezen a 2, 6, 10, ..., és a 3. lemezen a 3, 7, 11, ... sorszámú blokkok.

Ennek eredményeképpen minden lemez olvasási és írási terhelése megegyezik. Ha minden blokkot egyenlő valószínűséggel írunk, akkor egy írás esetén minden lemezen 1/4 az esély, hogy a blokk azon a lemezen van. Ha nincs rajta, akkor 1/3 az esélye, hogy ennek a blokknak ő a redundáns lemeze. Tehát a négy közül mindegyik lemez az írások $\frac{1}{4}+\frac{3}{4}\times\frac{1}{3}=\frac{1}{2}$ részében vesz részt. □

2.6.5. Mi a teendő, ha több lemez is tönkremehet?

A hibajavító kódok elmélete lehetővé teszi, hogy tetszőleges számú lemez (adat vagy redundáns lemez) tönkremenését kezelni tudjuk, ha elég sok redundáns lemezt használunk. Ez a stratégia jelenti a legmagasabb RAID-fokozatot, a *6. szintű RAID-et*. Adni fogunk egy egyszerű példát, amelyben két egyidejű tönkremenés kijavítható. Az ehhez használt stratégia a legegyszerűbb hibajavító kódon, a *Hamming-kódon* alapul.

A leírásunkban egy olyan rendszerrel foglalkozunk, amelynek 7 lemeze van, 1-től 7-ig számozva. Az első négy adatlemez, a maradék három pedig redundáns lemez. Az adatlemezek és a redundáns lemezek közti kapcsolatot a 2.17. ábrán látható 0-kból és 1-ekből álló 3 × 7-es mátrix írja le. Vegyük észre, hogy:

a) Minden lehetséges 0-kból és 1-ekből képezhető oszlop megjelenik a 2.17. ábra mátrixában, kivéve a csupa nullából álló oszlopot.
b) A redundáns lemezek oszlopaiban csak egy egyes szerepel.
c) Az adatlemezek oszlopaiban legalább két egyes szerepel.

	Adat				*Redundáns*		
Lemez száma	1	2	3	4	5	6	7
	1	1	1	0	1	0	0
	1	1	0	1	0	1	0
	1	0	1	1	0	0	1

2.17. ábra. *Redundanciaminta egy olyan rendszerre, amely két lemez egyidejű tönkremenését is helyre tudja hozni*

Ennek a nullákból és egyesekből álló három sornak a következő az értelme. Ha megnézzük mind a hét lemez megfelelő bitjeit, akkor azokra a lemezekre modulo-2 összegezve a biteket, ahol a sorban egyes szerepel, nullát kapunk. Másképpen elmondva, azok a lemezek, amelyekhez a sorban egyes tartozik, együttesen 4. szintű RAID-sémájú lemezhalmazt alkotnak. Tehát egy redundáns lemez bitjeit úgy számolhatjuk ki, hogy megnézzük melyik sorban szerepel egyes ennek a redundáns lemeznek az oszlopában, kiválasztjuk azokat a lemezeket, amelyeknél szintén egyes szerepel ebben a sorban, és ezeknek a lemezeknek a megfelelő bitjeit modulo-2 összeadjuk. Ebből a szabályból a 2.17. ábra mátrixára az alábbiak következnek:

1. Az 5. lemez bitjeit úgy kapjuk, hogy az 1., 2. és a 3. lemez megfelelő bitjeit modulo-2 összeadjuk.
2. Az 6. lemez bitjeit úgy kapjuk, hogy az 1., 2. és a 4. lemez megfelelő bitjeit modulo-2 összeadjuk.
3. Az 7. lemez bitjeit úgy kapjuk, hogy az 1., 3. és a 4. lemez megfelelő bitjeit modulo-2 összeadjuk.

Mindjárt látni fogjuk, hogy a mátrixban a biteknek ez a speciális választása egy egyszerű szabályt ad a kezünkbe, amellyel két lemez egyidejű tönkremenetelét is helyre tudjuk hozni.

Olvasás

Bármelyik adatlemezről normálisan olvashatjuk be az adatokat. A redundáns lemezeket figyelmen kívül hagyhatjuk.

Írás

Hasonló elvet követünk, mint a 2.6.4. részben vázolt írásra vonatkozó stratégiában, csak most lehet, hogy több redundáns lemezzel kel törődnünk. Ha egy adatlemezen írni akarunk egy új blokkot, akkor kiszámoljuk a blokk régi és új változatának a modulo-2 összegét. Megnézzük, hogy melyek azok a redundáns lemezek, amelyekhez van olyan sor a mátrixban, hogy az adatlemez oszlopában és a redundáns lemez oszlopában is egyes szerepel. Ezeknek a redundáns lemezeknek megfelelő blokkjaihoz modulo-2 hozzáadjuk az előbb kiszámolt biteket.

2.22. példa: Megint tegyük fel, hogy a blokkok csak 8 bit hosszúak, és fordítsuk a figyelmünket a 6. szintű RAID példában szereplő 7 lemez első blokkjaira. Először tegyük fel, hogy az első adat és redundáns blokkok a 2.18. ábrán látható módon vannak megadva. Vegyük észre, hogy az 5. lemez blokkja az első három lemez blokkjainak modulo-2 összege, a 6. sor az 1., 2. és 4. sorok modulo-2 összege, és az utolsó sor az 1., 3. és a 4. sorok modulo-2 összege.

Tegyük fel, hogy a 2. lemez első blokkját át akarjuk írni 00001111-re. Ha ezt

Lemez	*Tartalom*
1)	11110000
2)	10101010
3)	00111000
4)	01000001
5)	01100010
6)	00011011
7)	10001001

2.18. ábra. *A hét lemez első blokkja*

Lemez	*Tartalom*
1)	11110000
2)	00001111
3)	00111000
4)	01000001
5)	11000111
6)	10111110
7)	10001001

2.19. ábra. *A lemezek első blokkja azután, hogy a 2. lemezt újraírtuk és a redundáns lemezeket is megváltoztattuk*

modulo-2 hozzáadjuk a blokk régi értékéhez, azaz 10101010-hoz, akkor az 10100101 sorozatot kapjuk. Ha egy pillantást vetünk a 2.17. ábra 2. lemezének oszlopára, akkor azt látjuk, hogy az első két sorban szerepel egyes, a harmadikban nem. Mivel az első két sorban az 5. és a 6. redundáns lemez oszlopán látunk szintén egyest, ezért ezeknek az első blokkjaihoz kell modulo-2 hozzáadni a most kiszámolt 10100101 sorozatot. Emiatt ezekben a blokkokban az 1., 3., 6. és 8. pozíciókban szereplő értékeket az ellenkező értékre állítjuk. A 2.19. ábrán látható az összes lemez változás utáni első blokkjának tartalma. Vegyük észre, hogy továbbra is érvényben marad a 2.17. ábrán megfigyelt megszorítás, azaz a 2.17. ábra mátrixának bármelyik sora alapján kiválasztva azokat a lemezeket, amelyekre egyes szerepel a sorban, és ezeknek a lemezeknek a megfelelő blokkokat modul-2 összeadva, eredményül mindig a csupa 0 sorozatot kapjuk. □

Hibajavítás

Most nézzük meg, hogyan lehet az előbb vázolt redundanciasémát felhasználni arra, hogy két egyidejűleg tönkrement lemezt helyreállítsunk. Legyen a és b a két tönkrement lemez. Mivel a 2.17. ábra mátrixának minden oszlopa különböző, ezért lehet találni egy olyan r sort, amelynek az a és b oszlopa különbözik. Tegyük fel, hogy az r sorban az a értéke 0, míg a b értéke 1.

Ekkor a b helyes értékét kiszámolhatjuk úgy, hogy vesszük a b-n kívül az összes olyan lemezt, amelynél egyes szerepel az r sorban, és ezeknek a lemezeknek a megfelelő bitjeit modulo-2 összeadjuk. Vegyük észre, hogy az a nem szerepel ezek között a lemezek között, így nem fordulhat elő, hogy nem tudjuk elvégezni a modulo-2 összeadást. Miután ezt megtettük, ki kell számolnunk az a-t is a többi lemez felhasználásával. Mivel a 2.17. ábra mátrixának minden oszlopában legalább egy egyes szerepel, ezért vehetünk egy olyan sort, amelynek az a oszlopában egyes szerepel. Ha most vesszük az összes a-tól különböző lemezt, amelyre egyes szerepel ebben a sorban, és ezeknek a megfelelő bitjeit modulo-2 összeadjuk, akkor sikeresen újra kiszámoltuk az a lemez tartalmát.

2.23. példa: Tegyük fel, hogy egyszerre tönkrement a 2. és az 5. lemez. A 2.17. ábra mátrixát megvizsgálva észrevehetjük, hogy az ennek a két lemeznek megfelelő oszlopok különböznek a 2. sorban, ahol a 2. lemeznél egyes szerepel, míg az 5. lemeznél nulla. Tehát rekonstruálhatjuk a 2. lemezt úgy, hogy vesszük az 1., 4. és 6. lemezeket, vagyis, ahol szintén egyes szerepel ebben a sorban, és ezeknek a lemezeknek a megfelelő bitjeit modulo-2 összeadjuk. Vegyük észre, hogy a fenti három lemez között nem szerepel tönkrement lemez. Folytassuk például a 2.19. ábrának megfelelő helyzetet az első blokkokra vonatkozóan, azaz a 2. és 5. lemez tönkremenése után legyenek adva kezdetben a 2.20. ábra adatai.

Ha most az 1., 4. és 6. lemezek blokkjainak tartalmát modulo-2 összeadjuk, akkor a második lemeznek erre a blokkjára a 00001111-et kapjuk, amelyről a 2.19. ábrán ellenőrizhetjük, hogy tényleg ez a helyes blokk.

Most vegyük észre, hogy a 2.17. ábrán az 5. lemez oszlopában az első sorban egyes szerepel, ezért újra kiszámolhatjuk az 5. lemezt úgy, hogy vesszük a többi olyan lemezt, melyre szintén egyes szerepel az első sorban, így kapjuk az 1., 2. és 3. lemezeket, aztán ezek megfelelő bitjeit modulo-2 összeadjuk. Az első blokkra így 11000111 lesz az összeg. A számítás helyességéről meggyőződhetünk a 2.19. ábra segítségével. □

Lemez	*Tartalom*
1)	11110000
2)	????????
3)	00111000
4)	01000001
5)	????????
6)	10111110
7)	10001001

2.20. ábra. *A 2. és 5. lemez tönkremenése utáni helyzet*

2.6.6. Feladatok

2.6.1. feladat: Tegyük fel, hogy a 2.16. példához hasonlóan tükrözzük a lemezeket. A meghibásodási arány 4% évente. 8 órát vesz igénybe egy lemez cseréje. Mekkora a várható értéke az olyan lemezhibának, amely adatvesztéssel is jár?

*! **2.6.2. feladat:** Tegyük fel, hogy a lemezeknek a meghibásodási rátája egy F törtszám évente és H óra kell egy lemez cseréjéhez.

a) Ha tükrözött lemezeket használunk, akkor F és H függvényében mennyi az adatvesztési idő várható értéke?
b) Ha 4. vagy 5. szintű RAID-sémát használunk N számú lemezzel, akkor mennyi az adatvesztési idő várható értéke?

Lemez	*Tartalom*
1)	11110000
2)	00001111
3)	00111000
4)	01000001
5)	????????
6)	10111110
7)	10001001

2.21. ábra. *A 2. lemez helyreállítása után*

!! **2.6.3. feladat:** Tegyük fel, hogy három lemezt használunk tükrözött csoportként, azaz mindhárom lemez azonos adatokat tartalmaz. Tegyük fel, hogy egy lemez meghibásodási rátája egy F törtszám évente és H óra kell egy lemez helyreállításához. Az F és H függvényében mennyi az adatvesztési idő várható értéke?

További észrevételek a 6. szintű RAID-del kapcsolatban

1. Az 5. és 6. szintű RAID elvét össze is kombinálhatjuk, azaz a redundáns lemezeket a blokk vagy cilinder sorszáma szerint változtatjuk. Ha így teszünk, akkor elkerüljük azt az írásnál jelentkező problémát, hogy a redundáns lemezeket többet használjuk, mint az adatlemezeket, ugyanis a 2.6.5. részben leírt sémának a szűk keresztmetszetét a redundáns lemezek használata jelenti.
2. A 2.6.5. részben leírt séma nem csak négy lemezre használható. A lemezek száma egy 2-hatványnál eggyel kisebb szám lehet, azaz $2^k - 1$. Ekkor a lemezek közül k darab redundáns és a többi, azaz $2^k - k - 1$ darab pedig adatlemez. Emiatt a redundancia durván a lemezek számának logaritmusával arányosan nő. Tetszőleges k-ra egy 2.17. ábrának megfelelő mátrixot készíthetünk úgy, hogy vesszük az összes nullát és egyest tartalmazó lehetséges k dimenziós oszlopot, kivéve a csupa nullából álló oszlopot. Azok az oszlopok, amelyek egyetlen egyest tartalmaznak, a redundáns lemezeknek felelnek meg, az egynél több egyest tartalmazó oszlopok pedig az adatlemezeknek.

2.6.4. feladat: Tegyük fel, hogy 4. szintű RAID-sémát használunk négy adatlemezzel és egy redundáns lemezzel. A 2.17. példához hasonlóan tegyük fel, hogy a blokkok mérete egy bájt. Adjuk meg a redundáns lemez blokkját, ha a megfelelő blokkok az adatlemezeken a következők:

* a) 01010110, 11000000, 00111011 és 11111011.
b) 11110000, 11111000, 00111111 és 00000001.

2.6.5. feladat: Használjuk ugyanazt a 4. szintű RAID-sémát, mint a 2.6.4. feladatban, és tegyük fel, hogy tönkremegy az 1. lemez. Hozzuk helyre az elromlott lemez blokkját a következő körülmények mellett:

* a) A 2., 3. és 4. lemez tartalma 01010110, 11000000 és 00111011, a redundáns lemez tartalma pedig 11111011.
b) A 2., 3. és 4. lemez tartalma 11110000, 11111000 és 00111111, a redundáns lemez tartalma pedig 00000001.

2.6.6. feladat: Tegyük fel, hogy a 2.6.4. feladatban az első lemez blokkját 10101010-ra változtatjuk. Milyen változtatást kell tenni a többi lemez megfelelő blokkjain?

2.6.7. feladat: Tegyük fel, hogy a 2.22. példában szereplő 6. szintű RAID-sémával dolgozunk. A négy adatlemez blokkjai rendre 00111100, 11000111, 01010101 és 10000100.

a) Mik a redundáns lemezek megfelelő blokkjai?

Hibajavító kódok és a 6. szintű RAID

A redundáns lemezek tartalmának meghatározásához egy megfelelő mátrixot kell választanunk, amely olyan, mint a 2.17. ábra mátrixa. A mátrix kiválasztásában egy alaposan kidolgozott elmélet nyújt segítséget. Egy n hosszú bitvektorokból *(kódszavakból)* álló halmazt n hosszú *kódnak* hívunk. Két kódszó közti *Hamming-távolság* az a szám, ahány pozícióban a két kódszó különbözik. Egy kód *minimális távolsága* két tetszőleges, különböző kódszó közti legkisebb Hamming-távolság.

Legyen C egy tetszőleges n hosszú kód. Megkövetelhetjük, hogy n lemez megfelelő bitjei olyan sorozatokat alkossanak, amelyek a kód elemei. Vegyünk egy egyszerű példát, melyben egy lemezt és a tükörképét használjuk, azaz $n = 2$. Ekkor használhatjuk a $C = \{00, 11\}$ kódot, mert ez azt jelenti, hogy a két lemeznek a megfelelő bitjei meg kell hogy egyezzenek. Egy másik példaként nézzük a 2.17. ábra mátrixát. Ez a mátrix egy olyan kódot definiál, amely 16 darab 7 hosszú bitvektorból áll, melyek első négy bitje tetszőlegesen választható, de a maradék három bitet a három redundáns lemezre vonatkozó szabály határozza meg.

Legyen egy kód minimális távolsága d. Legyenek a lemezeink olyanok, melyek tartalmára igaz, hogy a lemezek megfelelő bitjeiből álló sorozatok mindig a kódnak valamelyik vektorával egyeznek meg. Ekkor ez a rendszer $d - 1$ lemez egyidejű meghibásodását is helyre tudja hozni. Ennek az az oka, hogy ha egy kódszóban kitörlünk $d - 1$ helyet, és feltennénk, hogy kétféleképpen is ki lehetne tölteni ezeket a pozíciókat úgy, hogy kódszavakat kapjunk, akkor az így kapott két kódszó legfeljebb $d - 1$ pozícióban különbözne, de akkor nem lehetett volna d a kód minimális távolsága. Például vegyük a 2.17. ábra mátrixát, mely a jól ismert *Hamming-kódot* definiálja, és amelynek 3 a minimális távolsága. Emiatt ezzel két lemez meghibásodását lehet kezelni.

b) Ha a harmadik lemez blokkját átírjuk 10000000-ra, akkor milyen lépéseket kell tennünk a többi lemez megváltoztatásának érdekében?

2.6.8. feladat: Legyen adott egy 6. szintű RAID-séma hét lemezzel. Írjuk le, hogy milyen lépéseket kell végrehajtani a helyreállításhoz, ha a következő lemezek sérülnek meg:

* a) 1. és 7. lemez.
b) 1. és 4. lemez.
c) 3. és 6. lemez.

2.6.9. feladat: Keressünk olyan 6. szintű RAID-sémát, amely 15 lemezre használható. A 15 lemezből 4 redundáns lemez. *Segítség:* Általánosítsuk a 7 lemezes Hamming-mátrixot.

2.6.10. feladat: Adjuk meg a 7 hosszú Hamming-kódnak mind a 16 kódszavát, azaz melyik az a 16 megengedett bitsorozat, amely előfordulhat 7 lemez megfelelő bitsorozataként, ha a 2.17. ábra mátrixán alapuló 6. szintű RAID-sémát használjuk?

2.6.11. feladat. Tegyük fel, hogy négy lemezünk van, az első kettő adatlemez, a 3. és 4. pedig redundáns. A 3. lemez az 1. lemez tükörképe. A 4. lemez a 2. és 3. lemez megfelelő bitjeinek paritás-ellenőrző bitjeit tartalmazza.

a) Fejezzük ki ezt a helyzetet azzal, hogy megadunk egy 2.17. ábrához hasonló paritás-ellenőrző mátrixot.
!! b) Bizonyos esetekben, de nem az összes lehetséges esetben, ez a rendszer is képes a helyreállításra, ha két lemez egyszerre megy tönkre. Határozzuk meg, hogy milyen párokra lehetséges a helyreállítás, milyen párokra nem lehetséges.

*! **2.6.12. feladat:** Tegyük fel, hogy nyolc adatlemezünk van, 1-től 8-ig számozva, és három redundáns lemezünk, a 9., 10. és 11. lemez. A 9. lemez az első négy adatlemez paritás-ellenőrzése, a 10. lemez pedig a további négy lemez paritás-ellenőrzése. Tegyük fel, hogy bármelyik két lemez egyforma valószínűséggel megy egyszerre tönkre. Mely lemezeknek legyen a 11. lemez a paritás-ellenőrzése, ha maximalizálni akarjuk annak a valószínűségét, hogy két lemez egyidejű meghibásodását is helyre tudjuk hozni?

!! **2.6.13. feladat:** Adjunk meg egy tíz lemezre vonatkozó 6. szintű RAID-sémát, melylyel lehetséges a helyreállítás akkor is, ha egyszerre három lemez megy tönkre. Használjunk annyi adatlemezt, amennyit csak lehet.

2.7. Összefoglalás

- *Memóriahierarchia:* Egy számítógépes rendszer többféle tárolóelemet használ. Ezek a sebesség, kapacitás és egy bitre jutó költség tekintetében különböző nagyságrendűek lehetnek. A legkisebbtől/legdrágábbtól a legnagyobbig/legolcsóbbig a sorrend a következő: cache, központi memória, másodlagos memória (lemez), harmadlagos memória.
- *Harmadlagos tárolás:* Az alapvető harmadlagos tárolóeszközök a következők: szalagos kazetták, szalagsilók (szalagos kazettákat kezelő mechanikus eszközök), lemeztárak vagy „juke box"-ok (CD-ROM-lemezeket kezelő mechanikus eszközök). Ezeknek a tárolóeszközöknek a kapacitása sok terabájt, de ezek a leglassabb tárolóeszközök.
- *Lemezek/másodlagos tárolás:* A másodlagos tárolóeszközök alapvetően sok gigabájt kapacitással rendelkező mágneses lemezek. A lemezegységek több kör alakú, mágneses anyaggal bevont tányérból állnak, melyek koncentrikus sávokban tárolják a biteket. A tányérok egy középen elhelyezett tengely körül forognak. A középponttól azonos távolságra elhelyezkedő sávok alkotnak egy cilindert.

- *Blokkok és szektorok:* A sávok szektorokra vannak felosztva, melyeket nem mágnesezett hézagok választanak el egymástól. A szektor a lemezről olvasás, illetve lemezre írás egysége. A blokkok a tárolás logikai egységei, melyeket alkalmazások, például adatbázis-kezelő rendszerek használnak. A blokkok tipikusan több szektorból állnak.
- *Lemezvezérlő:* A lemezvezérlő egy olyan processzor, mely egy vagy több lemezegységet vezérel. A lemezvezérlő felelős azért, hogy mikor olvasni vagy írni akarunk egy sávot, akkor a lemezfejeket a megfelelő cilinderre vigye. Feladata lehet még a versenyző igények ütemezése, és azoknak a blokkoknak a pufferezése, amiket írunk vagy olvasunk.
- *Lemezelérési idő:* Egy lemez késése az az idő, amely egy blokk olvasására vagy írására vonatkozó igény beérkezése és a blokkelérés végrehajtása között telik el. A késést alapvetően három tényező okozza: a keresési idő, amely ahhoz kell, hogy a fejek a megfelelő cilinderig eljussanak, a rotációs késés, amely ahhoz kell, hogy a kívánt blokk a lemez forgása közben a fej alá kerüljön, és végül az átviteli idő, amely ahhoz kell, hogy a fej alatti blokkot olvassuk vagy írjuk.
- *Moore törvénye:* Egy valósággal összhangban lévő irányzat szerint az olyan paraméterek, mint a processzor sebessége, a lemez kapacitása és a központi memória kapacitása minden 18 hónapban megduplázódik. Ezzel szemben hasonló időszak alatt a lemezelérési idő alig csökken, ha egyáltalán változik. Ennek egy fontos következménye, hogy a lemezelérés (relatív) költsége az évek során növekszik.
- *Másodlagos tárolókat használó algoritmusok:* Ha az adatmennyiség túl nagy, akkor nem fér el a központi memóriában. Ekkor olyan algoritmusokat kell használni az adatok kezelésére, melyek figyelembe veszik, hogy a lemez és memória között lezajló műveletek, lemezblokk olvasása, írása gyakran sokkal tovább tart, mint amennyi a központi memóriában szükséges ahhoz, hogy a memóriába bekerült adatokkal műveleteket végezzünk. A másodlagos memóriában tárolt adatokra vonatkozó algoritmusok értékelésénél ezért elsődlegesen azt kell vizsgálni, hogy mennyi lemez I/O-műveletet igényel az algoritmus.
- *Kétfázisú, többutas, összefésülő rendezés:* Ez a rendező algoritmus lemezen tárolt hatalmas adatmennyiséget képes rendezni. Minden adathoz csak két lemezolvasást és két lemezírást használ. A legtöbb adatbázis-alkalmazásban ezt a rendezési módszert választják.
- *A lemezelérés gyorsítása:* Több technika is használható arra, hogy bizonyos alkalmazásokhoz a lemezblokkokat gyorsabban lehessen elérni. Ezek közül a következőket emeljük ki: szétosztjuk az adatokat több lemezre (ezzel lehetségessé válik a párhuzamos elérés), tükrözzük a lemezeket (az adatok több példányának kezelése szintén megengedi a párhuzamos hozzáférést), azonos sávra vagy cilinderre helyezzük azokat az adatokat, amelyeket együtt akarunk elérni, korai beolvasást, dupla pufferezést használunk, azaz együtt olvasunk vagy írunk teljes sávokat vagy cilindereket.
- *Lift algoritmus:* Azzal is fel lehet gyorsítani az elérést, ha az elérési igényeket sorba állítjuk, és olyan sorrendben dolgozzuk fel őket, hogy a fejek mindig oda-vissza, végigmenjenek a teljes lemezen. Az igények kezelésére a fejek mindig megállnak, ha elérnek egy olyan cilinderhez, amely egy vagy több olyan blokkot tartalmaz, amelyre várakozási listán szereplő igény vonatkozik.

- *Lemezhibák típusai:* A rendszereknek kezelni kell tudniuk a hibákat azért, hogy elkerüljék az adatvesztést. Az alapvető lemezhibatípusok a következők: ideiglenes (ha elégszer megismételjük a műveletet, akkor egy idő után az írási vagy olvasási hiba nem fog előfordulni), állandó (adatok romlottak el a lemezen, és nem lehet helyesen olvasni őket), a lemez tönkremenése (a teljes lemez olvashatatlanná vált).
- *Ellenőrző összegek:* Ha egy plusz paritás-ellenőrző bittel egészítjük ki a bitsorozatainkat úgy, hogy a bitsorozatban szereplő egyesek számát az extra bit párossá teszi, akkor az ideiglenes és állandó hibákat felismerhetjük, bár nem tudjuk őket kijavítani.
- *Stabil tárolás:* Minden adatból két másolatot készítünk, és ügyelünk arra, hogy milyen sorrendben írjuk ezeket a másolatokat. Ezáltal egyetlen lemez használható arra, hogy kivédjünk majdnem minden, egy szektorra vonatkozó állandó hibát.
- *RAID:* Többféle séma létezik arra vonatkozóan, hogy lehet egy vagy több lemez hozzáadásával elérni, hogy az adatok túlélhessenek egy lemeztönkremenést. Az 1. szintű RAID a lemezek tükrözését jelenti. A 4. szintű RAID egy plusz lemez hozzáadását jelenti, amely az összes többi lemez megfelelő bitjeinek paritásellenőrzéseit tartalmazza. Az 5. szintű RAID váltogatja, hogy a paritás bit mikor melyik lemezre kerüljön, ezáltal megóv attól, hogy csak egyetlen paritáslemez legyen, ami az írás szűk keresztmetszetét jelentené. A 6. szintű RAID már hibajavító kódok használatát is magában foglalja, és lehetővé teszi, hogy az adatok több lemez egyidejű tönkremenését is túléljék.

2.8. Irodalomjegyzék

A RAID-elv a [6]-ban szereplő lemezsávozásra vezethető vissza. A hibajavító képesség neve az [5]-ből ered.

A 2.5. részben szereplő lemezmeghibásodási modell Lampson és Sturgis meg nem jelent [4] munkájában olvasható.

A fejezet lényeges elemeihez több hasznos áttekintés is létezik. A [2] a lemeztárolók és hasonló rendszerek irányzatait vizsgálja. A RAID-rendszerek összefoglalása az [1]-ben található. A [7] azokat az algoritmusokat tekinti át, melyek a számítás során másodlagos tárolási modellt (blokkmodellt) használnak.

A [3] egy fontos tanulmány arról, hogy egy bizonyos feladat végrehajtásához hogyan lehet optimalizálni egy processzorral, memóriával és lemezzel ellátott rendszert.

1. P. M. Chen et al., „RAID: high-performance, reliable secondary storage", *Computing Surveys*, **26**:2 (1994), pp. 145–186.
2. G. A. Gibson et al., „Strategic directions in storage I/O-issues in large-scale computing", *Computing Surveys*, **28**:4 (1996), pp. 779–793.
3. J. N. Gray and F. Putzolo, „The five minute rule for trading memory for disk accesses and the 10 byte rule for trading memory for CPU time", *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, (1987), pp. 395–398.

4. B. Lampson and H. Sturgis, „Crash recovery in a distributed data storage system", Technical report, Xerox Palo Alto Research Center, 1976.
5. D. A. Patterson, G. A. Gibson, and R. H. Katz, „A case for redundant arrays of inexpensive disks", *Proc. ACM SIGMOD Intl. Conf. on Management of Data,* (1988), pp. 109–116.
6. K. Salem and H. Garcia-Molina, „Disk striping", *Proc. Second Intl. Conf. on Data Engineering,* (1986), pp. 336–342.
7. J. S. Vitter, „External memory algorithms", *Proc. Seventeenth Annual ACM Symposium on Principles of Database Systems,* (1998), pp. 119–128.

3. fejezet

Adatelemek ábrázolása

Ez a fejezet összekapcsolja a másodlagos tárolók 2.3. részben bemutatott blokkmodelljét az adatbázis-kezelő rendszerek követelményeivel. Először is azzal kezdjük, hogy megnézzük, milyen módon ábrázolhatók a relációk vagy objektumhalmazok a másodlagos tárolón.

- Az attribútumokat rögzített vagy változó hosszú bájtsorozatokkal kell ábrázolni. Ezeket a sorozatokat „mezőknek" hívjuk.
- A mezőket sorban összetéve rögzített vagy változó hosszú csoportokat kapunk, melyeket „rekordoknak" hívunk, és ezek felelnek meg a relációsoroknak vagy objektumoknak.
- A rekordokat fizikai blokkokban kell tárolni. Többféle adatstruktúra is használható olyankor, ha a rekordokból álló blokkokat újra kell szervezni az adatbázis módosításakor.
- Azoknak a rekordoknak a csoportját, melyek egy relációt alkotnak vagy az osztálynak egy előfordulását (extent) alkotják, blokkok csoportjaként tároljuk, és ezt hívjuk *fájlnak*[1] (file). Ahhoz, hogy ezeket a csoportokat hatékonyan lehessen lekérdezni vagy módosítani, ráépítünk a fájlra egy „indexet" a számos szóba jöhető indexstruktúra közül. Ezekkel a struktúrákkal foglalkozik a 4. és 5. fejezet.

3.1. Adatelemek és mezők

Azzal fogjuk kezdeni, hogy megnézzük a legáltalánosabb adatelemek ábrázolását, vagyis azt, hogy miképpen ábrázolhatók a relációs vagy objektumorientált adatbázisrendszerben található attribútumok értékei. Ezeket ábrázoljuk „mezőkkel". Ezt követően látni fogjuk, hogyan kell a mezőket összetenni ahhoz, hogy a tárolórendszer nagyobb elemeit kapjuk: rekordokat, blokkokat és fájlokat.

[1] Az adatbázis fájlfogalma valamivel általánosabb, mint az operációs rendszer fájlfogalma. Bár az adatbázisfájl egy struktúrával nem rendelkező bájtfolyam is lehet, azért az a megszokottabb, hogy egy fájl a blokkoknak olyan csoportjából áll, amely valami hasznos módon szervezve van, például indexek segítségével vagy egyéb speciális elérési módszerekkel. Ezeket a szervezéseket a 4. fejezetben fogjuk tárgyalni.

3.1.1. Relációs adatbáziselemek ábrázolása

Tegyük fel, hogy egy SQL-rendszerben egy `CREATE TABLE` utasítással definiáltunk egy relációt úgy, mint ahogy a 3.1. ábrán látható. Az adatbázis-kezelő rendszer feladata, hogy a deklarációval leírt relációt ábrázolja és tárolja. Mivel a reláció nem más, mint relációsoroknak egy halmaza, és a relációsorok a rekordokhoz (esetleg C, illetve C++ terminológiában a „struct"-okhoz) hasonlóak, ezért úgy képzelhetjük el, hogy minden relációsor egy rekordként lesz tárolva a lemezen. A rekord valamilyen lemezblokkot vagy annak egy részét foglalja el. A rekordon belül a reláció minden attribútumához egy mező fog tartozni.

```
CREATE TABLE Filmszínész(
   név CHAR(30) PRIMARY KEY,
   cím VARCHAR(255),
   neme CHAR(1),
   születési_idő DATE
);
```

3.1. ábra. *Egy SQL-tábla deklarálása*

Bár az általános elv egyszerűnek tűnik, de „az ördög a részletekben lakozik", ezért meg kell majd vizsgálnunk a következő kérdéseket:

1. Hogyan ábrázoljuk az SQL-adattípusokat mezők formájában?
2. Hogyan ábrázoljuk a relációsorokat rekordok formájában?
3. Hogyan ábrázoljuk a rekordok vagy relációsorok csoportjait a memóriablokkokban?
4. Hogyan ábrázoljuk és tároljuk a relációkat blokkokból álló csoportok formájában?
5. Hogyan birkózhatunk meg az olyan problémákkal, hogy a rekordméretek a különböző relációsorokra különbözőek lehetnek, és/vagy a rekordméret nem osztja a blokkméretet?
6. Mi történik, ha megváltozik egy rekord mérete, mert az egyik mező módosult? Hogyan találunk üres helyet egy blokkon belül, például, mikor egy rekord mérete megnő?

Ennek a résznek a tárgya az első kérdés megválaszolása. A következő két kérdést a 3.2. rész taglalja. Az utolsó két kérdéssel a 3.4. és a 3.5. részekben fogunk foglalkozni. A negyedik kérdés, vagyis hogyan lehet úgy ábrázolni a relációkat, hogy a relációsorokat hatékonyan el lehessen érni, a 4. fejezet témája lesz.

Továbbá meg kell vizsgálnunk azt is, hogyan lehet speciális típusú adatokat ábrázolni, azaz például olyanokat, amelyek a modern objektumrelációs vagy objektumorientált rendszerekben találhatók. Itt olyanokra gondolunk, mint az objektumazonosítók (vagy másféle rekordmutatók), vagy a nagy bináris objektumok, „blob"-ok (binary, large objects). Ez utóbbi lehet például egy 2 gigabájtos MPEG formátumú film. Ezeket a kérdéseket a 3.3. és 3.4. részek válaszolják meg.

3.1.2. Objektumok ábrázolása

Ma már sok adatbázis-kezelő rendszer támogatja az „objektumokat". Ezek egyik válfaja valóban tiszta objektumorientált adatbázis-kezelő rendszer, amelyben egy C++-hoz hasonló objektumorientált nyelv, kibővítve egy objektumorientált lekérdezőnyelvvel, például OQL[2]-lel, használható befogadó- (host) és lekérdezőnyelvnek. A másik válfajba tartoznak a klasszikus relációs rendszerek objektumrelációs kiterjesztései. Ezek csak akkor támogatják az objektumokat, ha azok egy reláció attribútumainak értékei.

Első megközelítésben egy objektum egy relációsor, és a mezői vagy „példányváltozói" (instance variables) az attribútumok, bár van két fontos különbség köztük.

1. Az objektumok *metódusokkal,* azaz hozzájuk rendelt speciális célú függvényekkel is rendelkezhetnek. Ezeknek a függvényeknek a kódja az objektumokból álló osztály sémájának része.
2. Az objektumok *objektumazonosítóval,* OID-dal (object identifier) is rendelkezhetnek, mely egy címet jelent valamilyen globális címterületen, és amely egyértelműen erre az objektumra utal. Ezenkívül az objektumnak lehetnek más objektumokhoz kötődő kapcsolatai is. Ezeket a kapcsolatokat mutatókkal, vagy mutatókból álló listákkal ábrázolják. A relációs adatoknak nincsenek olyan értékei, amelyek címek lennének, bár látni fogjuk, hogy a kulisszák mögött a relációk megvalósítása a címek és mutatók sokféle kezelését követeli meg. A címek ábrázolásának kérdése összetett feladat, mind a nagy relációk, mind a nagy előfordulással rendelkező osztályok esetén. Ezzel a problémával a 3.3. részben foglalkozunk majd.

3.1. példa: A 3.2. ábrán a `Színész` osztály ODL definícióját látjuk. Ez is a filmszínészeket ábrázolja, bár a megadott információ kissé eltér attól, mint amit a 3.1. ábrán szereplő `FilmSzínész` reláció tartalmaz. Ugyanis nem ábrázoljuk a filmszínész nemét és születési idejét sem, ezzel szemben megadunk egy kapcsolatot a színészek és azok között a filmek között, amelyekben a színészek szerepeltek. Ezt a kapcsolatot a `szerepeltBenne` nevű kapcsolat ábrázolja, mely a színészektől indul, és a filmjeikhez vezet. Ennek inverze a `szereplői` nevű kapcsolat, mely egy filmtől indul és a film szereplőihez vezet. A kapcsolatban szereplő `Film` osztály definícióját most nem adjuk meg.

Egy `Színész` objektumot egy rekorddal lehet ábrázolni. Ennek a rekordnak lesznek olyan mezői, melyek a `név` és `cím` attribútumoknak felelnek meg. Mivel ez utóbbi egy struktúra, ezért egy `cím` nevű mező helyett inkább szívesebben használnánk két mezőt, az `utcát` és a `várost`. Problémásabb a `szerepeltBenne` kapcsolat ábrázolá-

[2] Az OQL (object query language) egy szabványos objektumorientált lekérdezőnyelv. A leírásához lásd: R. G. G. Cattell (ed.) *The Object Database Standard ODMG,* third edition, Morgan-Kaufmann, San Francisco, 1998. Ennek társnyelve az ODL (object description language). Ez utóbbi segítségével lehet adatbázissémákat megadni objektumorientált módon.

sa. Ez a kapcsolat egy olyan halmaz, amelynek elemei `Film` objektumokra mutató hivatkozások. Szükségünk van valamilyen módszerre, amellyel ezeknek a `Film` objektumoknak a helyét ábrázoljuk. Ez általában azt jelenti, hogy meg kell adnunk azt a helyet, valamilyen gép lemezén, ahol tárolják ezeket az objektumokat. Az ilyen címek ábrázolására szolgáló technikákat a 3.3. részben fogjuk tárgyalni. Arra is szükségünk van, hogy egy adott színészhez filmeknek változó hosszú listáját tudjuk ábrázolni. Ez a „változó hosszú rekordok" problémája, ami a 3.4. résznek lesz a témája. □

```
interface Színész {
    attribute string név;
    attribute Struct Címtípus {
        string utca, string város} cím;
    relationship Set<Film> szerepeltBenne
                 inverse Film::szereplői;
};
```

3.2. ábra. *A filmszínész osztály definíciója ODL-ben*

3.1.3. Adatelemek ábrázolása

Először tekintsük át, hogyan lehet az alapvető SQL-adattípusokat ábrázolni egy rekord mezőivel. Végeredményben minden adatot bájtok sorozatával ábrázolunk. Például egy `INTEGER` típusú attribútumot normálisan két vagy négy bájttal ábrázolunk, és egy `FLOAT` típusú attribútumot pedig rendszerint négy vagy nyolc bájttal. Az egészeket és valós számokat úgy kell bitláncokkal ábrázolni, hogy a reprezentáció speciálisan értelmezhető legyen a gép hardverje számára, azért, hogy a szokásos aritmetikai műveleteket végre tudja hajtani rajtuk.

Rögzített hosszú karakterláncok

A legegyszerűbb ábrázolandó karakterláncok azok, amelyeket az SQL `CHAR(`*n*`)` típusa ír le. Ezek rögzített hosszú, nevezetesen *n* hosszú karakterláncok. Egy ilyen típusú attribútumhoz rendelt mező egy *n* bájtból álló tömb, vagy másképpen vektor. Ha az attribútum értéke egy *n*-nél rövidebb lánc, akkor a tömböt speciális *töltelékkarakterekkel* (pad character) töltjük ki. A töltelékkarakterek 8 bites kódja nem egyezik semelyik olyan karakter kódjával, amely SQL-karakterláncokban használható.

3.2. példa: Ha egy *A* attribútumot `CHAR(5)` típusúnak deklaráltunk, akkor az *A*-nak megfelelő mező minden sorban egy ötkarakteres tömb. Ha egy sorban az *A* komponens értéke `'sas'`, akkor a tömb értéke a következő lenne:

```
sas⊥⊥
```


Egy megjegyzés a terminológiával kapcsolatban

A fájlrendszerekben, hagyományos C-hez hasonló programozási nyelvekben, relációs adatbázisnyelvekben (különös tekintettel az SQL-re) vagy az objektumorientált nyelvekben (például Smalltalk, C++ vagy objektumorientált adatbázisnyelv, mint az OQL) különböző elnevezéseket találhatunk a lényegében megegyező fogalmakra. A következő táblázat összegzi, hogy minek mi felel meg, bár lehetnek kisebb különbségek, például egy osztály rendelkezhet metódusokkal, míg egy reláció nem.

	Adatelem	*Rekord*	*Csoport*
Fájlok	mező	rekord	fájl
C	mező	struct	tömb, fájl
SQL	attribútum	relációsor	reláció
OQL	attribútum, kapcsolat	objektum	(egy osztály) előfordulása

Arra fogunk törekedni, hogy a fájlrendszer elnevezéseit használjuk (mező, rekord), kivéve, ha ezeknek a fogalmaknak valamilyen adatbázis-alkalmazásban előforduló speciális használatára akarunk utalni. Ez utóbbi esetben a relációs és/vagy objektumorientált elnevezéseket fogjuk használni.

Itt most a ⊥ töltelékkarakter foglalja el a tömb negyedik és ötödik bájtját. Jegyezzük meg, hogy bár egy SQL-programban idézőjeleket szükséges használni a karakterláncok jelölésére, de ezeket az idézőjeleket nem tároljuk el a karakterlánc értékével együtt. □

Változó hosszú karakterláncok

Időnként egy reláció oszlopában olyan karakterláncok szerepelnek értékként, melyek hossza széles határok között változik. Egy ilyen oszlop típusaként gyakran a `VARCHAR(`*n*`)` SQL-típust használjuk. Ezzel szemben először szándékosan úgy valósítjuk meg az így deklarált attribútumokat, hogy *n* + 1 bájtot rendelünk a karakterlánc értékéhez, függetlenül attól, hogy milyen hosszú. Ezzel az SQL `VARCHAR` típusa tulajdonképpen rögzített hosszú mezőket ábrázol, bár az értéke változó hosszú is lehet. Később, a 3.4. részben fogunk foglalkozni olyan karakterláncokkal, melyek reprezentációjának változhat a hossza. A `VARCHAR` karakterlánc reprezentációjának két szokásos módja a következő:

1. *Hossz plusz tartalom.* Lefoglalunk egy *n* + 1 bájt méretű tömböt. Az első bájt, mely egy 8 bites egész szám, a karakterlánc bájtjainak számával egyezik meg. A karak-

terlánc nem lehet hosszabb n karakternél, az n pedig nem lehet több 255-nél, mert különben nem tudnánk a hosszt egy bájtban ábrázolni.[3] A tömb bájtjai közül azokat, amelyeket nem használunk, mivel a karakterlánc rövidebb, mint a lehetséges maximum, figyelmen kívül hagyjuk. Ezeket a bájtokat véletlenül sem értelmezhetjük az érték részeként, mivel az első bájt megmondja nekünk, hogy hol végződig a karakterlánc.

2. *Nullértékkel végződő lánc.* Ismét azzal kezdjük, hogy lefoglalunk egy $n + 1$ bájt méretű tömböt. Ezt a tömböt a lánc karaktereivel feltöltjük, és az utolsó karakter után egy *nullkaraktert* teszünk. Ez egy olyan karakter, mely nem megengedett a karakterláncokban. Akár az első módszernél, a tömb fel nem használt helyeit most sem lehet félreérteni, azaz nem lehet az érték részeként értelmezni, mivel most a lezáró nullérték figyelmeztet bennünket, hogy nem kell tovább nézni a sorozatot. Ezzel a `VARCHAR` reprezentációját kompatibilissé tettük a C nyelv karakterláncainak reprezentációjával.

3.3. példa: Tegyük fel, hogy az A attribútumot `VARCHAR(10)`-ként deklaráltuk. Ekkor lefoglalunk egy 11 karakteres tömböt minden relációsornak megfelelő rekordban az A értékének. Tegyük fel, hogy a `'sas'` láncot kell ábrázolni. Az első módszer alapján az első bájtba 3-at tennénk, ezzel ábrázolnánk a lánc hosszát, és a többi három karakter lenne maga a lánc. Az utolsó hét pozíció lényegtelen. Tehát az érték a következőként fog kinézni:

```
3sas
```

Jegyezzük meg, hogy a „3" egy 8 bites egész szám, azaz 00000011, és nem pedig a `'3'` karakter.

A második módszer alapján az első három pozíciót töltjük fel a lánccal, és a negyedik helyre egy nullkaraktert teszünk (melyre a töltelékkarakterhez hasonlóan a ⊥ jelet használjuk), és a maradék hét hely most is lényegtelen. Tehát a

```
sas⊥
```

fogja a `'sas'` láncot ábrázolni. □

Dátumok és időpontok

Egy dátumot rendszerint valamilyen speciális formátumot követő, rögzített hosszú karakterláncként ábrázolunk. Tehát egy dátumot ugyanúgy lehet ábrázolni, ahogy bármilyen más rögzített hosszú karakterláncot ábrázolnánk.

[3] Természetesen használhatnánk két- vagy több-bájtos sémát a hossz jelölésére.

A 2000. év problémája (Y2K)[4]

Sok adatbázisrendszerben és más alkalmazások programjaiban is a dátumokban szereplő évet két számjeggyel ábrázolták, például `EEHHNN` a reprezentáció formátuma. Mivel ezeknek az alkalmazásoknak eddig soha sem kellett foglalkozniuk mással, csak XX. századi dátumokkal, ezért a 19-et mindig odaértették az évszám elé, így az 1948. május 14. ábrázolása `'480514'`-ként történt.

Ezekkel az alkalmazásokkal az a probléma, hogy kihasználják azt a tényt, hogy ha a d_1 dátum korábbi, mint a d_2, akkor d_1 olyan karakterlánccal van ábrázolva, amely lexikografikusan (azaz ábécérendben) kisebb, mint az a karakterlánc, amely a d_2-t ábrázolja. Ezt az észrevételt figyelembe véve, ha azokat a filmszínészeket akarjuk megkeresni a 3.1. ábrán deklarált `Filmszínész` relációjában, akik 1998. június 1. előtt születtek, akkor ezt a következő lekérdezéssel tehetjük meg:

```
SELECT név FROM Filmszínész WHERE születési_idő < '980601'
```

Ha viszont az adatbázisba bekerülnek olyan gyerekszínészek, akik már a 3. évezredben születtek, akkor az ő születésük is kisebb lesz lexikografikusan a `'980601'`-nél, és akkor a fenti lekérdezés nem azt adja eredményül, amit ki akartunk fejezni vele. Például, ha egy színész 2001. augusztus 31-én született, akkor a születés mező értéke `'010831'`, ami lexikografikusan kisebb, mint `'980601'`. Ez ellen a probléma ellen csak úgy védekezhetünk (legalábbis a 10000. évig), hogy azokban a rendszerekben, amelyek dátumokat hasonlítanak össze, újra kódoljuk a dátumokat úgy, hogy négy számjegyet használunk az év ábrázolására az SQL2-szabványnak megfelelően.

3.4. példa: Például az SQL2-szabvány a dátumokat 10 hosszú karakterláncokkal ábrázolja, és ezeknek a formátuma `EEEE-HH-NN` (év-hó-nap). Tehát az első négy számjegy ábrázolja az évet, az ötödik egy kötőjel, a hatodik és hetedik számjegy a hónapot ábrázolja (az egyjegyű szám elé nullát írva), a nyolcadik jegy egy másik kötőjel, és végül elértünk az utolsó két számjegyhez, mely a napot ábrázolja (itt is nullát írunk az egyjegyű szám elé). Például az `'1948-05-14'` az 1948. május 14-ét ábrázolja. □

Hasonlóan az időpontokat is ábrázolhatjuk úgy, mintha karakterláncok lennének. Például az SQL2-szabvány az egész számú másodperceket egy `00:PP:MM` (óra:perc:másodperc) formátumú 8 karakteres láncként ábrázolja. Tehát az első két karakter az órát ábrázolja. Ennek értéke egy egész szám 0-tól 23-ig úgy, hogy az egyjegyű számokat egy 0 előzi meg. A délelőtt 7 órát tehát a 07, az esti 7 órát pedig a 19 számjegyekkel ábrázoljuk. A kettőspont utáni két számjegy a perceket ábrázolja,

[4] A Y2K rövidítést használták erre a jelenségre, ahol Y a Year – év, K a Kilo – ezer rövidítése. *A fordító megjegyzése.*

Több mező egyetlen bájtba csomagolása

Ha ki akarjuk használni, hogy a mezők logikai értékűek, vagy kis halmazhoz tartozó felsorolási típusúak, akkor esetleg megpróbálkozhatunk azzal, hogy több mezőt egyetlen bájtba csomagolunk. Például, ha három mezőnk lenne, az egyik logikai típusú, a másikban a hét valamelyik napját ábrázolnánk, a harmadikban négy szín közül ábrázolnánk valamelyiket, akkor az elsőhöz elég lenne egy bitet használnunk, a másodikhoz 3 bitet, a harmadikhoz két bitet, ezeket együtt tennénk be egyetlen bájtba, és még maradna is két bit a bájtban. Semmi akadálya, hogy így tegyünk, de ezzel hibázásra hajlamosabbak, és költségesebbek lettek azok a műveletek, amelyeket akkor kell elvégeznünk, ha ki akarunk olvasni egy értéket a bájtba csomagolt mezők valamelyikéből, vagy új értéket akarunk egy ilyen mezőbe írni. A mezőknek ilyen jellegű összecsomagolása sokkal fontosabb volt, mikor még a tárolóterület sokkal drágább volt. Közönséges helyzetekben ma már nem tanácsoljuk a fenti módszert.

ismét jön egy kettőspont, majd az utolsó két számjegy következik, melyek a másodperceket ábrázolják. A percek és másodpercek esetében is 0-t írunk az egyjegyű számok elé. Például `'20:19:02'` a „két másodperccel múlt 20 óra 19 perc" időpontot ábrázolja.

Egy ilyen időpontot könnyen ábrázolhatunk 8 hosszú, rögzített hosszú karakterláncként. Az SQL2-szabvány viszont megenged egy `TIME` (idő) típusú értéket is, ami a másodperc törtrészét is tartalmazza. A fent megadott 8 karakter mögé teszünk egy pontot, és annyi számjegyet, amennyi a másodperc törtrészének leírásához szükséges. Például a „két és egy negyed másodperccel 20 óra 19 perc után" időpontot SQL2-ben a `'20:19:02.25'` ábrázolja. Mivel ezek a karakterláncok tetszőleges hosszúak lehetnek, ezért két választásunk van:

1. A rendszer korlátozhatja az időpontok pontosságát, és így az időpontokat úgy lehet tárolni, mintha `VARCHAR(`*n*`)` típusúak lennének, ahol *n* az időponthoz rendelt legnagyobb hossz, azaz 9 plusz annyi, amennyi tizedesjegy megengedett a másodperc töredékének megadásában.
2. Az időpontokat a 3.4. részben megadott módon valóban változó hosszú értékekként tároljuk.

Bitek

Egy bitsorozat esetén (ami egy olyan adat, melyet az SQL2-ben `BIT(`*n*`)` típusként írunk le) minden 8 bitet egy bájtba pakolhatunk. Ha az *n* nem osztható 8-cal, akkor a legjobb az utolsó bájt fel nem használt bitjeitől eltekinteni. Például, a `010111110011` ábrázolható úgy, hogy `01011111` az első bájt, és `00110000` a második, ahol az utolsó

négy 0 egyik mezőnek sem része. Speciális esetként Bool-értéket (logikai értéket), azaz egyetlen bitet is ábrázolni tudunk úgy, hogy `10000000` jelenti az igaz, `00000000` a hamis értéket. Bár bizonyos környezetben könnyebb lehet ellenőrizni egy logikai értéket, ha minden bitben különbözik a két reprezentáció. Ekkor `11111111`-et használunk az igaz, `00000000`-t a hamis ábrázolására.

Felsorolási típusok

Néha hasznos, ha van egy olyan attribútumunk, mely egy kis, rögzített értékhalmazból veheti fel az értékeit. Ezeknek az értékeknek szimbolikus neveket adunk, és ha egy típus ezekből a nevekből áll, akkor ezt a típust *felsorolási* (enumerated) *típusnak* hívjuk. Szokásos példák felsorolási típusra a hét napjai, például `(HÉT, KED, SZE, CSÜ, PÉN, SZO, VAS)`, vagy színek halmaza, például `(PIROS, ZÖLD, KÉK, SÁRGA)`.

Egy felsorolási típus értékeit egész kódokkal ábrázolhatjuk, de csak annyi bájtot használunk, amennyi feltétlenül szükséges. Például a `PIROSat` ábrázolhatjuk a 0-val, a `ZÖLDet` 1-gyel, a `KÉKet` 2-vel, a `SÁRGÁt` 3-mal. Ezeket az egész számokat mind ábrázolhatjuk két bittel, nevezetesen a 00, 01, 10 és 11 bitpárokkal. Kényelmesebb azonban teljes bájtokat használni, ha kis halmazba tartozó egész számokat akarunk ábrázolni. Például a `SÁRGA` ábrázolható a 3 egész számmal, mely 8 bites bájton `00000011`. Az olyan felsorolási típusok, melyek legfeljebb 256 különböző értékből álló halmaznak felelnek meg, mind ábrázolhatók egyetlen bájttal. Ha a felsorolási típusnak legfeljebb 2^{16} értéke van, akkor egy kétbájtos rövid egész szám elegendő lesz és ez így folytatható tovább.

3.2. Rekordok

Azzal kezdjük, hogy megvizsgáljuk, hogyan lehet a mezőket rekordokba csoportosítani. A vizsgálatainkat a 3.4. részben folytatjuk, ahol majd a változó hosszú mezőket és rekordokat nézzük meg.

Általában egy adatbázisrendszer által használt minden rekordtípus rendelkezik egy *sémával*, és ezt a sémát az adatbázis tárolja. A séma tartalmazza a rekord mezőinek neveit, adattípusait és a mezők kezdetét a rekordon belül. Ha a rekord komponenseit kell elérnünk, akkor szükségünk lesz a sémára.

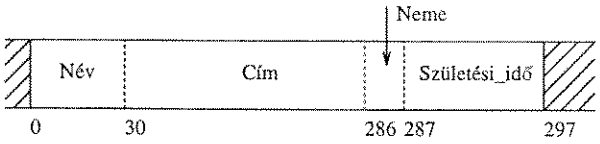
3.2.1. Rögzített hosszú rekordok építése

A relációsorokat rekordokkal ábrázoljuk. A rekordok olyan mezőket tartalmaznak, amilyenekkel a 3.1.3. részben foglalkoztunk. A legegyszerűbb eset, mikor a rekord összes mezője rögzített hosszú. Ekkor ezeket a mezőket összekapcsoljuk, és így állítjuk elő a rekordot.

3.5. példa: Vegyük a 3.1. ábrán szereplő `Filmszínész` reláció deklarációját. Ekkor négy mezőnk van:

1. `név`, mely egy 30 bájt hosszú karakterlánc.
2. `cím`, melynek típusa `VARCHAR(255)`. Ezt a mezőt a 3.3. példában tárgyalt séma felhasználásával 256 bájton ábrázolhatjuk.
3. `neme`, mely egyetlen bájt, amelyről feltesszük, hogy mindig vagy az `'F'` vagy a `'N'` karaktert tartalmazza, attól függően, hogy férfiról vagy nőről van szó.
4. `születési_idő`, mely `DATE`, azaz dátum típusú. Fel fogjuk tenni, hogy ezt a mezőt úgy ábrázoljuk, ahogy az SQL2 a dátumokat ábrázolja, azaz 10 bájton.

Tehát egy `Filmszínész` típusú rekordhoz 30 + 256 + 1 + 10 = 297 bájtra van szükség. Ezt mutatja a 3.3. ábra. Bejelöltük minden mező kezdetét, az *eltolási értéket* (offset), ami azzal a számmal egyezik meg, ahányadik bájton kezdődik a mező a rekord elejétől számolva. A `név` mező a 0. bájton kezdődik, a `cím` mező a 30. bájton, a `neme` a 286.-on, és a `születési_idő` a 287.-en. □



3.3. ábra. *Egy* `Filmszínész` *rekord*

Egyes számítógépek hatékonyabban tudják olvasni és írni azokat az adatokat, amelyek a központi memóriában speciális sorszámú bájton kezdődnek. Általában az szokott előnyös lenni, ha a cím négynek többszöröse (64 bites processzor esetén pedig a 8-nak többszöröse). Bizonyos típusú adatok, például egész számok esetén nagyon kívánatos, hogy 4-nek többszöröse legyen a kezdő cím, míg mások, például a dupla pontosságú valós számok esetén a kezdő cím inkább 8-nak kell hogy a többszöröse legyen.

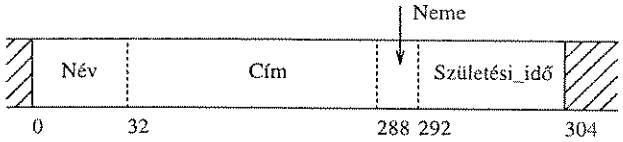
Igaz, hogy egy reláció sorait a lemezen tároljuk és nem a központi memóriában, de azért jó, ha tudunk erről. Ugyanis mikor egy blokkot a lemezről beolvasunk a központi memóriába, akkor a blokk első bájtja a központi memóriában biztosan egy olyan címen lesz, mely 4-nek többszöröse. Általában nem csak 4-nek, hanem 2 egy magasabb hatványának is többszöröse, például, ha a blokkok és lapok hossza $4096 = 2^{12}$, akkor a beolvasott blokkok első bájtjának címe 2^{12}-nek is többszöröse lesz. Így azt a követelményt, hogy bizonyos mezők úgy legyenek betöltve a központi memóriába, hogy az első bájtjuk címe a 4-nek vagy a 8-nak többszöröse legyen, úgy is fordíthatjuk, hogy ezeknek a mezőknek a blokkon belüli eltolási értéke legyen a 4, illetve a 8 az osztója.

Az egyszerűség kedvéért tegyük fel, hogy az adatokra vonatkozó egyetlen megkötésünk az, hogy a mezők a központi memóriában olyan címen kezdődnek, mely a 4-nek többszöröse. Ehhez elég, ha a következők teljesülnek:

a) minden rekord a blokkján belül olyan címen kezdődik, amely 4-nek többszöröse, és
b) a rekordon belül minden mezőnek a rekord elejétől mért eltolási értéke 4-nek valamilyen többszörösével megegyező bájt.

Másképpen elmondva, minden mezőnek és rekordnak a hosszát felkerekítjük a legközelebbi 4-gyel osztható számra.

3.6. példa: Tegyük fel, hogy a `Filmszínész` reláció sorait úgy kell ábrázolni, hogy minden mező 4-gyel osztható bájton kezdődjön. Ekkor a 4 mező eltolási értéke 0, 32, 288 és 292 lenne, és az egész rekord hossza 304 bájt lenne. A formátumot a 3.4. ábra mutatja.



3.4. ábra. *A* `Filmszínész` *relációsor beosztása, ha a mezőknek 4-gyel osztható bájton kell kezdődniük*

Például az első mező, a `név` 30 bájtos, de a második mezőt nem kezdhetjük el a következő 4-gyel osztható szám, azaz a 32 eltolási érték előtt. Tehát ebben a rekordformátumban a `cím` mezőnek 32 lesz az eltolási értéke. A második mező hossza 256 bájt, ami azt jelenti, hogy a `cím` utáni első rendelkezésre álló bájt a 288. A harmadik mező, a `neme`, csak egybájtos, de az utolsó mező csak 4 bájttal később kezdődhet, a 292. bájton. A negyedik mező, a `születési_idő`, 10 bájt hosszú, és így a mező a 301. bájton végződik, ezzel pedig a rekord hossza 302 lenne (ne feledjük, hogy az első bájt sorszáma 0.). Azonban, ha minden rekord minden mezőjének néggyel osztható bájton kell kezdődnie, akkor a 302. és 303. bájtok kihasználatlanul maradnak, így a rekord ténylegesen 304 bájtot használ el. Emiatt a 302. és 303. bájtot is a `születési_idő` mezőhöz fogjuk hozzárendelni, és így még véletlenül sem lehet őket más célra használni. □

3.2.2. Rekordfejlécek

Még egy fontos szempontot kell figyelembe venni, mikor a rekord formátumát akarjuk megtervezni. Gyakran a rekordban kell tárolnunk olyan információt is, amely egyik mezőnek sem az értéke. Például lehet, hogy a rekordban akarjuk tárolni a következőket:

1. A rekordsémát, vagy még valószínűbb, hogy csak egy mutatót arra a helyre, ahol az adatbázis-kezelő ehhez a rekordtípushoz tartozó sémát tárolja.
2. A rekord hosszát.

Miért szükséges a rekordséma?

Első ránézésre nem teljesen világos, hogy miért kell magában a rekordban is jelölnünk a rekordsémát, hiszen eddig csak rögzített formátumú rekordokat vizsgáltunk. Például a C-ben vagy hasonló nyelvekben programfutáskor egy „struct" mezői nem tárolják az eltolási értékeiket, hanem az eltolási értékeket a struktúrához hozzáférő alkalmazás programjába fordítjuk bele.

Annak viszont több oka is van, hogy miért kell a rekordsémát eltárolni, és az adatbázis-kezelő számára hozzáférhetővé tenni. Az egyik ok, hogy egy reláció sémája (és ennélfogva azoknak a rekordoknak a sémája, melyek a sorait ábrázolják) megváltozhat. A lekérdezéseknek az ezekhez a rekordokhoz tartozó aktuális sémát kell használniuk, ezért tudniuk kell, hogy mi is az aktuális séma. A másik ok, hogy léteznek olyan helyzetek, amikor nem tudjuk rögtön, egyszerűen megmondani, hogy mi a rekordtípus, ha csak a helyét ismerjük a tárolórendszerben. Például bizonyos tárolási szervezések esetén megengedett, hogy különböző relációk sorai a tárolónak ugyanabban a blokkjában legyenek.

3. Időbélyegzéseket, melyek azt mutatják, hogy a rekordot mikor módosították vagy olvasták utoljára.

De más információk tárolására is szükség lehet. Emiatt sok rekordformátum magában foglal néhány olyan bájtot is, ami ezeknek az információknak a tárolására szolgál. Ezek a bájtok jelentik a rekord *fejlécét* (header).

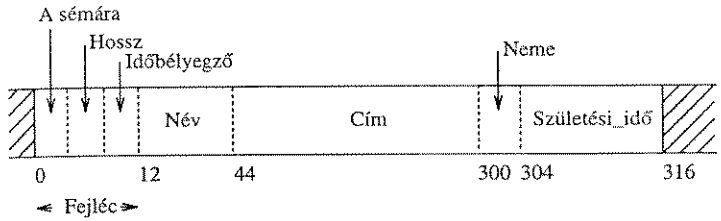
Az adatbázisrendszer tartja karban a *sémainformációt* (schema information). A sémainformáció lényegében az, amit a `CREATE TABLE` utasításban egy relációhoz megadunk:

1. A reláció attribútumai.
2. Az attribútumok típusai.
3. Az attribútumok sorrendje a soron belül.
4. Az attribútumokra és magára a relációra vonatkozó megszorítások, például az elsődleges kulcs deklarációja, vagy olyan kényszerfeltétel, hogy valamelyik egész attribútum csak egy bizonyos tartományból vehet fel értékeket.

Nem kell azonban minden információt egy rekord fejlécébe tennünk. Elég, ha egy mutatót helyezünk el, amely arra a helyre mutat, ahol a sor relációjára vonatkozó információt tároljuk.

Egy másik példa, hogy egy sor hosszát a sémájából ugyan ki tudjuk következtetni, de azért kényelmesebb lehet, ha ezt a hosszt magában a rekordban is tároljuk. Például lehet, hogy nem akarjuk a rekord tartalmát megvizsgálni, csak a következő rekord elejét szeretnénk gyorsan megtalálni. Ha a rekord hosszát kiolvashatjuk a rekordból, akkor elkerülhetjük, hogy a rekordsémát is be kelljen olvasni, mert ez egy lemez I/O-műveletbe is kerülhet.

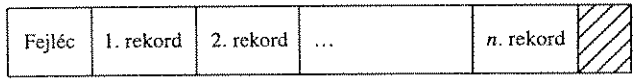
3.7. példa: Módosítsuk a 3.6. példában megadott rekordfelosztást úgy, hogy egy 12 bájtos fejlécet is tartalmazzon a rekord. Az első négy bájt a típus. Ez tulajdonképpen egy eltolási érték egy olyan területen, ahol az összes reláció sémáját tároljuk. A második a rekord hossza, egy 4 bájtos egész szám, és a harmadik egy időbélyegző, amely azt jelöli, hogy mikor szúrták be a rekordot vagy mikor módosították utoljára. Az időbélyegző is egy 4 bájtos egész szám. Az eredményül kapott felosztás a 3.5. ábrán látható. A rekord hossza most 316 bájt. □



3.5. ábra. *A* `Filmszínész` *reláció sorait ábrázoló rekordokat kiegészítjük valamilyen fejléc információval*

3.2.3. Rögzített hosszú rekordok blokkokba pakolása

Egy reláció sorait ábrázoló rekordokat a lemez blokkjaiban tároljuk. Ha el kell érnünk, vagy módosítanunk kell a rekordokat, akkor behozzuk őket a központi memóriába. A 3.6. ábrán látható egy rekordokat tartalmazó blokk felosztása.



3.6. ábra. *Rekordokat tartalmazó tipikus blokk*

Az opcionális *blokkfejléc* (block header) többek között az alábbi információkat is tartalmazhatja:

1. Hivatkozás (link) egy vagy több olyan blokkra, melyek egy blokkhálózat részét képezik. Ilyeneket a 4. fejezetben fogunk használni egy reláció soraihoz tartozó indexek készítésénél.
2. Információ arról, hogy ez a blokk milyen szerepet tölt be egy ilyen hálózatban.
3. Információ arról, hogy ennek a blokknak a rekordjai melyik relációhoz tartoznak.
4. Egy olyan „jegyzék" (directory), amely a blokk összes rekordjának az eltolási értékét tartalmazza.
5. Egy blokkazonosító; lásd a 3.3. részt.
6. Időbélyegzés(ek) jelölik a blokk utolsó módosítási és/vagy elérési idejét.

A legegyszerűbb eset, mikor a blokk egy reláció sorait tartalmazza, és a sorokhoz tartozó rekordok rögzített formátumúak. Ekkor a fejléc információját felhasználva annyi rekordot teszünk a blokkba, amennyit csak tudunk, és a megmaradó helyet felhasználatlanul hagyjuk.

3.8. példa: Tegyük fel, hogy a 3.7. példában továbbfejlesztett felosztással rendelkező rekordokat akarjuk tárolni. Ezek a rekordok 316 bájt hosszúak. Tegyük fel, hogy 4096 bájt méretű blokkokat használunk. Ebből 12 bájtot fogunk egy blokkfejlécre felhasználni, a maradék 4084 bájtot pedig az adatokhoz használjuk. Erre a területre az adott, 316 bájtos formátumú rekordból 12-t tudunk elhelyezni, és minden blokkban felhasználatlanul marad 292 bájt. □

3.2.4. Feladatok

* **3.2.1. feladat:** Tegyük fel, hogy egy rekord a következő mezőket tartalmazza, a megadott sorrendben: egy 15 hosszú karakterlánc, 2 bájtos egész szám, egy SQL2 dátumtípus, és egy SQL2 időtípus (tizedesvessző nélkül). Hány bájtot tesz ki egy rekord, ha:

a) A mezők tetszőleges bájton kezdődhetnek.
b) A mezőknek 4-gyel osztható bájton kell kezdődniük.
c) A mezőknek 8-cal osztható bájton kell kezdődniük.

3.2.2. feladat: Ismételjük meg a 3.2.1. feladatot a következő mezőlistával: egy 8 bájtos valós szám, egy 17 hosszú karakterlánc, egy egyedüli bájt és egy SQL2 dátumtípus.

* **3.2.3. feladat:** Tegyük fel, hogy a mezők ugyanazok, mint a 3.2.1. feladatban, de a rekordoknak fejlécük is van, mely egy 4 bájtos mutatót és egy karaktert tartalmaz. Számoljuk ki a rekord hosszát a mezők elhelyezésére vonatkozó azon feltételek mellett, amelyeket a 3.2.1. feladatban az a), b) és c) pontokban adtunk meg.

3.2.4. feladat: Ismételjük meg a 3.2.2. feladatot, ha a rekordok egy fejléccel is rendelkeznek, amely egy 8 bájtos mutatót és tíz 2 bájtos egész számot tartalmaz.

* **3.2.5. feladat:** Tegyük fel, hogy a rekordok olyanok, mint a 3.2.3. feladatban. A blokkok mérete 4096 bájt. A blokkfejléc tíz 4 bájtos egész számot tartalmaz. A blokkokba annyi rekordot akarunk elhelyezni, amennyit csak lehet. Hány rekordot tudunk egy blokkba tenni, a 3.2.1. feladatban megadott, mezőelhelyezésre vonatkozó a), b) és c) feltételek esetén?

3.2.6. feladat: Ismételjük meg a 3.2.5. feladatot a 3.2.4. feladat rekordjaival. Tegyük fel, hogy a blokkok 16 384 bájt hosszúak. A blokkfejlécek három 4 bájtos egész számot és egy olyan jegyzéket tartalmaznak, amelyben a blokk minden rekordjához egy 2 bájtos egész tartozik.

3.3. Blokkcímek és rekordcímek ábrázolása

Mielőtt még továbbmennénk, és azt vizsgálnánk, hogyan lehet bonyolultabb szerkezetű rekordokat ábrázolni, előtte meg kell néznünk, hogyan lehet címeket, mutatókat vagy rekord- és blokkhivatkozásokat ábrázolni. Ugyanis ilyen és ehhez hasonló mutatók gyakorta részét képezik a bonyolult rekordoknak. Más okokból is célszerű, ha megismerjük a másodlagos tárolók címreprezentációját. A 4. fejezetben fogjuk majd megnézni, hogy milyen hatékony struktúrákkal lehet a fájlokat vagy relációkat ábrázolni, és ekkor majd több fontos alkalmazást is látni fogunk a rekordcímek vagy blokkcímek felhasználására.

Miután egy blokkot betöltöttünk a központi memóriának egy pufferébe, azután a blokk első bájtjának virtuális memóriacímét tekinthetjük a blokk címének. Egy blokkon belüli rekord címének pedig a rekord első bájtjának virtuális memóriacímét tekinthetjük. Ezzel szemben a másodlagos tárolón a blokk nem része az alkalmazáshoz tartozó virtuális memória címterületének, hanem az adatbázisrendszer által elérhető adatok teljes rendszerén belül bájtok sorozata írja le a blokk helyét. Ez a sorozat a következőkből állhat: a lemezhez tartozó eszközazonosító, a cilinderszám stb. Egy rekordot úgy lehet azonosítani, hogy megadjuk a blokkját és a rekord első bájtjának a blokkon belüli eltolási értékét.

Hogy még bonyolultabb legyen a helyzet a címek ábrázoláskor, azt is elmondjuk, hogy újabban megfigyelhető egy törekvés az úgynevezett „objektumbrókerek" irányába, amelyek azt is megengedik, hogy több, együttműködő rendszer egymástól függetlenül készíthessen objektumokat. Ezek az objektumok rekordokkal ábrázolhatók. A rekordok ugyan egy objektumorientált adatbázis-kezelő rendszer részét alkotják, de úgy is gondolhatunk rájuk mint relációk soraira anélkül, hogy az alapvető elképzelésünket feladnánk. Mindazonáltal az a képesség, hogy függetlenül lehessen objektumokat vagy rekordokat készíteni, nagyon kényessé teszi azokat a mechanizmusokat, amelyek ezeknek a rekordoknak a címeit tartják karban.

Ebben a részben először a címterület vizsgálatát kezdjük el. Ez azért is fontos, mert ez kapcsolatban van az adatbázis-kezelők szokásos „kliens-szerver" felépítésével. Ezután megnézzük, milyen lehetőségeink vannak a címek ábrázolása, és végül foglalkozunk a mutatók helyreigazításával (pointer swizzling), amely egy módszer arra, hogy hogyan lehet az adatszerver világába tartozó címeket átalakítani a kliensalkalmazási programok világába tartozó címekre.

3.3.1. Kliens-szerver rendszerek

Rendszerint egy adatbázis tartalmaz egy *szerver* (server) folyamatot. Ez gondoskodik arról, hogy az adatok eljussanak a másodlagos tárolóról egy vagy több *kliens* (client) folyamathoz. A kliens folyamatok olyan alkalmazások, melyek adatokat használnak. A szerver és a kliens folyamatok lehetnek egy gépen is, vagy az is lehet, hogy a szervert és a különböző klienseket szétosztottuk sok gép között.

A kliensalkalmazások hagyományos, „virtuális" címterületet használnak, mely ti-

pikusan 32 bites, azaz körülbelül 4 milliárd különböző címet lehet megadni. Az operációs rendszer vagy az adatbázis-kezelő rendszer dönti el, hogy a címterületnek melyik része legyen aktuálisan a központi memóriában, és a hardver képezi le a virtuális címterületet a központi memória fizikai helyeire. Ettől kezdve nem foglalkozunk ezzel a virtuálisról fizikai címre fordítással, hanem a kliens címterületre úgy fogunk gondolni, mintha már magában a központi memóriában lenne.

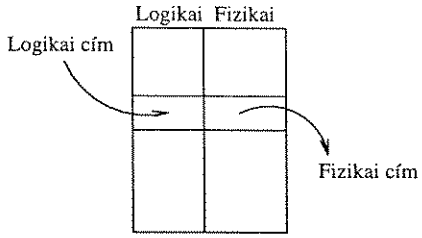
A szerver adatai az *adatbázis címterületén* (database address space) léteznek. Ennek a területnek a címei blokkokra vagy esetleg blokkon belüli eltolási értékekre vonatkoznak. Több mód is kínálkozik arra, ahogy ennek a címterületnek a címeit ábrázoljuk:

1. *Fizikai címek.* Ezek olyan bájtokból álló láncok, melyek segítségével meg lehet határozni, hogy a másodlagos tárolórendszeren hol lehet megtalálni a blokk vagy a rekord helyét. A fizikai címnek egy vagy több bájtja használatos arra, hogy megadja az összes alábbi információt:

   a) Melyik géphez (host) tartozik a tároló (abban az esetben, ha az adatbázist egynél több gépen tároljuk).
   b) Mi annak a lemeznek vagy más eszköznek az azonosítója, amelyen a blokk elhelyezkedik.
   c) A lemez cilinderének sorszáma.
   d) A cilinderen belül a sáv sorszáma (ha a lemeznek egynél több felülete van).
   e) A blokk sorszáma a sávon belül.
   f) (Bizonyos esetekben) a rekord kezdetének blokkon belüli eltolási értéke.

2. *Logikai címek.* Minden egyes blokknak vagy rekordnak van egy „logikai címe". A logikai cím tetszőleges rögzített hosszú bájtlánc lehet. A lemezen egy ismert helyen tárolják a *leképezési táblát* (map table), amely a 3.7. ábrán látható módon rendeli össze a logikai és fizikai címeket.

Vegyük észre, hogy a fizikai címek hosszúak. Nyolc bájtra minimum szükség van ahhoz, hogy a fenti listában felsorolt minden elemet tartalmazza, de egyes rendszerekben akár 16 bájtosak is lehetnek a fizikai címek. Például képzeljünk el egy objek-



3.7. ábra. *Egy leképezési tábla fordítja le a logikai címeket fizikai címekre*

tumokat tartalmazó adatbázist, amelyet úgy terveztek, hogy 100 évig létezzen. A jövőben ez az adatbázis megnőhet úgy, hogy egymillió gép tartozik majd hozzá, és tegyük fel, hogy minden gép elég gyors ahhoz, hogy egy objektumot készítsen minden nanoszekundumban[5]. Ez a rendszer körülbelül 2^{77} objektumot készítene el, melyek címeinek az ábrázolásához minimum tíz bájtra lenne szükség. Mivel valószínűleg jobban szeretnénk külön bájtokat lefoglalni a gépek, tárolóegységek stb. ábrázolására, így a címek jelölésére valószínűleg még 10-nél is sokkal több bájtot használnánk egy ekkora rendszer esetében.

3.3.2. Logikai és strukturált címek

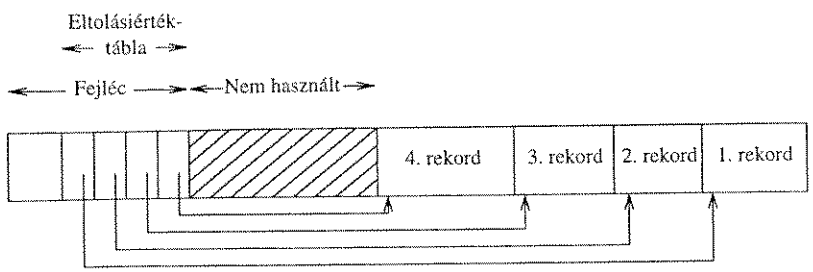
Most már biztos kíváncsiak vagyunk, hogy mire is kellhetnek a logikai címek. Minden olyan információ, ami szükséges egy fizikai címhez, a leképezési táblában található. Így ahhoz, hogy a logikai mutatókat követve eljussunk a rekordokhoz, először meg kell vizsgálnunk a leképezési táblát, és az innen kiolvasott információ segítségével tudunk elmenni a fizikai címre. Bár ez kissé körülményesnek tűnik, mégis a leképezési táblának ez az indirektségi szintje nagyfokú rugalmasságot tesz lehetővé. Például számos adatszervezési módszer esetén arra kényszerülünk, hogy a rekordokat ide-oda mozgassuk, vagy egy blokkon belül, vagy egyik blokkból egy másikba. Ha egy leképezési táblát használunk, akkor a rekordra mutató összes mutató erre a leképezési táblára hivatkozik, így ha elmozdítjuk vagy töröljük a rekordot, akkor csak annyi a teendő, hogy ennek a rekordnak a bejegyzését megváltoztatjuk a táblában.

A logikai és fizikai címeknek számos olyan kombinációja is lehetséges, amely *strukturált* címsémát ad meg. Például megtehetjük, hogy a szóban forgó rekordhoz megadjuk annak a blokknak a fizikai címét, amelyben a rekord szerepel, de nem a blokkon belüli eltolási értéket adjuk meg, hanem ehelyett a blokk fizikai címéhez hozzátesszük a rekord kulcsértékét. Ezután, ha ilyen strukturált című rekordot akarunk megtalálni, akkor a cím fizikai részét használva eljutunk ahhoz a blokkhoz, amely a rekordot tartalmazza, és megvizsgáljuk a blokk rekordjait, hogy megtaláljuk a megfelelő kulccsal rendelkező rekordot.

Persze a blokk rekordjainak végignézéséhez elég információval kell rendelkeznünk, hogy megtaláljuk őket. Az a legegyszerűbb eset, mikor a rekordok rögzített hosszúak, ismerjük a rekordok hosszát, és azt is tudjuk, hogy a rekordon belül milyen eltolási értéken kezdődik a kulcsmező. Ekkor az a teendő, hogy a blokk fejlécében meg kell találnunk azt a számlálót, amely azt mutatja, hogy mennyi rekord van a blokkban, és azt is pontosan tudjuk, hogy hol vannak a kulcsmezők, amelyek a megadott cím kulcs típusú részével megegyezhetnek. Az is igaz, hogy sokféle módon lehetne a blokkokat szervezni ahhoz, hogy a blokk rekordjait végignézhessük, mindjárt áttekintjük a többi lehetőséget is.

A fizikai és logikai címeknek egy nagyon hasznos, hasonló kombinációja az, amikor minden blokkban tárolunk egy *eltolásiérték-táblát* (offset table), amely a 3.8. áb-

[5] A nanoszekundum a másodperc egymilliárdnyi része. *A fordító megjegyzése.*



3.8. ábra. *Egy blokk, melyben egy eltolásiérték-tábla mondja meg minden rekordnak a blokkon belüli helyét*

rán látható módon a rekordoknak a blokkon belüli eltolási értékeit tartalmazza. Figyeljük meg, hogy ez a tábla a blokk elejétől a blokk vége felé nő, míg a rekordok a blokk végénél kezdődnek. Ez a stratégia akkor hasznos, ha a rekordok nem szükségképpen egyforma hosszúak. Ebben az esetben nem tudjuk előre, hogy mennyi rekordot fog a blokk tartalmazni, de nem is kell kezdetben rögzített nagyságú blokkfejlécet lefoglalni.

Egy rekord címe most is két részből áll, a rekord blokkjának a fizikai címéből és egy eltolási értékből. Ez utóbbi a rekordhoz tartozó bejegyzésnek az eltolási értéke a rekord blokkjának eltolásiérték-táblájában. A blokkon belüli indirektségi szint a logikai címek számos előnyét nyújtja továbbra is anélkül, hogy globális leképezési táblára lenne szükség.

- A rekordot ide-oda mozgathatjuk a blokkon belül, csak annyit kell tennünk, hogy módosítani kell a rekord bejegyzését az eltolásiérték-táblában; a rekordra hivatkozó mutatók alapján továbbra is meg tudjuk majd találni a rekordot.

A memória-címterület tulajdonjoga

Ebben a részben a másodlagos és a központi memória közti átvitelt a következő nézőpontból vizsgáltuk: minden egyes kliens saját memória-címterülettel rendelkezik, de az adatbázis-címterület közös. Az objektumorientált adatbáziskezelő rendszerekben ez a szokásos modell. Ezzel szemben a relációs rendszerek gyakran a memória-címterületet is megosztva kezelik. Emögött a helyreállíthatóság és a konkurencia támogatása húzódik meg, ahogy ezt majd a 8. és 9. fejezetekben látni fogjuk.

Egy használható kompromisszum, ha a szerver oldalán megosztjuk a memória-címterületet, és emellett legyen a kliensek oldalán is másolat a címterület részeiből. Ezzel a szervezéssel is támogatjuk a helyreállíthatóságot és a konkurenciát, de azt is lehetővé tesszük, hogy a feldolgozást „skálázható" módon megoszthassuk: minél több kliens van, annál több processzort működtethetünk.

- Még azt is megtehetjük, hogy a rekordot egy másik blokkba tesszük át. Ehhez csak az kell, hogy az eltolásiérték-tábla bejegyzései elég nagyok legyenek ahhoz, hogy tartalmazzanak a rekordhoz egy „következő címet" is.
- Végezetül megvan az a lehetőségünk is, hogy ha egy rekordot törlünk, akkor az eltolásiérték-táblában meghagyunk egy *sírkő* (tombstone) bejegyzést, ami egy speciális érték, és azt jelöli, hogy a rekordot törölték. A rekord törlése előtt lehet, hogy az adatbázisban több különböző helyen is tároltunk mutatókat erre a rekordra. A rekord törlése után egy ilyen mutatót követve eljutunk a sírkőhöz, emiatt a mutatót lecserélhetjük egy nullmutatóra (null pointer), vagy különben az adatstruktúrát kell módosítani, hogy tükrözze a rekord törlését. Ha nem hagytunk volna sírkövet a táblában, akkor megtörténhetne, hogy a mutató alapján valamilyen új rekordhoz jutnánk, ami meglepő és hibás eredményre vezetne.

3.3.3. Mutatók helyreigazítása

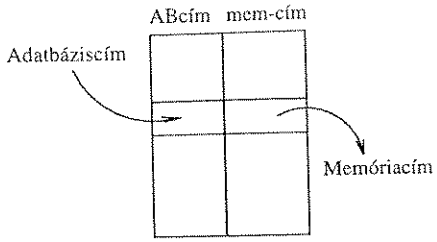
A mutatók és címek gyakran a rekordoknak részét képezik. Ez a helyzet általában nem azokra a rekordokra jellemző, melyek egy reláció sorait ábrázolják, hanem azokra a sorokra, amelyek objektumokat ábrázolnak. A modern objektumrelációs adatbázisrendszerekben megengedett a mutató típusú attribútumok használata is, melyeket hivatkozásoknak hívunk, így még relációs rendszerek esetén is szükség van arra, hogy ábrázolni tudjuk a sorokban elhelyezett mutatókat. Végezetül megemlítjük, hogy az indexstruktúrák blokkokból épülnek fel, és a blokkokban rendszerint mutatókat is találunk. Tehát szükséges tanulmányozni a mutatók kezelését, mikor a blokkokat mozgatjuk a központi memória és a másodlagos memória között. Ezt fogjuk megtenni ebben a részben.

Ahogy már korábban is említettük, minden blokk, rekord, objektum vagy más olyan adat, amire hivatkozni lehet, kétféle címmel rendelkezhet:

1. Az egyik címet *adatbáziscímnek* (database address) fogjuk hívni. Ez egy tipikusan 8 (esetleg másmilyen) hosszú bájtsorozat. Ez a cím a szerver adatbázisának címterületén van, és az adattétel helyét mutatja a rendszer másodlagos tárolóján.
2. Ha az adattételt jelenleg a virtuális memóriába pufferelték, akkor van egy címe a virtuális memóriában is. Ezek a címek tipikusan négybájtosak. Az ilyen címeket az adattétel *memóriacímének* (memory address) hívjuk.

Ha egy adattétel csak a másodlagos tárolón található, akkor biztosan az adattétel adatbáziscímét kell használnunk. Ezzel szemben, ha az adattétel a központi memóriában van, akkor hivatkozhatunk rá a memóriacímével vagy az adatbáziscímével is. Hatékonyabb, ha memóriacímet teszünk mindenhová, ahol az adattételben egy mutató szerepel, mivel ezeket a mutatókat egyszerű gépi utasítások segítségével lehet követni.

Az adatbáziscímek követése, éppen ellenkezőleg, sokkal időigényesebb. Kell egy tábla, amely a virtuális memóriában aktuálisan megtalálható adatbáziscímeket le tudja fordítani az aktuális memóriacímükre. Egy ilyen *fordítási táblát* (translation table) mutat be a 3.9. ábra. Ez emlékeztethet bennünket a 3.7. ábra leképezési táblájára,



3.9. ábra. *A fordítási tábla az adatbáziscímeket alakítja át velük ekvivalens memóriacímekké*

amely a logikai és fizikai címek közti fordítást mutatta be. A különbségek azonban a következők:

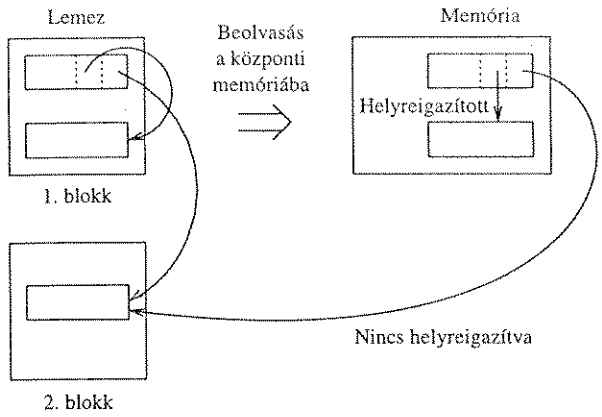
a) A logikai és a fizikai címek egyaránt az adatbáziscím reprezentációi. Ezzel szemben a fordítási táblában található memóriacímek a memóriába másolt megfelelő objektumra vonatkoznak.
b) Az adatbázis minden címezhető adattételéhez tartozik bejegyzés a leképezési táblában, míg a fordítási táblában csak azok az adattételek szerepelnek, amelyek jelenleg a memóriában találhatók.

Az adatbáziscímek ismételt lefordítása memóriacímekre nyilván költséges tevékenység. Ennek elkerülésére több technikát is kifejlesztettek, melyeket együttesen *mutatók helyreigazításának* (pointer swizzling) hívunk. Az az alapötlet, hogy amikor egy blokkot a másodlagos memóriából a központi memóriába olvasunk be, akkor a blokkon belüli mutatókat helyre lehet igazítani, vagyis az adatbázis címterületére hivatkozó mutatókat le lehet fordítani virtuális címterületre vonatkozó mutatókra. Tehát egy mutató végül is a következőkből áll:

1. Egy bit jelzi, hogy a mutató jelenleg egy adatbáziscím vagy egy (helyreigazított) memóriacím.
2. Egy alkalmas adatbázis vagy memóriamutató. Ugyanazt a helyet használjuk erre a célra, függetlenül attól, hogy az adott pillanatban melyik címformátum szerepel. Természetesen a memóriacím esetén nem használjuk fel az összes helyet, mivel a memóriacím tipikusan rövidebb, mint egy adatbáziscím.

3.9. példa: A 3.10. ábra egy egyszerű helyzetet mutat be. Az 1. blokkban van egy rekord, aminek van egy mutatója a blokkban szereplő második rekordra, és van még egy mutatója, amely egy másik blokkban szereplő rekordra mutat. Az ábrán az is látszik, hogy mi történhet, mikor az 1. blokkot bemásoljuk a memóriába. Az első mutató, mely az 1. blokkon belülre mutat, helyreigazítható, és így közvetlenül a megcélzott rekord memóriacímére fog mutatni.

Ezzel szemben, ha a 2. blokk pillanatnyilag nincs a memóriában, akkor a második mutatót nem tudjuk helyreigazítani, így helyreigazítatlanul marad, azaz a célba vett



3.10. ábra. *Egy mutató struktúrája abban az esetben, mikor elvégezhető a helyreigazítás*

rekord adatbáziscímére fog mutatni. Ha később behozzuk a 2. blokkot a memóriába, akkor elméletileg lehetővé válik, hogy az 1. blokk második mutatóját is helyreigazítsuk. A helyreigazító stratégiáktól függően lehet, hogy létezik egy olyan lista, amelyen azok a memóriában lévő mutatók szerepelnek, melyek a 2. blokkra hivatkoznak. Ha van ilyen lista, akkor lehetőségünk van a mutatók helyreigazítására.

Számos stratégia használható a helyreigazító mutató meghatározására. □

Automatikus helyreigazítás

Amint egy blokkot behozunk a központi memóriába, rögtön megkeressük az összes mutatóit és címeit, és bejegyezzük őket a fordítási táblába, ha még nem szerepelnek benne. Ezek a mutatók tartalmazzák azokat a mutatókat, amelyek a blokk rekordjai*ból* mutatnak valahová, és azokat is, amelyek magának a blokknak és/vagy a rekordjainak a címei, ha ezek egyáltalán címezhető adattételek. Szükségünk van egy olyan mechanizmusra, amely megkeresi a mutatókat a blokkon belül. Például:

1. Ha a blokk ismert sémájú rekordokat tartalmaz, akkor a séma megmondja, hogy a rekordban hol találhatók mutatók.
2. Ha a blokkot valamilyen indexstruktúrához használjuk, akkor a blokk mutatóinak elhelyezkedése ismert. Erről a 4. fejezetben lesz majd szó.
3. A blokk fejlécében tárolhatunk egy listát, amely megmondja, hogy hol vannak a mutatók.

Amikor a fordítási táblának megadjuk a központi memóriába éppen beolvasott blokknak és/vagy a rekordjainak a címeit, akkor pontosan tudjuk, hogy a memóriában a blokkot hová puffereltük. Tehát a fordítási tábla számára egyből elkészíthetjük az

ezekhez az adatbáziscímekhez tartozó bejegyzést. Mikor egy ilyen *A* adatbáziscímet akarunk beszúrni a fordítási táblába, akkor megtörténhet, hogy már megtaláljuk őt a táblában, mivel a blokkja jelenleg a memóriában van. Ebben az esetben a memóriába most beolvasott blokkban az *A*-t kicseréljük a megfelelő memóriacímre, és a „helyreigazított" bitnek igaz értéket adunk. Másrészt, ha az *A* még nem szerepel a fordítási táblában, akkor a blokkját sem olvastuk még be a memóriába, ezért aztán ezt a mutatót nem lehet helyreigazítani, hanem meghagyjuk a blokkban adatbázis-mutatónak.

Ha megpróbáljuk követni egy blokk *P* mutatóját, és a *P* még nincs helyreigazítva, vagyis még adatbázis-mutatóként szerepel, akkor meg kell győződnünk arról, hogy az a *B* blokk, amely azt az adattételt tartalmazza, amire a *P* mutat, a memóriában van (különben miért követnénk ezt a mutatót). A fordítási táblában megnézzük, hogy a *P* adatbáziscímnek szerepel-e a memóriára vonatkozó megfelelője. Ha nem, akkor bemásoljuk a *B* blokkot egy memóriapufferbe. Amint a *B* bekerült a memóriába, a *P* mutatót helyre tudjuk igazítani azáltal, hogy a *P* adatbázis formátumú címét az ekvivalens memória formátumú címre cseréljük ki.

Igény szerinti helyreigazítás

Egy másik lehetséges megközelítés, hogy amikor először hozzuk be a blokkot a memóriába, akkor még egyik mutatót sem igazítjuk helyre. A fordítási tábla számára megadjuk a blokknak és a mutatóinak a címét, valamint a nekik megfelelő memóriatípusú ekvivalens párjukat. Ha a memória valamelyik blokkjában szereplő mutatót akarjuk követni, csak akkor igazítjuk helyre a mutatót. Ehhez ugyanazt a stratégiát követjük, amit az automatikus helyreigazításnál használtunk, mikor egy helyreigazítatlan mutatót találtunk.

Az igény szerinti és az automatikus helyreigazítás között az a különbség, hogy az utóbbi esetben azonnal az összes mutatót megpróbáljuk gyorsan és hatékonyan helyreigazítani, amint a blokkot betöltjük a memóriába. Mérlegelnünk kell azt, hogy ugyan lehet, hogy időt takarítunk meg azzal, hogy egy blokk összes mutatóját egyszerre helyreigazítjuk, de az is lehet, hogy egyes helyreigazított mutatókat sohasem fogunk használni. Ebben az esetben kárba fog veszni minden olyan idő, amit egy ilyen mutató helyreigazításával vagy a helyreigazítás visszaalakításával töltöttünk.

Érdekes lehetőséget teremt, ha úgy intézzük, hogy minden adatbázis-mutató érvénytelen memóriacímként nézzen ki. Ebben az esetben rábízhatjuk a számítógépre, hogy bármelyik mutatót nyugodtan kövesse, mintha az memóriacím lenne. Ha történetesen a mutató nincs helyreigazítva, akkor a memóriahivatkozás egy hardvercsapda típusú eseményt fog előidézni. Ha az adatbázis-kezelő rendszer rendelkezik egy olyan függvénnyel, amelyet az ilyen csapda bekövetkezése hív meg, akkor ez a függvény helyreigazíthatja a mutatót a fent leírt módon, és azután már követheti egy-egy utasítással a helyreigazított mutatókat. Vegyük észre, hogy csak akkor kell időigényesebb dolgot végeznünk, mikor egy olyan mutatót követünk, amely nincs helyreigazítva.

Nincs helyreigazítás

Természetesen az is lehetséges, hogy sohasem igazítjuk helyre a mutatókat. Ehhez továbbra is kell a fordítási tábla, és akkor a mutatókat a helyreigazítatlan alakjukban is követhetjük. Ez a megközelítés két előnyt is nyújt. Az egyik, hogy a rekordokat a memóriában nem lehet feltűzni (pinned), ahogy azt majd a 3.3.5. részben tárgyalni fogjuk. A másik, hogy nem kell döntenünk arról, hogy melyik formájukban adjuk meg a mutatókat.

Programozó által vezérelt helyreigazítás

Bizonyos alkalmazások esetén az alkalmazás programozója lehet, hogy előre tudja, hogy egy blokkon belül a mutatókat valószínűleg majd követni kell. Ez a programozó explicite meghatározhatja, hogy egy memóriába betöltött blokk mutatói helyreigazítottak legyenek, vagy csak akkor kéri egy mutató helyreigazítását, amikor szükséges. Például, ha egy programozó tudja, hogy egy blokkhoz valószínűleg nagyon sokszor kell hozzáférni, ilyen lehet egy B-fa gyökérblokkja (amelyről a 4.3. részben olvashatunk), akkor a mutatókat helyre fogja igazítani. Ezzel szemben azokat a memóriába töltött blokkokat, amelyeket csak egyszer használunk, és aztán nagy valószínűséggel kidobunk a memóriából, nem fogja helyreigazítani.

3.3.4. Blokkok visszaírása a lemezre

Amikor egy blokkot a memóriából visszahelyezünk a lemezre, akkor a blokkon belül minden mutatót vissza kell állítani a helyreigazítás előtti állapotára, azaz, a blokk memóriacímeit le kell cserélni a megfelelő adatbáziscímekre. A fordítási tábla segítségével lehet a kétféle típusú címek közti megfeleltetést elvégezni mindkét irányban, így elvben meg lehet találni egy adott memóriacímhez a hozzárendelt adatbáziscímet.

Ezzel szemben nem akarjuk, hogy minden visszaállító művelet esetén a teljes fordítási táblát végig kelljen nézni a kereséshez. Noha nem beszéltünk eddig ennek a táblának a megvalósításáról, de azért azt képzelhetjük, hogy a 3.9. ábra táblája megfelelő indexekkel rendelkezik. Ha a fordítási táblára úgy gondolunk mint egy relációra, akkor azt a problémát, hogy az *x* adatbáziscímhez keressük a hozzá tartozó memóriacímet, kifejezhetjük az alábbi lekérdezéssel:

```
SELECT memCím
FROM FordításiTábla
WHERE abCím = x;
```

Például egy olyan tördelőtábla (hash-tábla), ahol a kulcs az adatbáziscím, alkalmas index lehetne az `abCím` attribútumra. A 4. fejezetben számos lehetséges adatstruktúrát fogunk majd javasolni.

Ha az ellentétes lekérdezést is támogatni akarjuk,

```
SELECT abCím
FROM FordításiTábla
WHERE memCím = y;
```

akkor a `memCím` attribútumra is kell egy index. A 4. fejezetben mutatunk majd olyan adatstruktúrákat, melyek alkalmasak ilyen indexnek. A 3.3.5. részben szó lesz majd a láncolt lista struktúráról is, ami bizonyos körülmények között arra is használható, hogy egy memóriacímből eljussunk az összes olyan központimemória-mutatóhoz, amely erre a címre mutat.

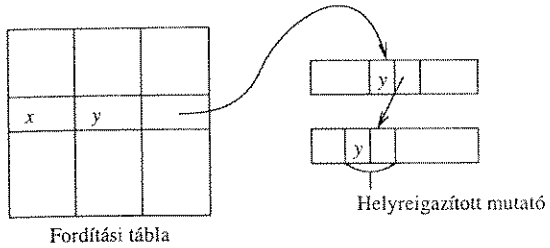
3.3.5. Feltűzött rekordok és blokkok

A memóriában egy blokkot *feltűzöttnek* (pinned) mondunk, ha pillanatnyilag nem lehet biztonságban visszaírni a lemezre. A blokk fejlécében el lehet helyezni egy bitet, amely azt mondja meg, hogy a blokk feltűzött-e vagy sem. Több ok is lehet arra, hogy egy blokk miért van feltűzve. Ezek között az okok között szerepelnek a helyreállító rendszerek követelményei is, ahogy ezt majd a 8. fejezetben látni fogjuk. A mutatók helyreigazítása is ad egy fontos indokot arra, hogy miért kell bizonyos blokkokat feltűzni.

Ha egy B_1 blokkon belül van egy helyreigazított mutató, mely a B_2 blokk valamelyik adatelemére mutat, akkor nagyon körültekintőnek kell lennünk, mikor a B_2 blokkot visszahelyezzük a lemezre, és újból fel akarjuk használni a számára a központi memóriában lefoglalt puffert. Amiatt kell vigyáznunk, hogy ha a B_1 blokk fenti mutatóját követnénk, akkor ez egy olyan pufferhez vezetne el bennünket, amely már nem tartalmazza a B_2 blokkot; végeredményben a mutató vége nem a kívánt helyre mutat. Az ilyen mutatóra azt mondjuk, hogy szabadon lógó mutatóvá[6] (dangling) vált. Emiatt egy olyan blokk, mint a B_2, vagyis amelyre valahonnan máshonnan egy helyreigazított mutató hivatkozik, feltűzött blokknak számít.

Amikor visszaírunk egy blokkot a lemezre, akkor nem csak az a teendőnk, hogy a blokkban szereplő összes helyreigazított mutatót visszaállítsuk, hanem arról is meg kell győződnünk, hogy a blokk nincs-e feltűzve. Ha fel van tűzve, akkor két dolgot tehetünk: vagy meg kell szüntetnünk a feltűzöttséget, vagy megengedjük, hogy a blokk a memóriában maradjon, és ezzel egy olyan területet foglaljon el, amit különben valamelyik másik blokk használhatna. Ha egy blokk azért feltűzött, mert kívülről helyreigazított mutatók hivatkoznak rá, akkor a feltűzöttség megszüntetése azt jelenti, hogy minden ilyen rámutató mutató helyreigazítását vissza kell állítani. Következésképpen a fordítási táblában a helyreigazított mutatókat is fel kell jegyezni, azaz minden egyes adatbáziscímre, melyhez tartozó adattétel előfordul a memóriában, és erre az adattételre helyreigazított mutatók hivatkoznak, fel kell jegyezni ezeknek a mutatóknak a memóriában elfoglalt helyét. Erre két lehetséges eljárást adunk:

[6] Szokásos még a fityegő, himbálódzó mutató elnevezés is. *A fordító megjegyzése.*



3.11. ábra. *Egy helyreigazított mutató előfordulásainak láncolt listája*

1. Készítsünk egy listát, amely az egy memóriacímre történő összes hivatkozást tartalmazza, és ezt a láncolt listát csatoljuk hozzá a fordítási táblában az ennek az adatbáziscímnek megfelelő bejegyzéshez.
2. Ha a memóriacímek jelentősen rövidebbek, mint az adatbáziscímek, akkor a láncolt listát azokon a területeken készíthetjük el, amelyeket magukhoz a mutatókhoz használunk fel. Vagyis minden adatbáziscímhez használt területet lecserélünk

   a) egy neki megfelelő helyreigazított mutatóra, és
   b) egy másik mutatóra, amely jellegét tekintve a helyreigazított mutató összes előfordulásából képzett láncolt lista struktúra mutatója.

A 3.11. ábra azt mutatja be, hogyan lehet egy y memória mutató összes előfordulását összeláncolni, kezdve a fordítási táblának annál a bejegyzésénél, ahol az x adatbáziscím, és a neki megfelelő y memóriacím szerepel.

3.3.6. Feladatok

* **3.3.1. feladat:** Hány bájtra van szükség ahhoz, hogy a *Megatron 747* lemez fizikai címeit ábrázolhassuk, ha külön bájtot vagy bájtokat akarunk lefoglalni a cilinderekre, a cilinder sávjaira és a sáv blokkjaira? Tegyünk valamilyen ésszerű feltételezést az egy sávon elhelyezkedő blokkok maximális számára; ne feledjük, hogy a *Megatron 747* esetén a szektorok/sávok száma változó.

3.3.2. feladat: Ismételjük meg a 3.3.1. feladatot a 2.2.1. feladatban leírt *Megatron 777* lemezzel.

* **3.3.3. feladat:** Ha nem csak a blokkcímeket, hanem a rekordcímeket is ábrázolni akarjuk, akkor további bájtokra van szükségünk. Tegyük fel, hogy úgy, mint a 3.3.1. feladatban, egy *Megatron 747* lemezhez akarunk címeket készíteni. Hány bájtra van szükség egy rekord címének megadásához, ha

* a) a fizikai címnek része a blokk bájtjainak a száma,
b) a rekordokhoz strukturált címeket használunk. Tegyük fel, hogy a tárolt rekordoknak van olyan kulcsuk, amely 4 bájtos egész szám.

3.3.4. feladat: Ma az IP (Internet Protocol)-címek 4 bájtosak. Tegyük fel, hogy egy világméretű címrendszerben a blokkok címei tartalmazzák a számítógép IP-címét, egy eszközszámot, amely egy 1 és 1000 közé eső szám, és a blokk címét az egyik eszközön, amelyről feltesszük, hogy egy *Megatron 747* lemez. Hány bájtra van szükség egy blokk címének megadásához?

3.3.5. feladat: A jövőben az IP-címek 16 bájtot fognak használni. Ezenfelül nem csak a blokkokra akarunk címeket megadni, hanem a rekordokra is. A rekordok egy blokkon belül tetszőleges bájton kezdődhetnek. Ezzel szemben a számítógépen belül az eszközöket nem kell külön ábrázolni (bár ez a 3.3.4. feladatban szükséges volt), mivel majd maguknak az eszközöknek lesz saját IP-címük. Ezen feltételezések mellett mennyi bájtra lenne szükség a címek ábrázoláshoz, ha továbbra is feltesszük, hogy az eszközeink *Megatron 747* lemezek?

! **3.3.6. feladat:** Tegyük fel, hogy egy *Megatron 747* lemez blokkjainak címeit valamilyen k-ra, k bájtos azonosítókat felhasználva logikailag akarjuk ábrázolni. Az is szükséges, hogy egy 3.7. ábrához hasonló leképezési táblát is magán a lemezen tároljunk. A leképezési tábla a logikai és fizikai címpárokból áll. A magához a leképezési táblához használt blokkok nem részei az adatbázisnak, ezért ezeknek a blokkoknak nincsen saját logikai címük a leképezési táblában. Tegyük fel, hogy a fizikai címek annyi bájtot használnak, amennyi a fizikai címekhez minimálisan szükséges (ennek értékét a 3.3.1. feladatban számoltuk ki). A logikai címekhez is annyi bájtot használunk, amennyi minimálisan szükséges. Hány blokkot fog elfoglalni a leképezési tábla a lemezen, ha a blokk mérete 4096 bájt?

*! **3.3.7. feladat:** Tegyük fel, hogy a blokkméret 4096 bájt, és a blokkban 100 bájtos rekordokat tárolunk. A blokk fejléce tartalmaz egy eltolásiérték-táblát a 3.8. ábrához hasonlóan. A táblában 2 bájtos mutatók tartoznak a blokk rekordjaihoz. Egy átlagos napon egy blokkba 2 rekordot szúrunk be, és 1 rekordot törlünk. Egy törölt rekord mutatóját a táblában helyettesítenünk kell egy „sírkővel", mert különben a rámutató mutatókból szabadon lógó mutatók keletkezhetnek. A pontosabb meghatározáshoz tegyük még fel azt is, hogy bármelyik napon a törlés mindig a beszúrások előtt történik. Ha egy blokk kezdetben üres, akkor hány nap múlva fordul elő, hogy nem marad hely benne több rekord beszúrásához?

! **3.3.8. feladat:** Ismételjük meg a 3.3.7. feladatot olyan feltevések mellett, hogy mindennap egy törlés és 1,1 beszúrás történik átlagosan.

3.3.9. feladat: Ismételjük meg a 3.3.7. feladatot olyan feltevések mellett, hogy a rekordok törlése helyett a rekordokat egy másik blokkba helyezzük át, és így egy 8 bájtos továbbítási címet kell az eltolásiérték-táblában a nekik megfelelő bejegyzésbe tenni. Tegyük fel a következők valamelyikét:

! a) Az eltolásiérték-táblában minden bejegyzéshez annyi bájtot használunk, amennyi maximálisan szükséges egy bejegyzéshez.

!! b) Megengedett, hogy az eltolásiérték-tábla bejegyzései változó hosszúak legyenek, csak az az elvárás, hogy minden bejegyzést meg lehessen találni, és helyesen lehessen értelmezni.

* **3.3.10. feladat:** Tegyük fel, hogyha minden mutatót automatikusan helyreigazítunk, akkor csak feleannyi idő szükséges a helyreigazításhoz, mint amennyire akkor lenne szükség, ha minden egyes mutatón külön-külön végeznénk el a helyreigazítást. Legyen p annak a valószínűsége, hogy egy mutatót a központi memóriában legalább egyszer követni fogunk. A p milyen értékére hatékonyabb az automatikus helyreigazítás az igény szerinti helyreigazításnál?

! **3.3.11. feladat:** Általánosítsuk a 3.3.10. feladatot. Egészítsük ki még azzal a lehetőséggel is, hogy a mutatók helyreigazítását sohasem végezzük el. Tegyük fel, hogy a fontos tevékenységek elvégzésére az alábbi idők szükségesek, tetszőleges időegységben mérve:

i) Egy mutató igény szerinti helyreigazítása: 30.
ii) A mutatók automatikus helyreigazítása: 20/mutató.
iii) Egy helyreigazított mutató követése: 1.
iv) Egy nem helyreigazított mutató követése: 10.

Tegyük fel, hogy a memóriamutatókat vagy $1 - p$ valószínűséggel nem követjük, vagy p valószínűséggel k alkalommal követjük. A k és p milyen értékeire nyújtja a legjobb átlagos teljesítményt az automatikus helyreigazítás, az igény szerinti helyreigazítás és az, amikor nem végzünk el semmilyen helyreigazítást?

3.4. Változó hosszú adatok és rekordok

Eddig azt az egyszerűsítő feltevést tettük, hogy minden adattételnek rögzített hossza van, a rekordoknak rögzített a sémájuk, és hogy a séma rögzített hosszú mezőkből álló lista. Ezzel szemben a gyakorlatban az élet ritkán ilyen egyszerű. Megadhatjuk a következőket is:

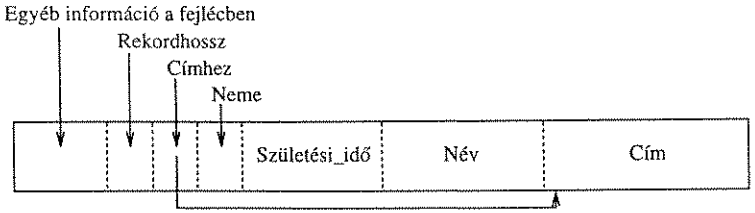
1. *Változó méretű adattételek.* Például a 3.1. ábrán láttunk egy `Filmszínész` relációt, amelynek van egy cím mezője, és ennek a hossza 255 bájtig bármekkora lehet. Noha lehetnek bizonyos címek ilyen hosszúak, de azért a címek nagy többsége valószínűleg 50 bájtos lesz vagy még rövidebb. Valószínűleg a `Filmszínész` sorainak tárolásához használt területnek több mint a felét megtakaríthatjuk, ha csak akkora területet használunk, amennyi az aktuális címhez szükséges.
2. *Ismétlődő mezők.* A 3.1. példában a filmszínész objektumoknak egy olyan osztályát vizsgáltuk, amely tartalmazott egy kapcsolatot a filmek halmazához, nevezetesen minden filmszínészhez hozzátartozik azoknak a filmeknek a halmaza, amelyben a filmszínész szerepelt. Ezeknek a filmeknek a száma színészről színészre változik, így az, hogy egy ilyen színész objektum rekord formájú tárolásához mekkora hely szükséges, szintén változó, és nem is lehet nyilvánvaló korlátot megadni rá.

3. *Változó formátumú rekordok.* Néha nem tudjuk előre, hogy mik lesznek egy rekordnak a mezői, vagy, hogy egy mezőnek hány előfordulása lesz a rekordban. Például bizonyos filmszínészek maguk is rendeznek filmeket, és ezért a rekordjaikhoz hozzá szeretnénk tenni olyan mezőket is, amelyek azokra a filmekre hivatkoznak, amelyeket a filmszínészek rendeztek. Hasonló a helyzet, ha más színészek például filmeket is gyártanak, vagy más formában vesznek részt egy film készítésében. Lehet, hogy ezeket az információkat is szeretnénk elhelyezni a rekordjaikba. Az is igaz viszont, hogy a legtöbb filmszínész se nem készít, se nem rendez filmeket, így nem szeretnénk az összes színész rekordban lefoglalni helyet ennek az információnak a számára.
4. *Hatalmas mezők.* A modern adatbázis-kezelő rendszerek olyan attribútumokat is támogatnak, amelyek értéke nagyon nagy adattétel. Például lehet, hogy azt akarjuk, hogy a filmszínész rekord egy kép attribútumot is tartalmazzon, amely egy GIF formátumú fénykép a színészről. Egy film rekordnak a szokásos mezők (filmcím és hasonlók) mellett pedig lehetne egy olyan mezője, amely magának a filmnek tartalmazná egy 2 gigabájtos MPEG formátumú kódolt változatát. Az ilyen mezők olyan nagyok, hogy ellentmondanak annak az elképzelésünknek, hogy a rekordok befférnek a blokkokba.

3.4.1. Változó hosszú mezőket tartalmazó rekordok

Ha egy rekord egy vagy több változó hosszúságú mezővel rendelkezik, akkor a rekordnak elegendő információt kell tartalmazni ahhoz, hogy megtalálhassuk a rekord bármelyik mezőjét. Egy egyszerű, de hatékony séma a következő: tegyük az összes rögzített hosszú rekordot a változó hosszú mezők elé. Ezután a rekord fejlécébe helyezzük el az alábbi információkat:

1. A rekord hossza.
2. Mutatók, vagyis eltolási értékek az összes változó hosszú mező elejére. Ha a változó hosszú mezők mindig ugyanabban a sorrendben szerepelnek, akkor közülük az elsőhöz nem kellene mutatót megadnunk, mivel tudjuk, hogy közvetlenül a rögzített hosszú mezők után kezdődik.



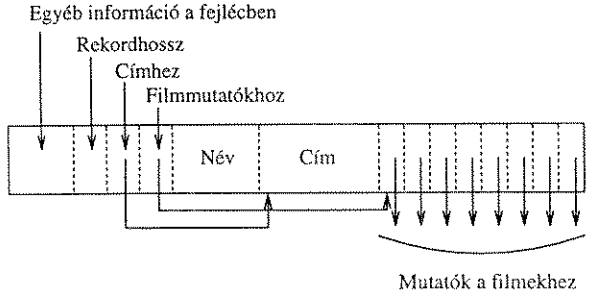
3.12. ábra. *Egy* `Filmszínész` *rekord, melyhez tartozó* `név` *és* `cím` *mezőket változó hosszú karakterláncokkal valósítjuk meg*

3.10. példa: Tegyük fel, hogy a filmszínész rekordokban név, cím, neme és születés mezők szerepelnek. Fel fogjuk tenni, hogy a neme és a születés mezők rögzített hosszúságúak, rendre 4, illetve 12 bájtosak. Ezzel szemben mind a név, mind a cím mezőket bármekkora, megfelelő hosszú karakterláncokkal fogjuk ábrázolni. A 3.12. ábra mutatja, hogy fog kinézni egy tipikus filmszínész rekord. A név mezőt mindig a cím elé fogjuk tenni. Így nem szükséges olyan mutató, amely a név kezdetére mutat; ez a mező mindig éppen a rekord rögzített hosszú része után fog kezdődni. □

3.4.2. Ismétlődő mezőket tartalmazó rekordok

Hasonló a helyzet, mikor egy F mezőből változó számú előfordulást tartalmaz egy rekord, bár maga az F mező rögzített hosszú. Ekkor elég, ha az F mező összes előfordulásából csoportot képezünk, és a rekord fejlécébe elhelyezünk egy mutatót, amely az első előfordulásra mutat. Ezután az F mező összes előfordulását (azaz az előfordulások eltolási értékeit) meg tudjuk találni az alábbi módon. Legyen az F mező egy előfordulásának a hossza L bájt. Ekkor az F eltolási értékéhez hozzáadjuk az L minden egész számú többszörösét (0, L, $2L$, $3L$ stb.) mindaddig, amíg el nem érjük az F-et követő mező eltolási értékét, és ekkor megállunk.

3.11. példa: Tegyük fel, hogy újratervezzük a filmszínész rekordjainkat úgy, hogy csak a név és cím mezőket tartalmazzák (melyek változó hosszú karakterláncok), valamint mutatókat a színész összes filmjére. A 3.13. ábra mutatja, hogyan lehetne az ilyen típusú rekordokat ábrázolni. A fejléc két mutatót tartalmaz. Ezek közül az egyik a cím mező elejére mutat (feltesszük, hogy a név mező közvetlenül a fejléc után kezdődik). A másik mutató pedig a film mutatók közül az elsőre mutat. A rekord hossza mondja meg, hogy mennyi ilyen filmmutató van a rekordban. □



3.13. ábra. *Egy rekord, melyben ismétlődő filmhivatkozások csoportja szerepel*

Egy másik lehetséges megadás, hogy a rekordok rögzített hosszúak maradnak, és a változó hosszú részüket – legyenek azok változó hosszú mezők vagy meghatározatlan számú mezőismétlődések – egy külön blokkba helyezzük. A rekordba magába pedig a következő információkat is eltároljuk:

Nullértékek ábrázolása

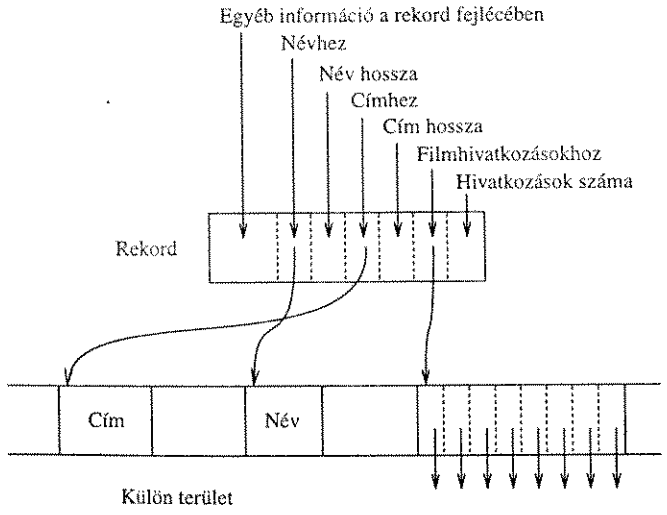
A soroknak gyakran lehetnek olyan mezőik, amelyek `nullértéket (NULL)` is tartalmazhatnak. A 3.12. ábra rekordformátuma egy kényelmes módot nyújt ahhoz, hogyan ábrázoljuk a `nullértékeket`. Ha egy mező, például a `cím` nullértékű, akkor nullmutatót (null pointer) teszünk arra a helyre, ahol a címre hivatkozó mutató következik. Ekkor nem kell hely a címnek, csak ennek a mutatónak kell hely. Ez az eljárás átlagban helyet takaríthat meg, még akkor is, ha a `cím` rögzített hosszú, de gyakran kap `nullértéket` értékül.

1. mutatókat arra a helyre, ahol minden ismétlődő mező kezdődik, és
2. vagy az ismétlődések számát, vagy hogy hol végződik az ismétlődés.

A 3.14. ábra a 3.11. példa problémájára mutat egy rekordelrendezést, de úgy, hogy a változó hosszú `név` és `cím` mezőket, valamint a `szerepeltBenne` ismétlődő (filmhivatkozások halmazát tartalmazó) mezőt egy vagy több különálló blokkban tárolja.

Előnyei és hátrányai is vannak annak, ha egy rekord változó hosszú komponenseit ilyen közvetett módon ábrázoljuk:

- Mivel a rekordot magát rögzített hosszú rekordként tároljuk, ezért a rekordok keresése sokkal hatékonyabb, minimalizálja a fejlécek költségét, és lehetővé teszi, hogy minimális erőfeszítéssel lehessen a rekordokat a blokkon belül vagy a blokkok között mozgatni.



3.14. ábra. *A változó hosszú mezőket a rekordtól külön tároljuk*

- Másrészt, ha a változó hosszú komponenseket egy másik blokkban tároljuk, akkor növekszik a lemez I/O-műveletek száma, mikor egy rekord összes komponensét meg akarjuk vizsgálni.

Egy kiegyenlített stratégia lehet a következő: a rekord rögzített hosszú része legyen akkora, hogy legyen benne elegendő hely a következők számára:

1. Az ismétlődő mezőkből ésszerű számú előfordulás.
2. Egy mutató arra a helyre, ahol az ismétlődő mező további előfordulásai találhatók.
3. Egy számláló, mely azt mutatja, hogy az ismétlődő mezőből mennyi további előfordulás létezik.

Ha az 1. pontban megadott számnál kevesebb az előfordulások száma, akkor valamekkora hely felhasználatlanul marad. Ha több van, mint amennyi a rögzített hosszú részbe fér, akkor a külön helyre mutató hivatkozás nem nullmutató lesz, és így ezt a mutatót követve meg tudjuk találni a többi előfordulást.

3.4.3. Változó formátumú rekordok

Még bonyolultabb a helyzet, mikor a rekordoknak nincs rögzített sémájuk, vagyis mikor az a reláció vagy osztály, aminek a sorát vagy objektumát ábrázolja a rekord, nem határozza meg teljesen a mezőket vagy a mezők sorrendjét. A változó formátumú rekordok ábrázolásának legegyszerűbb módja, mikor *címkézett mezők* (tagged fields) sorozatát adjuk meg. Minden címkézett mező a következőkből áll:

1. Információ a szóban forgó mező szerepéről, azaz
   a) az attribútum vagy mezőnév,
   b) a mező típusa, ha ez nem nyilvánvaló a mezőnévből és valami könnyen elérhető sémainformációból,
   c) a mező hossza, ha ez nem nyilvánvaló a típusból.
2. A mező értéke.

Legalább két okból értelmes a címkézett mezők használata.

1. *Információintegrációs alkalmazások.* Időnként egy relációt több, korábbi forrásból készítenek el, ráadásul ezekben a forrásokban különböző típusú információt tároltak; a részletesebb tárgyalásához lásd a 11.1. részt. Például a filmszínész információnk több helyről is származhat, melyek közül az egyik tárolja a születést, míg a többiek nem, egyesek megadják a címet, míg mások nem és így tovább. Ha nincsen túl sok mező, akkor valószínűleg akkor járunk a legjobban, ha `nullértékeket` hagyunk azokon az értékeken, amiket nem ismerünk. Viszont, ha sok forrásunk van, és ezek sok különböző típusú információt adnak, akkor lehet, hogy túl sok

! **3.4.6. feladat:** Emlékezzünk vissza, hogy a 2.3. példában kiszámoltuk, hogy egy *Megatron 747* lemez esetében egy 4096 bájtos blokk átviteli sebessége 1/2 milliszekundum. Egyórányi MPEG formátumú filmhez körülbelül egy gigabájt szükséges. Lehet-e úgy szervezni egy MPEG film blokkjait a *Megatron 747* lemezen, hogy a filmet valós időben tudjuk lejátszani? Ha nem, akkor hány *Megatron 747* lemez kellene ehhez? Hogyan tudnánk úgy szervezni a blokkokat, hogy a filmet, ha nem is valós időben, de csak egy kis késéssel lehessen lejátszani?

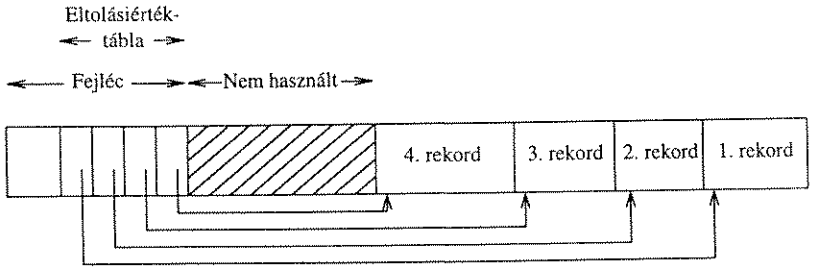
3.5. Rekordmódosítások

A beszúrás (insert), törlés (delete), módosítás (update) műveletek gyakran speciális problémákhoz vezetnek. Ezek a problémák akkor a legsúlyosabbak, mikor a rekordok hossza változik, de még akkor is előjöhetnek, ha a rekordok és a mezők egyaránt rögzített hosszúak.

3.5.1. Beszúrás

Először foglalkozzunk azzal, hogy új rekordokat akarunk beszúrni egy relációba (vagy ami ezzel ekvivalens, egy osztály aktuális előfordulásába). Ha egy relációban a rekordokat rendezetlenül tároljuk, azaz nincs különösebb sorrend meghatározva közöttük, akkor a beszúráshoz kereshetünk egy olyan blokkot, amelyben még van üres hely, vagy ha ilyen nincs, akkor kerítünk egy új blokkot, és abba tesszük a rekordot. Rendszerint létezik valamilyen mechanizmus arra, hogy hogyan lehet egy adott relációhoz vagy osztályhoz megtalálni az összes olyan blokkot, amelyek a relációsorokat, illetőleg az objektumokat tartalmazzák, de a 4.1. részig nem foglalkozunk azzal a kérdéssel, hogy miként lehet nyomon követni ezeket a blokkokat.

Problémásabb, mikor a sorokat valamilyen rögzített sorrend szerint kell tárolni, például az elsődleges kulcsuk szerint rendezve. Jó okunk van arra, hogy a rekordokat rendezve tároljuk, mivel bizonyos kérdések megválaszolását ez megkönnyítheti,



3.17. ábra. *Az eltolásiérték-tábla segítségével tudjuk a rekordokat elcsúsztatni a blokkon belül, hogy helyet készítsünk az új rekordoknak*

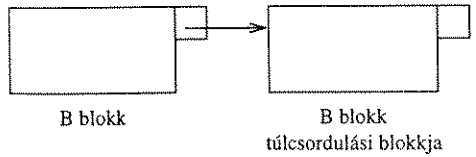
ahogy ezt majd a 4.1. részben látni fogjuk. Ha be kell szúrnunk egy új rekordot, akkor először meg kell keresnünk a rekordnak megfelelő blokkot. Ha véletlenül van még hely ebben a blokkban, akkor betesszük ide a rekordot. Mivel a blokkokat rendezve tároljuk, ezért lehet, hogy a blokkon belül a rekordokat el kell majd csúsztatni ahhoz, hogy a megfelelő pontnál helyet biztosítsunk az új rekordnak.

Ha a rekordokat el kel csúsztatni, akkor hasznos lehet az a blokkszervezés, amit a 3.8. ábrán mutattunk be, és amit most a 3.17. ábrán megismétlünk. Emlékezzünk vissza arra, hogy a 3.3.2. részben megtárgyaltuk, hogy készíthetünk egy „eltolásiérték-táblát" minden blokk fejlécében. Ez a tábla a blokkban szereplő minden rekordhoz tartalmaz egy mutatót, mely a megfelelő rekord helyére mutat. Azok a mutatók, amelyek egy rekordra a blokkon kívülről mutatnak, „strukturált címek". A strukturált címek két részből állnak. Ezek a blokk címe és annak a bejegyzésnek a helye az eltolásiérték-táblában, amely ennek a rekordnak felel meg.

Ha a blokkban egyből találunk helyet a beszúrt rekordnak, akkor egyszerűen elcsúsztatjuk a rekordokat a blokkon belül, és az eltolásiérték-táblában megfelelően módosítjuk a mutatókat, majd beszúrjuk az új rekordot a blokkba, és a blokk eltolásiérték-táblájához hozzáadunk egy új mutatót, mely az új rekordra mutat.

Igen ám, de előfordulhat, hogy nincs már hely a blokkban az új rekord számára. Ebben az esetben a blokkon kívül kell helyet keresni. Ennek a problémának a leküzdésére két megközelítést adunk meg, de ezeknek a kombinációja is alkalmazható.

1. *Keressünk helyet egy „közeli" blokkban.* Például ha a B_1 blokkban már nincs hely annak a rekordnak a számára, amelyet a rendezés szerinti sorrendben ebbe a blokkba kellene beszúrni, akkor megnézzük a blokkok rendezés szerinti sorrendjében a következő B_2 blokkot. Ha van hely a B_2-ben, akkor a B_1-ből a legnagyobb rendezési értékű rekordot vagy rekordokat átmozgatjuk a B_2-be, és mindkét blokkban megfelelően elcsúsztatjuk a rekordokat. Azonban, ha külső mutatók is vonatkoznak ezekre a rekordokra, akkor vigyáznunk kell arra, hogy nehogy elfelejtsünk a B_1 eltolásiérték-táblájában egy *továbbítási címet* (forwarding address) hagyni, amely azt mondja meg, hogy egy bizonyos rekordot a B_2-be mozgattunk át, és azt is megmondja, hogy hol van a neki megfelelő bejegyzés a B_2 eltolásiérték-táblájában. Ha ilyen továbbítási címeket is megengedünk, akkor általában több helyre van szükség az eltolásiérték-tábla bejegyzéseihez.
2. *Készítsünk egy túlcsordulási* (overflow) *blokkot.* Ebben a sémában minden B blokk fejlécében helyet tartunk fenn egy olyan mutató számára, amely egy *túlcsordulási*



3.18. ábra. *Egy blokk és az első túlcsordulási blokkja*

blokkra[7] mutat. Ez a blokk arra szolgál, hogy idehelyezhetjük azokat a további rekordokat, amelyek elméletileg a *B*-be tartoznak. A *B* túlcsordulási blokkja is mutathat egy második túlcsordulási blokkra és így tovább. A 3.18. ábra mutat egy ilyen helyzetet. Az ábrán a túlcsordulási blokkokra hivatkozó mutatót úgy ábrázoltuk, mintha a blokkon lenne egy kis dudor, de természetesen ez a mutató valójában a blokk fejlécének része.

3.5.2. Törlés

Amikor törlünk egy rekordot, akkor lehet, hogy vissza tudjuk nyerni a neki megfelelő helyet. Ha a 3.17. ábrához hasonló eltolásiérték-táblát használunk, és a rekordokat a blokkon belül elcsúsztathatjuk, akkor összetömöríthetjük a használt helyet a blokkon belül úgy, hogy egyetlen nem használt tartomány maradjon a blokk közepén, ahogy ez a 3.17. ábrán látható.

Ha nem tudjuk a rekordokat elcsúsztatni, akkor karban kell tartanunk egy listát a blokk fejlécében, amely a rendelkezésre álló helyeket mutatja meg. Ekkor tudni fogjuk, hogy hol vannak, és milyen nagyok a rendelkezésre álló területek, mikor egy új rekordot akarunk beszúrni a blokkba. Megjegyezzük, hogy normális esetben a blokk fejlécében nem kell tárolni a rendelkezésre álló helyek teljes listáját. Elég, ha a listának az első elemét tesszük a blokk fejlécébe, és magukat a szabad tartományokat használjuk arra, hogy tárolják a listának megfelelő hivatkozásokat, nagyjából úgy, ahogy ezt a 3.11. ábrán mutattuk.

Mikor törlünk egy rekordot, akkor lehet, hogy a túlcsordulási blokkot is megszüntethetjük. Ha egy *B* blokkból vagy egy olyan blokkból, amely a túlcsordulási láncához tartozik, törlünk egy rekordot, akkor megnézhetjük, hogy a lánc összes blokkjain mennyi a felhasznált összterület. Ha a rekordok kevesebb blokkban is elférnek, akkor nyugodtan átmozgathatjuk a rekordokat a lánc blokkjai között, és végrehajtjuk a teljes lánc újraszervezését.

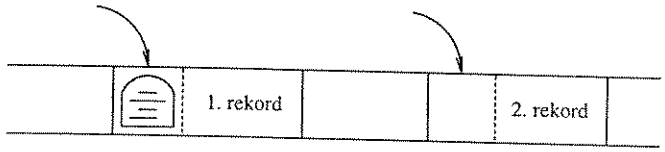
Van azonban még egy probléma a törléssel kapcsolatban, amire mindig figyelnünk kell, függetlenül attól, hogy milyen sémát használunk a blokkok szervezéséhez. Létezhetnek olyan mutatók, amelyek a törölt rekordra mutatnak. Ha vannak ilyen mutatók, akkor nem szeretnénk, hogy ezek a mutatók a törlés után szabadon lógóvá váljanak, vagy egy olyan új rekordra mutassanak, amelyet a törölt rekord helyére tettünk. Az ilyen problémák kezelésére szokásos technikát már bemutattuk a 3.3.2. részben, ezt most is alkalmazhatjuk, azaz helyezzünk egy *sírkövet* (tombstone) a rekord helyére. Ez a sírkő állandó; addig kell léteznie, amíg a teljes adatbázist újra nem szervezzük.

A rekordmutatók természetén múlik, hogy hová helyezzük a sírkövet. Ha a mutatók olyan rögzített helyekre mutatnak, ahonnan a rekord helyét megtaláljuk, akkor erre a rögzített helyre tesszük a sírkövet. Két példát mutatunk erre:

[7] Szokás ezt gyűjtőblokknak vagy egyszerűen gyűjtőnek is hívni. *A fordító megjegyzése.*

1. A 3.3.2. részben láttuk, hogy ha a 3.17. ábra sémájának eltolásiérték-tábláját használnánk, akkor a sírkő lehetne egy nullmutató az eltolásiérték-táblában, hiszen az erre a rekordra hivatkozó mutatók valójában az eltolásiérték-táblának a bejegyzéseire mutatnak.
2. Ha a 3.7. ábrán látható leképezési táblát használjuk arra, hogy a logikai rekord címeket lefordítsuk fizikai címekre, akkor a sírkő lehet egy nullmutató a fizikai cím helyén.

Ha rekordokat kell sírkövekre cserélnünk, akkor okos dolog lenne tárolni a rekord fejlécének legelején egy bitet, ami sírkőnek szolgál, vagyis amelynek az értéke 0, ha a rekordot *nem* töröltük, és 1, ha a rekordot töröltük. Ekkor csak ennek a bitnek kell megmaradnia ott, ahol a rekord kezdődött, és az ezutáni bájtok már újból felhasználhatók egy másik rekordhoz, ahogy ez a 3.19. ábrán[8] látszik. Ha egy törölt rekordhoz jutunk egy mutatót követve, akkor az első, amit észreveszünk, az a „sírkő" bit, amely elárulja nekünk, hogy a rekordot már törölték. Ezután már tudjuk, hogy a következő bájtokat nem kell megnézni.



3.19. ábra. *Az 1. rekord lecserélhető, de a sírkő megmarad; a 2. rekordnak nincs sírköve, és ezért látható, ha egy hozzávezető mutatót követünk*

3.5.3. Módosítás

Amikor egy rögzített hosszú rekordot módosítunk, akkor ennek nincs különösebb hatása a tárolórendszerre, mivel tudjuk, hogy pontosan ugyanazt a helyet fogja elfoglalni, mint a módosítás előtt. Ezzel szemben, mikor egy változó hosszú rekordot módosítunk, akkor a beszúrásnál és törlésnél keletkező összes problémával szembe kell nézni, kivéve, hogy sosem kell a rekord régi változata számára sírkövet készíteni.

Ha a módosított rekord hosszabb, mint a régi változata, akkor lehet, hogy több helyet kell a blokkjában készítenünk a számára. Ez a folyamat magában foglalhatja a rekordok elcsúsztatását, sőt egy túlcsordulási blokk készítését is. Ha a rekord változó hosszú részeit egy másik blokkban tároljuk, ahogy ezt a 3.14. ábrán láttuk, akkor előfordulhat, hogy az elemeket el kell mozgatnunk a blokkok belül, vagy hogy új blokkot kell készítenünk a változó hosszú mezők tárolására. Fordítva, ha a rekord a módosítás hatására kisebb lesz, akkor ahogy azt a törlésnél is láttuk, alkalmunk van visszaszerezni, vagy tömörebbé tenni a területet, vagy megszüntetni a túlcsordulási blokkokat.

[8] A 3.2.1. részben tárgyalt mezőigazítási probléma viszont arra kényszeríthet bennünket, hogy 4 vagy több bájtot is kihasználatlanul hagyjunk.

3.5.4. Feladatok

3.5.1. feladat: Tegyük fel, hogy a blokkjaink olyan rekordokból állnak, melyek a rendező kulcsmező alapján rendezettek, és a blokkok között is ez alapján a rendezés alapján vannak felosztva. Minden blokk esetében kívülről ismert, hogy mi a rendező kulcsainak a tartománya (erre a helyzetre mutat példát a 4.1.3. részben tárgyalt ritka indexstruktúra). Tegyük fel, hogy a rekordokra kívülről nem hivatkozik mutató, így a rekordokat a blokkok között nyugodtan mozgathatjuk, ha a szükség úgy kívánja. Megadunk néhány módszert arra, ahogy a beszúrásokat és törléseket kezelhetjük.

i) Ha túlcsordulás fordul elő, akkor vágjuk ketté a blokkot, és állítsuk be megfelelően a blokkhoz tartozó, rendezési kulcsra vonatkozó tartományokat.

ii) Tároljuk az egy blokkhoz tartozó rendezési kulcsok tartományát, és szükség esetén használjunk túlcsordulási blokkokat. Minden blokkhoz és túlcsordulási blokkhoz tároljunk egy eltolásiérték-táblát, mely az abban a blokkban szereplő rekordokra vonatkozik.

iii) Ugyanaz, mint ii), de most egy blokkhoz és az összes túlcsordulási blokkjához egy eltolásiérték-táblát tároljunk, méghozzá az első blokkban (vagy a túlcsordulási blokkokban, ha az eltolásiérték-táblához több hely szükséges). Megjegyezzük, hogy ha több hely kell az eltolásiérték-tábla számára, akkor az első blokkból átmozgathatunk rekordokat egy túlcsordulási blokkba, hogy több helyet teremtsünk.

iv) Ugyanaz, mint ii), de most egy mutatóval együtt a rendezési kulcsot is tároljuk az eltolásiérték-táblában.

v) Ugyanaz, mint iii), de most egy mutatóval együtt a rendezési kulcsot is tároljuk az eltolásiérték-táblában.

Válaszoljuk meg a következő kérdéseket:

* a) Tegyük fel, hogy egy adott rendezési kulcsú rekordot keresünk, és megtaláljuk azt a blokkot (vagy egy túlcsordulási blokkláncban az első blokkot), amelyben szerepelhet az adott kulcsú rekord. Hasonlítsuk össze az i) és ii) módszereket abban a tekintetben, hogy átlagosan mennyi lemez I/O-műveletre van szükség ahhoz, hogy ezután visszakapjuk az adott kulcsú rekordot.

b) Hasonlítsuk össze ismét az ii) és iii) módszereket, de most abban a tekintetben, hogy b paraméter függvényében átlagosan mennyi lemez I/O-művelet szükséges egy rekordmegkereséshez, ha a lánc b blokkból áll. Tegyük fel, hogy az eltolásiérték-tábla 10% helyet foglal el, és a maradék 90% helyet a rekordok foglalják el.

! c) Ugyanaz a feladat, mint b) esetében, de most a iv) és v) módszereket hasonlítsuk össze. Tegyük fel, hogy rendezési kulcs terjedelme a rekord hosszának 1/9-ed része. Megjegyezzük, hogy a rekordban nem is kell megismételni a rendezési kulcsot, ha ez az eltolásiérték-táblában is szerepel. Az előzőek miatt az eltolásiérték-tábla valójában 20% helyet fog használni, és a többi 80% hely marad a rekordok számára.

3.5.2. feladat: A relációs adatbázisrendszerek, ha ez lehetséges, mindig jobban szeretik a rögzített hosszú sorokat kezelni. Adjunk meg három indokot erre.

3.6. Összefoglalás

- *Mezők:* A mezők a legegyszerűbb adatelemek. Ezek közül sok esetben (például az egészek vagy rögzített hosszú karakterláncok esetében) egyszerűen megadunk egy megfelelő bájtszámot a másodlagos tárolón. A változó hosszú karakterláncokat kétféleképp kódoljuk. Az egyik esetben egy rögzített hosszú bájtsorozat tartalmaz egy „vége" jelet, a másik esetben az ilyen karakterláncokat a változó karakterláncok számára fenntartott területen tároljuk, és a hossznak megfelelő egész számot teszszük a karakterlánc elejére vagy egy „vége" jelet a végére.
- *Rekordok:* A rekordok néhány mezőből és egy rekordfejlécből épülnek fel. A fejléc a rekordról tartalmaz információkat. Ezek között szerepelhet időbélyegző, sémainformáció, rekordhossz.
- *Változó hosszú rekordok:* Ha a rekord egy vagy több változó hosszú mezőt tartalmaz, vagy egy mezőnek ismeretlen számú ismétlődését tartalmazza, akkor másmilyen struktúrát kell használni. A rekord fejlécében egy mutatókból álló jegyzéket (directory) használhatunk arra, hogy a rekordon belül megtaláljuk a változó hosszú mezőket. Egy másik lehetőség, hogy a változó hosszú vagy az ismétlődő mezőket olyan (rögzített hosszú) mutatókkal cseréljük fel, melyek egy rekordon kívüli helyre mutatnak, oda, ahol a mező értékét tároljuk.
- *Blokkok:* A rekordokat általában blokkokban tároljuk. A blokk területének egy részét a blokk fejléce foglalja el, melyben a blokkról tárolunk információkat, a blokk többi részét pedig egy vagy több rekord tölti ki, illetve üres hely is maradhat benne.
- *Átnyúló (spanned) rekordok:* Általában egy rekord egy blokkban helyezkedik el. Viszont ha a rekordok hosszabbak, mint a blokkok, vagy ha fel akarjuk használni a blokkon belüli maradék helyet, akkor a rekordot két vagy több darabra törjük, és egy töredéket teszünk minden blokkba. A töredéknek is kell hogy legyen fejléce, mert ennek segítségével lehet az egy rekordhoz tartozó töredékeket összekapcsolni.
- *Bináris, nagy objektumok (BLOB-ok):* A nagyon nagy méretű értékeket, olyanokat, mint a képek és filmek, bináris, nagy objektumoknak (binary, large objects – *BLOB*) hívjuk. Ezeket az értékeket több blokkon keresztül kell tárolnunk. A hozzáférésre vonatkozó elvárásoktól függően érdemes lehet a BLOB-ot egy cilinderen tárolni, mert ezzel csökkenteni lehet a BLOB elérési idejét. Szükség lehet arra is, hogy darabokra (stripe) szedjük szét a BLOB-ot, és ezeket a darabokat több lemezen helyezzük el. Ez a módszer lehetővé teszi a BLOB tartalmának párhuzamos visszanyerését.
- *Eltolási érték (offset) táblája:* A blokk fejlécében elhelyezhetünk egy eltolásiérték-táblát, amely mutatókat tartalmaz a blokk minden egyes rekordjához. Ezzel a rekordok törlését és beszúrását egyaránt támogatni lehet, és azokat a rekordokat is, melyeknek változhat a hossza a változó hosszú mezők módosítása miatt.
- *Túlcsordulási (overflow) blokk:* Szintén a beszúrások és megnagyobbodó rekordok támogatására szolgál a következő: egy blokk tartalmazhat egy hivatkozást egy túlcsordulási blokkra vagy blokkok láncára. Ezekben a túlcsordulási blokkokban olyan rekordokat tárolunk, amelyek logikailag az első blokkhoz tartoznak.
- *Adatbáziscím:* Egy adatbázis-kezelő rendszer által kezelt adatok általában több tárolóeszközön, tipikusan lemezen helyezkednek el. Ahhoz, hogy a blokkokat és rekor-

dokat ebben a tárolási rendszerben megtaláljuk, használhatunk fizikai címeket, melyek a következőket írják le: eszközszám, cilinder, sáv, szektor(ok) és egy szektoron belüli lehetséges bájt. Használhatunk logikai címeket is, amelyek tetszőleges karakterláncok. A logikai címeket egy leképezési táblával lehet lefordítani fizikai címekre.

- *Strukturált címek:* A rekordokat megtalálhatjuk a fizikai cím egy részének és egy további információnak a segítségével is. A fizikai cím része például tartalmazhatja annak a blokknak a helyét, amelyben a rekord található, a kiegészítő információ pedig lehet egy kulcsa a rekordnak vagy egy pozíció egy blokk eltolásiérték-táblájában, amely meghatározza a rekord helyét.
- *Mutatók helyreigazítása (swizzling):* Amikor egy lemezblokkot behozunk a memóriába, akkor az adatbáziscímeket le kell fordítani memóriacímekre, ha vannak olyan mutatók, amiket követni kell. Ezt a fordítást hívjuk helyreigazításnak. A helyreigazítást vagy automatikusan végezzük el akkor, amikor blokkokat hozunk be a memóriába, vagy igény szerint végezzük el, vagyis mikor először követünk egy ilyen mutatót.
- *Sírkövek:* Amikor törlünk egy rekordot, akkor ez azt okozhatja, hogy a rekordra hivatkozó mutatók szabadon lógóvá válnak, azaz a végük „felszabadult", és emiatt rossz helyre mutatnak. Egy sírkő figyelmeztet a törölt rekord helyén vagy annak egy részén, hogy ez a rekord már nincs ott.
- *Feltűzött (pinned) blokkok:* Különböző okokból kifolyólag (melyek közt szerepel az is, hogy egy blokk helyreigazított mutatókat is tartalmazhat), lehet, hogy nem szabad a memóriából egy blokkot egyszerűen visszamásolni a helyére, a lemezre. Az ilyen blokkot feltűzött blokknak hívjuk. Ha a feltűzöttség helyreigazított mutatóknak köszönhető, akkor mielőtt visszaírnánk a lemezre a blokkot, előbb vissza kell állítanunk a helyreigazítást.

3.7. Irodalomjegyzék

A [2]-ben egy adatstruktúrákról szóló, 1968-as klasszikus írást frissítettek fel nem is olyan régen. A [4]-ben hasznos információkat találunk azokról a struktúrákról, amelyek ebben és a 4. fejezetben fontos szerepet játszanak.

A törléssel foglalkozó technikák közül a sírkövek használata a [3]-ból ered. Az [1] a legfontosabb adatábrázolási kérdéseket fedi le; a címek és a helyreigazítás kérdéskörét az objektumorientált adatbázis-kezelő rendszerek vonatkozásában vizsgálja.

1. R. G. G. Cattell, *Object Data Management,* Addison-Wesley, Reading MA, 1994.
2. D. E. Knuth, *The Art of Computer Programming, Vol. I, Fundamental Algorithms, Third Edition,* Addison-Wesley, Reading MA, 1997. (Magyarul *A számítógép programozásának művészete, 1. kötet,* Műszaki Könyvkiadó, Budapest, 1987.)
3. D. Lomet, „Scheme for invalidating free references", *IBM J. Research and Development* **19**:1 (1975), pp. 26–35.
4. G. Wiederhold, *File Organization for Database Design,* McGraw-Hill, New York, 1987.

4. fejezet

Indexstruktúrák

Az eddigiekben láthattuk, hogy milyen lehetőségek állnak rendelkezésre a rekordok tárolására, nézzük most meg, hogy miként lehet tárolni teljes relációkat vagy osztály-kiterjedéseket. Nem elég csupán szétszórni a különböző blokkok között a reláció sorait reprezentáló rekordokat, illetve a kiterjesztés objektumait. Hogy lássuk miért, nézzük meg, miként tudnánk megválaszolni a következő legegyszerűbb lekérdezést: `SELECT * FROM R`. Meg kellene vizsgálnunk a háttértároló valamennyi blokkját, és az alábbi adatokra kellene támaszkodnunk:

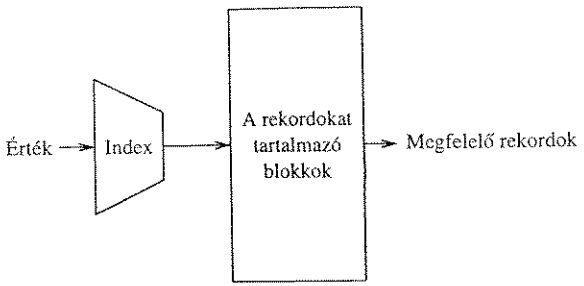
1. A blokkok fejléceiben tárolt információ arról, hogy hol kezdődnek az adott blokkban a rekordok.
2. A rekordok fejléceiben tárolt információ arról, hogy az adott rekord melyik relációhoz tartozik.

Valamivel jobb ötlet lefoglalni néhány blokkot, esetleg néhány teljes cilindert egy adott reláció számára. Ilyenkor e cilinderek valamennyi blokkja az adott reláció sorait reprezentáló rekordokat tartalmazza. Ez esetben legalább a reláció sorait megtalálhatjuk anélkül, hogy a teljes háttértárolót végig kellene pásztázni.

Azonban ez az elrendezés sem nyújt semmilyen segítséget abban az esetben, ha a következő, szintén egyszerű lekérdezést szeretnénk megválaszolni: „keressük azt a sort, amelyben az elsődleges kulcs értéke megegyezik egy előre megadott értékkel". Vegyük például a 3.1. ábra `Filmszínész` relációját, ahol a `név` elsődleges kulcs. Egy olyan lekérdezés esetén például, mint a

```
SELECT *
FROM Filmszínész
WHERE név = 'Jim Carrey';
```

végig kell pásztáznunk valamennyi olyan blokkot, ahol a `Filmszínész` reláció sorai előfordulhatnak. Az ilyen típusú lekérdezések megkönnyítésére gyakran hozunk létre relációkon egy vagy több *indexet.* Amint azt a 4.1. ábra is sugallja, az index egy olyan adatszerkezet, amelynek segítségével „könnyedén" megtalálhatunk adott tulajdonság-



4.1. ábra. *Egy index megkapja bizonyos mező(k) értékét, és megtalálja a megfelelő értékkel rendelkező rekordokat*

gal rendelkező rekordokat, ahol a tulajdonság jellegzetesen egy vagy több mező értékére vonatkozik. Az index lehetővé teszi, hogy egy rekord megkereséséhez az összes lehetséges rekordnak csak egy kis töredékét kelljen végignézni. Az index alapjául szolgáló mező(ke)t *keresési kulcsnak* (search key) nevezzük, de ha a szövegkörnyezetből egyértelműen kiderül, hogy indexről van szó, akkor nevezhetjük egyszerűen csak „kulcsnak".

Több különböző adatszerkezet szolgálhat indexként. A fejezet további részében indexek tervezésére és megvalósítására szolgáló módszereket fogunk megvizsgálni:

1. Egyszerű indexek rendezett fájlokon.
2. Másodlagos indexek nem rendezett fájlokon.
3. B-fák – közkeletű eljárás indexek építésére tetszőleges fájlon.
4. Tördelőtáblázatok – egy másik hasznos és fontos indexszerkezet.

4.1. Indexek szekvenciális fájlokon

Az indexek tanulmányozását annak az adatszerkezetnek a vizsgálatával kezdjük, amely talán a legegyszerűbb, és a következőképpen épül fel: egy *adatfájlnak* (data file) nevezett rendezett fájl, amelyhez tartozik egy kulcs-mutató párokból álló másik fájl, amit *indexfájlnak* (index file) nevezünk. Az indexfájl valamennyi K keresési kulcsa társítva van egy mutatóval, amely az adatfájl azon rekordjára mutat, amely tartalmazza a K keresési kulcsot. Ezek az indexek lehetnek „sűrűk", ami azt jelenti, hogy az adatfájl minden rekordjához létezik egy bejegyzés az indexfájlban, vagy lehetnek „ritkák", amikor az indexfájlban csak az adatfájl néhány rekordja van feltüntetve. Ez utóbbi esetben általában az adatfájl egy blokkjához az indexfájlban egyetlen bejegyzés tartozik.

4.1.1. Szekvenciális fájlok

Az egyik legegyszerűbb típusú index alapjául olyan fájl szolgál, amely rendezett az index attribútumára (attribútumaira) nézve. Az ilyen fájl neve *szekvenciális fájl* (sequential file). Ez a szerkezet különösen akkor hasznos, amikor a keresési kulcs a reláció elsődleges kulcsa, bár használható más attribútumok esetén is. A 4.2. ábrán egy szekvenciális fájlként ábrázolt relációt láthatunk.

Ebben a fájlban a sorok rendezve vannak az elsődleges kulcs szerint. Elképzeléseink szerint a kulcsok egész számok; csak a kulcsmezőket ábrázoljuk, és feltételezzük azt az egyáltalán nem tipikus helyzetet, hogy egy blokkban csak két rekord fér el. A fájl első blokkja például a 10-es és 20-as kulcsértékkel rendelkező rekordokat tartalmazza. Ebben és sok más példában is olyan kulcsot használunk, amelynek értékei a 10 egymást követő többszörösei, habár természetesen nem követelmény, hogy a kulcsok 10 többszörösei legyenek, mint ahogyan az sem, hogy valamennyi, a 10 többszörösét tartalmazó rekord jelen legyen.

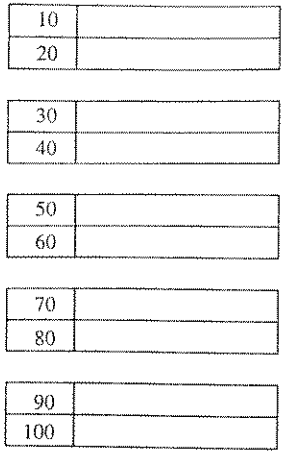
4.1.2. Sűrű indexek

Most, hogy már rendezettek a rekordjaink, felépíthetünk rajtuk egy *sűrű indexet* (dense index), amely nem más, mint olyan blokkok sorozata, amelyek csak a rekordok kulcsait és azokat a mutatókat tartalmazzák, amelyek az adott rekordokra mutatnak; a mutatók tulajdonképpen címek a 3.3. részben tárgyaltaknak megfelelően. Az indexet azért nevezzük „sűrűnek", mert az adatfájl valamennyi kulcsa megtalálható az indexben. Ezzel szemben a 4.1.3. részben tárgyalandó „ritka" index rendszerint egy adatblokkhoz egy kulcsot tartalmaz.

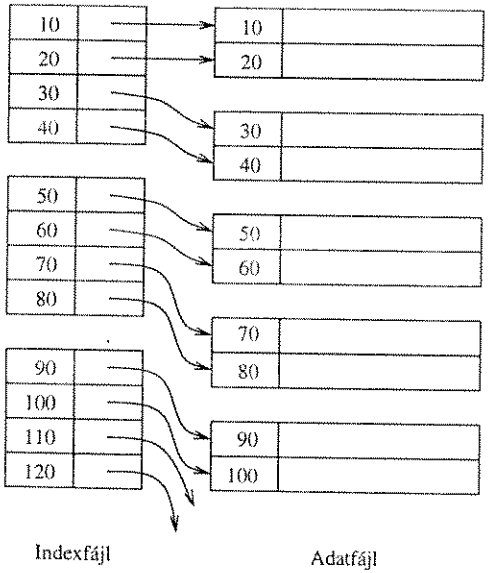
Kulcsok és még mindig kulcsok

A „kulcs" kifejezésnek több jelentése is van, és e könyvben a helyzettől függően valamennyi jelentését használjuk. Az olvasó számára bizonyára ismert a „kulcs" használata abban az értelemben, mint „egy reláció elsődleges kulcsa". Az ilyen kulcsokat SQL-ben deklaráljuk, és használatuk megköveteli, hogy a reláció nem tartalmazhat két olyan sort, amelyek megegyeznek az elsődleges kulcs attribútumán (attribútumain).

A 2.3.4. részben olvashattunk „rendezési kulcsokról", azokról az attribútumokról (illetve attribútumról), amelyek alapján egy rekordokból álló fájl rendezve van. Most „keresési kulcsokról" fogunk beszélni, azokról az attribútumokról (illetve attribútumról), amelyeknek értéket adunk, és egy index segítségével megkeressük a megfelelő értékekkel rendelkező sorokat. Abban az esetben, ha a „kulcs" jelentése nem világos, igyekszünk majd használni a megfelelő jelzőket – „elsődleges", „rendezési", illetve „keresési". A 4.1.2. és 4.1.3. részekben azonban sok esetben a háromtípusú kulcs egy és ugyanaz.



4.2. ábra. *Egy szekvenciális fájl*



4.3. ábra. *Sűrű index (balra) szekvenciális adatfájlon (jobbra)*

A sűrű index blokkjai ugyanolyan rendezett sorrendben tárolják a kulcsokat, mint az adatfájl maga. Mivel a kulcsok és mutatók feltehetően sokkal kevesebb helyet foglalnak, mint a teljes rekordok, ezért az index várhatóan jóval kevesebb blokkot használ majd, mint az eredeti fájl. Az index használata kiváltképpen akkor előnyös, amikor az adatfájl nem fér el az elsődleges memóriában, de az indexfájl igen. Ilyen esetben bármely rekord megtalálásához elegendő egyetlen lemez I/O-művelet.

4.1. példa: A 4.3. ábrán egy rendezett fájlon létrehozott sűrű indexet láthatunk. A rendezett fájl eleje ugyanaz, mint a 4.2. ábrán látható fájl. Az egyszerűség kedvéért feltételeztük, hogy a fájl a 10 további többszöröseit tartalmazó kulcsokkal folytatódik, habár a gyakorlatban nemigen számíthatunk ilyen szabályszerűséget követő kulcsokra. Feltételeztük továbbá, hogy egy indexblokkban csupán négy kulcs-mutató pár fér el. A gyakorlatban itt is más a helyzet, hiszen általában sokkal több ilyen pár fér el egy blokkban, meglehet, hogy több száz.

Az első indexblokkban azok a mutatók találhatók, amelyek az első négy rekordra mutatnak, a másodikban azok, amelyek a következő négy rekordra mutatnak és így tovább. A 4.1.6. részben olvashatunk majd azokról az okokról, amelyek miatt a gyakorlatban esetleg nem akarjuk majd teljesen kitölteni valamennyi indexblokkot. □

A sűrű index támogatja az olyan típusú lekérdezéseket, amelyek adott keresési-kulcs-értékkel rendelkező rekordokat keresnek. Adott K kulcsérték esetén megkeressük a K értékhez tartozó indexblokkokat, és amikor megtaláltuk, akkor követjük a K kulcshoz tartozó mutatót, amely a K kulcsú rekordra mutat. Úgy tűnhet, mintha a K megtalálásához az index valamennyi blokkját meg kellene vizsgálnunk, vagy átlagosan a blokkok felét. Van azonban néhány tényező, amely az index alapú keresést lényegesen hatékonyabbá teszi, mint ahogyan az első ránézésre tűnhet.

1. Az indexblokkok száma az adatblokkok számához képest rendszerint kicsi.
2. Mivel a kulcsok rendezettek, a K megtalálásához használhatunk bináris keresést. Ha n darab indexblokkunk van, akkor csupán $\log_2 n$ blokkot kell végignéznünk.
3. Az index olyan kicsi is lehet, hogy állandóan elsődleges memóriapufferekben tarthatjuk. Ha ez így van, akkor a K kulcs megtalálásához egyetlen elsődleges memória-hozzáféréshez van csak szükség, és ily módon elmaradnak a költséges lemez I/O-műveletek.

4.2. példa: Képzeljünk el egy 1 000 000 sorból álló relációt. Egy 4096 bájtból álló blokkban a reláció tíz sora fér el. Az adatok több mint 400 megabájt helyet foglalnak összesen, ami valószínűleg jóval több annál mintsem hogy beférjen az elsődleges memóriába. Feltételezzük azonban, hogy a kulcsmező mérete 30 bájt és a mutatók 8 bájtot foglalnak. Ésszerű blokkfejléc méretet feltételezve, 100 kulcs-mutató pár fér el egy 4096 bájtnyi blokkban.

Ily módon egy sűrű indexhez 10 000 blokk, azaz 40 megabájt szükséges. Ebben az esetben van rá esélyünk, hogy memóriapuffereket foglaljunk le ezekhez a blokkokhoz, attól függően, hogy mekkora az elsődleges memória mérete és ebből mennyi áll rendelkezésünkre. Továbbá, $\log_2(10\ 000)$ értéke körülbelül 13, tehát bináris kereséssel mindössze 13 vagy 14 blokkhoz kell hozzáférnünk ahhoz, hogy egy adott kulcsot megtaláljunk. Minden bináris keresés megtervezhető úgy, hogy a blokkoknak csak egy kis részhalmazához kelljen hozzáférni (a középső blokkhoz, az 1/4 és 3/4 pontoknál levő blokkokhoz, az 1/8, 3/8, 5/8 és 7/8 pontoknál levőkhöz és így tovább). Ily

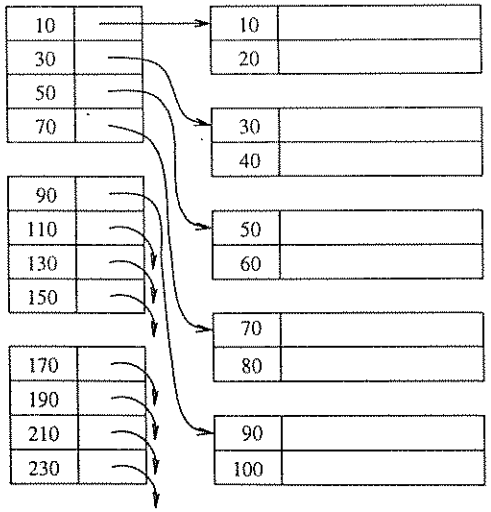
Indexblokkok helyének meghatározása

Mindeddig feltételeztük, hogy létezik valamilyen mechanizmus azon indexblokkok helyének meghatározására, amelyekből sűrű index esetén az adatfájl megfelelő sorai vagy ritka index esetén az adatfájl megfelelő blokkjai megtalálhatók. Az index helyének meghatározására több módszer is használható. Ha például az index kis helyen elfér, akkor tárolhatjuk a memória vagy a háttértároló előre rögzített részein. Ha az index nagyobb, akkor a 4.1.4. részben bemutatott módon készíthetünk fölé egy újabb indexet, és ezt tárolhatjuk a rögzített részekben. Ennek az ötletnek a végső kiterjesztését a 4.3. részben bemutatott B-fák jelentik, amelyek esetén elegendő csupán egyetlen úgynevezett gyökérblokk helyének az ismerete.

módon, ha nem is engedhetjük meg magunknak, hogy a teljes indexet a memóriában tartsuk, de a legfontosabb indexblokkok befférnek a memóriába, nos, még akkor is jelentősen kevesebb mint 14 lemez I/O-művelettel megtalálhatunk egy adott kulcsértékkel rendelkező rekordot. □

4.1.3. Ritka indexek

Ha a sűrű index túl nagy, akkor használhatjuk a *ritka indexnek* (sparse index) nevezett hasonló adatszerkezetet, amely kevesebb helyet foglal, de ennek az az ára, hogy valamivel több időt vesz igénybe egy adott kulcsértékkel rendelkező rekord megtalálása.



4.4. ábra. *Ritka index szekvenciális fájlon*

Egy ritka index egy adatblokkhoz csak egy kulcs-mutató párt tartalmaz, amint azt a 4.4. ábrán is láthatjuk. A kulcs az adatblokk első rekordjának a kulcsa.

4.3. példa: Ahogyan azt a 4.1. példában is tettük, feltételezzük, hogy az adatfájl rendezett és a kulcsok a tíz összes többszörösei valamely nagy számig bezáróan. Tegyük fel továbbá, hogy négy kulcs-mutató pár fér el egy indexblokkban. Ily módon az első indexblokk olyan bejegyzéseket tartalmaz, amelyek az első négy adatblokk első kulcsaihoz tartoznak, jelesül a 10-es, 30-as, 50-es és 70-es értékekhez. A második indexblokk a negyediktől nyolcadikig terjedő adatblokkok első kulcsaihoz tartozó bejegyzéseket tartalmazza, azaz, a kulcsok feltételezett szabályszerűségét folytatva, a 90-es, 110-es, 130-as és 150-es kulcsértékeket. Feltüntettük a harmadik indexblokkot is, amely a feltételezett kilencediktől tizenkettedikig terjedő adatblokkok első kulcsértékeit tartalmazza. □

4.4. példa: Egy ritka index sokkal kevesebb blokkot igényelhet, mint egy sűrű index. Ha a 4.2. példa sokkal életszerűbb paramétereit használjuk, azaz, hogy adott 100 000 adatblokk és 100 kulcs-mutató pár fér el egyetlen indexblokkban, akkor ritka index használata esetén mindössze 1000 indexblokkra van szükségünk. Ekkor az index mindössze négy megabájtot foglal le, ami jó eséllyel elhelyezhető majd az elsődleges memóriában.

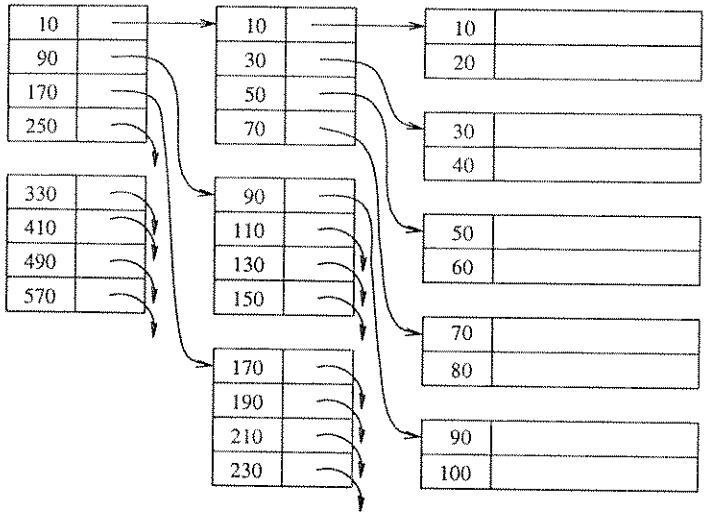
Másrészről azonban, egy sűrű index lehetővé teszi olyan típusú lekérdezések megválaszolását, mint „létezik-e K kulcsértékkel rendelkező rekord?" anélkül, hogy a rekordot tartalmazó blokkhoz hozzá kellene férni. Az a tény, hogy a K a sűrű indexben előfordul, garantálja a K kulcsértékű rekord létezését. Ugyanez a lekérdezés azonban ritka index használatával szükségessé tesz egy lemez I/O műveletet, amellyel beolvassuk azt a blokkot, amelyben a K *előfordulhat*. □

Ritka index esetén, egy K kulcsértékkel rendelkező rekord megtalálásához megkeressük azt a legnagyobb indexértéket, amely kisebb vagy egyenlő, mint a keresett K. Mivel az indexfájl rendezett a kulcs szerint, ismét használhatunk bináris keresést a megfelelő bejegyzés megtalálásához. Követjük az adatblokkra irányuló mutatót. Ekkor meg kell találnunk ebben a blokkban a K kulcsértékű rekordot. Természetesen a blokknak rendelkeznie kell elegendő információval a formátumra vonatkozóan ahhoz, hogy a rekordokat és azok tartalmát azonosítani lehessen. A 3.2. és 3.4. részekben bemutatott technikák bármelyikét használhatjuk, a helyzettől függően.

4.1.4. Többszintű indexelés

Amint azt a 4.2. és 4.4. példákban is láthattuk, egy index több blokkot is elfoglalhat. Ha ezek a blokkok nem egy előre rögzített helyen találhatók, például a háttértároló bizonyos cilinderein, akkor külön adatszerkezetre van szükségünk ahhoz, hogy megtaláljuk őket. Ha meg is tudjuk határozni az indexblokkok helyét, és bináris kereséssel a kívánt bejegyzést meg is találjuk, még akkor is igen sok lemez I/O-műveletre lehet szükség ahhoz, hogy a keresett rekordhoz hozzáférjünk.

Ha az indexre újabb indexet készítünk, akkor az első szintű index használatát még hatékonyabbá tehetjük. A 4.5. ábra kiterjeszti a 4.4. ábrát azzal, hogy egy második indexszintet ad hozzá (továbbra is feltételezzük, hogy a kulcsok a 10 többszörösei). Hasonló ötlettől vezérelve készíthetünk egy harmadik szintű indexet is a második szintre és így tovább. Ennek az ötletnek azonban megvannak a maga korlátai, így inkább a 4.3. részben ismertetett B-fákat részesítjük előnyben a többszintű indexek készítésekor.



4.5. ábra. *Második szintű ritka index készítése*

Ebben a példában az első szintű index ritka, habár választhattunk volna sűrű indexet is. A második és annál magasabb szintű indexek azonban kötelezően ritkák. Ennek oka az, hogyha az indexre egy sűrű indexet készítenénk, az ugyanannyi kulcs-mutató párt tartalmazna, mint az első szintű index, ezáltal ugyanakkora helyet is foglalna, mint az első szintű index. Ily módon a második szintű index egy újabb, de haszontalan adatszerkezet lenne csupán.

4.5. példa: Folytassuk a 4.4. példában használt reláció vizsgálatát. Tegyük fel, hogy készítünk egy második szintű indexet az első szintű ritka indexre. Az első szintű index 1000 blokkot foglal el és 100 kulcs-mutató pár fér el egy blokkban, ily módon a második szintű indexhez mindössze 10 blokkra van szükség.

Igen valószínű, hogy ez a 10 blokk elfér a memóriapufferben. Ha ez így van, akkor adott K kulcsértékű rekord megtalálásához a második szintű indexben megkeressük azt a legnagyobb kulcsértéket, ami kisebb vagy egyenlő, mint K. A megtalált mutatóval eljutunk az első szintű index egy olyan B blokkjához, amely minden bizonnyal elvezet a keresett rekordhoz. A B blokkot beolvassuk a memóriába, feltéve, hogy még nem olvastuk be. Ez az első lemez I/O-művelet. A B blokkban megkeressük azt a legnagyobb kulcsértéket, ami kisebb vagy egyenlő, mint K, és a megtalált kulcsérték megadja nekünk azt az adatblokkot, amely tartalmazza a K kulcsértékű rekordot, persze csak akkor, ha egyáltalán létezik ilyen rekord. Az adatblokk beolvasásához szükséges egy újabb lemez I/O-művelet. Ily módon mindössze két I/O-műveletet használtunk, és készen is vagyunk. □

4.1.5. Indexelés ismétlődő keresésikulcs-érték esetén

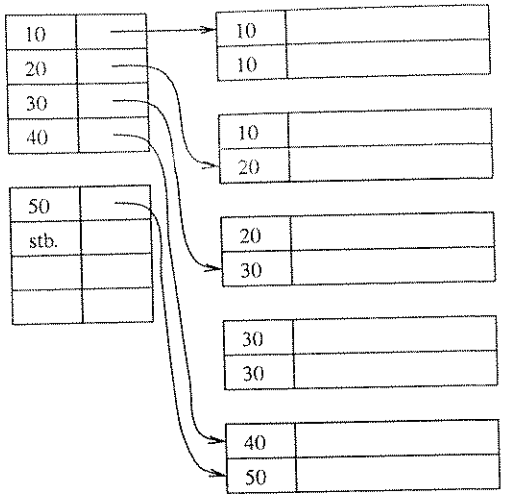
Mindeddig feltételeztük, hogy a keresési kulcs, amely az index alapját képezi, a reláció egy kulcsa, tehát adott kulcsértékhez legfeljebb egy rekord tartozhat. Gyakran használunk azonban indexeket nem kulcs attribútumokra is, így előfordulhat, hogy egy adott kulcsértékhez több mint egy rekord tartozik. Ha rendezzük a rekordokat a keresési kulcs szerint, az egyenlő kulcsértékkel rendelkező rekordokat tetszőleges sorrendben hagyva, akkor alkalmazhatjuk a korábbiakban bemutatott ötleteket olyan keresési kulcsokra is, amelyek nem kulcsai a relációnak.

Az előző ötletek talán legegyszerűbb kiterjesztése az, ha olyan sűrű indexet készítünk, amelyben az adatfájl valamennyi K keresésikulcs-értékkel rendelkező rekordjához tartozik egy K kulcsot tartalmazó bejegyzés. Ezzel tulajdonképpen engedélyeztük az ismétlődő keresési kulcsokat az indexfájlban. Az adott keresésikulcs-értékkel rendelkező valamennyi rekord megtalálása ily módon igen egyszerű: megkeressük az indexfájlban az első K értéket, ezáltal megtaláljuk a többi K értéket is, hiszen rögtön az első után helyezkednek el. Ezután követjük a megtalált kulcsokhoz tartozó mutatókat, eljutva ezzel a K keresésikulcs-értékkel rendelkező rekordokhoz.

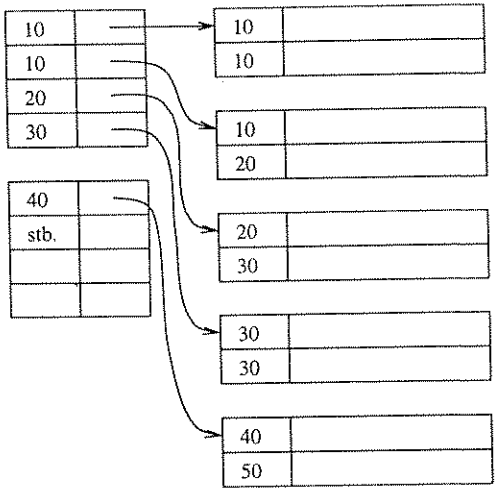
Némileg hatékonyabb az a megközelítés, amikor a sűrű indexben csak egy bejegyzés található valamennyi K keresési kulcshoz. A K kulcshoz olyan mutató tartozik, amely az első K értékkel rendelkező rekordra mutat. Ahhoz, hogy a többi K értékkel rendelkező rekordot is megtaláljuk, csupán el kell mozdulnunk előrefelé az adatfájlban, hiszen az adatfájl rendezett, és ezek a rekordok közvetlenül az első K érékű rekord után helyezkednek el. A 4.6. ábra ezt az ötletet mutatja be.

4.6. példa: Tegyük fel, hogy meg akarjuk találni a 4.6. ábrán az összes olyan rekordot, amelynek a keresési kulcsa 20. Megkeressük az indexben a 20-as értékhez tartozó bejegyzést, és követjük a hozzá tartozó mutatót, amely az első olyan rekordhoz vezet, amelyben a keresési kulcs értéke 20. Ezután elkezdünk előrefelé keresni az adatfájlban. Mivel a második blokk utolsó rekordján állunk, továbblépünk a harmadik blokkra.[1] Azt találjuk, hogy ennek a blokknak az első rekordja tartalmazza a 20-as kulcsértéket, a második rekord kulcsa azonban 30. Ezért aztán nem szükséges tovább keresni, megtaláltuk a 20-as kulcsértékkel rendelkező mindkét rekordot. □

[1] Ahhoz, hogy az adatfájl következő blokkját megtaláljuk, felfűzhetjük a blokkokat egy láncolt listára, például úgy, hogy valamennyi blokk végén elhelyezünk egy olyan mutatót, amelyik a következő blokkra mutat. Visszaléphetünk azonban az indexhez is, és követhetjük azt a mutatót, amely az adatfájl következő blokkjára mutat.



4.6. ábra. *Sűrű index, az ismétlődő keresési kulcsok megengedettek*



4.7. ábra. *Ritka index, amely a blokkok legkisebb keresési kulcsát tartalmazza*

A 4.7. ábrán egy ritka indexet láthatunk, amely a 4.6. ábrán látható adatfájlra épült. A ritka index meglehetősen hagyományos; kulcs-mutató párokat tartalmaz, az adatfájl valamennyi blokkjának első keresési kulcsainak megfelelően.

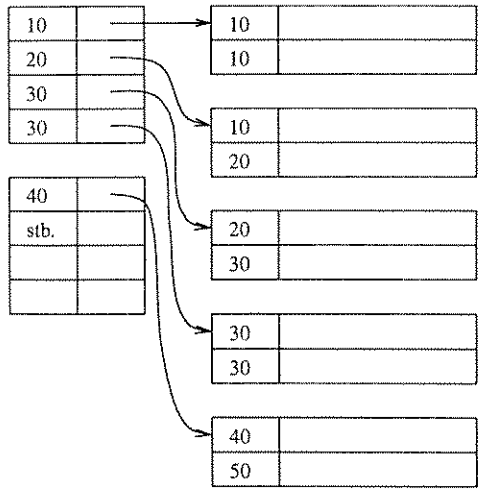
Ahhoz, hogy ezzel az adatszerkezettel megtaláljuk a K keresési kulccsal rendelkező rekordokat, meg kell keresnünk azt az utolsó bejegyzést az indexben, amelyben a

kulcs kisebb vagy egyenlő mint K. Nevezzük ezt a bejegyzést E_1-nek. Induljunk el az index eleje felé, és menjünk mindaddig, amíg el nem érünk az első bejegyzésig, vagy nem találunk egy olyan E_2 bejegyzést, amelyben a kulcs értéke szigorúan kisebb, mint K. Mindazon adatblokkok, amelyek tartalmazhatnak K keresési kulccsal rendelkező rekordot, elérhetők az E_2 és E_1 közötti bejegyzések mutatói segítségével, az E_1 bejegyzést is beleértve.

4.7. példa: Tegyük fel, hogy a 4.7. ábrán szemléltetett esetben szeretnénk megtalálni a 20-as kulcsértéket. Az első indexblokk harmadik bejegyzése az E_1; ez az utolsó olyan bejegyzés, ahol a kulcs ≤ 20. Amikor elkezdünk visszafelé keresni, rögtön találunk egy olyan bejegyzést, amelyben a kulcs kisebb, mint 20. Így, az E_2-nek az első indexblokk második bejegyzése felel meg. A bejegyzések két mutatója a második, illetve a harmadik adatblokkra mutat, és ez az a két blokk, amely tartalmaz 20-as keresésikulcs-értékkel rendelkező rekordokat.

Ha azonban például $K = 10$, akkor az E_1 az első indexblokk második bejegyzése, és az E_2 nem létezik, mivel nincs kisebb kulcsérték. Ily módon követjük az index valamennyi bejegyzésének mutatóit egészen a második bejegyzésig, azt is beleértve. Ezek a mutatók az első két adatblokkhoz vezetnek, ahol megtaláljuk az összes 10-es kulcsértékkel rendelkező rekordot. □

A 4.8. ábra egy némileg különböző helyzetet mutat be. Itt az egy adatblokkhoz tartozó indexbejegyzés a legkisebb *új* keresési kulcsot tartalmazza, azaz azt a legkisebb kulcsot, amely az előző blokkban nem szerepel. Ha egy blokkban nincs új keresési kulcs, akkor a hozzá tartozó indexbejegyzés az adott blokkban található keresési kulcs értékét tartalmazza. Ilyen feltételek mellett, a K keresésikulcs-értékkel rendelke-



4.8. ábra. *Ritka index, amely a blokkok legkisebb új keresési kulcsát tartalmazza*

ző rekordok megtalálásához meg kell keresnünk azt az első bejegyzést az indexben, amely:

a) egyenlő K-val, vagy
b) kisebb, mint K, de a következő kulcs nagyobb, mint K.

Követjük a bejegyzéshez tartozó mutatót, és ha a blokkban találunk legalább egy K keresésikulcs-értékkel rendelkező rekordot, akkor tovább keresünk előrefelé a következő blokkokban, egészen addig, amíg meg nem találjuk az összes K keresésikulcs-értékkel rendelkező rekordot.

4.8. példa: Tegyük fel, hogy a 4.8. ábrán látható esetben $K = 20$. A fenti szabály alapján az index második bejegyzését találjuk meg, amelynek mutatója elvezet minket az első olyan blokkhoz, amely tartalmaz 20-as kulcsértéket. Tovább kell keresnünk előrefelé, hiszen a következő blokkban is szerepel a 20.

Ha $K = 30$, a szabály alapján a harmadik bejegyzést találjuk meg. A bejegyzés mutatója a harmadik adatblokkhoz vezet, ahol a 30-as keresésikulcs-értéket tartalmazó rekordok kezdődnek. És végül, ha $K = 25$, akkor a kiválasztási szabály b) pontja alapján az index második bejegyzését találjuk meg. Ekkor a második adatblokkhoz jutunk. Ha léteznének olyan rekordok, amelyek keresési kulcsa 25, akkor ezen rekordok közül legalább az egyiknek ebben a blokkban kellene lenni a 20-as kulcsértékű rekordokat követve, hiszen tudjuk, hogy a harmadik adatblokk első új kulcsa 30. Mivel nincs 25-ös kulcsértékű rekord, keresésünk sikertelen. □

4.1.6. Indexek kezelése adatmódosításkor

Mindeddig úgy ábrázoltuk az adatfájlokat és az indexeket, mintha azok megfelelő típusú rekordokkal teljesen feltöltött blokkok sorozatából állnának. Mivel az adatok idővel változnak, várható, hogy rekordok kerüljenek beszúrásra, törlésre és néha módosításra. Következésképpen, egy olyan elrendezés, mint a szekvenciális fájl is változni fog, ily módon, ami egyszer elfért egyetlen blokkban, az többé már nem fog elférni. A 3.5. részben bemutatott technikákat használhatjuk az adatfájl újrarendezéséhez. Idézzük fel a 3.5. rész három fontos ötletét:

1. Hozzunk létre túlcsordulásblokkokat, amikor többlethelyre van szükségünk, vagy töröljünk túlcsordulásblokkokat, amikor elegendő rekord került törlésre, és nincs tovább szükség a helyre. A túlcsordulásblokkokhoz nem tartozik bejegyzés a ritka indexben. Sokkal inkább tekinthetők ezek a blokkok az elsődleges blokk kiterjesztéseinek.
2. A túlcsordulásblokkok helyett új blokkokat is beszúrhatunk a szekvenciális sorrendbe. Ha ezt tesszük, az új blokkhoz szükséges egy bejegyzés a ritka indexben. Emlékezzünk csak vissza, hogy egy index változása ugyanolyan problémákat okozhat az indexfájlban, mint a beszúrás és a törlés az adatfájlban. Ha új indexblokkokat hozunk létre, akkor ezeknek a blokkoknak meg kell tudnunk határozni valahogyan a helyét, például a 4.1.4. részben bemutatott újabb indexszint készítésével.
3. Ha már nincs hely, hogy beszúrjunk egy sort egy adott blokkba, akkor átcsúsztathatunk sorokat a szomszédos blokkokba. Fordítva, ha a szomszédos blokkok túl üressé válnak, akkor összevonhatjuk őket.

Azonban ha az adatfájlban változások állnak be, akkor az indexet is gyakran kell változtatni, hogy alkalmazkodjon a változásokhoz. A helyes megközelítés attól függ, hogy az index sűrű vagy ritka, és hogy az előbb tárgyalt három művelet közül melyiket használjuk. Azonban egy általános elvet megemlíthetünk:

- Az indexfájl tulajdonképpen egy speciális szekvenciális fájl; a kulcs-mutató párokat kezelhetjük úgy mint keresési kulcs szerint rendezett rekordokat. Ily módon módosításkor ugyanazokat a stratégiákat használhatjuk az indexfájlok esetén is, mint amit az adatfájlokra használunk.

A 4.9. ábrán összefoglaltuk azokat a tevékenységeket, amelyeket a ritka, illetve a sűrű indexen végre kell hajtanunk az adatfájlon elvégzett hét különböző művelet esetén. Ez a hét művelet magában foglalja üres túlcsordulásblokkok létrehozását és törlését, üres blokkok létrehozását és törlését a szekvenciális fájlban, rekordok beszúrását, törlését és mozgatását. Ne feledjük, hogy elfogadtuk: csak üres blokkokat lehet létrehozni, illetve törölni. Abban az esetben, ha olyan blokkot akarunk törölni, amely tartalmaz rekordokat, előbb törölnünk kell a rekordokat vagy másik blokkba kell őket áthelyeznünk.

Művelet	*Sűrű index*	*Ritka index*
Üres túlcsordulásblokk létrehozása	semmi	semmi
Üres túlcsordulásblokk törlése	semmi	semmi
Üres szekvenciális blokk létrehozása	semmi	beszúrás
Üres szekvenciális blokk törlése	semmi	törlés
Rekord beszúrása	beszúrás	módosítás (?)
Rekord törlése	törlés	módosítás (?)
Rekord mozgatása	módosítás	módosítás (?)

4.9. ábra. *A szekvenciális fájlon végzett műveletek hatásai az indexfájlra*

A táblázatban a következőket vehetjük észre.

- Üres túlcsordulásblokk létrehozása, illetve törlése nincs hatással egyik típusú indexre sem. A sűrű indexet azért nem érinti, mert az indexrekordokra vonatkozik. A ritka indexet pedig azért nem érinti, mert ritka indexben csak az elsődleges blokkokhoz tartozik bejegyzés, a túlcsordulásblokkokhoz nem.
- A szekvenciális fájl blokkjainak létrehozása, illetve törlése nem befolyásolja a sűrű indexet, az ok ismét az, hogy ez az indexrekordokra vonatkozik, nem blokkokra.

Felkészülés az adatok változására

Mivel a relációk és az osztálykiterjedések mérete idővel általában nő, gyakran bölcs dolog pluszhelyeket szétosztani a blokkok között, adatblokkok és indexblokkok között egyaránt. Ha a blokkok telítettsége kezdetben mondjuk 75%, akkor működhet a rendszerünk egy kevés ideig, mielőtt túlcsordulásblokkot kellene létrehoznunk, vagy rekordokat kellene átcsúsztatnunk blokkok között. Ha nincs túlcsordulásblokkunk, vagy csak kevés van, annak az az előnye, hogy az átlagos rekordhozzáféréshez mindössze egyetlen lemez I/O-műveletre van szükség. Minél több a túlcsordulásrekord, annál több blokkot kell megvizsgálnunk egy bizonyos rekord megtalálásához.

Befolyásolja azonban a ritka indexet, mivel a létrehozott vagy törölt blokkhoz létre kell hoznunk, illetve meg kell szüntetnünk egy indexbejegyzést.

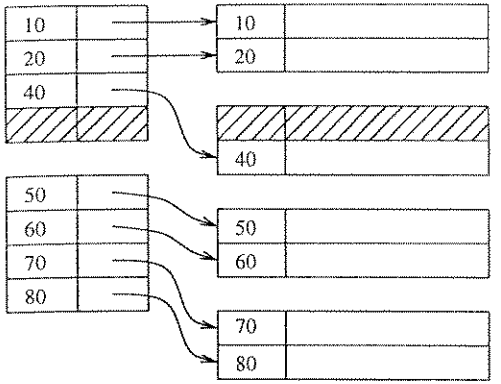
- Rekordok beszúrása és törlése éppolyan hatással van a sűrű indexre, mintha az adott rekordhoz egy kulcs-mutató párt szúrnánk be vagy törölnénk. Nincs azonban általában hatással a ritka indexre. Kivétel persze, ha az adott rekord éppen egy blokk első helyén áll, ilyenkor a ritka index megfelelő kulcsértékét módosítani kell. Ezért tettünk kérdőjelet a 4.9. ábrán a táblázat megfelelő soraiba, jelezvén, hogy módosítás lehetséges, de nem biztos.
- Hasonlóképpen, egy rekord mozgatása mindenképpen módosítást igényel a sűrű indexben, függetlenül attól, hogy a rekordot blokkok között vagy egy blokkon belül mozgattuk el. A ritka indexet viszont csak akkor érinti, ha a mozgatott rekord egy blokk első rekordja volt vagy éppen a mozgatás által került első helyre.

Azokat az algoritmusokat, amelyekre e szabályok céloznak, példákon keresztül mutatjuk be. Ezek a példák érintik a ritka és sűrű indexeket éppúgy, mint a „rekordok csúsztatását" és a túlcsordulásblokk alkalmazását.

4.9. példa: Vizsgáljuk meg először egy rekord szekvenciális fájlból történő törlését sűrű index esetén. Vegyük a 4.3. ábrán látható fájlt és indexet. Tegyük fel, hogy a 30-as kulcsértékű rekord törlésre kerül. A törlés eredményét a 4.10. ábrán láthatjuk.

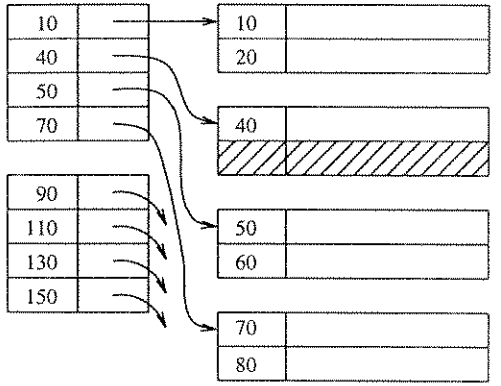
Először is töröljük a 30-as kulcsértéket tartalmazó rekordot a szekvenciális fájlból. Feltételezzük, hogy a blokk rekordjaira blokkon kívüli mutatók mutatnak, ily módon arra a döntésre jutottunk, hogy a blokk megmaradó rekordját nem csúsztatjuk előrébb a blokkban. Ehelyett inkább egy törlésre utaló jelet, egy úgynevezett sírkövet hagyunk a 30-as kulcsértékű rekord helyén.

Az indexben töröljük a 30-ashoz tartozó kulcs-mutató párt. Feltételezzük, hogy az indexrekordokra kívülről nem mutatnak mutatók, ily módon nem szükséges sírkövet tennünk a törölt pár helyére. Megvan tehát a lehetőségünk arra, hogy az indexblokkot konszolidáljuk, és a rekordokat előrébb csúsztassuk. □



4.10. ábra. *A 30-as keresésikulcs-értékű rekord törlése sűrű index esetén*

4.10. példa: Lássunk most két törlést egy ritka indexű fájlból. A 4.4. ábrával dolgozunk. Tegyük fel, hogy ismét a 30-as kulcsértékű rekord kerül törlésre. Feltételezzük, hogy nincs akadálya a rekordok blokkon belüli csúsztatásának, vagy azért, mert nincs olyan mutató, amely ezekre a rekordokra mutatna, vagy pedig azért, mert az ilyen csúsztatások támogatására a 3.17. ábrán bemutatott eltolásiérték-táblát használjuk.

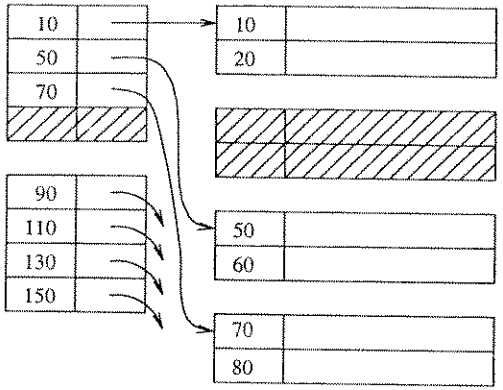


4.11. ábra. *A 30-as keresésikulcs-értékű rekord törlése ritka index esetén*

A 30-ast tartalmazó rekord törlésének eredményét a 4.11. ábrán láthatjuk. A rekord törlésre került, és a következő rekord, amelyik a 40-est tartalmazza, előrébb csúszott, konszolidálva ezáltal a blokk elejét. Mivel most a 40-es lett a második adatblokk első kulcsa, módosítanunk kell ennek a blokknak indexrekordját. A 4.11. ábrán láthatjuk, hogy a második adatblokkra utaló mutatóhoz tartozó kulcs értékét 30-asról 40-esre módosítottuk.

Tegyük most fel, hogy a 40-es kulcsértékű rekord szintén törlésre kerül. Ennek a

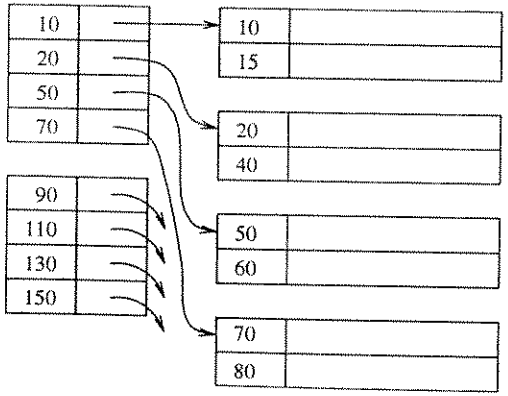
műveletnek az eredményét a 4.12. ábrán láthatjuk. A második adatblokkban nincs már egy rekord sem. Ha a szekvenciális fájl tetszőleges blokkokon van tárolva (és mondjuk nem egy adott cilinder egymás utáni blokkjain), akkor felfűzhetjük a nem használt blokkot a szabad helyek listájára.



4.12. ábra. *A 40-es keresésikulcs-értékű rekord törlése ritka index esetén*

A 40-est tartalmazó rekord törlését az index új viszonyokhoz történő beállításával fejezzük be. Mivel a második adatblokk többé nem létezik, töröljük a hozzá tartozó bejegyzést az indexből. A 4.12. ábrán azt is láthatjuk, hogy az első indexblokkot konszolidáltuk azáltal, hogy a törölt bejegyzés utáni rekordokat előrébb csúsztattuk. Ez a lépés opcionális. □

4.11. példa: Vizsgáljuk most meg egy beszúrás hatását. Induljunk ki a 4.11. ábrából, ahol egy ritka indexű fájlból éppen kitöröltük a 30-as rekordot, de a 40-es rekord



4.13. ábra. *Beszúrás ritka indexű fájlba, azonnali újrarendezést használva*

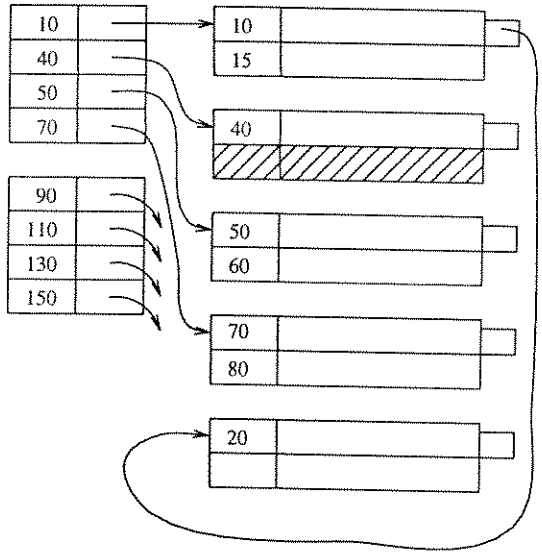
megmaradt. Most be fogunk szúrni egy 15-ös kulcsértékű rekordot. A ritka indexet megvizsgálva arra jutunk, hogy ez a rekord az első adatblokkhoz tartozik. Ez a blokk viszont tele van: a 10-es és 20-as rekordok vannak benne.

Az egyik lehetőség, hogy keresünk egy olyan szomszédos blokkot, amelyben van üres hely, ebben az esetben a második adatblokk ilyen. Ekkor hátracsúsztatjuk a rekordokat a fájlban, helyet készítve ezzel a 15-ös rekordnak. Az eredményt a 4.13. ábrán láthatjuk. A 20-as rekord átkerült a második adatblokkba, és a 15-ös rekord került a helyére. Ahhoz, hogy a 20-as rekord beférjen a második blokkba, és a rekordok rendezettsége is megmaradjon, a 40-es rekordot hátrébb csúsztattuk a második blokkban, és a 20-as blokkot helyeztük eléje.

Az utolsó lépés az, hogy módosítjuk a megváltozott blokkok indexbejegyzéseit. Szükség lehetne az első blokkhoz tartozó bejegyzés kulcsának megváltoztatására, de nem ebben az esetben, hiszen a beszúrt rekord nem került a blokk első helyére. Meg kell azonban változtatnunk a második adatblokkhoz tartozó indexbejegyzést, mivel ennek a blokknak az első rekordja a 40-es volt, de most a 20-as. □

4.12. példa: A 4.11. példában bemutatott stratégiával az a gond, hogy csak a szerencsén múlt, hogy találtunk üres helyet egy szomszédos adatblokkban. Ha a 30-as rekordot előzőleg nem töröltük volna, akkor mindhiába kerestünk volna üres helyet. Elvben a 20-as rekordtól kezdve minden egyes rekordot el kellett volna csúsztatnunk a fájl vége felé, egészen addig, amíg el nem értük volna a fájl végét és lett volna lehetőség újabb blokk létrehozására.

Éppen e kockázat miatt gyakran bölcsebb dolog engedélyezni a túlcsordulásblok-



4.14. ábra. *Beszúrás ritka indexű fájlba, túlcsordulásblokkokat használva*

kokat az olyan elsődleges blokkok kiegészítésére, amelyben túl sok a rekord. A 4.14. ábrán láthatjuk a 4.11. ábra szerkezetébe történő, 15-ös kulcsértékű rekord beszúrásának a hatását. Éppúgy, mint a 4.11. példában, az első adatblokk itt is túl sok rekordot tartalmaz. Ahelyett, hogy átcsúsztatnánk rekordokat a második blokkba, inkább készítünk egy túlcsordulásrekordot ehhez az adatblokkhoz. A 4.14. ábrán valamennyi rekordon egy „kinövés" látható, amely a blokkfejléc azon helyét ábrázolja, ahová elhelyezhető egy olyan mutató, amely túlcsordulásblokkra mutat. Akárhány túlcsordulásblokkot fel lehet fűzni ezen mutatóhelyek használatával.

A példánkban a 15-ös rekord az őt megillető helyre kerül, a 10-es rekord után. A 20-as rekord átcsúszik a túlcsordulásblokkba, hogy legyen hely a beszúrásra. Az indexben nincs szükség változtatásokra, hiszen az első adatblokk első rekordja nem változott. Ne feledjük, hogy a túlcsordulásblokkhoz nem készül indexbejegyzés. A túlcsordulásblokk az első adatblokk kiterjesztésének számít, nem pedig a szekvenciális fájl saját adatblokkjának. □

4.1.7. Feladatok

* **4.1.1. feladat:** Tegyük fel, hogy egy blokkban elfér 3 rekord vagy 10 kulcs-mutató pár. Ha n a rekordok száma, akkor hány blokkra van szükség az n függvényében az adatfájl és az indexfájl tárolásához:

a) sűrű index esetén,
b) ritka index esetén?

4.1.2. feladat: Ismételjük meg a 4.1.1. feladatot arra az esetre, ha egy blokkban 30 rekord vagy 200 kulcs-mutató pár fér el, de sem az adatblokkok, sem az indexblokkok telítettsége nem lehet több, mint 80%.

! **4.1.3. feladat:** Ismételjük meg a 4.1.1. feladatot arra az esetre, ha több indexszintet is használhatunk, egészen addig, míg az utolsó indexszint mindössze egyetlen blokkot foglal el.

*!! **4.1.4. feladat:** Tegyük fel, hogy egy blokkban elfér 3 rekord vagy 10 kulcs-mutató pár, ugyanúgy, mint a 4.1.1. feladatban, de az ismétlődő keresési kulcsok megengedettek. Hogy pontosabbak legyünk, az összes keresési kulcs 1/3-a egyetlen rekordban jelenik meg, másik 1/3-a pontosan két rekordban, és az utolsó 1/3-a pontosan három rekordban jelenik meg. Tegyük fel, hogy sűrű indexünk van, de egy keresésikulcsértékhez csak egy kulcs-mutató pár tartozik, amelynek mutatója az első olyan rekordra mutat, amely az adott kulcsot tartalmazza. Számítsuk ki egy adott K keresésikulcsértékkel rendelkező valamennyi rekord megtalálásához szükséges lemez I/O-műveletek átlagos számát, feltéve, hogy kezdetben egyetlen blokk sincs betöltve a memóriába. Feltételezhetjük, hogy a K kulcsot tartalmazó indexblokk helye ismert, noha a lemezen található.

! **4.1.5. feladat:** Ismételjük meg a 4.1.4. feladatot, ha:

a) Sűrű indexünk van, és valamennyi kulcsértékhez tartozik egy kulcs-mutató pár, beleértve az ismétlődő kulcsokat is.
b) Ritka indexünk van, amely az adatblokkok legkisebb kulcsaira mutat úgy, ahogyan a 4.7. ábrán látható.
c) Ritka indexünk van, amely az adatblokkok legkisebb *új* kulcsaira mutat úgy, ahogyan a 4.8. ábrán látható.

! **4.1.6. feladat:** Ha van egy sűrű indexünk a reláció elsődleges kulcs attribútumára, akkor lehetséges, hogy a sorokra (illetve a sorokat reprezentáló rekordokra) utaló mutatók az indexbejegyzésre mutassanak ahelyett, hogy magukra a rekordokra mutatnának. Milyen előnnyel jár az egyik, illetve a másik megközelítés?

4.1.7. feladat: Folytassuk a 4.13. ábrán elkezdett változtatásokat, abban az esetben, ha előbb töröljük a 60-as, 70-es és 80-as kulcsértékű rekordokat, majd beszúrunk 21-es, 22-es, 23-as, 24-es, 25-ös, 26-os, 27-es, 28-as és 29-es kulcsértékű rekordokat. Tegyük fel, hogy a szükséges hely előteremtéséhez:

* a) Túlcsordulásblokkokat készítünk az adatfájlhoz éppúgy, mint az indexfájlhoz.
b) Olyan távolra csúsztatjuk a rekordokat, amennyire csak szükséges, újabb blokkokat az adatfájl, illetve az indexfájl végéhez csatolhatunk, ha szükséges.
c) Szükség esetén a fájlok közepére szúrhatunk be új adat-, illetve indexblokkokat.

*! **4.1.8. feladat:** Tegyük fel, hogy az n rekordból álló adatfájlba történő beszúrást úgy oldjuk meg, hogy szükség esetén túlcsordulásblokkokat hozunk létre. Tegyük fel továbbá, hogy az adatblokkok átlagosan félig vannak telítve. Ha az új rekordok beszúrása véletlenszerű, hány rekordot kell beszúrnunk ahhoz, hogy egy adott kulccsal rendelkező rekord megtalálásához az átlagosan megvizsgálandó adatblokkok (beleértve a túlcsordulásblokkokat is) száma elérje a 2-t? Tegyük fel, hogy egy keresésnél először azt a blokkot nézzük meg, amelyre az index mutat. Utána sorban megnézzük a túlcsordulásblokkokat is, egészen addig, míg meg nem találjuk a keresett rekordot, amely egész biztosan a lánc valamelyik blokkjában található.

4.2. Másodlagos indexek

A 4.1. részben bemutatott adatszerkezeteket *elsődleges indexeknek* (primary index) nevezzük, mivel meghatározzák az indexelt rekordok helyét. A 4.1. részben a helyet az a tény határozta meg, hogy az index alapjául szolgáló fájl rendezett volt a keresési kulcs szerint. A 4.4. részben az elsődleges kulcs egy másik gyakori példáját láthatjuk majd: a tördelőtáblázatokat, amelyekben a keresési kulcs meghatározza azt a „kosarat", amelyikbe a rekord tartozik.

Gyakori azonban, hogy egy reláción több indexet is szeretnénk, hogy ezáltal gyor-

sabbá váljanak bizonyos lekérdezések. Vegyük például elő ismét a 3.1. ábrán deklarált `Filmszínész` relációt. Mivel úgy deklaráltuk, hogy a `név` elsődleges kulcs legyen, várható, hogy a relációs adatbázis-kezelő készít egy elsődleges kulcsot a színész nevére vonatkozó lekérdezések támogatására. Tegyük fel továbbá, hogy adatbázisunkat arra is fel szeretnénk használni, hogy gratuláljunk a sztároknak a kerek születésnapokon. Lehet, hogy futtatunk olyan lekérdezéseket, mint a következő:

```
SELECT név, cím
FROM Filmszínész
WHERE születési_idő = DATE '1950-01-01';
```

Ahhoz, hogy segítsük az ilyen jellegű lekérdezéseket, szükségünk van egy *másodlagos indexre* a `születési_idő` attribútumon. Egy SQL alapú rendszerben a következőhöz hasonló explicit utasítás segítségével hozhatunk létre ilyen indexet:

```
CREATE INDEX BDIndex ON Filmszínész(születési_idő);
```

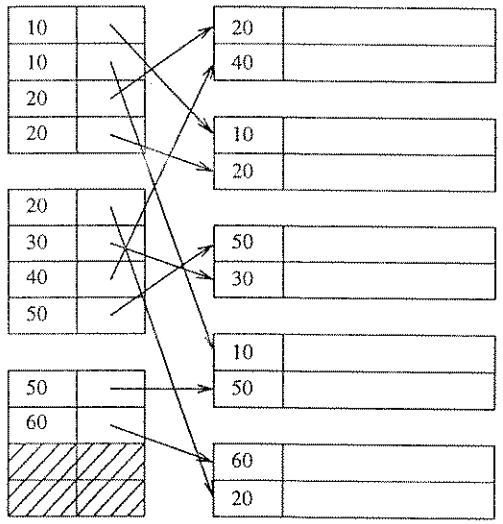
Egy másodlagos index megfelel bármely index céljainak: olyan adatszerkezet, amely megkönnyíti megadott értékű mezővel, illetve mezőkkel rendelkező rekordok megtalálását. A másodlagos index abban különbözik azonban az elsődleges indextől, hogy a másodlagos index nem határozza meg a rekordok adatfájlban elfoglalt helyét. Ehelyett a másodlagos index a rekordok aktuális helyét adja meg; ez a hely függhet valamely más mezőre vonatkozó elsődleges indextől. Az elsődleges és másodlagos indexek különbözőségének egy érdekes következménye az, hogy:

- Nincs értelme ritka, másodlagos indexről beszélni. Mivel a másodlagos index nem befolyásolja a rekordok elfoglalt helyét, ily módon nem is használhatjuk olyan rekord helyének megjósolására, amelynek kulcsa nem szerepel explicit módon az indexfájlban.
- Következésképpen, a másodlagos index mindig sűrű.

4.2.1. Másodlagos indexek tervezése

A másodlagos index sűrű index, amely általában tartalmaz ismétlődéseket. Éppúgy, mint az eddigiekben, az index kulcs-mutató párokból áll; a „kulcs" egy keresési kulcs, és nem szükséges, hogy egyedi legyen. Az indexfájlban levő párok rendezettek a kulcsérték szerint azért, hogy megkönnyítsék az adott kulccsal rendelkező bejegyzések megtalálását. Ha szeretnénk létrehozni ezen az adatszerkezeten egy második szintű indexet, akkor ez az index ritka lenne a 4.1.4. részben részletezett okok miatt.

4.13. példa: A 4.15. ábra egy jellegzetes másodlagos indexet mutat be. Az adatfájlban két rekord szerepel egy blokkban úgy, ahogyan az eddigi példákban is. A rekordoknak csak a keresési kulcsát tüntetjük fel; ezek az értékek olyan egész számok, amelyek a



4.15. ábra. *Másodlagos index*

10 többszörösei éppúgy, mint eddig. Vegyük észre, hogy itt az adatfájl nem rendezett a keresési kulcs szerint, nem úgy, mint a 4.1.5. részben.

Az indexfájlban azonban *rendezettek* a kulcsok. Ennek az az eredménye, hogy az egy indexblokkból kiinduló mutatók több különböző adatblokkra mutatnak ahelyett, hogy egy vagy néhány egymás utáni blokkra mutatnának. Ahhoz például, hogy megkeressük az összes 20-as keresésikulcs-értékkel rendelkező rekordot, nem elég megnézni két indexblokkot, hanem el kell zarándokolnunk ahhoz a három adatblokkhoz, amire a megfelelő mutatók hivatkoznak. Így a másodlagos index használata jóval több lemez I/O-műveletet eredményezhet, mintha ugyanannyi rekordot elsődleges index segítségével kellene megkapnunk. Erre a problémára azonban nincs megoldás, hiszen nem befolyásolhatjuk az adatblokkban levő sorok sorrendjét, hiszen azok valószínűleg már rendezve vannak egy vagy több másik attribútum szerint.

A 4.15. ábrához hozzáadhatnánk egy második szintű indexet. Ez a szint ritka lenne, és a 4.1.4. résznek megfelelően a párok az indexblokkok első vagy első új kulcsára hivatkoznának. □

4.2.2. Másodlagos indexek alkalmazása

A másodlagos indexek támogatják további indexek használatát a szekvenciális fájlokba szerveződő relációkon (illetve osztálykiterjedéseket). Ezenkívül azonban bizonyos adatszerkezetek esetén az elsődleges kulcs számára is másodlagos indexekre van szükség. Az egyik ilyen adatszerkezet a kupacszerkezet, amelyben a rekordok tárolása mindenféle rendezettség nélkül történik.

A második gyakori szerkezet, amelynek másodlagos indexre van szüksége, a *nyalábolt fájl* (clustered file). Az ilyen adatszerkezetben két vagy több relációt tárolunk oly módon, hogy a rekordok össze vannak keveredve. Példán keresztül fogjuk bemutatni, hogy bizonyos esetekben miért is lehet értelme az ilyen elrendezésnek.

4.14. példa: Tegyük fel, hogy van két relációnk. A relációk sémáit röviden a következőképpen írhatjuk le:

```
Film(cím, év, hossz, stúdiónév)
Stúdió(név, cím, elnök)
```

A `cím` és az `év` attribútumok együtt kulcsot alkotnak a `Film` relációban, a `név` viszont a `Stúdió` reláció kulcsa. A `Film` reláció `stúdiónév` attribútuma idegen kulcs és a `Stúdió` reláció `név` attribútumára utal. A továbbiakban feltételezzük, hogy a következő típusú lekérdezés gyakori:

```
SELECT cím, év
FROM Film
WHERE stúdiónév = 'zzz';
```

Ebben az esetben a `zzz` egy konkrét stúdió nevét hivatott reprezentálni, például azt, hogy `'Disney'`.

Ha meg vagyunk róla győződve, hogy a fenti lekérdezés tipikus, akkor ahelyett, hogy a `Film` sorait az elsődleges kulcs szerint rendeznénk (`cím` és `év`), rendezhetjük a sorokat a `stúdiónév` szerint. Ezután készíthetünk erre a szekvenciális fájlra egy ismétlődéseket megengedő elsődleges kulcsot úgy, ahogyan a 4.1.5. részben láthattuk. Ez azért jó, mert ha egy adott stúdióban készült filmekről szeretnénk információt lekérdezni, akkor a válasz sorai néhány blokkban találhatók majd, valószínűleg eggyel több blokkban, mint amennyi minimum szükséges lenne a tárolásukhoz. Ezzel minimizáljuk a lekérdezéshez szükséges lemez I/O-műveletek számát, sokkal hatékonyabbá téve ezáltal a lekérdezés végrehajtását.

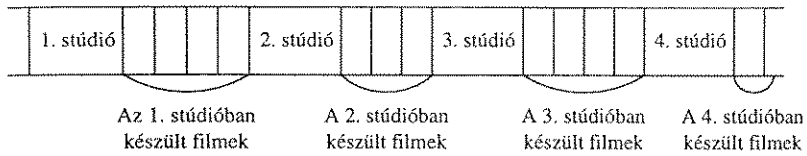
A `Film` reláció sorainak puszta rendezése egy, az elsődleges kulcstól különböző attribútum szerint, azonban nem segít abban az esetben, ha össze akarjuk kapcsolni a filmekről tárolt információkat a stúdiókról tárolt információkkal. Jó példa erre a következő lekérdezés, amelyben azt szeretnénk megtudni, hogy ki az elnöke annak a stúdiónak, amely a „Csillagok háborúja" című filmet készítette.

```
SELECT elnök
FROM Film, Stúdió
WHERE cím = 'Csillagok háborúja' AND
   Film.stúdiónév = Stúdió.név;
```

A következő lekérdezéssel a Hollywoodban készült összes filmet keressük:


```
SELECT cím, év
FROM Film, Stúdió
WHERE cím LIKE '%Hollywood%' AND
   Film.stúdiónév = Stúdió.név;
```

Ha biztosak vagyunk abban, hogy a `Film` és `Stúdió` relációk stúdiónév alapján történő összekapcsolása gyakori lesz, akkor hatékonnyá tehetjük ezeket az összekapcsolásokat azzal, hogy *nyalábolt fájl adatszerkezetet* választunk, ahol a `Film` sorai ugyanabban a blokksorozatban helyezkednek el, mint a `Stúdió` sorai. Pontosabban fogalmazva, mindegyik `Stúdió` sor után a `Film` reláció azon sorai foglalnak helyet, amely filmeket az adott stúdióban gyártottak. A szabályszerűséget a 4.16. ábra szemlélteti.



4.16. ábra. *Egy nyalábolt fájl, amelyben mindegyik stúdió össze van nyalábolva az általa készített filmekkel*

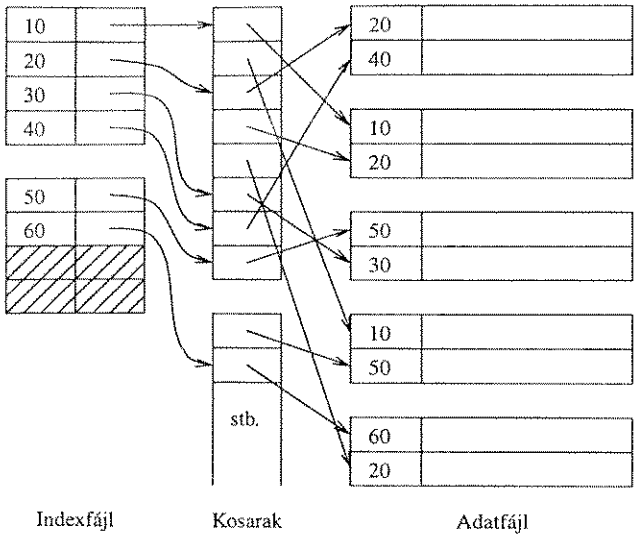
Ebben az esetben, ha annak a stúdiónak az elnökét keressük, amely egy adott filmet készített, jó esélyünk van arra, hogy a stúdióra és a filmre vonatkozó rekordok ugyanabban a blokkban találhatók, megtakarítva ezzel egy lemez I/O-műveletet. Ha azokat a filmeket keressük, amelyeket egy adott stúdió gyártott, szintén jó esélyünk van arra, hogy ezek a filmek ugyanabban a blokkban legyenek, mint a stúdió, ami szintén I/O-megtakarítást jelent.

Ha a fenti lekérdezések hatékonysága is fontos szempont, akkor egy adott filmet vagy stúdiót hatékonyan kell tudnunk megtalálni. Éppen ezért, ahhoz, hogy megtaláljuk a keresett filmet (vagy filmeket, hiszen több film is készülhet ugyanazzal a címmel), szükségünk van egy másodlagos indexre a `Film.cím` attribútumon, hiszen a keresett sor(ok) bárhol lehetnek a `Film` és `Stúdió` sorait tartalmazó blokkokban. Ahhoz, hogy egy adott stúdióhoz tartozó sort megtaláljunk, a `Stúdió.név` attribútumon is szükség van egy indexre. □

4.2.3. Közvetett másodlagos indexek

A 4.15. ábrán látható adatszerkezetben van némi fölösleg, amely talán jelentős méretű pazarlás. Ha egy keresési kulcs *n*-szer jelenik meg az adatfájlban, akkor az indexfájlba is *n*-szer fog bekerülni ez az érték. Jobb lenne, ha az összes olyan mutatóhoz, amely az adott kulcsértékű rekordra mutat, a kulcsértéket csak egyszer kellene beírnunk az indexbe.

Az ismétlődő értékek elkerülésének egy kényelmes módja az, ha beiktatunk egy *kosaraknak* (bucket) nevezett közvetett réteget a másodlagos indexfájl és az adatfájl



4.17. ábra. *Helymegtakarítás másodlagos indexben közvetett szint használatával*

közé. A 4.17. ábrán láthatjuk, hogy valamennyi K keresési kulcshoz egyetlen kulcs-mutató pár tartozik. A mutató a „kosárfájl" azon pozíciójára mutat, amely a K-hoz tartozó „kosarat" tartalmazza. E pozíció után egészen addig a pozícióig, amelyre az indexfájl következő kulcs-mutató párja mutat, azok a mutatók állnak, amelyek elvezetnek a K kulcsértékű összes rekordhoz.

4.15. példa: Kövessük például az indexfájl 50-es keresésikulcs-értékétől induló mutatóját a közbeeső „kosárfájlig". Ez a mutató történetesen a kosárfájl első blokkjának utolsó mutatójához vezet bennünket. Továbbmegyünk a következő blokk első mutatójához. Itt megállunk, hiszen az indexfájl következő mutatója, amely a 60-as keresésikulcs-értékhez tartozik, éppen a kosárfájl második blokkjának második mutatójára mutat. □

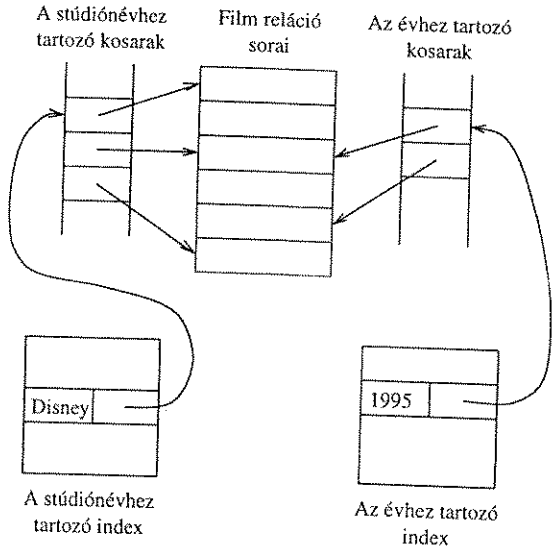
A 4.17. ábrán látható elrendezés mindaddig helymegtakarítást jelent, amíg a keresésikulcs-értékek több helyet foglalnak, mint a mutatók, és mindegyik kulcs átlagosan legalább kétszer megjelenik. A közvetett másodlagos indexek használatának akkor is van azonban egy fontos előnye, amikor a kulcsok és a mutatók mérete összemérhető: a lekérdezésekhez használhatjuk a kosarakban található mutatókat, ily módon nem szükséges végignéznünk az adatfájl összes rekordját egy lekérdezés megválaszolásához. Speciálisan, ha egy lekérdezés több feltételt is tartalmaz, és valamennyi feltételhez létezik egy másodlagos index, akkor az összes feltételt kielégítő rekordokhoz vezető, kosárból kiinduló mutatókat megkaphatjuk úgy, hogy a memóriában kiszámoljuk a mutatóhalmazok metszetét. Ily módon csak az eredményül kapott mutatókhoz tartozó rekordokat kell kinyernünk. Ezzel megtakarítjuk az olyan rekordok kinyeréséhez szükséges I/O-költséget, amelyek megfelelnek ugyan valamelyik feltételnek, de nem az összesnek.[2]

4.16. példa: Használjuk a 4.14. példa relációját:

```
Film(cím, év, hossz, stúdiónév)
Stúdió(név, cím, elnök)
```

Tegyük fel, hogy a `stúdiónév` és `év` attribútumokra egyaránt van közvetett kosarakat használó másodlagos indexünk. Tegyük fel továbbá, hogy a következő lekérdezéssel szeretnénk megtalálni az összes olyan filmet, amelyet 1995-ben a Disney stúdióban gyártottak.

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



4.18. ábra. *Kosarak metszése a memóriában*

[2] Ezt a trükköt a mutatóhalmazok metszésével olyankor is felhasználhatjuk, ha a mutatók közvetlenül az indexfájlból indulnak és nem a kosarakból. Azonban a kosarak használata gyakran jár lemez I/O-megtakarítással, mivel a mutatók kevesebb helyet igényelnek, mint a kulcs-mutató párok.

A 4.18. ábrán láthatjuk, hogy miként válaszolhatjuk meg ezt a lekérdezést az indexek felhasználásával. A `stúdiónév` attribútumhoz tartozó index segítségével megtaláljuk az összes Disney-filmre utaló mutatót, de még nem töltünk be egyetlen rekordot sem lemezről memóriába. Ehelyett, az `év` attribútumhoz tartozó index segítségével megtaláljuk az összes 1995-ben készült filmre utaló mutatót. Ezután kiszámoljuk a két mutatóhalmaz metszetét, megkapva ezáltal pontosan azokat a filmeket, amelyeket 1995-ben a Disney-stúdió készített. Most már betöltjük lemezről az összes olyan blokkot, amely egy vagy több ilyen filmet tartalmaz, ily módon a lehető legkevesebb adatblokkhoz nyúltunk hozzá. □

4.2.4. Dokumentumok visszakeresése és az invertált indexek

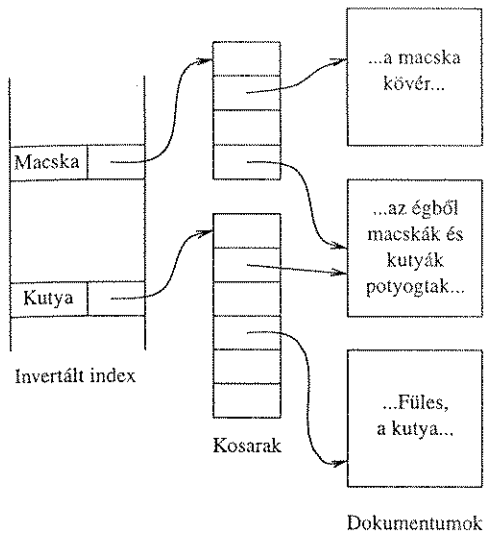
Az információszolgáltatók közössége több éve foglalkozik dokumentumok tárolásával és az adott kulcsszavakat tartalmazó dokumentumok hatékony visszakeresésével. A World Wide Web megjelenésével és a dokumentumok on-line elérhetőségének megvalósulásával az adott kulcsszavakat tartalmazó dokumentumok visszakeresése az egyik legnagyobb adatbázis-problémává vált. A tárgyhoz tartozó dokumentumok megtalálására igen sok fajta lekérdezés használatos, a legegyszerűbbek és a legáltalánosabbak azonban megoldhatók a következő relációs terminológiákkal:

- Egy dokumentumot elképzelhetünk úgy, mint egy `Doc` reláció egyetlen sorát. Ennek a relációnak nagyon sok attribútuma van, mindegyik attribútum a dokumentum egy lehetséges szavának felel meg. Valamennyi attribútum logikai típusú – a megfelelő szó vagy szerepel a dokumentumban vagy nem. Ily módon a reláció sémáját a következőképpen képzelhetjük el:

  ```
  Doc(vanbenneMacska, vanbenneKutya, ...)
  ```

  ahol a `vanbenneMacska` akkor és csak akkor igaz, ha a dokumentumban legalább egyszer szerepel az a szó, hogy „macska".
- A `Doc` valamennyi attribútumához tartozik egy másodlagos index. Megtakarítjuk azonban azt a fáradságot, amelyet azon sorok indexelése jelentene, amelyekben az attribútum értéke `FALSE`; ehelyett az index csak azon dokumentumokra mutat, amelyekben a keresett szó szerepel. Ez azt jelenti, hogy az indexben csak a `TRUE` keresésikulcs-értékekhez tartozik bejegyzés.
- Ahelyett, hogy valamennyi attribútumhoz (azaz minden szóhoz) külön indexet készítenénk, az indexeket összekombináljuk, ily módon egyetlen indexet kapunk, amit *invertált indexnek* nevezünk. Ez az index közvetett kosarakat használ a jobb helykihasználás végett, a 4.2.3. részben bemutatott módon.

4.17. példa: A 4.19. ábrán egy invertált indexet láthatunk. A rekordokból álló adatfájl helyett dokumentumok gyűjteményét láthatjuk. Valamennyi dokumentum tárolása egy vagy több lemezblokkon történhet. Az invertált index tulajdonképpen egy szó-mutató



4.19. ábra. *Invertált index dokumentumokon*

párokból álló halmaz; a szavak a keresési kulcs szerepét töltik be az indexben. Az invertált index tárolása éppúgy egy blokkszekvenciában történik, mint az eddig tárgyalt indexek esetében bármikor. A dokumentum-visszakereső alkalmazások némelyikében azonban az adat sokkal inkább statikus, mint egy átlagos adatbázis esetén, éppen ezért ezek az alkalmazások általában nem gondoskodnak a túlcsordulásblokkokról, illetve arról, hogy a változásokat átvezessék az indexbe is.

A mutatók a kosárfájl bizonyos pozícióira mutatnak. A 4.19. ábrán például a „macska" szó melletti mutató a kosárfájlra mutat. Ha követjük ezt a mutatót, akkor eljutunk a kosárfájl azon pozíciójáig, ahonnan kezdve azon mutatók találhatók, amelyek a „macska" szót tartalmazó összes dokumentumhoz elvezetnek. Ezek közül feltüntettünk néhányat az ábrán. Hasonlóképpen a „kutya" szó melletti mutatót is feltüntettük, amely azon mutatók listájára mutat, amelyek az összes „kutya" szót tartalmazó dokumentumhoz elvezetnek. □

A kosárfájl mutatói:

1. Mutathatnak magára a dokumentumra.
2. Mutathatnak a szó egy előfordulására. Ebben az esetben a mutató lehet egy olyan pár, amely tartalmazza a dokumentum első blokkját és egy egész számot, amely azt jelzi, hogy az adott szó hányadik szó az adott dokumentumban.

Ha már adott az ötlet, hogy mutatókból álló „kosarakat" használjunk valamennyi szó előfordulásához, akkor miért ne terjesztenénk ki az ötletet azzal, hogy a kosár tömbjében információkat tárolunk az előfordulásról. Ily módon a kosárfájl fontos

Az információ-visszakeresésről bővebben

Több olyan technika is létezik, amely az adott kulcsszavakat tartalmazó dokumentumok visszakeresésének hatékonyságát növeli. Mivel ezek teljes körű bemutatása túlmutat könyvünk céljain, lássunk két hasznos technikát:

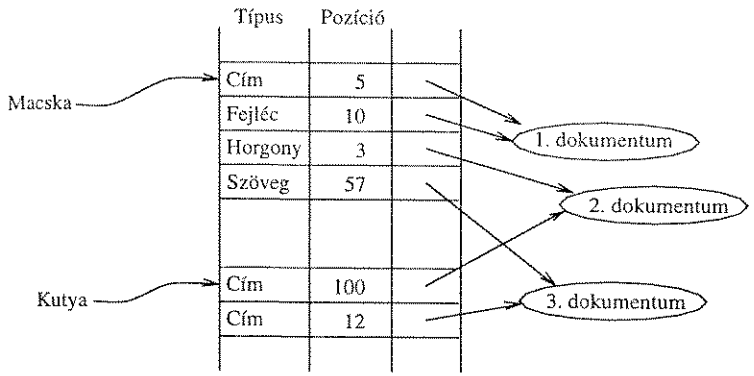
1. *Szótőképzés.* Mielőtt egy szó előfordulását bejegyeznénk az indexbe, eltávolítjuk a toldalékokat, hogy megkapjuk a „szótövet". A főnevek többes számát például úgy kezelhetjük, mintha egyes számban lennének. A 4.17. példában az invertált index természetesen szótőképzést használ, hiszen ha a „kutya" szót tartalmazó dokumentumokat keressük, akkor megkapjuk a „kutyák" szót tartalmazó dokumentumot is, nem csak a „kutya" szót tartalmazót.
2. *Töltelékszavak.* Az olyan leggyakrabban előforduló szavakat, mint az „és", „a" vagy „az" *töltelékszavaknak* nevezzük, és igen gyakran nem szerepelnek az invertált indexben. Ennek oka az, hogy a néhány száz leggyakoribb szó túl sok dokumentumban szerepel ahhoz, hogy hasznos legyen konkrét témájú dokumentumok megtalálásában. A töltelékszavak kiküszöbölése szintén jelentősen csökkenti az index méretét.

szerkezetű rekordok gyűjteményévé válik. Az ötlet régebbi felhasználása a következő eseteket különböztette meg: a szó a címben szerepel, az absztraktban vagy a dokumentum törzsében. A weben található nagyszámú dokumentum, kiváltképp a HTML, XML vagy egyéb hasonló szabványt használó dokumentumok, szintén indokolhatják a szavak melletti információk tárolását. Meg tudjuk például különböztetni a címekben, fejlécekben, táblázatokban és az ún. horgonyok között található szavakat éppúgy, mint a különböző fontokkal vagy méretben szedett szavakat.

4.18. példa: A 4.20. ábrán egy olyan kosárfájl látható, amelyet HTML-dokumentumokban előforduló szavak jelzésére szoktak használni. Az első oszlop a megjelenés típusára utal. A második és harmadik oszlop együtt alkotja a megjelenésre utaló mutatót. A harmadik oszlop mutat a dokumentumra, míg a második oszlopban található szám arra utal, hogy az adott szó hányadik szó a dokumentumban.

Ezt az adatszerkezetet igen sok, dokumentumokra vonatkozó lekérdezéshez használhatjuk, anélkül hogy meg kellene vizsgálnunk részletesen a dokumentumokat. Tegyük fel például, hogy szeretnénk megtalálni azokat a kutyákról szóló dokumentumokat, amelyek összehasonlítják a kutyákat a macskákkal. Anélkül, hogy a szöveget értelmeznénk, nemigen tudjuk pontosan megválaszolni ezt a lekérdezést. Azonban hasznos nyomon járhatunk, ha megkeressük azokat a dokumentumokat, amelyekben:

a) A címben szerepelnek kutyák.
b) Macskák is szerepelnek valamelyik horgonyban, ami valószínűleg egy link egy macskákról szóló dokumentumra.



4.20. ábra. *Több információk tárolása az invertált indexben*

A lekérdezést megválaszolhatjuk úgy, hogy képezzük a mutatók metszetét. Ez úgy történik, hogy követjük a „macska" szóhoz tartozó mutatókat és így megkapjuk e szó előfordulásait. A kosárfájlból kiválasztjuk azokat a „macska" szóhoz tartozó és dokumentumokra utaló mutatókat, amelyek típusa „horgony". Ezután megkeressük a „kutya" szóhoz tartozó kosár bejegyzéseket és kiválasztjuk közülük azokat, amelyek típusa „cím". Ha metsszük a két, mutatókból álló halmazt, akkor megkapjuk a feltételeket teljesítő dokumentumokat, azokat, amelyek címében szerepel a „kutya", és a „macska" szó horgonyban található bennük. □

4.2.5. Feladatok

* **4.2.1. feladat:** Az adatfájlba történő beszúrás vagy törlés esetén a másodlagos indexfájlnak is változnia kell. Javasoljunk néhány módszert arra, hogy miként lehet naprakészen tartani a másodlagos indexet az adatfájl változásai közepette.

! **4.2.2. feladat:** Tegyük fel, hogy egy blokkban elfér 3 rekord vagy 10 kulcs-mutató pár éppúgy, mint a 4.1.1. feladatban. Használjuk ezeket a blokkokat egy adatfájl és egy K keresési kulcsra épült másodlagos index tárolására. Minden egyes v értékre, amely szerepel a K mezőben, 1, 2 vagy 3 olyan rekord létezik a fájlban, amelyekben a K mező értéke v. Pontosabban, az értékek 1/3-a egyszer jelenik meg, másik 1/3-a pontosan kétszer, és az utolsó 1/3-a pontosan háromszor jelenik meg. Tegyük fel továbbá, hogy az indexblokkok és az adatblokkok egyaránt lemezen találhatók, létezik azonban egy olyan adatszerkezet, amely lehetővé teszi, hogy a K bármely v értékére megkapjuk azokat a mutatókat, amelyek az összes olyan indexblokkhoz elvezetnek, amelyben a v keresésikulcs-érték egy vagy több rekordban előfordul. (Esetleg van egy második szintű index a memóriában.) Számoljuk ki az összes v keresésikulcs-értékű rekord visszanyeréséhez szükséges lemez I/O-műveletek átlagos számát.

Beszúrás és törlés kosarakban

A 4.19. és a hozzá hasonló ábrákon úgy ábrázoltuk a kosarakat, mint megfelelő méretű tömör tömböket. A gyakorlatban azonban a kosár rekordokból áll, olyan rekordokból, amelyek egyetlen mezővel rendelkeznek (ez maga a mutató), és éppúgy blokkokban tároljuk, mint bármely más rekordokból álló gyűjteményt. Éppen ezért a mutatók beszúrására, illetve törlésére az eddig megismert bármelyik technikát alkalmazhatjuk, úgymint: extra helyet hagyunk a blokkokban a fájl kiterjesztésére, túlcsordulásblokkokat használunk, rekordokat mozgatunk a blokkon belül vagy a blokkok között. Az utóbbi esetben vigyázzunk, hogy a rekord mozgatásával egy időben az invertált indexben is változtassuk meg a megfelelő, kosárfájlra utaló mutatót.

*! **4.2.3. feladat:** Vegyünk egy, a 4.16. ábrához hasonló nyalábolt fájlt, és tegyük fel, hogy 10 film vagy stúdió rekord fér el egy blokkban. Tegyük fel továbbá, hogy az egy stúdióra eső filmek egyenletesen oszlanak meg 1 és m között. Fejezzük ki az m függvényében az egy stúdió és az ahhoz tartozó filmek visszanyeréséhez szükséges lemez I/O-műveletek átlagos számát. Mennyi lenne ez a szám, ha a filmek véletlenszerűen lennének szétszórva nagyszámú blokk között?

4.2.4. feladat: Tegyük fel, hogy egy blokkban elfér 3 rekord vagy 10 kulcs-mutató pár vagy 15 mutató. A 4.17. ábrán látható nem közvetlen kosarakat használva adjunk választ a következőkre:

* a) Ha az átlagos keresésikulcs-érték 10 rekordban jelenik meg, akkor hány blokkra van szükség 3000 rekord és az ahhoz tartozó másodlagos index tárolására? Hány blokkra lenne szükség, ha *nem* használnánk kosarakat?

! b) Ha nincs megszorítás az egy adott keresésikulcs-értékkel rendelkező rekordok számát illetően, akkor hány blokkra lenne szükség minimum és hány blokkra maximum?

! **4.2.5. feladat:** A 4.2.4. feladat a) pontjának feltevéseit használva, adjuk meg, hogy hány lemez I/O-műveletre lenne szükség átlagosan az adott keresésikulcs-értékkel rendelkező rekordok megtalálásához és visszanyeréséhez, a kosárszerkezet használatával, illetve kosarak nélkül. Tegyük fel, hogy kezdetben semmi nincs a memóriában, de az index-, illetve kosárblokkok helyét meg tudjuk határozni anélkül, hogy további lemez I/O-műveletre lenne szükség azon kívül, ami ezen blokkok memóriába történő beolvasásához szükséges.

4.2.6. feladat: Tegyük fel, hogy a 4.2.4. feladathoz hasonlóan, egy blokkban elfér 3 rekord vagy 10 kulcs-mutató pár vagy 15 mutató. Feltételezzük, hogy a `Film` reláció `stúdiónév` és `év` attribútumaira egyaránt van másodlagos indexünk, éppen úgy, mint a 4.16. példában. Tegyük fel, hogy van 51 Disney-film és 101 olyan film, ami 1995-ben készült. Ezek közül csak egy készült Disneyben. Számoljuk ki a 4.16. példa lekérdezésének (adjuk meg az összes olyan filmet, amelyet 1995-ben a Disney-stúdióban gyártottak) megválaszolásához szükséges lemez I/O-műveletek számát a következő esetekben:

* a) Mindkét másodlagos indexhez kosarakat használunk, a kosarakból kinyerjük a megfelelő mutatókat, a memóriában elkészítjük ezek metszetét, és csak azt az egyetlen rekordot olvassuk be, amely az 1995-ben a Disney-stúdióban készült filmhez tartozik.
  b) Nem használunk kosarakat, hanem a `stúdiónév` attribútumhoz tartozó index segítségével megkapjuk a Disney-filmekre utaló mutatókat, ezeket a filmeket beolvassuk és kiválasztjuk közülük azokat, amelyek 1995-ben készültek. Tegyük fel, hogy két Disney-filmhez tartozó rekord nincs ugyanabban a blokkban.
  c) Úgy járunk el, mint a b) esetben, viszont az `év` attribútumhoz tartozó indexszel kezdünk. Tegyük fel, hogy két 1995-ben készült filmhez tartozó rekord nincs ugyanabban a blokkban.

4.2.7. feladat: Tegyük fel, hogy van egy 1000 dokumentumból álló tárházunk és szeretnénk felépíteni hozzá egy 10 000 szót tartalmazó invertált indexet. Egy blokkban elfér 10 kulcs-mutató pár vagy 50 olyan mutató, amely vagy a dokumentumra mutat vagy a dokumentum egy pozíciójára. A szavak az ún. Zipfian-eloszlást követik (lásd a 7.4.3. részben a „Zipfian-eloszlás" című bekeretezett részt); az i-edik leggyakoribb szó előfordulásainak száma $100\,000/\sqrt{i}$, ahol $i = 1, 2, ..., 10\,000$.

* a) Hány szó található átlagosan egy dokumentumban?
* b) Tegyük fel, hogy az invertált indexünk minden szóhoz csak a dokumentumokat rögzíti, amelyekben a szó megtalálható. Maximum hány blokkra lenne szükség az invertált index tárolására?
  c) Tegyük fel, hogy az invertált indexünk minden egyes szó valamennyi előfordulásához tartalmaz egy mutatót. Hány blokkra van szükségünk az invertált index tárolására?
  d) Ismételjük meg a b) pontot abban az esetben, ha a 400 leggyakoribb szó (töltelékszavak) *nincs* benne az indexben.
  e) Ismételjük meg a c) pontot abban az esetben, ha a 400 leggyakoribb szó *nincs* benne az indexben.

4.2.8. feladat: Ha a 4.20. ábrához hasonló bővített indexet használunk, akkor igen sok különböző típusú keresést is végrehajthatunk. Adjunk javaslatokat, hogy miként lehetne ezt az indexet felhasználni a következő esetekben:

* a) Olyan dokumentumokat keresünk, amelyek 5 pozícióban tartalmazzák a „macska" és a „kutya" szavakat, és ezek a szavak mindegyik pozícióban ugyanolyan típusúak (pl. cím, szöveg vagy horgony).
  b) Olyan dokumentumokat keresünk, amelyek közvetlenül egymás után tartalmazzák a „macska" és a „kutya" szavakat.
  c) Olyan dokumentumokat keresünk, amelyek címében szerepelnek a „macska" és a „kutya" szavak.

4.3. B-fák

Egy vagy két indexszint használata gyakran igen hasznos a lekérdezések gyorsításában, van azonban egy ennél általánosabb, rendszerint kereskedelmi rendszerekben használatos adatszerkezet. Ezen adatszerkezetek közös családját *B-fának*, a leggyakrabban használt változatát *B+-fának* nevezzük. Lényegében:

- A B-fák automatikusan annyi indexszintet tartanak fenn, amennyi az indexelt fájl méretéhez szükséges.
- A B-fák úgy kezelik az általuk használt blokkokban az üres helyeket, hogy valamennyi blokk legalább félig ki van használva. Az indexhez soha nem kellenek túlcsordulásblokkok.

A következőkben „B-fákról" fogunk beszélni, de a részleteket mind ismertetjük a B+-fákhoz is. A többi B-fa-típust a feladatokban fogjuk ismertetni.

4.3.1. B-fák szerkezete

Ahogy a nevéből is következik, egy B-fa a blokkjait faszerkezetbe rendezi. A fa *kiegyensúlyozott*, ami azt jelenti, hogy valamennyi gyökértől levélig vezető út egyforma hosszú. Tipikusan három szint található egy B-fában: a gyökér, egy közbeeső szint és a levelek, de akárhány szint lehetséges. Hogy könnyebb legyen elképzelni egy B-fát, vessünk egy pillantást a 4.21. és 4.22. ábrákra, amelyeken B-fa-csúcsokat láthatunk, vagy a 4.23. ábrára, amely egy teljes B-fát mutat be.

Valamennyi B-fa-indexhez tartozik egy n paraméter, amely meghatározza a B-fa blokkjainak az elrendezését. Minden blokkban n keresési kulcsnak és $n + 1$ mutatónak van helye. Bizonyos értelemben egy B-fa hasonló a 4.1. részben bemutatott indexblokkhoz, kivéve, hogy a B-fa tartalmaz egy pluszmutatót az n darab kulcs-mutató pár mellett. Az n értékét úgy választjuk meg, hogy egy blokkban elférjen $n + 1$ mutató és n kulcs.

4.19. példa: Tegyük fel, hogy a blokkjaink mérete 4096 bájt. A kulcsok legyenek 4 bájtot lefoglaló egész számok és a mutatók 8 bájtot foglaljanak le. Ha a blokkokban nincsenek fejléc-információk, akkor azt a legnagyobb egész n értéket keressük, amelyre $4n + 8(n + 1) \leq 4096$. Ez az érték az $n = 340$. □

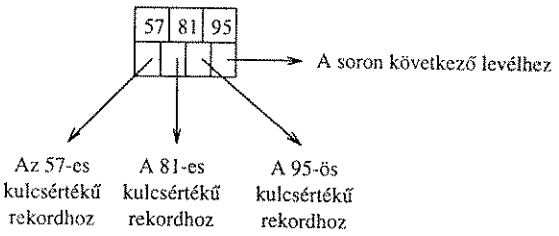
Van néhány olyan fontos szabály, amely megszorításokat jelent arra nézve, hogy egy B-fa blokkjai mit tartalmazhatnak.

- A gyökérben van legalább két használatban levő mutató.[3] Minden mutató a B-fa következő szintjének blokkjaira mutat.

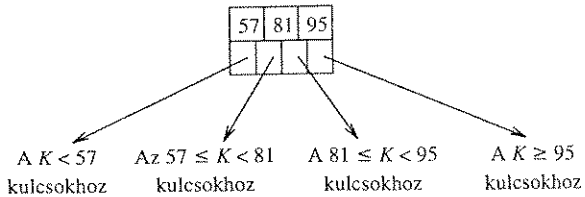
[3] Technikailag lehetséges, hogy a teljes B-fa mindössze egy mutatót tartalmazzon, akkor, ha az adatfájl egyetlen rekordból áll. Ebben az esetben az egész fa egy olyan gyökérblokk, amely egyben levél is, és ez a blokk egyetlen kulcsot és egyetlen mutatót tartalmaz. Az elkövetkezőkben eltekintünk ettől a triviális esettől.

- A levelekben az utolsó mutató a következő (jobbra) levélblokkra mutat, azaz arra a blokkra, amely a soron következő nagyobb kulcsokat tartalmazza. Egy levélblokk többi n mutatójából legalább $\left\lceil \frac{n+1}{2} \right\rceil$ használatban van, és adatrekordra mutat; a nem használatos mutatókat elképzelhetjük úgy, mintha nullák lennének, és nem mutatnának sehová. Az i-edik mutató, ha használatban van, akkor az i-edik kulccsal rendelkező rekordra mutat.
- A közbeeső szinteken levő csúcsokban mind az $n + 1$ mutató a B-fa következő szintjére mutat. Közülük legkevesebb $\left\lceil \frac{n+1}{2} \right\rceil$ használatban van (de ha a csúcs maga a gyökér, akkor csak annyit követelünk meg, hogy 2 mutató legyen használatban, függetlenül attól, hogy mekkora az n). Ha j mutató van használatban, akkor $j-1$ kulcs van, mondjuk $K_1, K_2, \ldots, K_{j-1}$. Az első mutató a B-fa olyan részére mutat, ahol a K_1 kulcsnál kisebb kulcsokat tartalmazó rekordok találhatók. A második mutató a fa azon részére mutat, ahol azok a rekordok találhatók, amelyeknek kulcsa nagyobb vagy egyenlő, mint K_1, de kisebb, mint K_2 és így tovább. És végül, a j-dik mutató a fa azon részére mutat, ahol azok a rekordok találhatók, amelyeknek kulcsa nagyobb vagy egyenlő, mint K_{j-1}. Észrevehetjük, hogy bizonyos rekordok, amelyeknek kulcsa sokkal kisebb, mint K_1, vagy jóval nagyobb, mint K_{j-1}, nem érhetők el egyáltalán ebből a blokkból, elérhetők azonban ennek a szintnek egy másik blokkján keresztül.
- Tegyük fel, hogy egy B-fát a fákra jellemző hagyományos módon ábrázolunk, azaz egy adott csúcs gyermekeit sorrendben balról („első gyermek") jobbra („utolsó gyermek"). Ily módon, ha bármely szinten megnézzük balról jobbra a B-fa csúcsait, akkor a csúcsok kulcsai nem csökkenő sorrendben jelennek meg.

4.20. példa: Ebben és a B-fákról szóló további példákban is azt használjuk, hogy $n = 3$. Azaz, a blokkokban 3 kulcs és 4 mutató fér el, ami igen kevés és nemigen jellemző. A kulcsok egész számok. A 4.21. ábrán egy teljes kihasználtságú levelet láthatunk. 3 kulcs van benne, 57, 81 és 95. Az első 3 mutató a megfelelő kulcsértékű rekordra mutat. Az utolsó mutató a közvetlenül jobbra következő levélre mutat, mint mindig, ha levelekről van szó. Sorrendben az utolsó levél esetén ez a mutató 0.



4.21. ábra. *Egy B+-fa jellegzetes levele*



4.22. ábra. *Egy B+-fa jellegzetes belső csúcsa*

Egy levélnek nem kell szükségszerűen tele lenni, a mi példánkban azonban $n = 3$, és legalább 2 kulcs-mutató párt tartalmaznia kell. Ily módon a 4.21. ábrán hiányozhat a 95-ös kulcs és vele együtt a harmadik mutató is, az, amelyet „a 95-ös kulcsértékű rekordhoz" felirattal láttunk el.

A 4.22. ábra egy jellegzetes belső csúcsot ábrázol. 3 kulcsa van; ugyanazokat a kulcsokat választottuk, mint a levelet bemutató példában: 57, 81 és 95.[4] A csúcs 4 mutatót is tartalmaz. Az első mutató a B-fa olyan részére mutat, ahonnan az 57-nél kisebb kulcsokat tartalmazó rekordok érhetők el. A második mutató azon rekordokhoz vezet, amelyek kulcsértéke az első és második kulcs között található, a harmadik mutató azokhoz, amelyek kulcsértéke a blokk második és harmadik kulcsa között található, és a negyedik mutató lehetővé teszi azon rekordok elérését, amelyek kulcsértéke nagyobb vagy egyenlő, mint a blokk harmadik kulcsa.

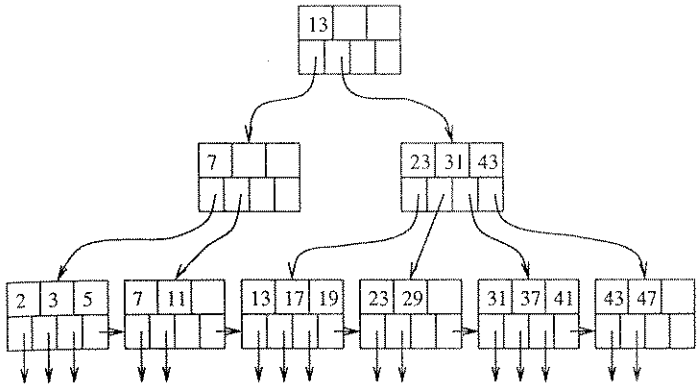
Éppúgy, mint a leveleknél, itt sem szükséges, hogy a kulcsok és mutatók tárolására fenntartott minden hely ki legyen töltve. Azonban $n = 3$, és legalább 1 kulcs és 2 mutató jelen kell legyen egy belső csúcsban. Ennek legszélsőségesebb esete az lenne, amikor az egyetlen kulcs az 57, és ekkor csak az első 2 mutató lenne használatban. Ebben az esetben az első mutató az 57-nél kisebb kulcsokhoz vezetne, és a második mutató az 57-nél nagyobb vagy egyenlő kulcsokhoz vezetne. □

4.21. példa: A 4.23. ábra egy teljes, háromszintű B+-fát[5] mutat be, a 4.20. példában leírt csúcsok segítségével. Feltételeztük, hogy az adatfájl olyan rekordokból áll, melyek kulcsai az összes 2 és 47 közötti prímszámok. Figyeljük meg, hogy a levelekben sorban minden kulcs megjelenik egyszer. Mindegyik levélblokk két vagy három kulcs-mutató párt tartalmaz, plusz egy mutatót, amely a sorban következő levélre mutat. A kulcsok rendezett sorrendben vannak, ahogyan azt láthatjuk is, ha balról jobbra végignézzük a leveleket.

A gyökérben csak 2 mutató van, ennyi a minimum, azonban lehetne 4 is. A gyökérben levő kulcs elkülöníti az első mutatón keresztül elérhető kulcsokat a második mutatón keresztül elérhető kulcsoktól. Ez azt jelenti, hogy azok a kulcsok, amelyek értéke kisebb, mint 12, a gyökér első részfájában találhatók, míg azok, amelyek értéke nagyobb vagy egyenlő, mint 13, a második részfában találhatók.

[4] Habár a kulcsok ugyanazok, de a 4.21. ábrán látható levélnek és a 4.22. ábrán látható belső csúcsnak semmi köze nincs egymáshoz. Sőt soha nem is szerepelhetnek ugyanabban a B-fában.

[5] Ne feledjük, hogy a B-fák, amelyeket ebben az részben bemutattunk, mind B+-fák, de a jövőben eltekintünk a „+" jelöléstől, amikor hivatkozunk rájuk.



4.23. ábra. *B+-fa*

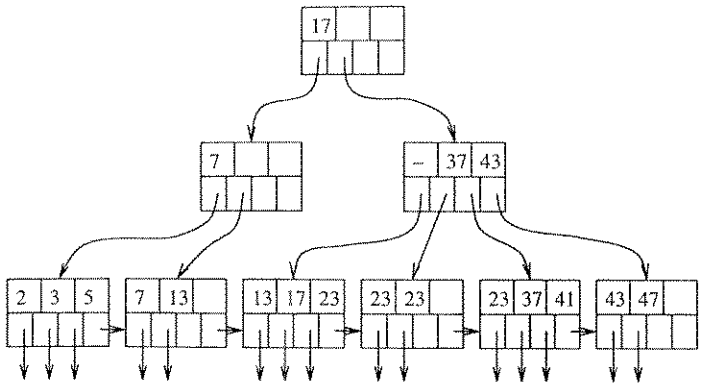
Ha megnézzük a gyökér első gyermekét, amelyben a 7-es kulcs található, ismét 2 mutatót találunk. Az egyik azokhoz a kulcsokhoz vezet, amelyek kisebbek, mint 7, a másik azokhoz, amelyek értéke nagyobb vagy egyenlő, mint 7. Vegyük észre, hogy ennek a csúcsnak a második mutatója csak a 7-es és 11-es kulcsokhoz vezet el bennünket és nem az *összes* olyan kulcshoz, amely ≥ 7. Például a 13-as kulcshoz nem vezet el. (Igaz ugyan, hogy eljuthatunk a nagyobb kulcsokhoz, ha követjük a levelek következő blokkra utaló mutatóit.)

Végezetül, a gyökér második gyermekében mind a 4 mutatóknak fenntartott hely használatban van. Az első elvezet minket azokhoz a kulcsokhoz, amelyek értéke kisebb, mint 23, nevezetesen a 13-as, 17-es és 19-es kulcsokhoz. A második mutató az összes olyan K kulcshoz elvezet, amelyre $23 \leq K < 31$; a harmadik mutató az összes olyan K kulcs elérését biztosítja, amelyre $31 \leq K < 43$, és a negyedik kulcs olyan kulcsokhoz vezet, amelyek ≥ 43 (ebben az esetben az összes ilyen kulcshoz elvezet). □

4.3.2. B-fák alkalmazása

A B-fa egy erőteljes eszköz az indexek készítéséhez. A rekordokhoz vezető mutatók sorozata, amely a levelekben található, betöltheti a 4.1. és 4.2. részekben bevezetett indexfájlokból előálló mutatósorozat szerepét. Lássunk néhány példát:

1. A B-fa keresési kulcsa az adatfájl elsődleges kulcsa, és az index sűrű. Ez azt jelenti, hogy az adatfájl mindegyik rekordjához tartozik egy olyan kulcs-mutató pár, amelyik levélben található. Az adatfájl vagy rendezett az elsődleges kulcs szerint, vagy nem.
2. Az adatfájl rendezett az elődleges kulcs szerint, és a B+-fa egy ritka index, amelyben az adatfájl mindegyik blokkjához tartozik egy olyan kulcs-mutató pár, amelyik levélben található.



4.24. ábra. *B-fa ismétlődő kulcsokkal*

3. Az adatfájl egy olyan attribútum szerint rendezett, amely nem kulcs. Ez az attribútum a B+-fa keresési kulcsa. Az adatfájlban megjelenő valamennyi K keresési-kulcs-értékhez tartozik egy olyan kulcs-mutató pár, amelyik levélben található. A mutató arra az első rekordra mutat, amelynek rendezési kulcsértéke K.

A B-fák változatainak vannak olyan további alkalmazásai, amelyek megengedik a keresési kulcs[6] ismétlődését a levelekben. A 4.24. ábrán láthatjuk, hogy miként nézne ki egy ilyen B-fa. A kiterjesztés analóg azokkal a 4.1.5. részben bemutatott indexekkel, amelyek tartalmaznak ismétlődéseket.

Ha megengedjük egy keresési kulcs ismétlődő megjelenéseit, akkor némileg módosítanunk kell a belső csúcsok kulcsaira vonatkozó, 4.3.1. részben megadott definíciót. Tegyük fel, hogy egy belső csúcs kulcsai a $K_1, K_2, \ldots, K_n$. Ily módon a K_i lesz annak a részfának a legkisebb új kulcsa, amely az $(i + 1)$-edik mutató segítségével érhető el. Az „új" azt jelenti, hogy a K_i kulcs nem jelenik meg a fának abban a részében, ami az $(i + 1)$-edik részfától balra található, viszont a részfa tartalmazza a K_i legalább egy előfordulását. Jegyezzük meg, hogy bizonyos esetekben nem lesz ilyen kulcs, ilyenkor a K_i-t nullának vesszük. A hozzá tartozó mutatóra azonban szükség van, mivel az a fa egy olyan fontos részére mutat, amely történetesen egy kulcsértéket tartalmaz.

4.22. példa: A 4.24. ábrán egy olyan B-fát láthatunk, amely hasonlít a 4.23. ábrához, viszont tartalmaz ismétlődő értékeket. Tulajdonképpen a 11-es kulcsértéket helyettesítettük 13-mal, míg a 19-es, 29-es és 31-es kulcsértékek mindegyikét 23-mal helyettesítettük. Ennek eredményeképpen a gyökérben levő kulcs 17 lett, és nem 13. Ennek oka az, hogy bár a gyökér második részfájának most is a 13-as a legkisebb kulcsértéke, viszont a 13 most nem *új* kulcs az adott részfában, mivel megjelenik az első részfában is.

[6] Ne feledjük, hogy egy „keresési kulcs" nem feltétlenül „kulcs" abban az értelemben, hogy nem kell feltétlenül egyedinek lennie.

A gyökér második gyermekében is kellett változtatásokat végeznünk. A második kulcsot 37-re változtattuk, mivel a harmadik gyermeknek (balról az ötödik levélnek) ez az első új kulcsa. Még érdekesebb, hogy az első kulcs 0. Ennek oka az, hogy a második gyermek (negyedik levél) egyáltalán nem tartalmaz új kulcsot. Másképpen közelítve, ha egy kulcs keresése közben a gyökér második gyermekéhez érkezünk, soha nem akarunk majd annak második gyermeke felé elindulni. Ha a 23-as vagy ennél kisebb kulcsértéket keressük, akkor az első gyermek irányába indulunk tovább, ahol vagy megtaláljuk, amit keresünk (ha az a 17), vagy megtaláljuk az első előfordulását annak, amit keresünk (ha az a 23). Jegyezzük meg, hogy:

- Ha a 13-at keressük, akkor nem jutunk el a gyökér második gyermekéhez, ehelyett már a gyökérből az első gyermekhez leszünk irányítva.
- Ha 24 és 36 közötti kulcsot keresünk, akkor a harmadik levélhez leszünk irányítva, de ha nem találjuk egyetlen előfordulását sem a keresett kulcsnak, akkor tudjuk, hogy nem kell tovább keresnünk jobbra. Ha például lenne egy 24-es kulcs a levelek között, akkor az vagy a negyedik levélben lenne, és ekkor a gyökér második gyermekében a 0 kulcs helyett 24 állna, vagy az ötödik levélben lenne, és ekkor a gyökér második gyermekének 37-es kulcsa helyett a 24 állna.

□

4.3.3. Keresés B-fában

Térjünk most vissza az eredeti feltevésünkhöz, mely szerint a levelekben nincsenek ismétlődő kulcsok. Ez a feltevés megkönnyíti a B-fa műveleteinek tárgyalását, de nem feltétlenül szükséges a műveletekhez. Tegyük fel, hogy adott egy B-fa-index, és meg akarunk találni egy K keresésikulcs-értékű rekordot. Rekurzív módon keressük a K-t, a gyökértől kezdünk, és egy levélnél fogunk megállni. A keresési eljárás a következő:

Kiindulási pont: Ha egy levélnél vagyunk, akkor végignézzük annak kulcsait. Ha az i-edik kulcs a K, akkor az i-edik mutató elvezet minket a keresett rekordhoz.

Indukció: Ha egy $K_1, K_2, \ldots, K_n$ kulcsokkal rendelkező belső csúcsnál vagyunk, akkor a 4.3.1. részben bemutatott szabályokat használjuk annak eldöntésére, hogy a csúcs melyik gyermekét vizsgáljuk meg a következőkben. Ez azt jelenti, hogy csak egy olyan gyermek van, amely elvezethet egy K kulcsot tartalmazó levélhez. Ha $K < K_1$, akkor ez az első gyermek, ha $K_1 \leq K < K_2$, akkor ez a második gyermek és így tovább. Az így megkapott gyermekre rekurzív módon alkalmazzuk a keresési szabályt.

4.23. példa: Tegyük fel, hogy adott a 4.23. ábrán látható B-fa, és szeretnénk találni egy olyan rekordot, amelynek keresési kulcsa 40. Elindulunk a gyökérből, ahol egyetlen kulcs van, a 13. Mivel $13 \leq 40$, ezért a második mutatót követjük, amely a 23, 31 és 43 kulcsokkal rendelkező, második szinten található belső csúcshoz vezet bennünket.

Ennél a csúcsnál $31 \leq 40 < 43$, így a harmadik mutatót követjük. Ily módon a 31, 37 és 41 kulcsokat tartalmazó levélhez jutunk. Ha lenne az adatfájlban olyan rekord, amelynek keresési kulcsa 40, akkor a 40-es kulcsot ebben a levélben találnánk. Mivel nem találtunk 40-es kulcsot, levonjuk a következtetést, miszerint az alapul szolgáló adatok nem tartalmaznak 40-es kulcsú rekordot.

Figyeljük meg, hogyha olyan rekordot kerestünk volna, amelynek kulcsa 37, akkor ugyanezeket a döntéseket hoztuk volna, de amikor eljutottunk volna a levélhez, megtaláltuk volna a 37-es kulcsot. Mivel ez a második kulcs a levélben, a második mutatót követve eljutunk a 37-es kulcsú adatrekordhoz. □

4.3.4. Tartományra vonatkozó lekérdezések

A B-fák nem csak olyan lekérdezések esetén hasznosak, amelyekben a keresési kulcs egy konkrét értékére keressük, hanem olyankor is, amikor értékek egy tartományára vonatkozik a kérdés. A *tartományra vonatkozó lekérdezések* a WHERE záradékban jellegzetesen tartalmaznak egy olyan kifejezést, amely az = és < > operátoroktól eltérő összehasonlító operátort tartalmaz. Példák *k* keresési kulcs attribútumot használó tartományt eredményező lekérdezésekre:

```
SELECT *
FROM R
WHERE R.k > 40;
```

vagy

```
SELECT *
FROM R
WHERE R.k >= 10 AND R.k <= 25;
```

Ha meg akarjuk találni egy B-fa leveleiben az összes $[a, b]$ tartományba tartozó kulcsot, akkor végrehajtunk egy keresést az a megtalálására. Függetlenül attól, hogy létezik-e vagy sem, eljutunk egy olyan levélhez, ahol az a előfordulhatna, és megkeressük a levélben azokat a kulcsokat, amelyek nagyobbak vagy egyenlők, mint az a. Minden ilyen kulcshoz találunk egy mutatót, amely egy olyan rekordra mutat, amelynek kulcsa a kívánt tartományba tartozik.

Ha nem találunk olyan kulcsot, amely nagyobb, mint b, akkor használjuk a levélnek azt a mutatóját, amely a következő levélre mutat. Megtartjuk a megvizsgált kulcsokat, valamint követjük a hozzájuk tartozó mutatókat, mindaddig, amíg:

1. Találunk egy olyan kulcsot, amely nagyobb, mint b, és ekkor megállunk.
2. Elérjük a levél végét, ekkor továbblépünk a következő levélre, és megismételjük az eljárást.

A fenti keresési algoritmus akkor is működik, ha b végtelen, azaz csak egy alsó határ van megadva, felső határ nincs. Ebben az esetben végignézzük az összes levelet attól a levéltől kezdve, amelyik tartalmazhatná az a kulcsot, egészen a levelek végéig. Ha az a értéke $-\infty$ (azaz a tartománynak csak felső határa van, alsó határa nincs), akkor a „mínusz végtelen" kulcs keresése a B-fa valamennyi csúcsa esetén az első gyermekhez vezet majd bennünket, azaz tulajdonképpen az első levelet találjuk majd meg. A keresés a továbbiakban ugyanúgy történik, mint fentebb, megállni akkor kell majd, ha túlléptük a b kulcsot.

4.24. példa: Tegyük fel, hogy adott a 4.23. ábrán látható B-fa, és a (10, 25) tartományba eső kulcsokat keressük. Elkezdjük a 10-es kulcs keresését, és eljutunk a második levélhez. Az első kulcs kisebb, mint 10, de a második 11, ami nagyobb vagy egyenlő, mint 10. Követjük a hozzá tartozó mutatót, hogy megkapjuk a 11-es kulcsú rekordot.

Mivel nincs több kulcs a második levélben, követjük a levelek láncolatát, és eljutunk a harmadik levélhez, melynek kulcsai 13, 17 és 19. Mindegyik kisebb vagy egyenlő, mint 25, ezért követjük a hozzájuk tartozó mutatókat, és megkapjuk azokat a rekordokat, amelyek ezekkel a kulcsértékekkel rendelkeznek. Végezetül átmegyünk a negyedik levélbe, ahol először 23-as kulcsot találunk. A levél következő kulcsa azonban 29, ami nagyobb, mint 25, ezért itt be is fejezzük a keresést. Ily módon megkaptuk azt az öt rekordot, melynek kulcsai 11, 13, 17, 19 és 23. □

4.3.5. Beszúrás B-fában

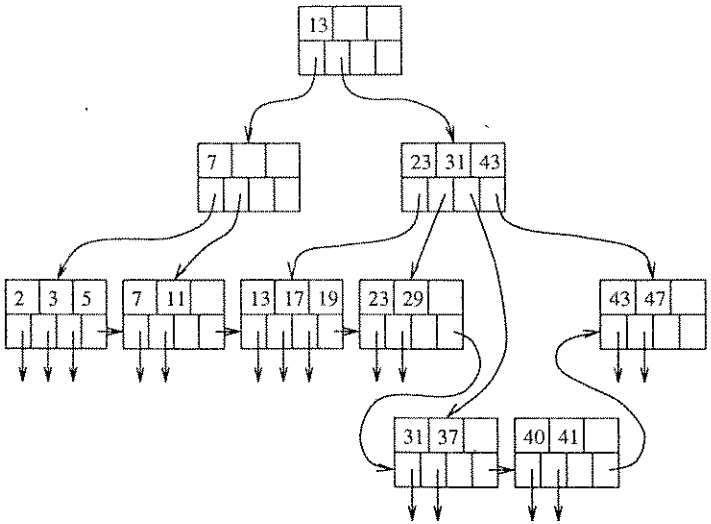
A B-fáknak vannak előnyei az egyszerűbb többszintű indexekkel szemben, ezek közül láthatunk néhányat, miközben áttekintjük, hogy miként kell beszúrni egy B-fába egy új kulcsot. A megfelelő rekordot a 4.1. részben bemutatott módszerek valamelyikével beszúrjuk a B-fával indexelt fájlba; itt most azt tekintjük át, hogy a B-fa ennek megfelelően miként változik. A beszúrás alapjában véve rekurzív:

- Megpróbálunk találni egy helyet az új kulcs számára a megfelelő levélben, és ha van szabad hely, akkor ide tesszük.
- Ha nincs hely a megfelelő levélben, akkor kettévágjuk a levelet, és szétosztjuk a kulcsokat a két új csúcs között, így mindkettő félig lesz telítve vagy éppen csak egy kissé jobban.
- Egy csúcs szétvágása egy adott szinten hatással van a fölötte levő szintre is, oly módon, hogy egy új kulcs-mutató párt kell beszúrni ezen a felsőbb szinten. Ily módon rekurzívan alkalmazhatjuk ezt a stratégiát a magasabb szinten történő beszúrásra: ha van hely, beszúrjuk amit kell; ha nincs, akkor szétvágjuk a szülő csúcsot és megyünk tovább fölfelé a fában.
- Van egy kivétel: ha a gyökérbe próbálunk beszúrni és nincs hely, akkor szétvágjuk a gyökeret két csúcsra, és létrehozunk egy új gyökeret a következő szinten; az új gyökérnek a szétvágás következtében két gyermek csúcsa lesz. Emlékezzünk vissza, hogy bármekkora is az n (az egy csúcsba tehető kulcsoknak fenntartott helyek száma), a gyökér számára mindig engedélyezett, hogy csak egy kulcsa és két gyermeke legyen.

Amikor szétvágunk egy csúcsot, és beszúrunk a szülő csúcsba, vigyáznunk kell arra, hogy miként kezeljük a kulcsokat. Először is, tegyük fel, hogy az N egy olyan levél, amelynek kapacitása n kulcs. Tegyük fel továbbá, hogy szeretnénk beszúrni egy $(n + 1)$-edik kulcsot és a hozzá tartozó mutatót. Készítünk egy új M csúcsot, amely az N testvére lesz, közvetlenül jobbra tőle. Az első $\left\lceil \frac{n+1}{2} \right\rceil$ kulcs-mutató pár a kulcsok rendezett sorrendjében az N csúcsban marad, míg a többi kulcs-mutató pár átköltözik az M csúcsba. Figyeljük meg, hogy az M és az N csúcs egyaránt elegendő számú kulcs-mutató párral rendelkezik, legkevesebb $\left\lfloor \frac{n+1}{2} \right\rfloor$ párral.

Most tegyük fel, hogy az N egy olyan belső csúcs, melynek kapacitása n kulcs és $n + 1$ mutató, de az N csúcshoz $n + 2$ mutató kellene tartozzon egy csúcs alsóbb szinten történt szétvágása miatt. A következőket tesszük:

1. Készítünk egy új M csúcsot, amely az N testvére lesz, közvetlenül jobbra tőle.
2. Az első $\left\lceil \frac{n+2}{2} \right\rceil$ mutató, a kulcsok rendezett sorrendjében az N csúcsban marad, míg a többi $\left\lfloor \frac{n+2}{2} \right\rfloor$ átköltözik az M csúcsba.
3. Az első $\left\lceil \frac{n}{2} \right\rceil$ kulcs az N csúcsban marad, míg a többi $\left\lfloor \frac{n}{2} \right\rfloor$ átköltözik az M csúcsba.



4.25. ábra. *A 40-es kulcs beszúrásának kezdete*

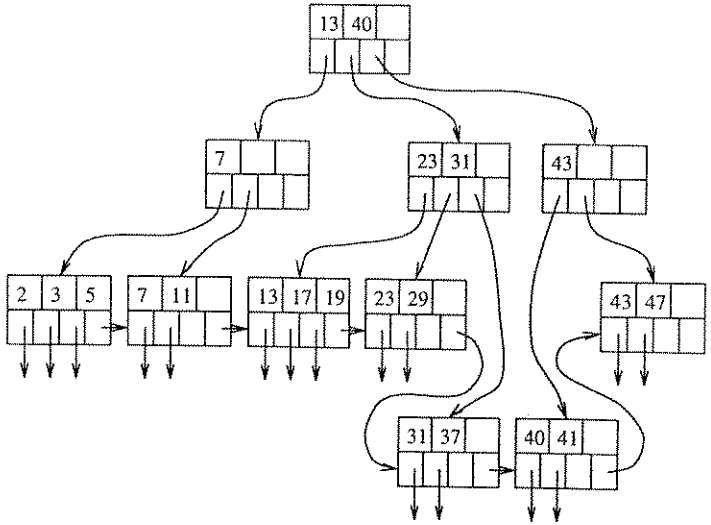
Figyeljük meg, hogy középen marad egy kulcs, amely nem jelenik sem az N, sem az M csúcsban. A maradék K kulcs azt a legkisebb kulcsot jelöli, amely az M első gyermekén keresztül elérhető. Habár ez a kulcs nem jelenik meg sem az N, sem az M csúcsban, mindamellett az M csúcshoz tartozik abban az értelemben, hogy az M csúcson keresztül elérhető legkisebb kulcsot jelöli. Ekképpen a K-t az N és M csúcsokhoz tartozó szülő fogja használni, hogy megossza a kereséseket a két csúcs között.

4.25. példa: Szúrjuk be a 4.23. ábrán látható B-fába a 40-es kulcsot. A beszúráshoz megfelelő levelet a 4.3.3. részben leírt eljárással keressük meg. Ahogyan a 4.23. példában is láttuk, a beszúrás az ötödik levélbe történik. Mivel $n = 3$, és ez a levél most négy kulcs-mutató párt tartalmaz – 31, 37, 40 és 41 – szét kell vágnunk a levelet. Az első lépés az, hogy készítünk egy új csúcsot, és a két legnagyobb kulcsot (40 és 41) áttesszük ebbe az új csúcsba. A 4.25. ábrán láthatjuk ezt a szétvágást.

Megjegyzendő, hogy habár most négy sorban ábrázoljuk a csúcsokat, a fának valójában három szintje van, és a hét levél foglalja el az ábra két alsó sorát. A levelek össze vannak kötve az utolsó mutatóik segítségével, amelyek most is egy balról jobbra tartó láncot alkotnak.

Be kell most szúrnunk egy új mutatót az új levélhez (ahhoz, amelynek kulcsai a 40 és a 41) a fölötte levő csúcsba (amelynek kulcsai 23, 31 és 43). Ehhez a mutatóhoz társítanunk kell a 40-es kulcsot, amely az új levélen keresztül elérhető legkisebb kulcs. Sajnos, a szétvágott csúcs szülője is tele van; nincs benne hely egy újabb kulcsnak vagy mutatónak. Ily módon ezt is szét kell vágnunk.

Kezdjük azokkal a mutatókkal, amelyek az utolsó öt levélre mutatnak, és a négy utolsó levél legkisebb kulcsainak a listájával. Tehát adottak a P_1, P_2, P_3, P_4, P_5 mu-



4.26. ábra. *A 40-es kulcs beszúrásának befejezése*

tatók azokhoz a levelekhez, amelyeknek legkisebb kulcsai 13, 23, 31, 40 és 43, és adott egy, a mutatók elválasztására szolgáló kulcssorozatunk: 23, 31, 40, 43. Az első három mutató és az első két kulcs a szétvágott belső csúcsban marad, míg az utolsó két mutató és az utolsó kulcs átmegy az új csúcsba. A megmaradt kulcs, a 40-es, az új csúcson keresztül elérhető legkisebb kulcsot jelöli.

A 4.26. ábra a 40-es kulcs beszúrásának befejezését mutatja be. A gyökérnek most három gyermeke van; a két utolsó a szétszedett belső csúcsból származik. Figyeljük meg, hogy a 40-es kulcs, amely a szétszedett csúcsok második csúcsán keresztül elérhető legkisebb kulcs, a gyökérben került elhelyezésre, hogy szétválassza a gyökér második és harmadik gyermekeinek a kulcsait. □

4.3.6. Törlés B-fában

Ha ki akarunk törölni egy *K* kulcsú rekordot, akkor ahhoz meg kell találnunk a rekordot és a hozzá tartozó kulcs-mutató párt a B-fa leveleiben. A törlési folyamat ezen része tulajdonképpen egy olyan keresés, amit a 4.3.3. részben már láthattunk. Ezután kitöröljük a rekordot az adatfájlból és a kulcs-mutató párt a B-fából.

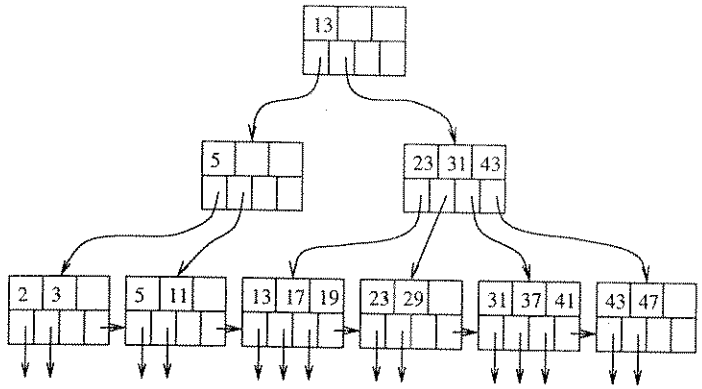
Ha a B-fának az a csúcsa, amelyben a törlés megtörtént, még így is tartalmaz legalább annyi kulcsot és mutatót, amennyit minimum tartalmaznia kell, akkor készen is vagyunk.[7] Lehetséges azonban, hogy a csúcs törlés előtti kihasználtsága minimális volt, így a törlés után a kulcsok számára vonatkozó megszorítás nem teljesül. Ilyen esetben egy olyan *N* csúcsra, amely nem tartalmazza a szükséges minimumot, a következők egyikét meg kell tennünk; az egyik eset rekurzív törlést igényel fölfelé a fában:

1. Ha az *N* csúcs egyik szomszédos testvére több kulcsot és mutatót tartalmaz, mint amennyi minimum szükséges, akkor egy kulcs-mutató párt áttehetünk az *N* csúcsba, miközben a kulcsok sorrendjét érintetlenül hagyjuk. Lehetséges, hogy az *N* csúcs szülőjében levő kulcsokat az új helyzethez kell igazítanunk. Ha például az *N* csúcs szülőjében levő kulcsokat az új helyzetünk *M*-nek, biztosít egy fölösleges kulcsot és mutatót, akkor az *M*-ből a legkisebb kulcsot tesszük át az *N*-be. Az *M* és *N* szülőjében van egy olyan kulcs, amely az *M* csúcson keresztül elérhető legkisebb kulcsot jelöli; ilyenkor ezt föl kell emelni.
2. A nehéz eset az, amikor egyik szomszédos testvért sem használhatjuk arra, hogy biztosítson egy fölösleges kulcsot az *N* számára. Ebben az esetben azonban van két olyan egymás melletti testvér, *N* és *M*, amelyik közül az egyik a minimum szükséges számú kulccsal rendelkezik, a másik eggyel kevesebbel. Ily módon együttvéve sem rendelkeznek több kulccsal és mutatóval, mint amennyi egy csúcsban megengedett (éppen ezért választottuk a minimumnak a félig történő telítettséget a B-fák csúcsaira). Összeolvasztjuk a két csúcsot úgy, hogy gyakorlatilag töröljük az egyiket. A szülőben levő kulcsokat az új helyzethez kell igazítanunk, és ezután ki kell törölnünk egy kulcsot a szülőből. Ha a szülő még így is eléggé telítve van, akkor készen vagyunk. Ha nem, akkor rekurzívan alkalmazzuk a szülőre a törlési algoritmust.

[7] Ha egy levél legkisebb kulcsához tartozó rekordot töröljük, akkor opcionálisan fölemelhetjük a levél egyik ősének megfelelő kulcsát, de ez nem kötelező; minden keresés a megfelelő levélhez fog vezetni enélkül is.

4.26. példa: Kezdjünk a 40-es kulcs beillesztése előtti eredeti B-fával, amelyet a 4.23. ábrán láthatunk. Tegyük fel, hogy töröljük a 7-es kulcsot. Ez a kulcs a második levélben található. Kitöröljük a kulcsot, a hozzá tartozó mutatót és a megfelelő rekordot.

Sajnos, a második levél már csak egy kulcsot tartalmaz, és nekünk legalább két kulcsra van szükségünk valamennyi levélben. De meg vagyunk mentve, hiszen a balra lévő testvér, az első levél tartalmaz egy fölösleges kulcs-mutató párt. Ily módon áttehetjük a legnagyobb kulcsot és a hozzá tartozó mutatót a második levélbe. Az eredményül kapott B-fát a 4.27. ábrán láthatjuk. Figyeljük meg, hogy mivel a második levél legkisebb kulcsa most 5, ezért a két első levél szülőjében a 7-es kulcs 5-re változott.

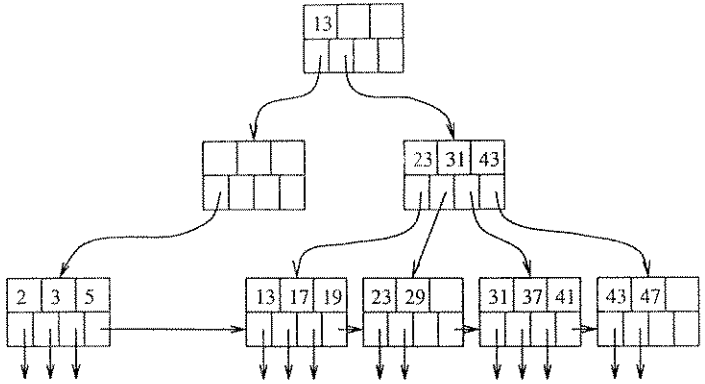


4.27. ábra. *A 7-es kulcs törlése*

A következőkben tegyük fel, hogy a 11-es kulcsot töröljük. Ennek a törlésnek hasonló hatása van a második levélre; ezzel a kulcsok száma ismét a minimum alá csökkent. Ezúttal azonban nem kérhetünk kölcsön az első levéltől, hiszen abban pontosan a minimum számú kulcs található. Ráadásul jobbról nincs is testvér, akitől kölcsönkérhetnénk.[8] Ily módon össze kell olvasztanunk a második levelet egy testvérével, nevezetesen az első levéllel.

Az első két levél három megmaradó kulcs-mutató párja befér egyetlen levélbe, így áttesszük az 5-ös kulcsot az első levélbe, és a második levelet kitöröljük. A szülőben levő kulcsokat és mutatókat a gyermekek új helyzetéhez igazítjuk, azaz a két mutatót egy mutató váltja fel (amely a megmaradt levélre mutat), és az 5-ös kulcs többé már nem fontos, ezért kitöröljük. Ezt a helyzetet a 4.28. ábrán mutatjuk be.

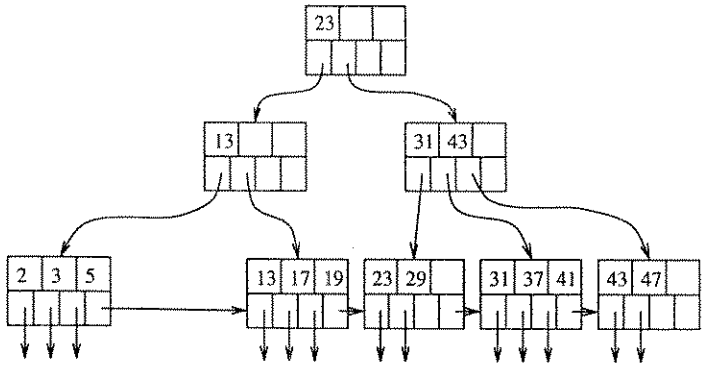
[8] Figyeljük meg, hogy a jobbra levő levél, amelynek kulcsai 13, 17 és 19 nem testvér, hiszen nem ugyanaz a két levél szülője. Ezzel együtt persze „kölcsönözhetünk" ettől a csúcstól, azonban a fa kulcsainak a helyzethez történő igazítása sokkal bonyolultabbá válik. Ezt a lehetőséget meghagyjuk feladatnak.



4.28. ábra. *A 11-es kulcs törlésének kezdete*

Sajnos, a levél törlése kedvezőtlenül befolyásolta a szülőt, amely a gyökér bal oldali gyermeke. Ez a csúcs ily módon nem is tartalmaz kulcsot, és mutatóból is csak egy van neki, amint a 4.28. ábrán láthatjuk. Ezért megpróbálunk szerezni egy fölösleges kulcsot és mutatót az egyik szomszédos testvértől. Ebben az esetben könnyű dolgunk van, hiszen a gyökér másik gyermeke megengedheti magának, hogy átadja a legkisebb kulcsot és egy mutatót.

A változás a 4.29. ábrán látható. Az a mutató, amely a 13, 17 és 19 kulcsokat tartalmazó levélre mutat, átköltözött a gyökér második gyermekéből az első gyermekbe. A belső csúcsokban is megváltoztattunk azonban néhány kulcsot. A 13-as kulcsot, amely eddig a gyökérben volt, és azt jelölte, hogy ez a legkisebb kulcs, amelyet az átköltöztetett mutatón keresztül el lehet érni, át kellett tenni a gyökér első gyermekébe. Másrészről, a 23-as kulcs, amely eddig a gyökér második gyermekének első és második gyermekét volt hivatott szétválasztani, most a gyökér második gyermekéből elérhető legkisebb kulcsot jelöli. Ily módon a 23-as kulcs átkerült magába a gyökérbe. □



4.29. ábra. *A 11-es kulcs törlésének befejezése*

4.3.7. B-fák hatékonysága

A B-fák lehetővé teszik rekordok keresését, beszúrását és törlését oly módon, hogy nagyon kevés az egy fájlra eső lemez I/O-műveletek száma. Először is, megfigyelhetjük, hogy ha az n, az egy blokkban tárolható kulcsok száma megfelelően nagy, mondjuk 10 vagy több, akkor ritkán lesz szükség blokkok szétvágására, illetve összeolvasztására. Továbbá, ha ilyen műveletre szükség van, akkor az majdnem mindig a levelekre korlátozódik, azaz mindössze két levél és azok szülője érintett. Ily módon lényegében elhanyagolható a B-fák újrarendezésének I/O-költsége.

Azonban adott keresési kulccsal rendelkező rekord(ok) megkereséséhez a gyökérből kell elindulnunk, és el kell jutnunk egy levélhez, hogy megtaláljuk a keresett rekordhoz tartozó mutatót. Mivel a B-fa blokkjait csak beolvassuk, ezért a lemez I/O-műveletek száma meg fog egyezni a B-fa szintjeinek a számával, plusz a rekord manipulásához szükséges lemez I/O-műveletek számával, ami egy vagy kettő, attól függően, hogy keresésről van-e szó avagy beszúrásról, illetve törlésről. Joggal kérdezhetjük: hány szintje van egy B-fának? A kulcsok, mutatók és blokkok tipikus méretével a legnagyobb adatbázisokat kivéve is elegendő három szint. Ily módon általában 3-nak fogjuk venni egy B-fa szintjeinek a számát. A következő példából láthatjuk ennek az okát.

4.27. példa: Idézzük fel a 4.19. példa elemzését, ahol meghatároztuk, hogy a példa adatai esetén 340 kulcs-mutató pár fér el egyetlen blokkban. Tegyük fel, hogy egy átlagos blokk telítettségi középértéke valahol a minimum és a maximum között van, azaz egy tipikus blokk 255 mutatót tartalmaz. Egy gyökér, 255 gyermek és $255^2 = 65\,025$ levél esetén a levelekből 255^3 mutató indulhat ki, ami körülbelül 16,6 millió rekordra utaló mutató. Ily módon egy 3 szintű B-fa akár 16,6 millió rekordot tartalmazó fájlba is alkalmazható. □

Azonban a B-fában történő kereséshez háromnál kevesebb lemez I/O-műveletet is használhatunk. A B-fa gyökérblokkjának állandóan az elsődleges memóriában történő tárolása egy nagyon jó választás. Ilyen esetben a keresés egy 3 szintű B-fában mindössze két lemezolvasást igényel. Valójában, bizonyos körülmények között a B-fa második szintű csúcsait is tarthatjuk az elsődleges memóriában, egyre csökkentve ily módon a kereséshez szükséges lemez I/O-műveletek számát, ehhez jön természetesen esetenként az adatfájl blokkjainak manipulálásához szükséges lemez I/O-műveletek száma.

4.3.8. Feladatok

4.3.1. feladat: Tegyük fel, hogy egy blokkban elfér tíz rekord vagy 99 kulcs és 100 mutató. Tegyük fel továbbá, hogy a B-fa átlagos csúcsának a telítettsége 70%; azaz 69 kulcsot és 70 mutatót tartalmaz. A B-fákat felhasználhatjuk több különböző adatszerkezet részeként. Az alábbiakban bemutatott valamennyi adatszerkezetre határozzuk meg i) a szükséges adatblokkok számát egy olyan fájl esetén, amelyik 1 000 000 re-

Szükséges-e törölnünk B-fákból?

Vannak olyan B-fa-megvalósítások, ahol a törlést egyáltalán nem szervezik meg. Ha egy levélben túl kevés kulcs és mutató van, megengedett, hogy így maradjon. Az igazság az, hogy a legtöbb fájl egyenletesen nő, és ha alkalmanként elő is fordul olyan törlés, amellyel egy levél a minimum alá csökken, a levél valószínűleg hamarosan ismét visszahízik, és ismét eléri a kulcs-mutató párok számára vonatkozó alsó határt.

Továbbá, ha rekordokra a B-fa indexen kívül máshonnan is mutatnak mutatók, akkor a rekordot egyszerűen egy ún. „sírkővel" helyettesítjük, és nem kell feltétlenül kitörölni B-fának azt a mutatóját, amely erre a rekordra mutat. Bizonyos körülmények között, ha garantált, hogy a kitörölt rekordra csak a B-fán keresztül történik hozzáférés, akkor a B-fa levelében a rekordra utaló mutató helyére sírkövet tehetünk. Ily módon, a rekord által elfoglalt hely újra felhasználható.

kordot tartalmaz, valamint ii) adott keresésikulcs-értékkel rendelkező rekord megtalálásához szükséges átlagos lemez I/O-műveletek számát. Feltételezhetjük, hogy kezdetben semmi sincs a memóriában, és hogy a keresési kulcs a rekordok elsődleges kulcsa.

* a) Az adatfájl egy szekvenciális fájl, amely a keresési kulcs alapján rendezett, és 10 rekord van egy blokkban. A B-fa egy sűrű index.
b) Ugyanaz, mint az a) esetben, viszont az adatfájl rekordjai nem rendezettek, és 10 rekord van egy blokkban.
c) Ugyanaz, mint az a) esetben, de a B-fa egy ritka index.
! d) Ahelyett, hogy a B-fa leveleiben olyan mutatók lennének, amelyek az adatrekordokra mutatnak, a B-fa levelei magukat a rekordokat tartalmazzák. Egy blokkban tíz rekord fér el, viszont átlagosan egy levélblokk 70% telítettségű; azaz 7 rekord van egy levélblokkban.
* e) Az adatfájl szekvenciális fájl, és a B-fa egy ritka index, viszont az adatfájl valamennyi elsődleges blokkja rendelkezik egy túlcsordulásblokkal. Átlagosan az elsődleges blokk tele van, és a túlcsordulásblokk félig telített. A rekordok azonban rendezetlenek az elsődleges és a túlcsordulásblokkokban.

4.3.2. feladat: Ismételjük meg a 4.3.1. feladatot arra az esetre, ha a lekérdezés olyan tartományt eredményez, amelybe 1000 rekord tartozik.

4.3.3. feladat: Tegyük fel, hogy a mutatók 4 bájt hosszúak, és a kulcsok 12 bájt hosszúak. Hány kulcsot és mutatót fog tartalmazni egy 16 384 bájtból álló blokk?

4.3.4. feladat: Mennyi a kulcsok, illetve a mutatók minimális száma egy B-fa i) belső csúcsaiban, illetve ii) leveleiben, ha:

* a) $n = 10$; azaz egy blokk 10 kulcsot és 11 mutatót tartalmaz.
b) $n = 11$; azaz egy blokk 11 kulcsot és 12 mutatót tartalmaz.

4.3.5. feladat: Hajtsuk végre a következő műveleteket a 4.23. ábrán. Írjuk le a változásokat azoknál a műveleteknél, amelyek módosítják a fát.

a) A 41-es kulcsértékű rekord megkeresése.
b) A 40-es kulcsértékű rekord megkeresése.
c) Az összes olyan rekord megkeresése, amelyek halmaza a 20 és 30 közötti tartományba tartozik.
d) Az összes olyan rekord megkeresése, amelynek kulcsa kisebb, mint 30.
e) Az összes olyan rekord megkeresése, amelynek kulcsa nagyobb, mint 30.
f) Az 1-es kulcsértékű rekord beszúrása.
g) A 12-es, 15-ös és 16-os kulcsértékű rekordok beszúrása.
h) A 23-as kulcsértékű rekord törlése.
i) A 23-as és annál nagyobb kulcsértékű rekordok törlése.

! **4.3.6. feladat:** Említettük, hogy a 4.21. ábrán látható levél és a 4.22. ábrán látható belső csúcs soha nem jelenhet meg ugyanabban a B-fában. Magyarázzuk meg, hogy miért.

4.3.7. feladat: Ha egy B-fában megengedettek az ismétlődő kulcsok, akkor szükség van néhány módosításra azokban az algoritmusokban, amelyeket ebben a fejezetben mutattunk be a keresésre, beszúrásra, illetve törlésre. Adjuk meg a módosításokat:

* a) keresésre,
b) beszúrásra,
c) törlésre.

! **4.3.8. feladat:** A 4.26. példában említettük, hogy lehetőség lenne bal (vagy jobb) oldali nem testvértől is kulcsot kölcsönözni, amennyiben a belső csúcsok kulcsainak karbantartására egy jóval bonyolultabb algoritmust használnánk. Adjunk meg egy olyan algoritmust, amely a kiegyenlítést az ugyanazon a szinten levő szomszédos csúcsoktól való kölcsönzéssel oldja meg, függetlenül attól, hogy az a csúcs, amelytől a kölcsönzés történt, testvére-e vagy sem a túl sok vagy túl kevés kulcs-mutató párt tartalmazó csúcsnak.

4.3.9. feladat: Ha 3 kulcsos, 4 mutatós csúcsokat használunk a fejezetben levő példákhoz, akkor hány különböző B-fa létezik, ha az adatfájlban:

*! a) 6 rekord van,
!! b) 10 rekord van,
!! c) 15 rekord van.

*! **4.3.10. feladat:** Tegyük fel, hogy olyan B-fa csúcsaink vannak, amelyekben 3 kulcs és 4 mutató számára van hely, éppúgy, mint a fejezet példáiban. Tegyük fel továbbá, hogy amikor szétvágunk egy levelet, akkor a mutatókat 2 és 2 arányban osztjuk meg, míg ha egy belső csúcsot vágunk szét, akkor az első 3 mutató az első (bal oldali) csúcshoz kerül, az utolsó 2 mutató pedig a második (jobb oldali) csúcshoz kerül. Egyetlen levéllel kezdünk, amely az 1, 2 és 3 kulcsokat tartalmazza. Ezután sorban beszúrjuk a 4, 5, 6 kulcsokat és így tovább. Melyik kulcs beszúrásánál éri el a B-fa szintjeinek száma először a négyet?

!! **4.3.11. feladat:** Adott egy B+-fa szerkezetű index. A levél csúcsok mutatói összesen N rekordra mutatnak, és valamennyi, az indexet felépítő blokk m mutatót tartalmaz. Szeretnénk megválasztani az m értékét úgy, hogy az minimalizálja a keresési időket azon a lemezen, amely a következő sajátosságokkal rendelkezik:

i) Egy adott blokk memóriába történő beolvasásának ideje körülbelül $70 + 0{,}5m$ milliszekundum. A 70 milliszekundum a beolvasás keresési és lappangási idejét jelenti, míg a $0{,}5m$ milliszekundum az átviteli idő. Ily módon, ha az m nő, a blokk mérete is nő, és több időbe kerül a memóriába való beolvasás.
ii) Ha a blokk egyszer már a memóriában van, akkor bináris keresést használunk a megfelelő mutató megtalálásához. Ily módon, egy blokk feldolgozási ideje az elsődleges memóriában $a + b \log_2 m$ milliszekundum, bizonyos a és b konstansokra.
iii) Az elsődleges memória a konstansa sokkal kisebb, mint a 70 milliszekundum, ami a lemez keresési és lappangási ideje.
iv) Az index tele van, így keresésenként $\log_m N$ számú blokkot kell megvizsgálni.

Adjunk feleletet a következőkre:

a) Milyen m érték minimalizálja egy adott rekord keresési idejét?
b) Mi történik, ha a lemez keresési és lappangási ideje (70 ms) csökken? Például, ha ez a konstans a felére csökken, hogyan változik az m optimális értéke?

4.4. Tördelőtáblázatok

Igen sok indexként hasznos olyan adatszerkezet létezik, amelyik magában foglal egy tördelőtáblázatot. Feltételezzük, hogy az olvasó találkozott már a tördelőtáblázattal, mint elsődleges memóriában használt adatszerkezettel. Egy ilyen szerkezetben van egy *tördelőfüggvény*, amely argumentumként megkap egy keresési kulcsot (amelyet *tördelőkulcsnak* is nevezhetünk), és eredményül ad egy 0 és $B - 1$ közötti egész számot, ahol B a *kosarak* száma. A *kosártömb* egy olyan tömb, amelynek indexei 0 és $B - 1$ között vannak, és B számú, a tömb valamennyi kosarához egy-egy, a láncolt lista fejlécét tartalmazza. Ha egy rekord keresési kulcsa K, akkor a rekordot a $h(K)$-val számozott kosárnál láncoljuk a kosárlistához, ahol h a tördelőfüggvény.

4.4.1. Másodlagos tárolón tárolt tördelőtáblázatok

Egy tördelőtáblázat, amely olyan sok rekordot tartalmaz, hogy többnyire másodlagos tárolón kell tartani, apró, de fontos dolgokban különbözik az elsődleges memóriában tárolt változattól. Először is, a kosártömb blokkokból áll, és nem a listák fejléceire mutató mutatókból. Azok a rekordok, amelyeket a h tördelőfüggvény egy bizonyos kosárba tördel, az adott kosárhoz tartozó blokkban vannak. Ha egy kosár *túlcsordul*, azaz nem képes befogadni az összes hozzá tartozó rekordot, akkor egy *túlcsordulásblokkokból* álló láncot lehet hozzáadni a kosárhoz, hogy több rekordot be tudjon fogadni.

Feltételezzük, hogy bármely i kosár első blokkja megtalálható, adott i érték esetén. Például, az elsődleges memóriában lehet egy olyan tömbünk, amelyik a kosarak sorszámával indexelt és a blokkokra mutató mutatókból áll. Egy másik lehetőség, hogy valamennyi kosár első blokkját rögzített, egymás utáni lemezterületekre tesszük, így kiszámolható az i kosár első blokkja, adott i érték esetén.

4.28. példa: A 4.30. ábrán egy tördelőtáblázatot láthatunk. A példa átláthatóságának megőrzése érdekében feltételezzük, hogy egy blokkban csak két rekord fér el, és, hogy $B = 4$; azaz a h tördelőfüggvény 0 és 3 közötti értékeket ad vissza. A tördelőtáblázatot benépesítő rekordokat feltüntettük. A 4.30. ábrán látható kulcsok a és f közötti betűk. Feltételezzük, hogy $h(d) = 0$, $h(c) = h(e) = 1$, $h(b) = 2$ és $h(a) = h(f) = 3$. A hat rekord ily módon megoszlik a blokkok között, amint az látható is. □

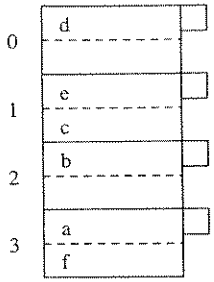
Figyeljük meg, hogy a 4.30. ábra valamennyi blokkjának jobb szélén egy „kinövés" látható. Ez a kinövés további információkat reprezentál a blokk fejlécében.

Tördelőfüggvény megválasztása

A tördelőfüggvénynek úgy kell „tördelnie" a kulcsot, hogy az eredményül kapott egész szám látszólag véletlenszerűen függjön a kulcstól. Ily módon a kosarak közel egyenlő számú rekordot fognak tartalmazni, ami javítja a rekordhozzáférések átlagos idejét, amint arról a 4.4.4. részben olvashatunk majd. Mindamellett a tördelőfüggvény könnyen kiszámítható kell legyen, mivel sokszor kerül majd kiszámításra.

- Ha a kulcsok egész számok, akkor a tördelőfüggvényt általában úgy választjuk meg, hogy a K/B maradékát számolja ki, ahol K a kulcsérték és B a kosarak száma. A B-t általában úgy választjuk meg, hogy prímszám legyen, habár indokolt lehet a B-t 2 valamilyen hatványának választani, ahogyan arról a 4.4.5. résztől kezdve olvashatunk.
- Ha a keresési kulcsok karakterláncok, akkor valamennyi karaktert kezelhetjük úgy, mint egész számot, összegezzük ezeket az egész számokat, és vegyük az összeg B-vel történő osztásának a maradékát.

Használhatjuk túlcsordulásblokkok összeláncolásához, és a 4.4.5. résztől kezdődően használjuk egyéb, blokkra vonatkozó kritikus információk tárolására.



4.30. ábra. *Tördelőtáblázat*

4.4.2. Beszúrás tördelőtáblázatba

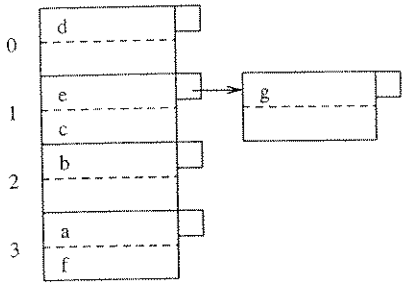
Ha egy K keresésikulcs-értékű új rekordot kell beszúrni, akkor kiszámoljuk a $h(K)$-t. Ha a $h(K)$ jelzőszámú kosárban van szabad hely, akkor a rekordot beszúrjuk a kosárhoz tartozó blokkba, vagy ha az első blokkban nincs hely, akkor a kosárhoz tartozó lánc valamelyik túlcsordulásblokkjába. Ha a $h(K)$ sorszámú kosárhoz tartozó lánc egyik blokkjában sincs szabad hely, akkor hozzáadunk a lánchoz egy újabb túlcsordulásblokkot, és ide szúrjuk be az új rekordot.

4.29. példa: Tegyük fel, hogy a 4.30. ábrán látható tördelőtáblázathoz szeretnénk hozzáadni a g kulcsértékű rekordot, és $h(g) = 1$. Így az új rekordot az 1-es sorszámú kosárba kell helyeznünk, amely felülről a második kosár. Az ehhez a kosárhoz tartozó blokkban azonban már van két rekord. Ezért hozzáadunk egy új blokkot, és az 1-es kosárhoz tartozó eredeti blokkhoz láncoljuk. A g kulcsértékű rekord ebbe a blokkba kerül, ahogyan azt a 4.31. ábrán is láthatjuk. □

4.4.3. Törlés tördelőtáblázatban

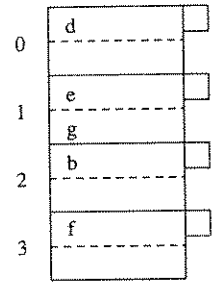
A K keresésikulcs-értékű rekord (illetve rekordok) törlése hasonló szabályszerűséget követ. Vesszük a $h(K)$ sorszámú kosarat, és megkeressük benne a megfelelő kulcsértékű rekordokat. Valamennyi megtalált rekordot töröljük. Ha van lehetőségünk a blokkok közötti rekordmozgatásra, akkor opcionálisan konszolidálhatjuk egy lánc blokkjait.[9]

[9] Ha egy lánc blokkjainak konszolidálása lehetséges, az azzal a veszéllyel jár, hogy egy oszcillálás, amikor felváltva szúrunk be és törlünk rekordokat a kosárból, azt fogja okozni, hogy mindegyik lépésnél egy blokkot létre kell hozni vagy ki kell törölni.



4.31. ábra. *A tördelőtáblázat egyik kosarához hozzáadunk egy további blokkot*

4.30. példa: A 4.32. ábrán láthatjuk a c kulcsértékű rekord 4.31. ábrából történő törlésének eredményét. Emlékezzünk vissza, hogy $h(c) = 1$, tehát az 1-es jelzőszámú (azaz második) kosárhoz kell mennünk, és végig kell keresnünk a hozzá tartozó valamennyi blokkot, hogy megtaláljuk a c kulcsértékű rekordot (illetve rekordokat, ha a keresési kulcs nem elsődleges kulcs). Az 1-es jelzőszámú kosárhoz tartozó lánc első blokkjában megtaláljuk. Most van hely arra, hogy a második blokk g kulcsértékű rekordját áttegyük a lánc első blokkjába, és ily módon töröljük a második blokkot.



4.32. ábra. *Tördelőtáblázatból történő törlések eredménye*

Bemutatjuk az a kulcsértékű rekord törlését is. Ezt a kulcsot a 3-as jelzőszámú kosárban találjuk, kitöröljük, és a megmaradó rekordot a blokk elejére „konszolidáljuk". □

4.4.4. Tördelőtáblázat-indexek hatékonysága

Ideális esetben elég sok kosarunk van ahhoz, hogy a legtöbb kosár egy blokkból álljon. Ha ez így van, akkor a tipikus keresés csak egy lemez I/O-művelettel jár, míg a beszúrás és a törlés két lemez I/O-műveletet igényel. Ez a szám jelentősen jobb, mint a hagyományos ritka vagy sűrű indexeknél, illetve a B-fa-indexeknél (habár a tördelőtáblázatok a B-fákkal ellentétben nem támogatják a 4.3.4. részben bemutatott tartományt eredményező lekérdezéseket).

Ha azonban a fájl nő, előbb-utóbb eljutunk egy olyan helyzethez, amikor egy átlagos kosárhoz tartozó lánc sok blokkot tartalmaz. Ha ez így van, akkor blokkok hosszú listáit kell végignéznünk, amely legalább egy lemez I/O-műveletet jelent blokkonként. Jó okunk van tehát rá, hogy az egy kosárra eső blokkok számát alacsonyan tartsuk.

Az eddig vizsgált tördelőtáblázatokat *statikus tördelőtáblázatoknak* nevezzük, mivel a B, a kosarak száma soha nem változik. Léteznek azonban különböző *dinamikus tördelőtáblázatok* is, ahol a B változhat, azaz a B megközelíti azt a számot, amelyet úgy kapunk, ha elosztjuk a rekordok számát azon rekordok számával, amelyek elférnek egy blokkban; ez azt jelenti, hogy körülbelül egy blokk tartozik egy kosárhoz. Két ilyen módszert fogunk bemutatni:

1. a kiterjeszthető tördelést a 4.4.5. részben és
2. a lineáris tördelést a 4.4.7. részben.

Az első úgy növeli a B értékét, hogy megduplázza azt, valahányszor túl kevésnek bizonyul, és a második mindig 1-gyel növeli a B értékét, valahányszor a fájl statisztikái alapján növelésre van szükség.

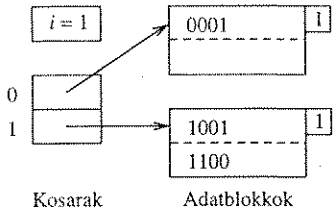
4.4.5. Kiterjeszthető tördelőtáblázatok

A dinamikus tördelés első megközelítését *kiterjeszthető tördelőtáblázatoknak* nevezzük. Az egyszerűbb statikus tördelőtáblázathoz képest a főbb kiegészítések:

1. A kosarakhoz egy közvetett szintet vezetünk be. Ez azt jelenti, hogy a kosarakat egy olyan, mutatókból álló tömb reprezentálja, ahol a mutatók blokkokra mutatnak ahelyett, hogy a tömb magukat az adatblokkokat tartalmazná.
2. A mutatókból álló tömb nőhet. Hossza mindig a 2 valamilyen hatványa, tehát egy növekedési lépésben a kosarak száma megduplázódik.
3. Mindamellett nem kell valamennyi kosárnak rendelkeznie egy adatblokkal; bizonyos kosarak osztozhatnak egy blokkon, ha a kosarakban levő rekordok elférnek ebben a blokkban.
4. A h tördelőfüggvény valamennyi kulcs esetén egy k bitből álló sorozatot számol ki, ahol a k elég nagy, mondjuk 32. A kosárjelzőszámok azonban mindenkor kevesebb számú bitet használnak, mondjuk i bitet a sorozat elejéről. Ily módon a kosártömbnek 2^i bejegyzése lesz, ahol i a felhasznált bitek száma.

4.31. példa: A 4.33. ábrán egy kisméretű, kiterjeszthető tördelőtáblázatot láthatunk. Az egyszerűség kedvéért tegyük fel, hogy $k = 4$; azaz a tördelőfüggvény egy mindösszesze négy bitből álló sorozatot ad vissza. Jelenleg ezen bitekből csak egy használatos, ahogyan azt fel is tüntettük a kosártömb fölötti dobozban. A kosártömbnek ily módon mindössze két bejegyzése van, egy a 0-hoz és egy az 1-hez.

A kosártömb bejegyzései két blokkra mutatnak. Az első tartalmazza az összes olyan aktuális rekordot, amelynek keresési kulcsa 0-val kezdődő bitsorozatot tördel, a



4.33. ábra. *Kiterjeszthető tördelőtáblázat*

második pedig azokat tartalmazza, amelyek keresési kulcsa 1-gyel kezdődő sorozatot tördel. A kényelem kedvéért a rekordok kulcsait úgy ábrázoljuk, mintha azok megegyeznének azokkal a teljes bitsorozatokkal, amelyekké a tördelőfüggvény konvertálja őket. Ily módon az első blokk egy rekordot tartalmaz, amelynek kulcsa a 0001-et tördeli, a második blokk azokat a rekordokat tartalmazza, amelyek kulcsai az 1001-et és az 1100-t tördelik. □

A 4.33. ábrán megfigyelhetjük, hogy valamennyi blokk kinövésében megjelenik az 1-es szám. Ez a szám, amely tulajdonképpen a blokk fejlécében is megjelenhet, azt jelzi, hogy a tördelőfüggvény által visszaadott sorozatból hány bit használatos annak eldöntésére, hogy egy rekord az adott blokkhoz tartozik-e vagy sem. A 4.31. példában valamennyi blokk és rekord esetén egy bit használatos, de amint azt majd látni fogjuk, a különböző blokkokhoz használatos bitek száma változhat, ahogyan a tördelőtáblázat növekszik. Ily módon a kosártömb mérete az aktuálisan használt bitek száma által meghatározott, de előfordulhat, hogy bizonyos blokkok kevesebbet használnak.

4.4.6. Beszúrás kiterjeszthető tördelőtáblázatokba

Egy kiterjeszthető tördelőtáblázatba történő beszúrás ugyanúgy kezdődik, mint egy statikus tördelőtáblázatba történő beszúrás. A K keresésikulcs-értékű rekord beszúrásához kiszámoljuk a $h(K)$ bitsorozatot, ennek vesszük az első i bitjét, és a kosártömb azon bejegyzéséhez megyünk, amelynek jelzőszáma ez az i bit. Megjegyzendő, hogy az i-t azért tudjuk meghatározni, mert a tördelő-adatszerkezetben el van tárolva.

Követjük a kosártömb ezen bejegyzésének mutatóját, és elérkezünk egy B blokkhoz. Ha van szabad hely a B-ben az új rekord elhelyezésére, akkor ezt megtesszük, és készen is vagyunk. Ha nincs szabad hely, akkor a j értékétől függően két lehetőségünk van; a j azt jelzi, hogy a tördelőfüggvény által kiszámolt érték hány bitje használatos annak eldöntésére, hogy a rekord a B blokkhoz tartozik vagy nem (ne feledjük, hogy a j érték az ábrán a blokkok kinövéseiben található).

1. Ha $j < i$, akkor a kosártömbbel semmit nem kell tennünk. Amit meg kell tennünk:

   a) A B blokkot kettévágjuk.
   b) A B rekordjait szétosztjuk a két blokk között, a $(j + 1)$-edik bit értéke alapján –

azok a rekordok, amelyek kulcsa 0 az adott bitnél, maradnak a B blokkban, míg azok a rekordok, melyek ezen a helyen 1-et tartalmaznak, az új blokkba kerülnek.

c) Mindegyik blokk „kinövésébe" $j + 1$ kerül, jelezvén, hogy ennyi bit használatos az odatartozás eldöntésére.

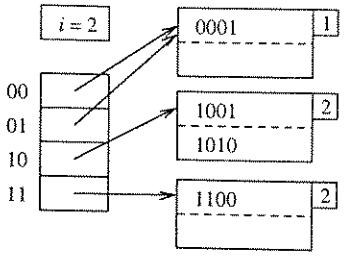
d) A kosártömb mutatóit az új helyzethez igazítjuk úgy, hogy azok a bejegyzések, amelyek azelőtt a B-re mutattak, most vagy a B-re vagy az új blokkra mutassanak, a $(j + 1)$-edik bittől függően.

Megjegyzendő, hogy a B blokk szétvágása nem biztos, hogy megoldja a problémát, hiszen a véletlen hozhatja úgy, hogy a B valamennyi rekordja a két blokk egyikébe kerül a szétvágás után. Ha ez így van, akkor meg kell ismételnünk az eljárást a j következő értékére, és arra a blokkra, amelyik még mindig tele van.

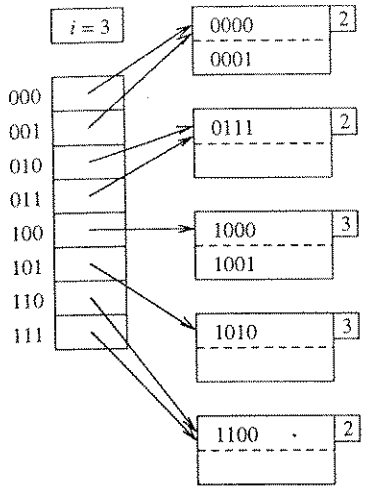
2. Ha $j = i$, akkor először meg kell növelnünk 1-gyel az i értékét. Megduplázzuk a kosártömb hosszát, hiszen most 2^{i+1} bemenetet tartalmaz. Tegyük fel, hogy a w egy i számú bitből álló sorozat, amely az előző kosártömb egyik bejegyzésének volt a jelzőszáma. Az új kosártömbben a $w0$ és $w1$ jelzőszámú bejegyzések (azaz a w 0-val és 1-gyel történő kiterjesztéséből származó két új szám) ugyanarra a blokkra mutatnak, mégpedig arra, amelyikre a w bejegyzés mutatott. Ez azt jelenti, hogy a két új bejegyzés megosztozik a blokkon, és a blokk maga nem változik. A blokkhoz való tartozás ugyanúgy meghatározott, mint a bitek előzőleg használt száma esetében. Végül kettévágjuk a B blokkot ugyanúgy, mint az 1. esetben. Mivel az i most nagyobb, mint j – alkalmazható az előző eset.

4.32. példa: Tegyük fel, hogy beszúrunk a 4.33. ábrán látható táblába egy olyan rekordot, melynek kulcsa az 1010 sorozatot tördeli. Mivel az első bit 1, a rekord a második blokkhoz tartozik. Azonban ez a blokk már tele van, tehát szét kell vágni. Mivel ebben az esetben $j = i = 1$, ezért először meg kell dupláznunk a kosártömböt a 4.34. ábrán látható módon. Az ábrán feltüntettük azt is, hogy $i = 2$.

Figyeljük meg, hogy mindkét 0-val kezdődő bejegyzés arra a blokkra mutat, amelyben a tördelt kulcsok 0-val kezdődnek, és ez a blokk még mindig az 1-es egész számot tartalmazza a „kinövésben", ami azt jelenti, hogy csak az első bit használatos a blokkhoz való tartozás eldöntésére. Azonban az 1-gyel kezdődő rekordok blokkját ketté kell vágnunk, így a hozzá tartozó rekordokat is szét kell válogatnunk az 10-val



4.34. ábra. *Most a tördelőfüggvény két bitje van használatban*



4.35. ábra. *A tördelőtáblázat most a tördelőfüggvény három bitjét használja*

és az 11-gyel kezdődő rekordokra. Mindkét blokkban egy 2-es jelzi, hogy az odatartozást két bit határozza meg. Szerencsére a szétvágás sikeres, mivel a két új blokk mindegyike tartalmaz legalább egy rekordot, így nem kell rekurzívan tovább vágnunk.

Most tegyük fel, hogy beszúrjuk azokat a rekordokat, amelyek kulcsai 0000-t és 0111-et tördelnek. Mindkét rekord a 4.34. ábra első blokkjába kerül, amely ezzel túlcsordul. Mivel ebben a blokkban csak egy bit használatos az odatartozás eldöntésére, és $i = 2$, ezért a kosártömbbel nem kell foglalkoznunk. Egyszerűen csak kettévágjuk a blokkot, a 0000 és 0001 a régiben maradnak, és a 0111 az új blokkba kerül. A kosártömb 01 bejegyzését beállítjuk, hogy az új blokkra mutasson. Ismét szerencsénk volt, hogy nem az összes rekord került az új blokkok valamelyikébe, így nem kellett rekurzívan tovább vágnunk.

Most tegyük fel, hogy egy olyan rekordot szúrunk be, amelynek kulcsa 1000-t tördel. Az 10-hoz tartozó blokk túlcsordul. Mivel ez már eleve 2 bitet használ az odatartozás eldöntésére, itt az ideje, hogy a kosártömböt ismét megnöveljük, és beállítsuk, hogy $i = 3$. A 4.35. ábra ezt az adatszerkezetet mutatja be. Figyeljük meg, hogy az 10-hoz tartozó blokkot kettévágtuk az 100-hoz és az 101-hez tartozó blokkokra, míg a többi blokk továbbra is két bitet használ az odatartozás eldöntésére. □

4.4.7. Lineáris tördelőtáblázatok

A kiterjeszthető tördelőtáblázatoknak van néhány fontos előnye. A legjelentősebb az, hogy egy rekord megtalálásához mindig csak egy adatblokkban kell keresnünk. A kosártömb egy bejegyzését is meg kell vizsgálnunk, ha azonban a kosártömb elég kicsi ahhoz, hogy elférjen az elsődleges memóriában, akkor nincs szükség lemez I/O-műve-

letre ahhoz, hogy hozzáférjünk a kosártömbhöz. Azonban a kiterjeszthető tördelőtáblázatok rendelkeznek néhány hiányossággal is:

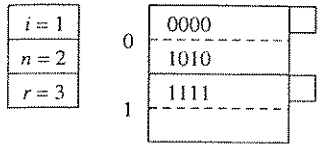
1. Amikor meg kell dupláznunk a kosártömböt, akkor tekintélyes mennyiségű munkát kell végezni (ha az i nagy). Ez a munka megszakítja az adatfájlhoz való hozzáférést, vagy igen lassúvá tesz bizonyos beszúrásokat.
2. Ha a kosártömb méretét megduplázzuk, lehet, hogy nem fér majd el az elsődleges memóriában, vagy esetleg kiszorít olyan adatokat, amelyeket az elsődleges memóriában szeretnénk tartani. Ennek eredményeként egy eddig jól működő rendszer hirtelen elkezd sokkal több lemez I/O-műveletet használni, és elér egy észrevehető teljesítménycsökkenést.
3. Ha a blokkonkénti rekordok száma kicsi, akkor valószínű, hogy lesz egy olyan blokk, amelyet ketté kell majd vágni jóval előbb, mint ahogyan logikailag itt lenne az ideje. Ha például éppúgy, mint az eddigi példákban, két rekord van egy blokkban, akkor lehetséges, hogy három rekordnál ugyanaz a 20 bitből álló sorozat található, még akkor is, ha a rekordok összesen jóval kevesebben vannak, mint 2^{20}. Ebben az esetben $i = 20$ és egymillió bejegyzést kell használnunk a kosártömbben, még akkor is, ha a rekordokat tartalmazó blokkok száma jóval kevesebb, mint egymillió.

Egy másik stratégia, amit lineáris tördelőtáblázatoknak hívunk, jóval lassabban növeli a kosarak számát. A lineáris tördelés legfontosabb új elemei:

- n, a kosarak száma mindig úgy alakul ki, hogy a rekordok blokkonkénti átlagos száma a blokkot megtöltő rekordoknak egy állandó hányadát képezze, mondjuk 80%-át.
- Mivel a blokkokat nem lehet mindig szétvágni, ezért a túlcsordulásblokkok megengedettek, habár az egy kosárra eső túlcsordulásblokkok átlagos száma jóval kevesebb lesz, mint 1.
- A kosártömb bejegyzéseinek megszámozására használt bitek száma $\lceil \log_2 n \rceil$, ahol n a kosarak aktuális száma. Ezeket a biteket mindig a tördelőfüggvény által visszaadott bitsorozat *jobb* széléről vesszük (alacsony prioritás).
- Tegyük fel, hogy a tördelőfüggvény i bitje használatos a tömb bejegyzéseinek megszámozására, és hogy egy K kulcsú rekordot szánunk az $a_1a_2...a_i$ kosárba; ez azt jelenti, hogy a $h(K)$ utolsó i bitje $a_1a_2...a_i$-t. Tekintsük az $a_1a_2...a_i$-t, mint egy i bitből álló bináris egészet, és jelöljük m-mel. Ha $m < n$, akkor az m jelzőszámú kosár létezik, és a rekordot ebben a kosárban helyezzük el. Ha $n \leq m < 2^i$, akkor az m jelzőszámú kosár még nem létezik, így a rekordot az $m - 2^{i-1}$ jelzőszámú kosárban helyezzük el, amit úgy kapnánk, ha kicserélnénk az a_1-et (aminek 1-nek kell lennie) 0-ra.

4.33. példa: A 4.36. ábrán egy lineáris tördelőtáblázatot láthatunk, ahol $n = 2$. Jelenleg a tördelési értéknek csak egy bitjét használjuk a rekordok kosarakhoz való tartozásának eldöntésére. A 4.31. példában bemutatott törvényszerűséget felhasználva tegyük fel, hogy a h tördelőfüggvény 4 bitet hoz létre, és a rekordokat azzal az értékkel ábrázoljuk, amit a rekord keresési kulcsára alkalmazott h függvény eredményez.

A 4.36. ábrán két kosarat láthatunk, mindkettő egy blokkból áll. A kosarak jelzőszámai 0 és 1. Minden olyan rekord, amelynek tördelési értéke 0-ra végződik, az első kosárba kerül, míg azok, amelyeknek tördelési értéke 1-re végződik, a második kosárba kerülnek.



4.36. ábra. *Lineáris tördelőtáblázat*

Az adatszerkezet részét képezi még az i paraméter (a tördelőfüggvényből használatos bitek száma), az n (a kosarak aktuális száma) és az r (a tördelőtáblázat rekordjainak aktuális száma). Az r/n arány korlátozott lesz, így az átlagos kosár körülbelül egy blokkot igényel majd. Az n megválasztásakor azt az elvet követjük, mely szerint a fájlban legfeljebb $1{,}7n$ rekord van, azaz $r \leq 1{,}7n$. Ily módon, mivel a blokkok két rekordot tartalmaznak, egy kosár átlagos kihasználtsága nem haladja meg egy blokk kapacitásának a 85%-át. □

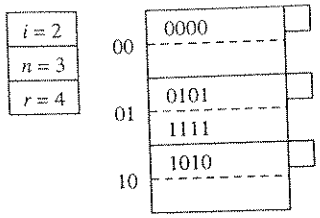
4.4.8. Beszúrás lineáris tördelőtáblázatokba

Amikor egy új rekordot szúrunk be, akkor a megfelelő kosarat a 4.4.7. részben vázolt algoritmussal határozzuk meg. Ez azt jelenti, hogy kiszámoljuk a $h(K)$-t, ahol K a rekord kulcsa, és meghatározzuk azt a számot, ahány bitet figyelembe kell vennünk a $h(K)$ bitsorozat végéről, hogy aztán a kosár jelzőszámaként használjuk. A rekordot vagy ebbe a kosárba tesszük, vagy (ha a kosár jelzőszáma nagyobb vagy egyenlő, mint n) abba a kosárba, amit úgy kapunk, hogy a vezető bitet 1-ről 0-ra cseréljük. Ha a kosárban nincs szabad hely, akkor készítünk egy túlcsordulásblokkot, hozzáláncoljuk a kosárhoz, és ebbe tesszük a rekordot.

Minden egyes beszúrásnál összehasonlítjuk a rekordok aktuális számát, az r-et, az r/n hányados felső határával, és ha ez a hányados túl magas, akkor hozzáadjuk a táblához a következő kosarat. Megjegyzendő, hogy az általunk hozzáadott kosárnak nincs semmi köze ahhoz a kosárhoz, amelybe a beszúrás történik! Ha a hozzáadott kosár jelzőszámának bináris reprezentációja $1a_2...a_i$, akkor szétszedjük a $0a_2...a_i$ jelzőszámú kosarat, oly módon, hogy annak rekordjait egyik vagy másik kosárba tesszük, az utolsó i bitjüktől függően. Figyeljük meg, hogy valamennyi rekord tördelési értéke $a_2...a_i$-re végződik, és csupán jobbról az i-edik bit fog változni.

Az utolsó fontos részlet, hogy mi történik, ha az n túllépi a 2^i értéket. Ekkor az i-t megnöveljük eggyel. Technikailag valamennyi kosár jelzőszáma kap egy 0-t a saját bitsorozata elé, de semmi más fizikai változtatásra nincs szükség, hiszen ezek a bitsorozatok, egész számként értelmezve, ugyanazok maradnak.

4.34. példa: Folytatjuk a 4.33. példát, és megnézzük, mi történik, ha egy olyan rekordot szúrunk be, amelynek kulcsa a 0101 értéket tördeli. Mivel ez a bitsorozat 1-re



4.37. ábra. *Egy harmadik kosár hozzáadása*

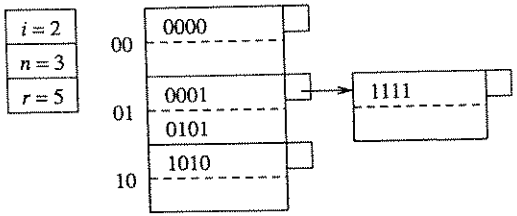
végződik, a rekord a 4.36. ábra második kosarába kerül. Van szabad hely a rekord számára, így nem kell túlcsordulásblokkot készíteni.

Mivel azonban most 4 rekord van 2 kosárban, túlléptük az 1,7 hányadost, ezért fel kell emelnünk az n értékét 3-ra. Mivel $\lceil \log_2 3 \rceil = 2$, kezdhetünk úgy gondolni a 0 és 1 kosarakra, mint 00 és 01 kosarakra, de nem szükséges módosítani az adatszerkezetet. Hozzáadjuk a táblához a következő kosarat, amelynek a jelzőszáma 10 lesz. Ezután szétszedjük a 00 kosarat, azt a kosarat, amelynek jelzőszáma csak az első bitben különbözik a hozzáadott kosártól. Amikor elvégezzük a szétszedést, akkor az a rekord, amelynek kulcsa 0000-t tördel, marad a 00-s kosárban, míg az a rekord, amelynek kulcsa 1010-t tördel, átmegy az 10-s kosárba, mivel a végződések így kívánják. Az eredményül kapott tördelőtáblázatot a 4.37. ábrán láthatjuk.

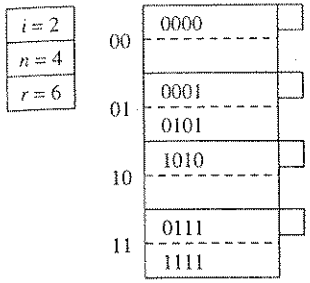
Most tegyük fel, hogy egy olyan rekordot akarunk beszúrni, amelynek keresési kulcsa 0001-et tördel. A két utolsó bit 01, így ebbe a kosárba tesszük, amely jelenleg létezik. Sajnos a kosár blokkja tele van, így hozzáadunk egy túlcsordulásblokkot. A három rekordot elosztjuk a kosár két blokkja között; a tördelt kulcsok numerikus sorrendjében helyezzük el őket, de a sorrend nem igazán fontos. Mivel a táblában a rekordok és a blokkok aránya 5/3, és ez kevesebb, mint 1,7, ezért nem készítünk új kosarat. Az eredményt a 4.38. ábrán láthatjuk.

Végül, nézzük meg egy olyan rekord beszúrását, amelynek keresési kulcsa 0111-et tördel. Az utolsó két bit 11, de az 11-es kosár nem létezik. Ily módon átirányítjuk ezt a rekordot a 01-es kosárba, amely szám csak a 0-s első bitben különbözik a keresett értéktől. Az új rekord befér a kosár túlcsordulásblokkjába.

Azonban a rekordok kosarakhoz viszonyított aránya meghaladja az 1,7-et, ezért létre kell hoznunk egy új kosarat, 11 jelzőszámmal. Ez a kosár történetesen az, amit az új rekord számára kerestünk. Szétosztjuk a 01-es kosár négy rekordját, a 0001-es és 0101-es marad, a 0111-es és az 1111-es átmegy az új kosárba. Mivel a 01-es kosár



4.38. ábra. *Szükség esetén túlcsordulásblokkokat használunk*



4.39. ábra. *Egy negyedik kosár hozzáadása*

jelenleg két rekordot tartalmaz, ezért eltörölhetjük a túlcsordulásblokkot. A tördelőtáblázat ezen állapotát a 4.39. ábrán láthatjuk.

Figyeljük meg, hogy a 4.39. ábrába történő következő beszúrásnál a rekordok és a kosarak aránya meg fogja haladni az 1,7-et. Ekkor meg fogjuk növelni az n értékét 5-re, és az i értéke 3 lesz. □

4.35. példa: Egy lineáris tördelőtáblázatban történő keresés megegyezik azzal az eljárással, amellyel kiválasztjuk azt a kosarat, amelybe a beszúrni kívánt rekord kerülni fog. Ha a keresett rekord nincs ebben a kosárban, akkor sehol máshol sem lehet. Szemléltetésképpen nézzük meg a 4.37. ábra helyzetét, ahol $i = 2$ és $n = 3$.

Először tegyük fel, hogy egy olyan rekordot akarunk megtalálni, amelynek kulcsa az 1010-t tördeli. Mivel $i = 2$, ezért a két utolsó bitet nézzük, ami 10, és ezt bináris egészként értelmezünk, nevezetesen $m = 2$. Mivel $m < n$, ezért az 10-s kosár létezik, itt fogjuk keresni. Ne feledjük, hogy csupán az a tény, hogy találunk egy olyan rekordot, amelynek tördelési értéke 1010, még nem jelenti azt, hogy ez az a rekord, amit keresünk; ahhoz, hogy ebben biztosak legyünk, ellenőriznünk kell a rekord teljes kulcsát.

Másodszor nézzük meg egy olyan rekordnak a keresését, amelynek kulcsa az 1011-et tördeli. Most olyan kosárban kell keresnünk, amelynek jelzőszáma 11. Mivel ez a szám bináris egészként $m = 3$, és $m \geq n$, az 11-es kosár nem létezik. Átmegyünk a 01-es kosárba, kicserélvén a vezető bitet 1-ről 0-ra. A 01-es kosárnak azonban nincs olyan rekordja, amelynek a kulcsa az 1011-et tördelné, így a keresett rekord biztosan nincs a tördelőtáblázatban. □

4.4.9. Feladatok

4.4.1. feladat: Mutassuk meg, hogy mi történik a 4.30. ábra kosaraival a következő beszúrások és törlések bekövetkeztével:

i) A g, h, i, j rekordok beszúrása a 0, 1, 2, 3 kosarakba.
ii) Az a és b rekordok törlése.
iii) A k, l, m, n rekordok beszúrása a 0, 1, 2, 3 kosarakba.
iv) A c és d rekordok törlése.

4.4.2. feladat: Nem mutattuk be, hogy miként lehet törléseket végrehajtani lineáris, illetve kiterjeszthető tördelőtáblázatokban. A törlendő rekordok megtalálásának mechanizmusa kézenfekvő. Milyen módszert javasolnánk a törlés végrehajtására? Ebben az esetben milyen előnyökkel, illetve hátrányokkal jár a tábla átrendezése, ha a törlés utáni kisebb méret lehetővé teszi bizonyos blokkok tömörítését?

! **4.4.3. feladat:** Ebben az részben feltételeztük, hogy a keresési kulcsok egyediek. Apró módosításokra van azonban csak szükség ahhoz, hogy ezek a technikák ismétlődő kulcsokra is alkalmazhatók legyenek. Írjuk le a beszúrás, törlés és keresés algoritmusaiban elvégzendő változtatásokat, és vázoljuk az ismétlődések okozta főbb problémákat:

* a) Egyszerű tördelőtáblázat esetén.
b) Kiterjeszthető tördelőtáblázat esetén.
c) Lineáris tördelőtáblázat esetén.

! **4.4.4. feladat:** Bizonyos tördelőfüggvények nem működnek olyan jól, mint ahogyan elméletileg lehetséges lenne. Tegyük fel, hogy olyan tördelőfüggvényt használunk, amely egész kulcsokra értelmezett, és a következőképpen definiáltuk: $h(i) = i^2 \bmod B$.

* a) Mi a gond ezzel a tördelőfüggvénnyel, ha $B = 10$?
b) Mennyire jó ez a tördelőfüggvény, ha $B = 16$?
c) Léteznek-e olyan B értékek, amelyekre ez a tördelőfüggvény hasznos?

4.4.5. feladat: Egy kiterjeszthető tördelőtáblázatban, amely blokkonként n rekordot tartalmaz, mi a valószínűsége annak, hogy egy túlcsordulásblokkot rekurzívan kelljen kezelni; azaz, hogy a blokk valamennyi rekordja a szétvágás által létrehozott két blokk közül ugyanabba kerüljön?

4.4.6. feladat: Tegyük fel, hogy a kulcsok négy bitből álló sorozatot tördelnek éppúgy, mint ezen rész kiterjeszthető és a lineáris tördeléssel foglalkozó példáiban. Tegyük fel azonban, hogy ezek a blokkok három rekordot képesek befogadni és nem kettőt, mint az eddigi példák blokkjai. Ha olyan tördelőtáblázattal kezdünk, amely két üres blokkot tartalmaz (0 és 1), mutassuk be a tördelőtáblázat szerkezetét a következő kulcsértékekkel rendelkező rekordok beszúrása után:

* a) 0000, 0001, ..., 1111, és a módszer a kiterjeszthető tördelés.
b) 0000, 0001, ..., 1111, és a módszer a lineáris tördelés, 75%-os kapacitásküszöbbel.
c) 1111, 1110, ..., 0000, és a módszer a kiterjeszthető tördelés.
d) 1111, 1110, ..., 0000, és a módszer a lineáris tördelés, 75%-os kapacitásküszöbbel.

* **4.4.7. feladat:** Tegyük fel, hogy lineáris vagy kiterjeszthető tördelési módszert használunk, de vannak olyan mutatók, amelyek a rekordokra mutatnak kívülről. Ezek a mutatók megakadályoznak bennünket abban, hogy rekordokat mozgassunk blokkok

között, ami pedig ezeknél a módszereknél néha szükséges. Javasoljunk néhány olyan eljárást, amelyek mellett módosítható a szerkezet, és engedélyezettek a külső mutatók is.

!! **4.4.8. feladat:** Egy lineáris tördelőelrendezés, k darab rekordot tartalmazó blokkokkal, olyan c küszöbállandót használ, hogy a kosarak n aktuális száma és a rekordok r aktuális száma közötti összefüggés $r = ckn$. Például a 4.33. példában azt használtuk, hogy $k = 2$ és $c = 0{,}85$, így 1,7 rekord volt egy kosárban, azaz $r = 1{,}7n$.

a) Az egyszerűség kedvéért tegyük fel, hogy mindegyik kulcs pontosan annyiszor jelenik meg, amennyi a várható értéke.[10] A túlcsordulásblokkokat is beleértve, hány blokkra van szükség ehhez az adatszerkezethez a c, k és az n függvényében?
b) A kulcsok általában nem egyenletesen oszlanak meg, sokkal inkább *Poisson-eloszlást* követ az egy adott kulccsal (vagy adott végződésű kulccsal) rendelkező rekordok száma. Ez azt jelenti, hogy ha egy adott végződésű kulccsal rendelkező rekordok várható száma λ, akkor annak a valószínűsége, hogy ezen rekordok aktuális száma i legyen, $e^{-\lambda}\lambda^i/i!$. Ilyen előfeltételek mellett számítsuk ki a felhasznált blokkok várható számát, a c, k és az n függvényében.

*! **4.4.9. feladat:** Tegyük fel, hogy van egy 1 000 000 rekordból álló fájlunk, amit egy 1000 kosárból álló táblázatba szeretnénk tördelni. Egy blokkban 100 rekord fér el, és a blokkokat szeretnénk minél jobban teletenni, de két kosár nem osztozhat ugyanazon a blokkon. Hány blokkra lehet szükségünk minimum és maximum ennek a tördelőtáblázatnak a tárolására?

4.5. Összefoglalás

- *Szekvenciális fájlok:* Különféle egyszerű fájlszerkezetek, amelyekben az adatfájlok rendezettek valamilyen keresési kulcs szerint, és ennek a fájlnak a tetejére kerül egy index.
- *Sűrű indexek:* Ezek az indexek az adatfájl valamennyi rekordjához tartalmaznak egy kulcs-mutató párt. Ezen párok tárolása rendezett a kulcsértékeik szerint.
- *Ritka indexek:* Ezek az indexek az adatfájl valamennyi blokkjához tartalmaznak egy kulcs-mutató párt. A blokkra utaló mutatóhoz tartozó kulcs tulajdonképpen a blokkban található első kulcs.
- *Többszintű indexek:* Néha hasznos az indexfájlra is indexet készíteni, erre az indexre újabb indexet és így tovább. Az index magasabb szintjei kötelezően ritka indexek.
- *Fájlok kibővítése:* Ahogyan egy adatfájl és a hozzá tartozó indexfájl (illetve fájlok) mérete növekszik, szükséges néhány intézkedés további blokkok fájlhoz történő hozzáadására. Az egyik lehetőség túlcsordulásblokkok hozzáadása az eredeti blok-

[10] Ez a feltevés nem jelenti azt, hogy valamennyi kosár ugyanannyi rekordot tartalmaz, mivel bizonyos kosarak kétszer annyi kulcsot jelölnek, mint mások.

kokhoz. Az adatfájl vagy az indexfájl blokkjai közé is beszúrhatunk további blokkokat, kivéve, ha a fájl blokkjainak egymás után kell elhelyezkedniük a lemezen.

- *Másodlagos indexek:* Egy K keresési kulcson akkor is készíthető index, ha az adatfájl nem rendezett K szerint. Egy ilyen index mindig sűrű.
- *Invertált indexek:* A dokumentumok és az őket felépítő szavak közötti kapcsolatot gyakran ábrázolják úgy, mint egy szó-mutató párokból alkotott indexet. A mutató a kosárfájl azon helyére mutat, ahol a szó előfordulására utaló mutatók listája található.
- *B-fák:* Ezek tulajdonképpen többszintű indexek, a méret növekedésére vonatkozó könnyed lehetőségekkel. Egy fát n kulcsból és $n + 1$ mutatóból álló blokkok alkotnak, és a levelek mutatnak a rekordokra. Valamennyi blokk telítettsége minden időben valahol a félig telítettség és a teljes telítettség között van.
- *Tartományra vonatkozó lekérdezések:* Azokat a lekérdezéseket, amelyekkel olyan rekordokat keresünk, amelyek keresésikulcs-értékei egy megadott tartományba tartoznak, az indexelt szekvenciális fájlok és a B-fa-indexek támogatják, de a tördelőtáblázat-indexek nem.
- *Tördelőtáblázatok:* Tördelőtáblázatokat készíthetünk másodlagos memóriában blokkokból nagyjából ugyanúgy, ahogyan elsődleges memóriában készíthetünk tördelőtáblázatokat. Egy tördelőfüggvény leképezi kosarakba a keresésikulcs-értékeket, gyakorlatilag több kisebb csoportba (kosarakba) particionálva ezáltal az adatfájl rekordjait. A kosarak megvalósítása egy blokkal és további lehetséges túlcsordulásblokkokkal történik.
- *Dinamikus tördelés:* Mivel a tördelőtáblázat hatékonysága csökken, ha túl sok rekord van egy kosárban, ezért a kosarak számának növelése indokolttá válhat az idő múlásával. A méret növekedésére vonatkozó könnyed lehetőségekkel két módszer rendelkezik: a kiterjeszthető és a lineáris tördelés. Mindkettő úgy kezdődik, hogy hosszú bitsorozatokká tördeli a keresésikulcs-értékeket, és ezekből váltakozó számú bitet használ fel a rekordhoz tartozó kosár meghatározására.
- *Kiterjeszthető tördelés:* Ez a módszer lehetővé teszi a kosarak számának megduplázását, valahányszor egy kosár túl sok rekordot tartalmaz. A kosarak ábrázolására egy blokkokra utaló mutatókból álló tömböt használ. A túl sok blokk elkerülése érdekében bizonyos kosarak osztozhatnak ugyanazon a blokkon.
- *Lineáris tördelés:* Ez a módszer megnöveli eggyel a kosarak számát, valahányszor a rekordok kosarakhoz viszonyított aránya meghalad egy bizonyos küszöbértéket. Mivel egyetlen kosár megtelése nem okozhatja a tábla növekedését, ezért néha a kosarakban szükség van túlcsordulásblokkokra.

4.6. Irodalomjegyzék

A B-fa Bayer és McCreight [2] eredeti ötlete volt. A most bemutatott B+-fákkal szemben, ezek belső csúcsai éppúgy mutathattak rekordokra, mint a levelekre. A [3] a különböző B-fák áttekintését tartalmazza.

A tördelés mint adatszerkezet Peterson [8] munkájáig nyúlik vissza. A kiterjeszthető tördelés kidolgozása a [4]-ben szerepel, míg a lineáris tördelés a [7]-ből való. A Knuth-könyv [6] sok információt tartalmaz az adatszerkezetekről, a tördelőfüggvények megválasztásától és a tördelőtáblázatok tervezésétől kezdve egészen a különböző B-fákkal foglalkozó ötletekig. A B+-fa változat (kulcsértékek nélküli belső csúcsok) a [6] 1973-as kiadásában jelent meg.

A másodlagos indexeket és a dokumentumok visszakeresésének egyéb technikáit a [9] mutatja be. Az [5] és az [1] szintén a szöveges dokumentumok indexelési módszereinek áttekintését tartalmazzák.

1. R. Baeza-Yates, „Integrating contents and structure in text retrieval," *SIGMOD Record* **25**:1 (1996), pp. 67–79.
2. R. Bayer and E. M. McCreight, „Organization and maintenance of large ordered indexes," *Acta Informatica* **1**:3 (1972), pp. 173–189.
3. D. Comer, „The ubiquitous B-tree," *Computing Surveys* **11**:2 (1979), pp. 121–137.
4. R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, „Extendible hashing – a fast access method for dynamic files," *ACM Trans. on Database Systems* **4**:3 (1979), pp. 315–344.
5. C. Faloutsos, „Access methods for text," *Computing Surveys* **17**:1 (1985), pp. 49–74.
6. D. E. Knuth, *The Art of Computer Programming, Vol. III, Sorting and Searching, Third Edition,* Addison-Wesley, Reading MA, 1998.
7. W. Litwin, „Linear hashing: a new tool for file and table addressing," *Proc. Intl. Conf. on Very Large Databases* (1980) pp. 212–223.
8. W. W. Peterson, „Addressing for random access storage," *IBM J. Research and Development* **1**:2 (1957), pp. 130–146.
9. G. Salton, *Introduction to Modern Information Retrieval,* McGraw-Hill, New York, 1983.

5. fejezet

Többdimenziós indexek

Az eddig tárgyalt indexstruktúrák *egydimenziósak* voltak; azaz mindegyik egyetlen keresési kulcsot tartalmazott, és olyan rekordokat adott vissza, amelyek megfeleltek az adott keresésikulcs-értéknek. Azt gondolhatjuk, hogy a keresési kulcs mindig egyetlen attribútum vagy mező. Azonban olyan index is lehet egydimenziós, amelynek keresési kulcsa több mező egyesítése. Ha egy egydimenziós indexet akarunk, amelynek a keresési kulcsa az $(F_1, F_2, ..., F_k)$ mezőkből áll, akkor képezhetjük a keresésikulcs-értéket az egyes mezőértékek egymás után helyezésével, elsőnek az F_1 értékét, másodiknak az F_2 értékét és így tovább. Ezeket az értékeket valami speciális jellel elválaszthatjuk, hogy a képzett érték és az $F_1, F_2, ... F_k$ értékek listája közötti megfeleltetést egyértelművé tegyük.

5.1. példa: Ha az F_1 és F_2 mező értéke szöveg, illetőleg egész szám, és a # karakter nem fordulhat elő a szöveg mezőben, akkor az F_1 = `'abcd'` és az F_2 = 123 értékek egyesítését az `'abcd#123'` karaktersorozattal ábrázolhatjuk. □

A 4. fejezetben az egydimenziós kulcstér előnyeit többféle módon is kihasználtuk:

- A szekvenciális állományok és a B-fák indexeinél feltételeztük, hogy a kulcsok rendezett sorrendben vannak.
- A tördelőtáblák használata megköveteli, hogy a keresési kulcs teljes egészében ismert legyen bármely kereséskor. Ha egy kulcs több mezőből áll, és azokból akár csak egy is ismeretlen, már nem tudjuk a tördelőfüggvényt alkalmazni, hanem helyette végig kell keresni az összes kosarat.

Sok alkalmazás azt igényli tőlünk, hogy az adatokat kétdimenziós vagy néha magasabb dimenziós térben ábrázolhatónak tekintsük. Némelyeket ezek közül ki tud szolgálni egy szokásos adatbázis-kezelő rendszer, de vannak speciális rendszerek, amelyeket eleve többdimenziós alkalmazásokhoz terveztek. Egy fontos dolog, ami megkülönbözteti ezeket a szakosított rendszereket a többitől az, hogy olyan adatstruktúrákat használnak, amelyek támogatnak bizonyosfajta lekérdezéseket, amelyek nem általánosak az SQL-alkalmazásokban. Az 5.1. rész bemutat tipikus lekérdezéseket, amelyek olyan indexeket használnak, amiket arra terveztek, hogy többdimenziós

adatokat és többdimenziós lekérdezéseket támogassanak. Azután az 5.2. és az 5.3. részben a következő adatstruktúrákat tárgyaljuk:

1. *Rácsos állományok*, amelyek az egydimenziós tördelőtáblák egyfajta többdimenziós kiterjesztései.
2. *Particionált tördelőfüggvények*, ez egy másik módszer, amellyel többdimenziós adatokra alkalmazzuk a tördelőtáblák ötletét.
3. *Többkulcsos indexek*, ahol egy A attribútum indexe elvezet egy másik B attribútum indexeihez, az A minden lehetséges értékére.
4. *kd-fák*, amelyek a B-fák általánosításai ponthalmazokra.
5. *Quad-fák*, olyan fák, amelyben egy csomópont minden gyereke egy többdimenziós kockának felel meg.
6. *R-fák*, a B-fák olyan általánosítása, amely alkalmas területgyűjtemények kezelésére.

Végül, az 5.4. részben a *bittérképindexnek* nevezett struktúrát tárgyaljuk. Ezekben az indexek tömör kódok, amelyek adott mezőben adott értéket tartalmazó rekordok elérésére használhatók. Ezek a megoldások napjainkban kezdenek megjelenni a nagy kereskedelmi adatbázis-kezelő rendszerekben, és időnként kiváló választásnak bizonyulnak egydimenziós indexek kezelésére. Azonban bizonyosfajta többdimenziós lekérdezések megválaszolására is hatékony eszközök lehetnek.

5.1. Többdimenziós alkalmazások

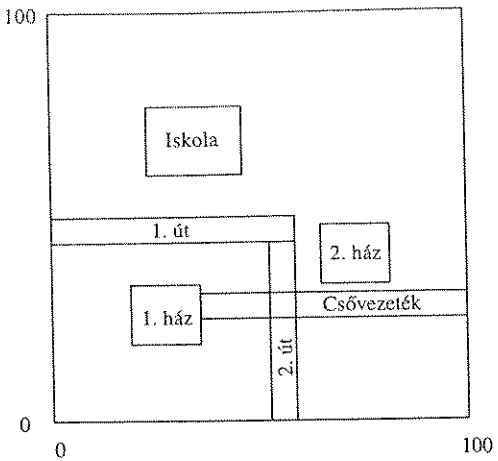
A többdimenziós alkalmazások két általános osztályát fogjuk áttekinteni. Az egyik *földrajzi* természetű, ahol az adatok a két- vagy néha a háromdimenziós világ elemei. A másik a dimenziók egy elvontabb fogalmát foglalja magában. Kissé pongyolán fogalmazva, tekinthetjük egy reláció minden attribútumát egy-egy dimenziónak, és minden sorát egy pontnak abban a térben, amelyet ezek a dimenziók meghatároznak.

Ebben a részben elemezzük azt is, hogyan használhatók a hagyományos indexek, például a B-fák, a többdimenziós lekérdezések támogatására. Bár bizonyos esetekben megfelelőek, arra is vannak példák, ahol a speciális struktúrák messze felülmúlják őket.

5.1.1. Térinformatikai rendszerek

Egy *térinformatikai rendszer* az objektumokat (tipikusan) egy kétdimenziós térben tárolja. Az objektumok lehetnek pontok vagy alakzatok. Ezek az adatbázisok gyakran térképek, ahol a tárolt objektumok házakat, utakat, hidakat, csővezetékeket és sok más fizikai tárgyat ábrázolhatnak. Egy ilyen térképre látunk mintát az 5.1. ábrán.

Azonban számos másfajta felhasználás is lehetséges. Például egy integrált áramkör terve is területek kétdimenziós térképe, ezek gyakran meghatározott anyagból készült téglalapok, úgynevezett rétegek. Hasonlóan, az ablakokat és ikonokat egy képernyőn szintén tekinthetjük kétdimenziós objektumok gyűjteményének.



5.1. ábra. *Néhány objektum egy kétdimenziós térben*

A térinformatikai rendszerek lekérdezései nem tipikus SQL-lekérdezések, de némi erőfeszítéssel sok kifejezhető SQL-ben. Ilyen típusú lekérdezésekre példák az alábbiak:

1. *Lekérdezések részleges egyezéssel.* Értékeket adunk meg egy vagy több dimenzióra, és az olyan pontokat keressük, amelyek megfelelnek a megadott értékeknek, a megadott dimenziókban.
2. *Tartománylekérdezések.* Tartományokat adunk meg egy vagy több dimenzióra, és az olyan pontok halmazát kérjük, amelyek ezeken a tartományokon belül vannak, vagy ha alakzatokat ábrázolunk, az olyan alakzatok halmazát, amelyek részben vagy teljesen a tartományon belül vannak. Ezek a lekérdezések a 4.3.4. részben tárgyalt egydimenziós tartománylekérdezések általánosításai.
3. *Legközelebbi szomszéd-lekérdezések.* Egy adott ponthoz legközelebbi pontot keressük. Például, ha egy pont egy várost jelent, és mi egy adott kisvároshoz legközelebbi 100 000-nél nagyobb lélekszámú várost akarjuk megtalálni.
4. *Hol-vagyok-én-lekérdezések.* Egy adott pontban vagyunk, és tudni akarjuk, hogy melyik alakzatban van benne ez a pont, ha van egyáltalán olyan alakzat, amiben benne van. Egy ismert példa erre, ami akkor történik, amikor az egérrel kattintunk, és a rendszer meghatározza, hogy mely látható elemre kattintottunk.

5.1.2. Adatkockák

Az utóbbi időkben megjelent az adatbázis-kezelő rendszerek azon családja, amit időnként *adatkocka*rendszereknek neveznek, amelyek az adatot többdimenziós térben létezőnek tekintik. Ezeket a 11.4. részben tárgyaljuk majd részletesebben, de a következő példa rávilágít az alapötletükre.

Többdimenziós adatokat sok vállalat gyűjt a *döntéstámogató* alkalmazások számára, amelyekkel elemzik az információkat, olyanokat mint az eladások, hogy jobban megértsék a vállalat működését. Például egy áruházlánc feljegyezheti minden eladásáról az alábbiakat:

1. a dátumot és időt,
2. az áruházat, amelyben az eladás történt,
3. a vásárolt árucikket,
4. az árucikk színét,
5. az árucikk méretét,

és esetleg az eladás egyéb tulajdonságait.

Szokásos az adatokat egy relációnak tekinteni, ahol minden tulajdonság a reláció egy attribútuma. Ezeket az attribútumokat egy többdimenziós tér, az „adatkocka" dimenzióinak tekinthetjük. Minden sor egy pont ebben a térben. Az elemzők aztán olyan kérdéseket tesznek fel, amelyek rendszerint csoportosítják az adatokat néhány dimenzió mentén, és a csoportokat összegzik valamilyen összesítő függvénnyel (aggregátorral). Egy jellemző példa lehet: „add meg a rózsaszínű ingek 1998-as eladásait áruházanként, havi bontásban".

5.1.3. Többdimenziós lekérdezések SQL-ben

Lehetséges a fent említett alkalmazások mindegyikét mint hagyományos, relációs adatbázist megvalósítani, és a felmerült lekérdezéseket SQL-ben megfogalmazni. Nézzünk néhány példát.

5.2. példa: Tegyük fel, hogy a legközelebbi szomszéd-lekérdezést akarjuk megválaszolni egy kétdimenziós térben lévő ponthalmazon. A pontokat ábrázolhatjuk valós számpárokból álló relációként.

```
Pontok(x, y)
```

Ennek két attribútuma van, x és y, amelyek a pont x és y koordinátái. A `Pontok` reláció további – itt nem mutatott – attribútumai esetleg a pontok egyéb jellemzőit ábrázolják.

Tegyük fel, hogy a (10.0, 20.0) ponthoz legközelebbi pontot keressük. Az 5.2. ábrán szereplő lekérdezés megtalálja a legközelebbi pontot, illetve pontokat, ha több ilyen is van. Minden egyes p pontra megnézi, létezik-e olyan másik q pont, amelyik közelebb van a (10.0, 20.0) ponthoz. A távolságok összehasonlítása úgy történik, hogy a (10.0, 20.0) pont és a lekérdezésben szereplő pont x, illetve y koordinátáinak különbségét négyzetre emeli és összeadja. Megjegyezzük, hogy nem kell gyököt vonni az összegből, hogy megkapjuk a tényleges távolságot, mert a távolság négyzetének összehasonlítása ugyanazt az eredményt adja, mintha a távolságértékeket magukat hasonlítanánk össze. □

```
SELECT *
FROM PONTOK p
WHERE NOT EXISTS(
    SELECT *
    FROM PONTOK q
    WHERE (q.x-10.0)*(q.x-10.0)+(q.y-20.0)*(q.y-20.0)<
          (p.x-10.0)*(p.x-10.0)+(p.y-20.0)*(p.y-20.0)
);
```

5.2. ábra. *Azon pontok keresése, amelyeknél nincs közelebbi a (10.0, 20.0) ponthoz*

5.3. példa: A téglalapok szokásos alakzatok a térinformatikai rendszerekben. Egy téglalapot többféleképpen ábrázolhatunk. Egyik népszerű forma a bal alsó sarok és a jobb felső sarok koordinátáinak megadása. Ezután a `Téglalapok` relációval ábrázolhatjuk a téglalapok gyűjteményét, ennek attribútumai a téglalap azonosítója, a négy koordinátája, ami leírja a téglalapot, és esetleg bármi egyéb jellemzője a téglalapnak, amit rögzíteni szeretnénk. A következő relációt fogjuk használni ebben a példában:

```
Téglalapok(az, xba, yba, xjf, yjf)
```

Az attribútumok sorrendben a téglalap azonosítója, a bal alsó sarok x koordinátája, a bal alsó sarok y koordinátája, illetve a jobb felső sarok két koordinátája.

Az 5.3. ábrán szereplő lekérdezés azokat a téglalapokat keresi, amelyek tartalmazzák a (10.0, 20.0) pontot. A `WHERE` feltétel egyszerű. A (10.0, 20.0) pontot akkor tartalmazza a téglalap, ha a bal alsó sarkának x koordinátája 10.0 vagy attól balra van, és az y koordinátája 20.0 vagy az alatti. A jobb felső sarokra egyúttal az x = 10.0 vagy attól jobbra, és y = 20.0 vagy a feletti érték kell legyen. □

```
SELECT az
FROM Téglalapok
WHERE xba   <= 10.0  AND   yba <=      20.0    AND
      xjf   >= 10.0  AND   yjf >=      20.0;
```

5.3. ábra. *Téglalap(ok) keresése, amely(ek) tartalmaz(nak) egy adott pontot*

5.4. példa: Az adatkockarendszereknek megfelelő adatok általában *ténytáblába* – ezekben az alapadatok vannak rögzítve (pl. minden eladás) – és *dimenziótáblákba* – ezek tartalmazzák az egyes értékekhez tartozó jellemzőket az egyes dimenziókban – vannak szervezve. Például, ha az áruház, amely eladott valamit, az egy dimenzió, akkor az áruházhoz tartozó dimenziótábla megadhatja az áruház vezetőjének a címét, a telefonszámát és a nevét.

Ebben a példában csak a ténytáblával foglalkozunk, amelynek feltevésünk szerint az 5.1.2. részben javasolt dimenziói vannak. Tehát a ténytábla a következő reláció:

```
Eladások(nap, áruház, cikk, szín, méret)
```


Az „összegezd a rózsaszínű ingek eladását áruházanként, napi bontásban" lekérdezés az 5.4. ábrán látható. A lekérdezés csoportosítja az eladásokat a `nap` és az `áruház` dimenzió értékei szerint, miközben összefogja a többi dimenziót a `COUNT` összesítő függvénnyel. Az adatkocka csak azon részeire figyelünk, ami bennünket érdekel, így a `WHERE` feltétel csak a rózsaszín ingek sorait választja ki. □

```
SELECT nap, áruház, COUNT(*) AS összEladás
FROM Eladások
WHERE cikk  =  'ing' AND
      szín  =  'rózsaszín'
GROUP BY nap, áruház;
```

5.4. ábra. *A rózsaszínű ingek eladásának összegzése*

5.1.4. Tartománylekérdezések végrehajtása hagyományos indexekkel

Most tekintsük át, hogy a 4. fejezetben leírt indexek milyen mértékben segíthetik a tartománylekérdezések megválaszolását. Az egyszerűség kedvéért tegyük fel, hogy két dimenziónk van. Az x és az y dimenziók mindegyikére tehetünk egy másodlagos indexet. Mindkettőhöz B+-fát használva különösen könnyen lehet értékek egy tartományát visszanyerni bármelyik dimenzió esetén.

Ha mindkét dimenzióra adott egy tartomány, akkor kezdhetjük az x-hez tartozó B-fával, hogy visszanyerjük az összes olyan rekord mutatóját, amely az x-hez megadott tartományba esik. Azután az y B-fáját használva megkapjuk az összes olyan ponthoz tartozó rekord mutatóját, amelynek az y koordinátája az y-hoz megadott tartományba esik. Végül vesszük a mutatók metszetét, a 4.2.3. rész ötletét használva. Ha a mutatók elférnek a központi memóriában, akkor a szükséges lemez I/O-műveletek száma megegyezik a két B-fában a megvizsgálandó levélcsomópontok számával, plusz még néhány lemez I/O-művelet, amíg megtaláljuk a B-fákban lefelé az utat (lásd a 4.3.7. részt). Ehhez kell még hozzáadni a megfelelő rekordok eléréséhez szükséges lemez I/O-műveletek számát.

5.5. példa: Tekintsük 1 000 000 pontnak egy feltételezett halmazát, amely pontok véletlenszerűen oszlanak el a kétdimenziós térben, és az x, illetve y koordinátáik egyaránt 0 és 1000 közé esnek. Tegyük fel, hogy 100 pont rekordja fér el egy blokkban, és egy átlagos B-fa-levél körülbelül 200 kulcs-mutató párt tartalmaz (emlékeztetünk rá, hogy egy B-fa-blokk nem feltétlen minden bejegyzése foglalt minden időpontban). Feltételezzük továbbá, hogy van B-fa-index x-hez és y-hoz is.

Képzeljünk el egy olyan tartománylekérdezést, amely egy a tér közepén levő 100 egység oldalú négyzetbe eső pontok számát kérdezi le: $450 \leq x \leq 550$ és $450 \leq y \leq 550$. Az x-hez tartozó B-fát használva megkaphatjuk az összes rekord mutatóját, amely az x szerinti tartományba esik, ez körülbelül 100 000 mutató lehet, és ez elférhet a központi memóriában. Hasonlóan, az y-hoz tartozó B-fát használva megkaphatjuk az összes olyan rekord mutatóját is, amely y szerint a kívánt tartományba esik, ezekből ismét körülbelül 100 000 lesz. E két halmaz metszete körülbelül 10 000 mutató, és ezzel

a metszetben szereplő 10 000 mutatóval elérhető rekordok alkotják a választ a lekérdezésünkre.

Most becsüljük meg a tartománylekérdezés megválaszolásához szükséges lemez I/O-műveletek számát. Először is, ahogy a 4.3.7. részben rámutattunk, általában megvalósítható, hogy mindegyik B-fa gyökerét a központi memóriában tartsuk. Mivel keresésikulcs-értékek egy tartományát keressük a B-fákban, és a mutatók a levelekben e szerint a keresési kulcs szerint rendezettek, így mindössze annyit kell tennünk ahhoz, hogy dimenziónként hozzáférjünk a kb. 100 000 mutatóhoz, hogy átvizsgálunk egy köztes szintű csomópontot, valamint az összes levelet, amely a kívánt mutatókat tartalmazza. Mivel feltételeztük, hogy egy levél körülbelül 200 kulcs-mutató párt tartalmaz, így mindkét B-fa esetén körülbelül 500 levélblokkot kell megnéznünk. Ha ehhez hozzáadjuk B-fánként az egy köztes szintű csomópontot, akkor összesen 1002 lemez I/O-műveletet kapunk.

Végül vissza kell nyerni a blokkokat, amelyek a 10 000 kívánt rekordot tartalmazzák. Ha ezek véletlenszerűen vannak tárolva, azt várhatjuk, hogy ezek közel 10 000 különböző blokkban helyezkednek el. Mivel az egymilliós teljes állomány – a feltevés szerint 100 rekord tölt meg egy blokkot – 10 000 blokkban tárolódik, ezért nagyjából az adatállomány minden blokkját végig kell néznünk. Így, legalábbis ebben a példában, a hagyományos indexek nem, vagy csak alig segítették a tartománylekérdezés megválaszolását. Természetesen, ha a tartomány kisebb lett volna, a mutatóhalmazok metszetének létrehozása lehetővé tehette volna számunkra, hogy a keresés az adatállomány blokkjainak töredékére korlátozódjon. □

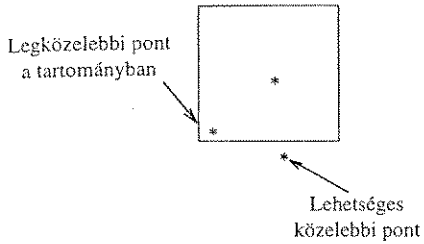
5.1.5. Legközelebbi szomszéd-lekérdezések végrehajtása hagyományos indexekkel

Majdnem minden általunk használt adatstruktúra lehetővé teszi a legközelebbi szomszéd-lekérdezések megválaszolását azáltal, hogy kiválasztunk egy tartományt minden dimenzióhoz, megválaszoljuk a tartománylekérdezést, majd kiválasztjuk a célhoz legközelebbi pontot a tartományon belül. Két probléma merülhet fel:

1. Nincs pont a kiválasztott tartományban.
2. A tartományon belüli legközelebbi pont lehet, hogy nem a legközelebbi.

Vizsgáljuk meg mindkét esetet az 5.2. példa legközelebbi szomszéd-lekérdezésével kapcsolatban, használva az 5.5. példában bemutatott x és y dimenzióra feltételezett indexeket. Ha jó okunk van feltételezni, hogy a (10.0, 20.0)-hoz d távolságon belül létezik pont, akkor használhatjuk az x-hez tartozó B-fát, hogy visszanyerjük azon pontok mutatóit, amelyek x koordinátája $10 - d$ és $10 + d$ között van. Azután használhatjuk az y-hoz tartozó B-fát, hogy visszanyerjük azon pontok mutatóit, amelyek y koordinátája $20 - d$ és $20 + d$ között van.

Ha van egy vagy több pont a metszetben, és minden mutatóhoz feljegyeztük az x, illetve az y koordinátát is (ami a keresési kulcs volt az indexhez), akkor a metszetben



5.5. ábra. *Legközelebbi pont a tartományban, ám lehet közelebbi pont a tartományon kívül*

rendelkezésünkre áll minden egyes pont koordinátája. Ezért meghatározhatjuk, hogy melyik a (10.0, 20.0)-hoz legközelebbi pont, és visszanyerhetjük annak a rekordját. Sajnos nem lehetünk biztosak benne, hogy van pont az adott ponthoz d távolságon belül, ezért lehet, hogy meg kell ismételni az eljárást egy nagyobb d értékkel.

Azonban, ha van is pont az átnézett tartományban, bizonyos esetekben az adott ponthoz – mely példánkban a (10.0, 20.0) – legközelebbi pont a tartományon belül, lehet, hogy a d távolságnál messzebb van. Egy ilyen helyzetet mutat az 5.5. ábra. Ha ez a helyzet, akkor meg kell növelnünk a tartományt, és újra kell keresnünk, hogy meggyőződjünk arról, hogy nincs közelebbi pont. Ha az eddig talált legközelebbi pont és az adott pont távolsága d', és $d' > d$, akkor meg kell ismételnünk a keresést d'-t használva d helyett.

5.6. példa: Tekintsük az 5.5. példában szereplő adatokat és indexeket. Ha a legközelebbi szomszédját keressük az adott $P = (10.0, 20.0)$ pontnak, választhatjuk a $d = 1$ értéket. Átlagosan egy pont jut a terület egy egységére, és $d = 1$ esetén megtalálunk minden olyan pontot, amely a P pont körüli 2.0 oldalú négyzetbe esik, ezeknek a várható száma 4.

Ha megvizsgáljuk az x koordinátához tartozó B-fát a $9.0 \leq x \leq 11.0$ tartománylekérdezés esetén, akkor a lekérdezés körülbelül 2000 pontot fog találni, így legalább 10 levelet be kell járnunk, de valószínűbb, hogy 11-et (mivel valószínűtlen, hogy az $x = 9.0$ értékű pontok éppen egy levél elején kezdődnek). Mint az 5.5 példában, várhatóan a B-fák gyökereit a központi memóriában tudjuk tartani, így csak egy lemez I/O-műveletre van szükségünk egy köztes szintű csomópont eléréséhez, és 11 lemez I/O-műveletre a levelekhez. További 12 lemez I/O-művelet kell, hogy az y koordináta B-fájában megkeressük azokat a pontokat, amelyeknek az y koordinátája 19.0 és 21.0 között van.

Ha a megközelítőleg 4000 mutató metszetét képezzük a központi memóriában, várhatóan négy rekordunk lesz, amelyek közül kikerülhet a (10.0, 20.0) pont legközelebbi szomszédja. Feltéve, hogy van legalább egy ilyen rekord, a mutatókhoz tartozó x és y koordinátákból meghatározhatjuk melyik a legközelebbi szomszéd. Még kell egy lemez I/O-művelet, hogy visszanyerjük a kívánt rekordot, így összesen 25 lemez I/O-művelet kellett, hogy befejezzük a lekérdezést. Azonban, ha $d = 1$ értékhez tartozó négyzetben nincs pont, vagy a legközelebbi pont távolsága az adott ponttól nagyobb, mint 1, meg kell ismételnünk a lekérdezést egy nagyobb d értékkel. □

Azt a következtetést vonhatjuk le az 5.6. példából, hogy a hagyományos indexek nem feltétlenül rosszak a legközelebbi szomszéd-lekérdezésekhez, de lényegesen több lemez I/O-műveletet igényelnek, mint amennyit használnánk, ha mondjuk egy rekordot adott kulcs és a kulcshoz tartozó B-fa alapján (amely valószínűleg csak két vagy három lemez I/O-művelettel járna) keresnénk meg. Az ebben a fejezetben ajánlott módszerek általában jobb teljesítményt nyújtanak, és ezeket használják a többdimenziós adatokat támogató szakosított adatbázis-kezelő rendszerek is.

5.1.6. A hagyományos indexek további korlátjai

A fent említett struktúrák költsége nem jobb tartománylekérdezésekre sem, mint a legközelebbi szomszéd-lekérdezések esetén. Gyakorlatilag az 5.6. példában úgy közelítettünk a legközelebbi szomszéd-lekérdezés megoldásához, hogy tulajdonképpen – egy kisméretű tartománnyal minden dimenzióra – tartománylekérdezéssé alakítottuk át azt, és reméltük, hogy a tartomány mérete elégséges ahhoz, hogy legalább egy pont beleessen. Következésképpen, ha egy tartománylekérdezéshez nagyobb tartományokat választanánk, és az adatstruktúrák indexek lennének minden dimenzióra, akkor a megfelelő rekordok mutatóinak eléréséhez szükséges lemez I/O-műveletek száma minden dimenzióban több lenne, mint amennyit az 5.6. példában találtunk.

Az 5.4. ábrán látható lekérdezés többdimenziós összegzése szintén nem szerencsés. Ha van indexünk a cikk és a szín attribútumra, akkor megtalálhatjuk a rózsaszínű ingek eladásához tartozó összes rekordot, a metszetüket véve, ahogyan az 5.6. példában tettük. Azonban az olyan lekérdezések esetében, amelyeknél a szín és a cikk attribútumok mellett más attribútumok is adottak, inkább az azokra az attribútumokra megadott indexekre lenne igény.

Sőt amíg az adatállományt rendezetten tudjuk tartani az öt attribútum valamelyike szerint, már nem tudjuk két attribútum szerint sorrendben tartani, nem is szólva az ötről. Így az 5.4. példában mutatott alakú lekérdezések többségénél szükség lenne az adatállomány minden vagy majdnem minden blokkjának az elérésére. Az ilyen típusú lekérdezések végrehajtása rendkívül költséges lenne, különösen ha az adatok háttértárolón vannak.

5.1.7. A többdimenziós indexstruktúrák áttekintése

A többdimenziós adatok lekérdezését támogató legtöbb adatstruktúra a következő két kategória egyikébe tartozik:

1. tördelőtábla alapú,
2. fastruktúrájú.

Mindkét fenti struktúránál feladunk valamit abból, amivel rendelkeztünk a 4. fejezet egydimenziós struktúráinál.

- A tördelőtábla alapú elgondolásoknál – rácsos állományok és particionált tördelőfüggvények az 5.2. részben – a továbbiakban nem lesz meg az az előnyünk, hogy a lekérdezésünkre adott válasz pontosan egy kosárban van. Azonban minden ilyen elgondolásnál a keresés a kosarak egy részhalmazára korlátozódik.
- A fastruktúrájú elgondolásoknál feladunk legalább egyet a következő B-fa-tulajdonságok közül:

1. A fa kiegyensúlyozottságát, ahol az összes levél ugyanazon a szinten van.
2. A facsomópontok és a lemezblokkok közötti megfeleltetést.
3. A sebességet, amellyel az adatok módosítását végre lehet hajtani.

Amint látni fogjuk az 5.3. részben, a fák gyakran mélyebbek lesznek bizonyos részeken, mint máshol, és gyakran a mélyebb részek sok pontot tartalmazó területeknek felelnek meg. Szintén látni fogjuk, hogy gyakran egy fa csomópontjához tartozó információ lényegesen kisebb méretű annál, mint ami elfér egy blokkban. Így valamilyen hasznos módon blokkokba célszerű csoportosítani a csomópontokat.

5.1.8. Feladatok

5.1.1. feladat: Írjunk SQL-lekérdezéseket az 5.3. példából vett

```
Téglalapok(az, xba, yba, xjf, yjf)
```

relációt használva, amelyek megválaszolják a következő kérdéseket:

* a) Keressük meg azon téglalapok halmazát, amelyeknek van közös része azzal a téglalappal, amelynek bal alsó sarka a (10.0, 20.0), a jobb felső sarka pedig a (40.0, 30.0) pont.
b) Keressük meg azon téglalappárokat, melyek átfedik egymást.
c) Keressük meg azokat a téglalapokat, amelyek teljesen tartalmazzák az a)-ban említett téglalapot.
d) Keressük meg azokat a téglalapokat, melyek teljesen benne vannak az a)-ban említett téglalapban.
! e) Keressük meg azokat a „téglalapokat" a `Téglalapok` relációban, amelyek valójában nem téglalapok, azaz fizikailag nem létezhetnek.

Mindegyik lekérdezésnél mondjuk meg, mely indexek – ha vannak ilyenek – segíthetnek a kívánt sorok elérésében.

5.1.2. feladat: Az 5.4. példából vett

```
Eladások(nap, áruház, cikk, szín, méret)
```

relációt használva adjuk meg a következő lekérdezéseket SQL-ben:

* a) Listázzuk ki az ingek színeit, és az eladások összesített számát az olyan színekre, amelyekből 1000-nél többet adtak el.
b) Listázzuk ki az ingek eladásait áruházanként és színenként.
c) Listázzuk ki az összes árucikk eladásait áruházanként és színenként.
! d) Listázzuk ki minden árucikkre és színre azt az áruházat, amely a legtöbbet adta el, és listázzuk ki ezeknek az eladásoknak a számát is.

5.1.3. feladat: Oldjuk meg újra az 5.5. példát azzal a feltételezéssel, hogy a tartománylekérdezés egy középen lévő n egység oldalú négyzetre vonatkozik, ahol n egy tetszőleges 1 és 1000 közötti érték. Hány lemez I/O-művelet szükséges? Milyen n értékeknél segítenek az indexek ténylegesen?

* **5.1.4. feladat:** Ismételjük meg az 5.1.3. feladatot úgy, hogy a rekordok adatállománya rendezett x-re.

!! **5.1.5. feladat:** Tegyük fel, hogy egy négyzetben véletlenszerűen elosztott pontjaink vannak (ahogy az 5.6. példában), és egy legközelebbi szomszéd-lekérdezést akarunk végrehajtani. Választunk egy d távolságot, és megkeressük az összes olyan pontot, ami abba a $2d$ oldalú négyzetbe esik, amelynek a középpontja az adott pont. A keresésünk akkor sikeres, ha találunk a négyzetben legalább egy pontot, amelynek a távolsága az adott ponttól d vagy annál kisebb.

* a) Ha egységnyi területen átlagosan egy pont van, adjuk meg annak a valószínűségét d függvényében, hogy sikeresek leszünk.
b) Ha sikertelenek vagyunk, meg kell ismételnünk a keresést egy nagyobb d-vel. Tegyük fel az egyszerűség kedvéért, hogy minden esetben, amikor sikertelenek vagyunk, akkor megduplázzuk a d-t, és kétszer annyi a költségünk, mint amennyi az előző kereséskor volt. Ismét csak feltesszük, hogy egységnyi területen átlagosan egy pont van. Melyik az a d kezdőérték, ami a minimális várható keresési költséget adja?

5.2. Tördelésen alapuló struktúrák többdimenziós adatokhoz

Ebben a részben áttekintünk két olyan adatstruktúrát, amely általánosítja az egyetlen kulcs használatára épülő tördelőtáblákat. Mindkét esetben a ponthoz rendelt kosár az összes attribútum, illetve dimenzió függvénye. Az egyik szerkezet az ún. „rácsos állomány", ez általában nem tördeli az értékeket a dimenzió mentén, hanem inkább felosztja a dimenziót, rendezve az értékeket a dimenzió mentén. A másik az ún. particionált tördelés, ami tényleg tördeli a különböző dimenziókat, és minden egyes dimenzió részt vesz a kosársorszám kialakításában.

5.2.1. Rácsos állományok

Az egyik legegyszerűbb adatstruktúra, amely gyakran felülmúlja teljesítményben az egydimenziós indexeket a többdimenziós adatokat magában foglaló lekérdezéseknél, a *rácsos állomány* (grid file). Gondoljunk egy pontokból álló térre, amelyet rácsok osztanak fel. Minden egyes dimenzióban *rácsvonalak* (grid lines) osztják fel a teret *sávokra* (stripes). Azokat a pontokat, amelyek egy rácsvonalra esnek, ahhoz a sávhoz tartozónak tekintjük, amelynek a rácsvonal az alsó határa. A különböző dimenziókhoz különböző számú rácsvonal tartozhat, és a szomszédos rácsvonalak között különböző lehet a távolság, még azonos dimenzión belül is.

5.7. példa: Bevezetjük e fejezet állandó példáját: a lekérdezésünk legyen „ki vásárol arany ékszert?". Képzeljük el az arany ékszerek vásárlóinak adatbázisát, amely számos dolgot mond nekünk minden egyes vásárlóról – a nevét, címét stb. Azonban, hogy a dolgokat egyszerűbbé tegyük, feltételezzük, hogy csak a vásárló életkora és fizetése a lényeges attribútum. A példa adatbázisunkban 12 vásárló van, akiket a következő életkor-fizetés párokkal ábrázolhatunk.

(25, 60)	(45, 60)	(50, 75)	(50, 100)
(50, 120)	(70, 110)	(85, 140)	(30, 260)
(25, 400)	(45, 350)	(50, 275)	(60, 260)

Az 5.6. ábrán látható a 12 pont elhelyezkedése egy kétdimenziós térben. Néhány rácsvonalat is választottunk mindegyik dimenzióban. Ehhez az egyszerű példához két rácsvonalat választottunk mindegyik dimenzióban, ezzel a teret 9 téglalap alakú területre osztottuk, de persze semmi nem indokolja, hogy ugyanannyi vonalat használjunk minden dimenzióban. Azt szintén megengedtük, hogy a vonalak között különböző távolságok legyenek. Például az életkor dimenziónál, a három terület – amire a két függőleges vonal osztja a teret – szélessége 40, 15 és 45.

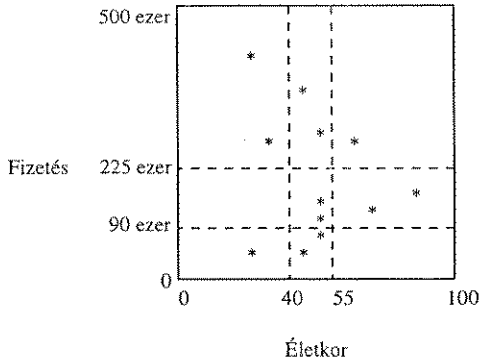
Ebben a példában nem esik pont a rácsvonalakra. De általánosságban a pont a téglalaphoz tartozik, ha az alsó vagy a bal oldali élére esik, és nem tartozik hozzá, ha a felső vagy a jobb oldali élén van. Például az 5.6. ábrán lévő középső téglalap azokat a pontokat tartalmazza, amelyekre $40 \leq \textit{életkor} < 55$ és $90 \leq \textit{fizetés} < 225$. □

5.2.2. Keresés rácsos állományban

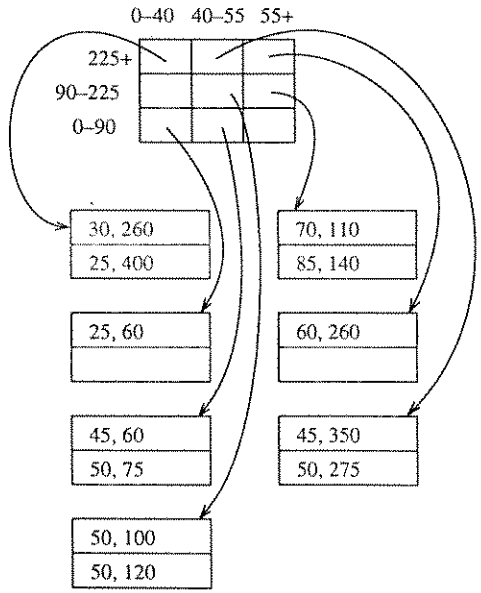
Mindegyik terület – amire a teret felosztottuk – úgy tekinthető, mint egy kosár egy tördelőtáblában, és minden ezen a területen lévő pontnak megvan a rekordja az ahhoz a kosárhoz tartozó valamely blokkban. Túlcsordulásblokkok használhatók – ha szükséges – a kosár méretének növelésére.

A kosarak egydimenziós tömbje helyett – ahogyan azt a hagyományos tördelőtáblánál láttuk – a rácsos állomány olyan tömböt használ, amelynek a dimenziószáma megegyezik az adatállományéval. Ahhoz, hogy egy ponthoz a megfelelő kosarat meg-

határozzuk, tudnunk kell minden dimenzióra az értékek listáját, ahol a rácsvonalak vannak. Egy pont hasítása így némileg különbözik attól, mint amikor egy tördelőfüggvényt alkalmazunk a változóinak értékeire. Vesszük a pont komponenseit, és meghatározzuk a pont helyzetét az ahhoz a dimenzióhoz tartozó rácson. A pont dimenziónként meghatározott helyzetei együtt határozzák meg magát a kosarat.



5.6. ábra. *Egy rácsos állomány*



5.7. ábra. *Egy rácsos állomány, amely az 5.6. ábra pontjait ábrázolja*

5.8. példa: Az 5.7. ábrán láthatjuk az 5.6. ábra adatainak elhelyezését a kosarakban. Mivel a rácsok a teret mindkét dimenziónál három területre osztják, a kosarak tömbje egy 3 × 3 -as mátrix. Kettő a kosarak közül üres:

1. A fizetés 90 ezer és 225 ezer dollár között, és az életkor 0 és 40 év között,

valamint

2. A fizetés 90 ezer dollár alatt, és az életkor 55 év fölött.

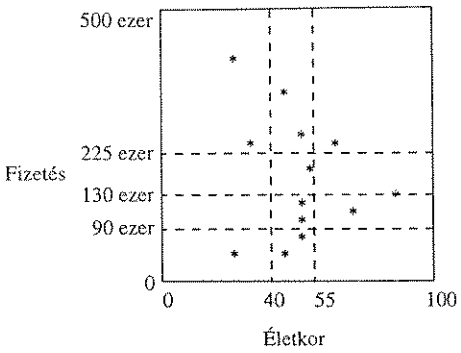
Ezért nem is jelölünk blokkot ehhez a két kosárhoz. A többi kosárhoz tartozó blokkban maximum két pont szerepelhet, ami egy mesterségesen alacsonyan tartott érték. Ebben az egyszerű példában nincs olyan kosár, amelynek kettőnél több tagja lenne, így túlcsordulásblokkokra sincs szükség. □

5.2.3. Beszúrás rácsos állományba

Amikor rekordot szúrunk be egy rácsos állományba, először a rekordkeresés eljárását követjük, és az új rekordot az így kapott kosárba tesszük. Ha van hely a kosár blokkjában, nincs is más teendőnk. Akkor van gond, ha nincs hely a kosárban. Két általános megközelítés lehetséges:

1. Adjunk túlcsordulásblokkokat a kosárhoz, ha szükséges. Ez a megoldás jól működik, egészen addig, amíg a kosár blokkjainak lánca nem válik túl naggyá. Ez utóbbi esetben a kereséshez, beszúráshoz vagy a törléshez szükséges lemez I/O-műveletek száma végül elfogadhatatlanul nagyra nőhet.
2. Szervezzük újra a struktúrát rácsvonalak hozzáadásával vagy elmozgatásával. Ez a megoldás hasonló a 4.4. részben tárgyalt dinamikus tördelési technikához, de vannak további problémái, mert a kosarak tartalma függ a dimenziótól. Azaz egy rácsvonal hozzáadása szétvágja az adott vonal mentén lévő összes kosarat. Ennek következtében elképzelhető, hogy nem lehetséges olyan rácsvonalat választani, amely mindegyik kosár számára a legjobb lenne. Például, ha egy kosár túl nagy, lehet hogy nem tudunk egy szétvágandó dimenziót vagy egy vágási pontot választani anélkül, hogy sok üres kosarat ne képeznénk vagy számos telit ne hagynánk meg.

5.9. példa: Tegyük fel, hogy valaki, aki 52 éves és 200 ezer dollár jövedelme van, arany ékszert vásárol. Ez a vásárló az 5.6. ábrán a középső téglalapba tartozik. Azonban, most már három rekord van a kosárban. Egyszerűen hozzáadhatunk egy túlcsordulásblokkot. Ha szét akarjuk vágni a kosarat, ki kell választanunk vagy az életkor, vagy a fizetés dimenziót, és választanunk kell egy új rácsvonalat, ami létrehozza a felosztást. Csak három lehetőség van, hogy egy olyan rácsvonalat vezessünk be, amely úgy vágja ketté a középső kosarat, hogy két pont kerüljön az egyik oldalra, és egy a másikra, amely esetünkben a legegyenletesebb lehetséges szétvágás.



5.8. ábra. *Az (52, 200) pont beszúrása a kosarak szétvágásával*

1. Egy függőleges vonal, mint például az életkor = 51, ami elválasztja a két 50 évest az 52-től. Ez a vonal nem vág szét semmit az alatta és felette lévő kosarakból, mivel mindkét pontja ennek a két további – 40–55 életkorhoz tartozó – kosárnak az életkor = 51 vonal bal oldalára esik.
2. Egy vízszintes vonal, ami elválasztja a középső kosárban a fizetés = 200 pontot a másik két ponttól. Választhatunk olyan számot, mondjuk a 130-at, amely szétvágja a jobbra lévő kosarat is (azaz az életkor 55–100 és fizetés 90–225 kosarat).
3. Egy vízszintes vonal, ami elválasztja a fizetés = 100 pontot a másik két ponttól. Ismét javasolhatunk olyan számot, mondjuk a 115-öt, amely szétvágja a tőle jobbra lévő kosarat is.

Az 1. lehetőség valószínűleg nem javasolt, mert nem vág szét egyetlen további kosarat sem; több üres kosarunk lesz, és egyetlen foglalt kosárnak sem csökkenti a méretét. A 2. és a 3. lehetőség egyformán jó, de mi a 2.-at választanánk, mivel a hozzáadott vízszintes fizetés = 130 rácsvonal közelebb van a 90 és 225 alsó, illetve felső határ közepéhez, mint a 3. választásakor. A kosarak eredményül kapott felosztását az 5.8. ábra mutatja. □

5.2.4. A rácsos állományok hatékonysága

Tekintsük át mennyi lemez I/O-műveletet igényel a rácsos állomány a különféle típusú lekérdezések esetén. Eddig a rácsos állomány kétdimenziós változatára koncentráltunk, bár használható tetszőleges számú dimenzió esetén. Egyik fő probléma a sokdimenziós eseteknél, hogy a kosarak száma exponenciálisan nő a dimenziók számával. Ha a tér nagy darabjai üresek, akkor sok üres kosár lesz. Megvilágíthatjuk a problémát akár két dimenzióban is. Tegyük fel, hogy szoros az összefüggés az életkor és a fizetés között, akkor az 5.6. ábra összes pontja az átló mentén fekszik, így mindegy hová helyezzük a rácsvonalakat, az átlótól távolabb lévő kosarak üresek lesznek.

Rácsos állomány kosarainak elérése

Míg egy háromszor hármas rácsban – amilyen az 5.7. ábrán látható – könnyű megtalálni egy pont megfelelő koordinátáit, ne feledjük: egy rácsos állománynak nagyon sok sávja lehet mindegyik dimenziójában. Ha így van, akkor indexet kell létrehoznunk minden egyes dimenzióhoz. Egy ilyen index keresési kulcsa az adott dimenzió felosztási értékeinek halmaza.

Adott egy v érték valamely koordinátára, és keressük azt a legnagyobb w kulcsértéket, amely kisebb vagy egyenlő mint v. Az indexben a w-hez rendelt érték a mátrix azon sora, vagy oszlopa lesz, amelyikbe v esik. Megadva az értékeket mindegyik dimenzióhoz, megtalálhatjuk azt a helyet, ahova a mátrixban a kosár mutatója esik. Ezzel a mutatóval aztán közvetlenül elérhetjük a blokkot.

Szélsőséges esetekben a mátrix olyan nagy, hogy a kosarak nagy része üres, és nem áll módunkban tárolni az üres kosarakat. Ekkor úgy kell kezelnünk a mátrixot, mint egy relációt, amelynek az attribútumai a nem üres kosarak sarkai, és egy záró attribútum ábrázolja a mutatót a kosárra. Az ilyen relációban való keresés is többdimenziós, de a mérete kisebb, mint magának az adatállománynak.

Azonban, ha az adatok eloszlása jó, és az adatállomány maga nem túl nagy, akkor ki tudjuk választani a rácsvonalakat úgy, hogy:

1. Kellően kisszámú kosár van ahhoz, hogy a kosármátrixot a központi memóriában tartsuk, így nem kell külön lemez I/O-művelet ahhoz, hogy megnézzük a mátrixot, vagy új sorokat, illetve oszlopokat adjunk hozzá, amikor új rácsvonalat veszünk fel.
2. A rácsvonalak értékeinek indexét is a memóriában tudjuk tartani mindegyik dimenzióban (lásd a „Rácsos állomány kosarainak elérése" című bekeretezett részt), vagy el tudjuk kerülni az indexeket, és a dimenziók rácsvonalait meghatározó értékeken bináris keresést tudunk alkalmazni a központi memóriában.
3. Egy tipikus kosárnak legfeljebb néhány túlcsordulásblokkja lehet, tehát ez nem okoz túl sok további lemez I/O-műveletet, amikor végignézzük a kosarat.

Ezen előfeltevések mellett, bemutatjuk a rácsos állományok viselkedését a lekérdezések néhány fontos osztályára.

Adott pont keresése

A megfelelő kosárhoz vezet bennünket, így a szükséges lemez I/O-művelet csak ennek a kosárnak a beolvasása. Beszúrás vagy törlés esetén egy további lemezírás szükséges. Ha a beszúrás túlcsordulásblokk létrehozásához vezet, az egy további írási műveletet jelent.

Lekérdezések részleges egyezéssel

Az ilyen lekérdezésre példák: „keresd meg az összes 50 éves vásárlót", vagy „keresd meg az összes vásárlót, akinek a fizetése 200 ezer dollár". Most meg kell néznünk az összes kosarat a kosármátrix egy sorában vagy oszlopában. A lemez I/O-műveletek száma igen magas lehet, ha sok kosár van abban a sorban, illetve oszlopban.

Tartománylekérdezések

Egy tartománylekérdezés a rács egy téglalap alakú területét határozza meg, és az összes pont, amit a területen belüli kosarakban találunk a lekérdezés eredményéhez tartozik, néhány olyan pont kivételével, amelyek a kijelölt terület határára eső kosarakban találhatók. Például, ha meg akarjuk találni a 35–45 éves vásárlókat, akiknek a fizetése 50 ezer–100 ezer dollár, akkor négy kosarat kell megnéznünk az 5.6. ábra bal alsó részén. Ebben az esetben minden kosár a határokon van, így jó sok pontot meg kell néznünk, amelyek esetleg nem tartoznak a kérdésre adott válaszhoz. Azonban ha olyan területet vizsgálunk, amely sok kosarat tartalmaz, akkor ezek nagy része szükségképpen belső, így minden pontjuk a válaszhoz tartozik. Tartománylekérdezéseknél a lemez I/O-műveletek száma nagy is lehet, mivel esetleg sok kosarat kényszerülünk megvizsgálni. Bár a tartománylekérdezések hajlamosak nagy eredményhalmazt produkálni, általában nem kell sokkal több blokkot megvizsgálnunk, mint a minimális blokkszám, amennyibe az eredmény egyáltalán elhelyezhető tetszőleges szervezés esetén.

Legközelebbi szomszéd-lekérdezések

Adott egy P pont, a keresést azzal a kosárral kezdjük, amelyikhez ez a pont tartozik. Ha találunk legalább egy pontot abban, akkor van egy Q jelöltünk a legközelebbi szomszédra. Azonban lehetnek a szomszédos kosarakban olyan pontok, amelyek közelebb vannak P-hez mint a Q; a helyzet hasonló, mint amit az 5.5. ábra mutat. Meg kell vizsgálnunk, vajon a P és az őt tartalmazó kosár határai közötti távolság kisebb-e, mint P és Q távolsága. Ha vannak ilyen határok, akkor minden ilyen határ túloldalán lévő szomszédos kosarat szintén át kell néznünk. Valójában, ha a kosarak olyan téglalapok, amelyek sokkal hosszabbak az egyik dimenzió mentén, mint a másikban, akkor szükséges lehet megvizsgálni még olyan kosarakat is, amelyek nem is szomszédosak a P pontot tartalmazóval.

5.10. példa: Tegyük fel, hogy az 5.6. ábrán, a $P = (45, 200)$ ponthoz legközelebbi pontot keressük. A kosárban az (50, 120) pont van hozzá a legközelebb, a távolsága 80.2. Az alsó három kosárban nem lehet egy pont sem közelebb a (45, 200)-hoz, mivel a fizetés részük legfeljebb 90, így ezeket kihagyhatjuk a keresésből. Azonban a másik öt kosarat át kell néznünk, és azt találjuk, hogy valójában két egyformán közeli

pont van: (30, 260) és (60, 260), P-től 61.8 távolságra. Általában, a legközelebbi szomszéd keresését néhány kosárra, és így néhány lemez I/O-műveletre lehet korlátozni. Azonban, mivel a P ponthoz legközelebbi kosarak lehetnek üresek, nem tudunk könnyen felső korlátot adni a keresés költségére. □

5.2.5. Particionált tördelőfüggvények

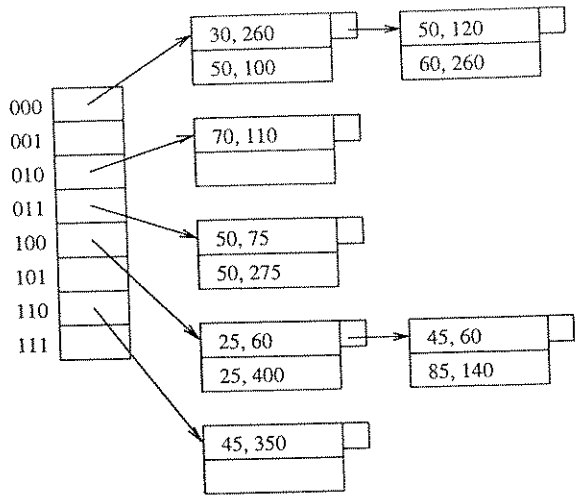
A tördelőfüggvényeknek az argumentuma lehet attribútumértékek egy listája, bár a tipikus az, hogy csak egyetlen attribútumból képzik a tördelőértékeket. Például, ha a egy egész értékű attribútum, b pedig egy szöveg értékű attribútum, akkor hozzáadhatjuk a értékéhez b minden egyes karakterének ASCII kód értékét, majd az így kapott értéket elosztjuk a kosarak számával, és vesszük ennek maradékát. Az eredményt használhatjuk egy tördelőtábla kosárszámaként, mint egy indexet az (a, b) attribútumpáron.

Azonban az ilyen tördelőtábla csak olyan lekérdezésekhez használható, amelyekben az a és a b értéke egyaránt adott. Célszerűbb választás úgy tervezni a tördelőfüggvényt, hogy állítson elő, mondjuk k számú bitet. Ezt a k bitet felosztjuk az n darab attribútum között úgy, hogy a tördelő érték k_i darab bitjét készítjük el az i-edik attribútumból, $i = 1, 2, ..., n$ értékekre, ahol $\sum_{i=1}^{n} k_i = k$. Pontosabban, a h tördelőfüggvény valójában a $(h_1, h_2, ..., h_n)$ tördelőfüggvények olyan listája, ahol a h_i-t az i-edik attribútum értékére alkalmazzuk, és az k_i hosszú bitsorozatot állít elő. Azt a kosarat, amelybe a tördelőérték képzésében részt vevő n attribútumra a $(v_1, v_2, ..., v_n)$ értéket felvevő sort kell helyezni, úgy kapjuk meg, hogy egymás után összekapcsoljuk a $h_1(v_1)h_2(v_2)...h_n(v_n)$ bitsorozatokat.

5.11. példa: Ha van egy tördelőtáblánk 10 bites kosárszámokkal (1024 kosár), akkor felhasználhatunk ebből négy bitet az a, és a maradék hat bitet a b attribútumhoz. Tegyük fel, hogy van egy sorunk, amelyben az a értéke A és a b értéke B, esetleg még vannak más attribútumok is, amelyek nem vesznek részt a tördelésben. Az A értéket leképezzük – az a attribútumhoz tartozó – h_a tördelőfüggvénnyel, és így kapunk négy bitet, mondjuk: 0101. Azután a B-t képezzük le a h_b tördelőfüggvénnyel, legyen a kapott hat bit: 111000. Ehhez a sorhoz rendelt kosárszám ezért 0101111000, azaz a két bitsorozat egymás után helyezésével kapott érték.

A tördelőfüggvényt ilyen módon felosztva hasznos számunkra, ha bármely – egy vagy több – olyan attribútumértékét ismerjük, amelyik szerepel a tördelőfüggvényben. Például, ha adott az a attribútum egy A értéke, és erre $h_a(A) = 0101$, akkor tudjuk, hogy azok a sorok, amelyeknél az a értéke A, csak abban a 64 kosárban lehetnek, amelyeknek a száma 0101... alakú, ahol a ... tetszőleges hat bitet jelölhet. Hasonlóan, ha a b attribútum B értéke adott, kiszűrhetjük azt a 16 kosarat, amely az ilyen sorokat tartalmazhatja, mert a kosárszámuk a $h_b(B)$ hat hosszú bitsorozattal végződik. □

5.12. példa: Vegyük az 5.7. példa „arany ékszer" adatait, amelyeket egy particionált tördelőtáblában akarunk tárolni, amelynek nyolc kosara van (azaz hárombitesek a ko-



5.9. ábra. *Egy particionált tördelőtábla*

sárszámok). Feltesszük, mint korábban is, hogy mindössze két rekord fér egy blokkba. Egy bitet szánunk az életkor attribútumnak, és a maradék két bitet a fizetés attribútumnak.

Az életkor tördelőfüggvényének az életkort 2-vel osztva keletkező maradékot választjuk, azaz a páros életkor olyan kosárba kerül, aminek a száma 0*xy* alakú, ahol *x* és *y* tetszőleges bit lehet. A páratlan életkort tartalmazó rekord olyan kosárba kerül, amelynek száma 1*xy* alakú. A fizetés tördelőfüggvénye legyen a fizetés (ezresekben) maradéka 4-gyel osztva. Például az 57 ezer dollár fizetés maradéka 1, ha 4-gyel osztjuk, ez olyan kosárba kerül, amelynek a száma *z*01, ahol *z* tetszőleges bitérték.

Az 5.9. ábrán az 5.7. példa adatait látjuk, ebben a tördelőtáblában elhelyezve. Figyeljük meg, mivel leginkább a 10-zel osztható életkorokat és fizetéseket használtunk, a tördelőfüggvény nem osztja el a pontokat túl jól. A nyolc kosárból kettőben négy-négy rekord van, így ezekhez túlcsordulásblokk szükséges, miközben három másik kosár üres. □

5.2.6. A rácsos állományok és a particionált tördelés összehasonlítása

Az ebben a részben tárgyalt két adatstruktúra hatékonysága teljesen eltérő. Íme az összehasonlítás főbb pontjai.

- A particionált tördelőtáblák gyakorlatilag teljesen használhatatlanok legközelebbi szomszéd- vagy tartománylekérdezések esetén. A gond az, hogy a pontok fizikai távolságát a kosárszámok közelsége nem tükrözi. Természetesen tervezhetünk egy tördelőfüggvényt úgy, hogy egy a attribútum legkisebb értékéhez az első bitsorozatot rendelje (csupa 0), a sorrendben következő értékhez a következő bitsorozatot (00...01) és így tovább. Ha így tennénk, akkor újra feltalálnánk a rácsos állományt.
- Egy jól megválasztott tördelőfüggvény véletlenszerűen osztja el a pontokat a kosarak között, így a kosarak foglaltsága nagyjából egyenletes lesz. Azonban a rácsos állományok – különösen, ha a dimenziók száma nagy – várhatóan sok kosarat hagynak üresen vagy majdnem üresen. A sejthető oka ennek az, hogy ha sok attribútum van, valószínű, hogy bizonyos összefüggés van legalább néhányuk között. Például, ahogy az 5.2.4. részben említettük, az életkor és a fizetés közötti összefüggés azt okozhatja, hogy az 5.6. ábra legtöbb pontja az átlóhoz közel fekszik, miközben a téglalap nagy része üres. Következésképpen kevesebb kosarat használhatunk, és/vagy kevesebb túlcsordulásblokkra van szükség egy particionált tördelőtáblában, mint egy rácsos állományban.

Ebből következően, ha csak a lekérdezések részleges egyezéssel típusú lekérdezést kell támogatnunk, amelyeknél néhány attribútumértéke adott, a többi teljesen meghatározatlan, akkor a particionált tördelőfüggvény valószínűleg jobb teljesítményt nyújt, mint a rácsos állomány. Megfordítva, ha legközelebbi szomszéd- vagy tartománylekérdezéseket kell gyakran végrehajtanunk, akkor előnyben részesíthetjük a rácsos állományok használatát.

5.2.7. Feladatok

5.2.1. feladat: Az 5.10. ábrán 12 PC leírása látható. Tegyük fel, hogy csak a sebességre és a merevlemez méretére akarunk indexet tervezni.

* a) Válasszunk öt rácsvonalat (összesen a két dimenzióhoz) úgy, hogy ne kerüljön kettőnél több pont egyik kosárba sem.

Modell	*Sebesség*	*Memória*	*Merevlemez*
A	300	32	6,0
B	333	64	4,0
C	400	64	12,7
D	350	32	10,8
E	450	96	14,0
F	400	128	12,7
G	450	128	18,1
H	233	32	4,0
I	266	64	6,0
J	300	64	6,0
K	350	64	12,0
L	400	128	6,0

5.10. ábra. *Néhány PC és a tulajdonságai*

Kis kosarak kezelése

Általában úgy gondolunk a kosárra, mint ami körülbelül egyblokknyi adatot tartalmaz. Azonban vannak helyzetek, amikor annyi kosarat kellene létrehoznunk, hogy egy átlagos kosár csak töredékét tárolja annak a rekordszámnak, ami elfér egy blokkban. Például, sokdimenziós adatok esetén sok kosárra lesz szükségünk, ha sok részre készülünk osztani mindegyik dimenzió mentén. Így, e rész struktúráinál és az 5.3. rész fa alapú szerkezeteinél is, lehet, hogy azt választjuk, hogy több kosarat (vagy facsomópontot) rakunk egy blokkba. Ha ezt tesszük, néhány fontos dologra emlékeznünk kell.:

- A blokkfej információjának tartalmaznia kell, hogy melyik rekord hol van, és hogy melyik kosárhoz tartozik.
- Ha egy rekordot szúrunk be a kosárba, lehet, hogy nincs hely abban a blokkban, amely a kosarat tartalmazza. Ha így van, szét kell vágnunk a blokkot valamilyen módon. El kell döntenünk, melyik kosár melyik blokkba kerüljön, meg kell találni a kosarak rekordjait, és a megfelelő blokkba tenni, valamint beállítani a kosártáblát, hogy a megfelelő blokkra mutasson.

! b) Szét tudjuk-e választani a pontokat úgy, hogy legfeljebb kettő lehet egy kosárban, ha csak négy rácsvonalat használunk? Mutassuk meg hogyan, vagy bizonyítsuk be, hogy ez nem lehetséges.

! c) Javasoljunk egy particionált tördelőfüggvényt, amely szétosztja ezeket a pontokat négy kosárba úgy, hogy legfeljebb négy pont lehet egy kosárban.

! **5.2.2. feladat:** Tegyük fel, hogy az 5.10. ábra adatait egy háromdimenziós rácsos állományban akarjuk elhelyezni, amelyik a sebesség, a memória és a merevlemez attribútumokon alapul. Ajánljunk egy felosztást minden egyes dimenzióra, ami jól osztja el az adatokat.

5.2.3. feladat: Válasszunk egy particionált tördelőfüggvényt, amelyik egy-egy bitet használ a sebesség, a memória és a merevlemez attribútumokhoz, és jól osztja el az 5.10. ábra adatait.

5.2.4. feladat: Tegyük fel, hogy az 5.10. ábra adatait egy rácsos állományba tesszük, amelynek csak a sebesség és a memória a dimenziói. A felosztások a 310, 375 és 425 sebességeknél, és a 40 és 75 memóriaértékeknél vannak. Tegyük fel azt is, hogy csak két pont fér egy kosárba. Javasoljunk jó vágást arra az esetre, ha az alábbi pontot szúrjuk be:

* a) Sebesség = 250 és memória = 48.

b) Sebesség = 333 és memória = 48.

5.2.5. feladat: Tegyük fel, hogy az $R(x, y)$ relációt egy rácsos állományban tároljuk. Mindkét attribútum 0 és 1000 közötti értékeket vehet fel. Ennek a rácsos állománynak a felosztása történetesen egyenletes, x-re minden 20 egységnél van egy felosztás, azaz az osztópontok 20, 40, 60 és így tovább, y-ra pedig 50 egységenként, azaz 50, 100, 150 és így tovább.

a) Hány kosarat kell megvizsgálnunk, hogy megválaszoljuk a következő tartománylekérdezést?

```
SELECT *
FROM R
WHERE 310 < x AND x < 400 AND 520 < y AND y < 730;
```

*! b) Legközelebbi szomszéd-lekérdezést akarunk végrehajtani a (110, 205) pontra. Azzal a kosárral kezdjük a keresést, amelyiknek a bal alsó sarka (100, 200) és a jobb felső sarka (120, 250), és azt találjuk, hogy a legközelebbi pont ebben a kosárban, a (115, 220). Mely kosarakat kell még átnéznünk ahhoz, hogy megbizonyosodjunk róla, hogy ez a legközelebbi pont?

! **5.2.6. feladat:** Tegyük fel, hogy van egy rácsos állományunk három rácsvonallal (azaz négy sávval) minden dimenziójában. Azonban az (x, y) pontoknak történetesen van valami különleges tulajdonságuk. Mondjuk meg a nem üres kosarak lehetséges legnagyobb számát, ha:

* a) A pontok egy vonalon vannak, azaz létezik két konstans, a és b, úgy, hogy $y = ax + b$ minden egyes (x, y) pontra.

b) A pontok között másodfokú összefüggés van, azaz létezik három konstans, a, b és c, úgy hogy $y = ax^2 + bx + c$ minden egyes (x, y) pontra.

5.2.7. feladat: Tegyük fel, hogy az $R(x, y, z)$ relációt egy particionált tördelőtáblában tároljuk, aminek 1024 kosara van (azaz 10 bites a kosárcím). Az R-re vonatkozó lekérdezések mindegyike pontosan egyet ad meg az attribútumok közül, és bármelyiket a három attribútum közül egyforma valószínűséggel adhatják meg. Ha a tördelőfüggvény 5 bitet készít az x alapján, 3 bitet az y alapján és 2 bitet a z alapján, akkor mennyi a kosarak átlagos száma, amelyeket meg kell vizsgálni egy lekérdezés megválaszolásához?

!! **5.2.8. feladat:** Tegyük fel, hogy van egy tördelőtáblánk, amelynek a kosarai 0-tól $2^n - 1$-ig vannak számozva, azaz a kosárcím n bit hosszú. A táblában egy olyan relációt akarunk tárolni, amelynek az attribútumai x és y. Egy lekérdezés vagy az x-et, vagy az y-t határozza meg, de sohasem mind a kettőt egyszerre. Legyen p annak a valószínűsége, hogy az x értéke a megadott.

a) Tegyük fel, hogy a tördelőfüggvényt úgy osztjuk fel, hogy m bitet használunk x-re, és a maradék $n - m$ bitet y-ra. Mi a megvizsgálandó kosarak várható száma m, n és

p függvényében, ami szükséges ahhoz, hogy megválaszoljunk egy véletlenszerű lekérdezést?

b) Mely m értékre (mint n és p függvénye) lesz minimális a kosarak várható száma? Ne aggódjunk, hogy ez az m valószínűleg nem egész szám lesz.

*! **5.2.9. feladat:** Tegyük fel, hogy van egy $R(x, y)$ relációnk, 1 000 000 véletlenszerűen elosztott ponttal. x és y egyaránt 0 és 1000 közötti értékeket vehet fel. Az R relációnak 100 sora fér egy blokkba. Úgy döntünk, hogy egy rácsos állományt használunk, mindegyik dimenzióban egyforma lépésközzel elhelyezett rácsvonalakkal, ahol a sávok szélessége m. Olyan m-et akarunk választani, amelyre minimális a lemez I/O-műveletek száma, ami ahhoz kell, hogy beolvassuk az összes olyan kosarat, amely egy 50 egység oldalú négyzetre vonatkozó tartománylekérdezés megválaszolásához kell. Feltehető, hogy a négyzet oldalai sosem esnek egybe a rácsvonalakkal. Ha az m-et túl nagyra választjuk, akkor sok túlcsordulásblokkunk lesz minden kosárhoz, és sok pontja a kosárnak a lekérdezés tartományán kívül lesz. Ha az m-et túl kicsire választjuk, akkor túl sok kosár lesz, és valószínűleg nem lesznek tele adattal a blokkok. Mi az m legjobb értéke?

5.3. Faszerű struktúrák többdimenziós adatokhoz

Most további négy struktúrát fogunk áttekinteni, amelyek hasznosak többdimenziós adatokra vonatkozó tartomány- vagy legközelebbi szomszéd-lekérdezések esetén. E célból tárgyaljuk a:

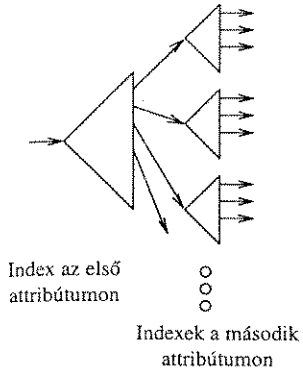
1. többkulcsos indexeket,
2. *kd*-fákat,
3. Quad-fákat,
4. R-fákat.

Az első hármat ponthalmazokhoz szánták. Az R-fa általában területek halmazának ábrázolására használatos, de pontokhoz is hasznos lehet.

5.3.1. Többkulcsos indexek

Tegyük fel, hogy van valahány attribútumunk, amik az adatpontjaink dimenzióit képviselik, és támogatni akarjuk a tartománylekérdezéseket vagy a legközelebbi szomszéd-lekérdezéseket ezeken a pontokon. Egy egyszerű faszerű szerkezet ezen pontok eléréséhez az indexek indexe, vagy általánosabban egy fa, amelyben a csomópontok, minden egyes szinten valamelyik attribútum indexei.

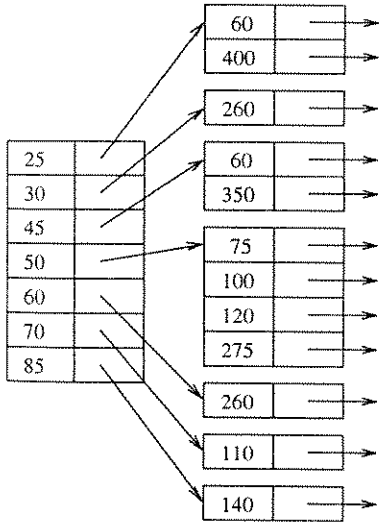
Az ötletet az 5.11. ábra mutatja két attribútum esetén. A „fa gyökere" egy index, a



5.11. ábra. *Egymásba ágyazott indexek használata különböző kulcsokon*

két attribútum közül az elsőre. Ez az index lehet bármelyik fajta szokásos index, mint a B-fa vagy a tördelőtábla. Az index hozzárendel minden egyes keresésikulcs-értékhez – azaz, az első attribútum értékeihez – egy mutatót a másik indexre. Ha V az első attribútum egy értéke, akkor az index, amit elérünk, a V értéket és annak mutatóját követve, az egy olyan pontok halmazának indexe, amelyeknél az első attribútum értéke V, a második attribútum értéke pedig tetszőleges.

5.13. példa: Az 5.12. ábra egy többkulcsos indexet mutat a megszokott „arany ékszer" példánkhoz, ahol az életkor az első attribútum, a második pedig a fizetés. A



5.12. ábra. *Többszintű indexek az életkor/fizetés adatokhoz*

gyökérindex az életkoron, az 5.12. ábra bal oldalán látható. Még nem mutattuk meg azt, hogy hogyan működik az index. Például, a kulcs-mutató párok, amelyek az index hét sorát alkotják, a B-fa levelein szétterjedhetnek. Azonban lényeges, hogy csak azok a kulcsok szerepelnek az életkorok között, amelyekhez van egy vagy több adatpont, és az index egyszerűvé teszi egy adott értékhez rendelt mutató megtalálását.

Az 5.12. ábra jobb oldalán hét index van, amelyek magukhoz a pontokhoz való hozzáférést teszik lehetővé. Például, ha követjük azt a mutatót, amely a gyökérindexben az 50 életkor értékhez van rendelve, elérünk egy kisebb rekordszámú indexhez, amelyben a fizetés a kulcs, és a négy kulcsérték az a négy fizetésérték, ami az életkor = 50 értékhez tartozó pontokban van. Megint csak nem jelöltük az ábrán, hogyan van megvalósítva az index, csak a kulcs-mutató párosítást adtuk meg. Amikor követjük a mutatókat, amelyek az egyes értékekhez (75, 100, 120 és 275) vannak rendelve, megkapjuk az ábrázolt egyedi rekordokat. Például, ha a 100 értékhez rendelt mutatót követjük, megtaláljuk azt a személyt, akinek az életkora 50, és a fizetése 100 ezer dollár. □

Többkulcsos index esetén a második vagy további szintű indexek nagyon kicsik lehetnek. Például az 5.12. ábrán található négy második szintű index, amiben csak egyetlen pár van. Ezért célszerű úgy megvalósítani ezeket, mint egyszerű táblákat, amelyekből többet berakhatunk egy blokkba, ahogyan azt az 5.2.5. részben a „Kis kosarak kezelése" című keretes rész javasolta.

5.3.2. A többkulcsos indexek hatékonysága

Nézzük meg milyen hatékony egy többkulcsos index a különféle többdimenziós lekérdezések esetén. Két attribútum esetére koncentrálunk, bár az általánosítás kettőnél több attribútum esetére magától értetődő.

Lekérdezések részleges egyezéssel

Ha az első attribútum van megadva, az elérés igen hatékony. A gyökérindexet használjuk annak az egy alindexnek a megtalálására, amely az elérni kívánt pontokhoz vezet. Például, ha a gyökér egy B-fa-index, akkor két vagy három lemezműveletet végzünk, amíg megkapjuk a megfelelő alindexet, és aztán a többi lemez I/O-művelet ahhoz szükséges, hogy elérjük teljes egészében az indexet, és magukat a pontokat az adatállományban. Másrészt azonban, ha az első attribútum értéke nem adott, akkor végig kell nézni az összes alindexet, ami időigényes eljárás lehet.

Tartománylekérdezések

A többkulcsos index nagyon jó a tartománylekérdezéshez, amennyiben az egyes indexek maguk támogatják a tartománylekérdezést a saját attribútumukon (azaz, ha B-fa-indexek). Egy tartománylekérdezés megválaszolásához használjuk a gyökérindexet és az első attribútum tartományát, így megtalálhatjuk az összes olyan alindexet, amely

tartalmazhat megfelelő pontokat. Azután ezeket az alindexeket vizsgáljuk meg a második attribútumhoz adott tartományt használva.

5.14. példa: Tegyük fel, hogy az 5.12. ábra szerinti többkulcsos indexünk van, és a tartománylekérdezésünk a $35 \leq$ életkor ≤ 55, és a $100 \leq$ fizetés ≤ 200. Megvizsgálva a gyökérindexet, a 45 és az 50 kulcsértékeket találjuk az életkori határok között. Kövessük a kapcsolódó mutatókat a két alindexhez a fizetésen. A 45 éves életkorhoz nincs fizetés a 100–200 tartományban, míg az 50 éves korhoz tartozó indexben van két ilyen fizetés: a 100 és a 120. Tehát csak két pont van az adott tartományban: az (50, 100) és az (50, 120). □

Legközelebbi szomszéd-lekérdezések

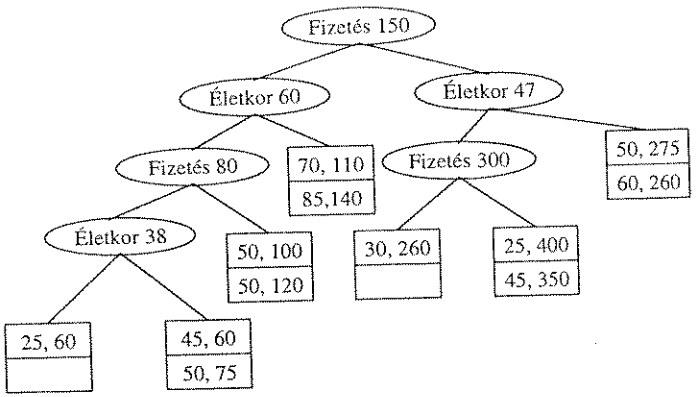
A legközelebbi szomszéd-lekérdezés megválaszolása többkulcsos index segítségével ugyanazt az eljárást követi, mint a legtöbb adatstruktúra esetén ebben a fejezetben. Az (x_0, y_0) pont legközelebbi szomszédjának megkereséséhez választunk egy d távolságot, amelyre várható, hogy találunk néhány pontot az (x_0, y_0) ponttól d távolságon belül. Azután megválaszoljuk az $x_0 - d \leq x \leq x_0 + d$ és az $y_0 - d \leq y \leq y_0 + d$ tartománylekérdezést. Ha az jön ki, hogy nincs pont ebben a tartományban, vagy ha van is pont, de a legközelebbi pont távolsága is nagyobb (x_0, y_0)-tól, mint d (azért még lehet közelebbi pont a tartományon kívül, ahogyan megtárgyaltuk az 5.1.5. részben), akkor meg kell növelnünk a tartományt, és újra kell keresnünk. Viszont tudjuk olyan sorrendben végezni a keresést, hogy a közelebbi helyeket nézzük át először.

5.3.3. *kd*-fák

Egy *kd*-fa (k dimenziós keresőfa) egy központi memóriabeli adatstruktúra, amely a bináris keresőfa általánosítása többdimenziós adatokra. Bemutatjuk az ötletet, majd megtárgyaljuk, hogyan lehet alkalmazni blokk módú tárolókra. A *kd*-fa egy bináris fa, amelyben minden belső csomóponthoz hozzá van rendelve egy a attribútum, és egy V érték, ami szétválasztja az adatpontokat két részre: az egyik rész, azon adatpontokból áll, amelyekre az a érték kisebb V-nél, és a másik rész, amelyben az a érték nagyobb vagy egyenlő mint V. A fa egymás alatti szintjein az attribútumok különböznek, miközben a szintek váltogatják az attribútumokat, körbe járva az összes dimenzión.

A klasszikus *kd*-fában, az adatpontok a csomópontokban vannak elhelyezve csakúgy, mint a bináris keresőfában. Azonban az alapötleten két módosítást fogunk végrehajtani annak érdekében, hogy a blokk módú tárolók bizonyos, korlátozott előnyeit kihasználhassuk.

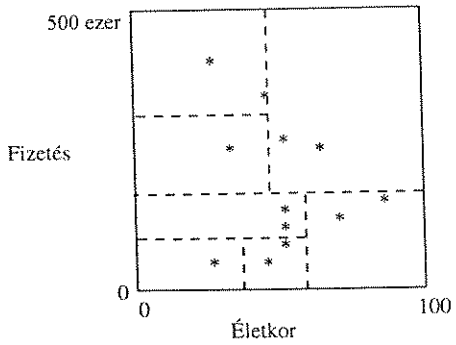
1. A belső csomópontokban csak egy attribútum lesz, egy elhatároló érték ehhez az attribútumhoz, és a mutatók a bal és a jobb gyerekre.
2. A levelek blokkok lesznek, annyi rekordnak biztosítva helyet, amennyi a blokkban elfér.



5.13. ábra. *Egy* kd*-fa*

5.15. példa: Az 5.13. ábrán egy *kd*-fa látható a már megszokott „arany ékszer" példánk tizenkét pontjával. Az egyszerűség kedvéért olyan blokkokat használunk, amelyekbe csak két rekord fér el, ezeket a blokkokat és a tartalmukat a négyzet alakú levelek mutatják. A belső csomópontok ovális alakúak egy attribútummal – vagy életkor, vagy fizetés – és egy értékkel. Például a gyökér a fizetés szerint vág ketté, a bal oldali részfában minden rekordban a fizetés kisebb, mint 150 ezeR dollár, míg a jobb oldali részfában miden rekord legalább 150 ezer dollár fizetés értékű.

A második szinten a vágás életkor szerinti. A gyökér bal oldali gyereke a 60 éves életkornál választ el, tehát minden rekordra ennek a bal oldali részfájában az életkor kisebb mint 60, és a fizetés 150 ezer dollárnál kevesebb. A jobb oldali részfájában az életkor legalább 60, és a fizetés 150 ezer dollárnál kevesebb. Az 5.14. ábra mutatja, hogy a különböző belső csomópontok hogyan osztják fel a pontok terét levélblokkokra. Például, a vízszintes vonal a fizetés = 150 értéknél a gyökérnél lévő vágást mutatja. A vonal alatti rész függőlegesen a 60 éves életkornál válik ketté, míg a felette lévő rész – a gyökér jobb oldali gyerekében lévő határoló értéknek megfelelően – a 47 éves életkornál. □



5.14. ábra. *Az 5.13. ábrán látható fának megfelelő felosztások (partíciók)*

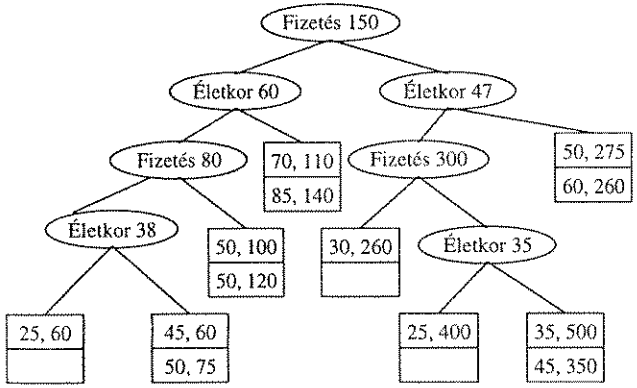
5.3.4. Műveletek a *kd*-fákon

Egy sor keresése, ha adott minden dimenzió értéke, ugyanúgy történik, mint a bináris keresőfákban. Minden belső csomópontnál eldöntjük, hogy merre kell továbbmenni, és így eljutunk egy levélhez, amelynek a blokkját kerestük.

Egy beszúrás végrehajtásához úgy járunk el, mint a keresésnél. Végül egy levélhez jutunk, és ha annak a blokkjában van hely, az új adatpontot betesszük oda. Ha nincs hely, kettévágjuk a blokkot, és megosztjuk a tartalmát, aszerint, hogy azon a szinten, amin a szétvágandó levél van, melyik a megfelelő attribútum. Létrehozunk egy új belső csomópontot, amelynek a gyerekei a két új blokk, és úgy állítjuk be az elhatároló értéket a belső csomópontban, hogy megfeleljen a vágásnak, amit épp most készítettünk.[1]

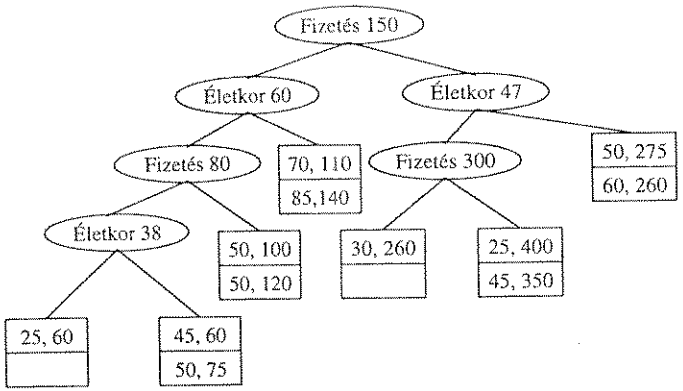
5.16. példa: Tegyük fel, hogy valaki, aki 35 éves, és 500 ezer dollár a fizetése, arany ékszert vásárol. A gyökérnél kezdünk, tudjuk, hogy a fizetés legalább 150 ezer dollár, tehát jobbra megyünk. Ott a csomópontnál összehasonlítjuk a 35 éves kort a 47-tel, ez balra irányít bennünket. A harmadik szinten ismét a fizetéseket hasonlítjuk össze, és a mi értékünk nagyobb, mint a 300 ezer dollár elválasztó érték. Így egy levélhez értünk, ami a (25, 400) és a (45, 350) pontokat tartalmazza, és ez lenne a helye az új, (35, 400) pontnak.

Nincs hely a blokkban három pont számára, tehát szét kell vágnunk. A negyedik szint az életkor szerinti vágás, tehát egy életkort kell választanunk, ami olyan egyenletesen osztja el a rekordokat, amennyire csak lehetséges. A középső érték a 35, egy jó választás, tehát a levelet helyettesítjük egy belső csomóponttal, amely az életkor = 35-



5.15. ábra. *A* kd*-fa a (35, 500) rekord beszúrása után*

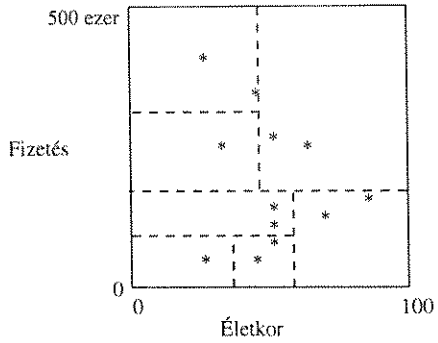
[1] Felmerülhet az a gond, hogy olyan sok pont van azonos értékkel az adott dimenzióban, hogy az adott kosárban csak egy érték van ahhoz a dimenzióhoz, és így nem tudjuk szétvágni. Ekkor megpróbálhatjuk szétvágni egy másik dimenzió mentén, vagy használhatunk túlcsordulásblokkot.



5.13. ábra. *Egy* kd*-fa*

5.15. példa: Az 5.13. ábrán egy *kd*-fa látható a már megszokott „arany ékszer" példánk tizenkét pontjával. Az egyszerűség kedvéért olyan blokkokat használunk, amelyekbe csak két rekord fér el, ezeket a blokkokat és a tartalmukat a négyzet alakú levelek mutatják. A belső csomópontok ovális alakúak egy attribútummal – vagy életkor, vagy fizetés – és egy értékkel. Például a gyökér a fizetés szerint vág ketté, a bal oldali részfában minden rekordban a fizetés kisebb, mint 150 ezeR dollár, míg a jobb oldali részfában miden rekord legalább 150 ezer dollár fizetés értékű.

A második szinten a vágás életkor szerinti. A gyökér bal oldali gyereke a 60 éves életkornál választ el, tehát minden rekordra ennek a bal oldali részfájában az életkor kisebb mint 60, és a fizetés 150 ezer dollárnál kevesebb. A jobb oldali részfájában az életkor legalább 60, és a fizetés 150 ezer dollárnál kevesebb. Az 5.14. ábra mutatja, hogy a különböző belső csomópontok hogyan osztják fel a pontok terét levélblokkokra. Például, a vízszintes vonal a fizetés = 150 értéknél a gyökérnél lévő vágást mutatja. A vonal alatti rész függőlegesen a 60 éves életkornál válik ketté, míg a felette lévő rész – a gyökér jobb oldali gyerekében lévő határoló értéknek megfelelően – a 47 éves életkornál. □



5.14. ábra. *Az 5.13. ábrán látható fának megfelelő felosztások (partíciók)*

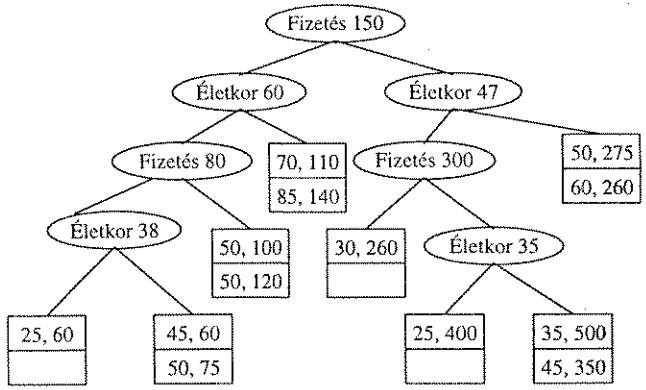
5.3.4. Műveletek a *kd*-fákon

Egy sor keresése, ha adott minden dimenzió értéke, ugyanúgy történik, mint a bináris keresőfákban. Minden belső csomópontnál eldöntjük, hogy merre kell továbbmenni, és így eljutunk egy levélhez, amelynek a blokkját kerestük.

Egy beszúrás végrehajtásához úgy járunk el, mint a keresésnél. Végül egy levélhez jutunk, és ha annak a blokkjában van hely, az új adatpontot betesszük oda. Ha nincs hely, kettévágjuk a blokkot, és megosztjuk a tartalmát, aszerint, hogy azon a szinten, amin a szétvágandó levél van, melyik a megfelelő attribútum. Létrehozunk egy új belső csomópontot, amelynek a gyerekei a két új blokk, és úgy állítjuk be az elhatároló értéket a belső csomópontban, hogy megfeleljen a vágásnak, amit épp most készítettünk.[1]

5.16. példa: Tegyük fel, hogy valaki, aki 35 éves, és 500 ezer dollár a fizetése, arany ékszert vásárol. A gyökérnél kezdünk, tudjuk, hogy a fizetés legalább 150 ezer dollár, tehát jobbra megyünk. Ott a csomópontnál összehasonlítjuk a 35 éves kort a 47-tel, ez balra irányít bennünket. A harmadik szinten ismét a fizetéseket hasonlítjuk össze, és a mi értékünk nagyobb, mint a 300 ezer dollár elválasztó érték. Így egy levélhez értünk, ami a (25, 400) és a (45, 350) pontokat tartalmazza, és ez lenne a helye az új, (35, 400) pontnak.

Nincs hely a blokkban három pont számára, tehát szét kell vágnunk. A negyedik szint az életkor szerinti vágás, tehát egy életkort kell választanunk, ami olyan egyenletesen osztja el a rekordokat, amennyire csak lehetséges. A középső érték a 35, egy jó választás, tehát a levelet helyettesítjük egy belső csomóponttal, amely az életkor = 35-



5.15. ábra. *A* kd*-fa a (35, 500) rekord beszúrása után*

[1] Felmerülhet az a gond, hogy olyan sok pont van azonos értékkel az adott dimenzióban, hogy az adott kosárban csak egy érték van ahhoz a dimenzióhoz, és így nem tudjuk szétvágni. Ekkor megpróbálhatjuk szétvágni egy másik dimenzió mentén, vagy használhatunk túlcsordulásblokkot.

nél vág. A belső csomóponttól balra egy levélblokk lesz a (25, 400) rekorddal, míg a jobbra lévő levél blokkban lesz a másik két rekord, ahogy az 5.15. ábrán látható. □

Az ebben a fejezetben tárgyalt bonyolultabb lekérdezéseket szintén támogatják a *kd*-fák. Íme az alapvető ötletek és az algoritmusok áttekintése.

Lekérdezések részleges egyezéssel

Amennyiben adottak bizonyos attribútumok értékei, akkor ha olyan szinten vagyunk, amelyhez tartozó attribútum értéke ismert, akkor mehetünk valamelyik konkrét irányba. Ha nem tudjuk az aktuális csomóponthoz tartozó attribútum értékét, akkor meg kell vizsgálni mindkét gyerekét. Például, ha az 5.13. ábrán, azokat a pontokat keressük, amelyekre az életkor = 50, akkor meg kell néznünk a gyökér mindkét gyerekét, mivel a gyökér a fizetés szerint vág. Azonban, a gyökér bal oldali gyereknél csak balra kell továbbmenni, míg a gyökér jobb oldali gyereknél csak a jobb oldali részfát kell felderítenünk. Tegyük fel a példa kedvéért, hogy van egy tökéletesen kiegyensúlyozott, kétdimenziós fánk, amelynek sok szintje van, és a kereséshez az egyik dimenzió meg van adva. Akkor a másikhoz tartozó szinteken mindig meg kell vizsgálnunk mindkét utat, végül is el kell érnünk körülbelül annyi levelet, amennyi a levelek teljes számának a négyzetgyöke.

Tartománylekérdezések

Néha a tartomány lehetővé teszi számunkra, hogy csak a csomópont egyik gyereke felé menjünk tovább, de ha a tartomány tartalmazza a csomópont elhatároló értékét, akkor mindkét gyereket meg kell néznünk. Például, ha az életkor tartománya 35 és 55 közötti, a fizetésé 100 ezer dollártól 200 ezer dollárig, akkor az 5.13. ábra fáját a következőképpen járhatjuk be: A fizetés tartománya tartalmazza a gyökérnél lévő 150 ezer dollár értéket, így mindkét gyereket meg kell néznünk. A bal oldali gyereknél a tartomány teljesen a bal oldalra esik, tehát ahhoz a csomóponthoz megyünk, ahol a fizetés 80 ezer dollár. Most a tartomány teljesen jobbra esik, így elértük a levelet, amiben az (50, 100) és az (50, 120) rekordok vannak, ezek mindketten megfelelnek a tartománylekérdezésnek. Visszatérve a gyökér jobb oldali gyerekéhez, az elválasztó érték az életkor = 47, ezért meg kell néznünk mindkét részfát. A 300 ezer dollár fizetésértékű csomópontnál csak balra mehetünk, és így megtaláljuk a (30, 260) pontot, ami kívül esik a tartományon. Az életkor = 47 csomópont jobb oldali gyerekénél találunk további két pontot, de ezek is kívül esnek a tartományon.

Legközelebbi szomszéd-lekérdezések

Használjuk ugyanazt a megközelítést, amit az 5.3.2. részben tárgyaltunk. Kezeljük úgy a feladatot, mint egy tartománylekérdezést a megfelelő tartománnyal, és ismételjük meg egy nagyobb tartománnyal, ha szükséges.

5.3.5. A *kd*-fák alkalmazása másodlagos tárolók esetén

Tegyük fel, hogy egy n levelű *kd*-fát egy állományban tárolunk. Ebben az esetben a gyökértől a levélig tartó út átlagos hossza $\log_2 n$, mint minden bináris fánál. Ha minden egyes csomópontot egy blokkban tárolunk, amikor bejárunk egy utat csomópontonként egy lemez I/O-műveletet kell végrehajtanunk. Például, ha $n = 1000$, akkor körülbelül 10 lemez I/O-műveletre lesz szükségünk, ami sokkal több, mint egy tipikus B-fa esetén szükséges – még ennél sokkal nagyobb állományok esetén is – 2 vagy három lemez I/O-művelet. Ráadásul, mivel a *kd*-fák belső csomópontjaiban viszonylag kevés információ van, a blokk nagy része elpazarolt hely.

Nem tudjuk teljesen megoldani a hosszú út és kihasználatlan terület kettős problémáját. Viszont itt van két megközelítés, ami némi javulást fog hozni a hatékonyságban.

Többutas elágazások a belső csomópontokban

A *kd*-fák belső csomópontjai jobban hasonlítanának a B-fa-csomópontokra, ha több kulcs-mutató párjuk lenne. Ha n kulcsunk lenne egy csomópontban, akkor az a attribútum értékeit $n + 1$ tartományra oszthatnánk. Ha $n + 1$ mutató lenne, követhetnénk az egy megfelelőt ahhoz a részfához, amely csak olyan pontokat tartalmaz, amelyekre az a attribútum értéke abba a tartományba esik. Problémák akkor lépnek fel, amikor megpróbáljuk újraszervezni a csomópontokat azzal a céllal, hogy megtartsuk az eloszlást és az egyensúlyt, ahogyan a B-fáknál tettük. Például, tegyük fel van egy csomópontunk, ami az életkor szerint választ szét, és össze kell olvasztanunk a két gyere-

Semmi sem tart örökké

Az ebben a fejezetben tárgyalt adatstruktúrák lehetővé teszik, hogy beszúráskor és törléskor helyi döntéseket hozzunk, hogyan kell átszervezni a struktúrát. Sok adatbázis-módosítás után, ezen helyi döntések hatása valamiféle kiegyensúlyozatlanságot vihet a struktúrába. Például túl sok üres kosara lehet egy rácsos állománynak, vagy egy *kd*-fa erősen kiegyensúlyozatlan lehet.

Minden adatbázisnál teljesen szokásos, hogy időnként újraszervezik. Az adatbázis visszatöltésekor megvan a lehetősége annak, hogy úgy hozzuk létre az indexstruktúrákat, hogy – legalábbis abban a pillanatban – olyan kiegyensúlyozottak és hatékonyak legyenek, amennyire csak lehetséges az adott típusú indexeknél. Az ilyen újraszervezés költségét a kiegyensúlyozatlansághoz vezető sok módosítás számlájára lehet írni, így az egy módosításra eső költség kicsi. Azonban ehhez az kell, hogy „le tudjuk kapcsolni" az adatbázist, azaz elérhetetlenné tudjuk tenni a visszatöltés idejére. Az ilyen helyzet vagy okoz gondot, vagy nem az alkalmazástól függően. Például sok adatbázist leállítanak éjszakára, amikor senki sem használja őket.

két, amelyek a fizetés szerint vágnak. Nem készíthetünk egyszerűen egy csomópontot, amely tartalmazza a két gyerek fizetés tartományait, mivel ezek a tartományok általában átfedik egymást. Vegyük észre, mennyivel könnyebb lenne, ha (mint a B-fáknál) a két gyerek egyaránt tovább finomítaná az életkori felosztást.

A belső csomópontok blokkokba csoportosítása

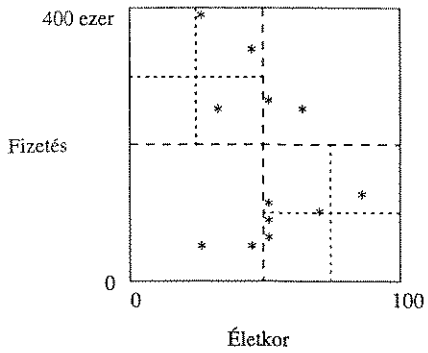
Az előzőek helyett megtarthatjuk azt az elvet, hogy a fa csomópontjainak csak két gyereke van. Több belső csomópontot tehetünk egyetlen blokkba. Abból a megfontolásból, hogy minimalizáljuk a lemezről beolvasandó blokkok számát, miközben lefelé haladunk egy útvonalon, az a legjobb, ha egy csomópont és – bizonyos szint mélységig – annak az összes leszármazottja egyetlen blokkban van. Így, ha egyszer lekérdezünk egy adott csomópontot tartalmazó blokkot, biztosak lehetünk abban, hogy fel tudunk használni néhány további csomópontot is ugyanebből a blokkból, ezzel lemez I/O-műveleteket takarítunk meg. Például tegyük fel, hogy három belső csomópontot tudunk tárolni egy blokkban. Akkor az 5.13. ábrán lévő fa esetén berakhatjuk a gyökeret és két gyerekét egy blokkba. Aztán a fizetés = 80-hoz tartozó csomópontot, és a bal oldali gyerekét betehetjük egy másik blokkba, így csak a fizetés = 300 csomópont maradt, ami egy újabb blokkba kerülhet; ezt például elhelyezhetnénk az előző két csomópont blokkjába is, viszont az osztozkodás jelentős munkát igényel, amikor a fa nő vagy csökken. Így, ha a (20, 60) pontot akarjuk megtalálni, csak két blokkot kell bejárnunk, bár négy csomóponton haladunk át.

5.3.6. Quad-fák

Egy quad-fában minden egyes belső csomópont megfelel egy négyzet alakú területnek kétdimenziós esetben, illetve egy k dimenziós kockának k dimenzió esetén. Mint a fejezet más adatstruktúrái esetén is, elsősorban a kétdimenziós esetet tekintjük át[2]. Ha egy négyzetbe eső pontok száma nem több, mint ami elfér egy blokkban, akkor úgy gondolhatunk erre a négyzetre, mint egy fa levelére, és egy blokkal ábrázoljuk, ami a pontjait tartalmazza. Ha túl sok a pont ahhoz, hogy beleférjen egy blokkba, akkor úgy kezelhetjük a négyzetet, mint egy belső csomópontot, amelynek a gyerekei felelnek meg a négyzet négy negyedének.

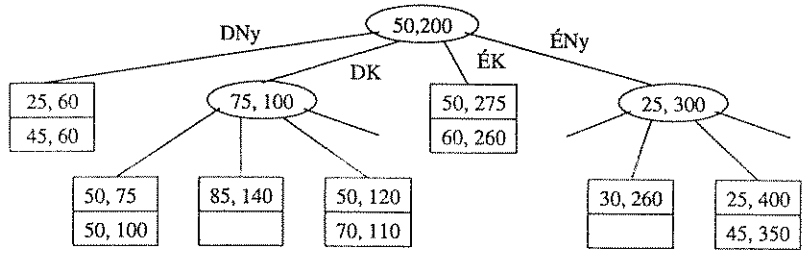
5.17. példa: Az 5.16. ábra az arany ékszer adatpontjait mutatja területekbe szervezve, amelyek megfelelnek egy quad-fa csomópontjainak. Az egyszerű számolás kedvéért, a szokásos teret leszűkítettük úgy, hogy a fizetés 0 és 400 ezer dollár közé essen, és nem 500 ezer dollárig, mint a fejezet többi példájában. Továbbra is feltesszük, hogy két rekord fér egy blokkba.

[2] Ilyenkor használatos a négyágú fa elnevezés is. *A fordító megjegyzése.*



5.16. ábra. *Quad-faként szervezett adatok*

A fát pontosan mutatja az 5.17. ábra. Az iránytű jelöléseit használjuk a negyedekhez, és a csomópontok gyerekeihez (azaz DNy a délnyugati negyedet jelöli, azaz a középponttól balra és lefelé lévő pontokat). A gyerekek sorrendje is mindig olyan, mint a gyökérnél feltüntetett. A belső csomópontok a terület középpontjának a koordinátáit jelölik.



5.17. ábra. *Egy quad-fa*

Mivel az egész tér 12 pontból áll, és csak két pont fér egy blokkba, szét kell vágnunk a teret részekre, amit a szaggatott vonal mutat az 5.16. ábrán. A kapott negyedek közül kettőnek – a délnyugatinak és az északkeletinek –, csak két pontja van. Ezek ábrázolhatók levélként, és nem kell tovább darabolni őket.

A maradék két negyednek kettőnél több pontja van. Mindkettőt tovább negyedeljük, ahogy azt az 5.16. ábrán a szaggatott vonal mutatja. Az eredményül kapott részeknek már csak kettő vagy kevesebb pontja van, így nem szükséges a további darabolás. □

Mivel k dimenzió esetén egy quad-fában egy belső csomópontnak 2^k gyereke van, található k-nak egy olyan tartománya, amelyre a csomópontok kényelmesen elhelyezhetők egy blokkban. Például, ha 128, azaz 2^7 mutató fér el egy blokkban, akkor $k = 7$ megfelelő dimenziószám. A kétdimenziós esetben azonban, a helyzet nem sokkal

jobb, mint a *kd*-fáknál; egy belső csomópontnak négy gyereke van. Továbbá, míg a *kd*-fa esetén a csomópont számára megválaszthatjuk az elhatároló pontot, addig itt kénytelenek vagyunk a quad-fa terület közepét választani, ami vagy egyenletesen osztja szét a terület pontjait, vagy nem. Várhatóan – különösen akkor, ha a dimenzószám nagy – sok üres mutató lesz a belső csomópontokban (ami az üres részeknek felel meg). Természetesen valamivel okosabban is ábrázolhatjuk a sokdimenziós csomópontokat úgy, hogy csak a nem üres mutatókat tároljuk, és egy leírót készítünk, mely megadja, hogy melyik részre vonatkozik, így jelentős helyet takaríthatunk meg.

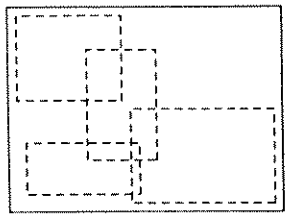
Nem megyünk bele a részletekbe az alapvető műveleteket illetően, amiket a *kd*-fák esetén az 5.3.4. részben tárgyaltunk. Az algoritmusok quad-fa esetén hasonlóak a *kd*-fákéhoz.

5.3.7. R-fák

Egy *R-fa* (régiófa) egy olyan adatstruktúra, amely a B-fa néhány alapötletét általánosítja többdimenziós adatokra. Emlékezzünk arra, hogy a B-fa-csomóponthoz egy kulcshalmaz tartozik, amely egy egyenest szakaszokra oszt. Az egyenes pontjai pontosan egy szakaszhoz tartoznak, ahogy az 5.18. ábra mutatja. Tehát a B-fa megkönnyíti a pontok elérését; ha feltételezzük, hogy a pont a B-fa-csomópont által ábrázolt egyenesen van, akkor egyértelműen meg tudjuk határozni a csomópontnak azt a gyerekét, ahol a pont megtalálható.

5.18. ábra. *A B-fa csomópontjai az egy egyenes mentén levő kulcsokat diszjunkt darabokra osztják*

Egy R-fa viszont olyan adatokat ábrázol, amelyek két- vagy többdimenziós területek lehetnek, ezeket *adatrégiónak* nevezzük. Egy R-fa belső csomópontja egy *belső régiónak* felel meg, vagy egyszerűen csak „régió"-nak, ami általában nem adatrégió. Elméletileg a régió lehet bármilyen alakú, de a gyakorlatban általában téglalap vagy valami más egyszerű alakzat. A R-fa-csomópontnak – kulcsok helyett – alrégiói vannak, amelyek gyerekeinek a tartalmát ábrázolják. Az 5.19. ábra bemutat egy R-fa-csomópontot, amely a nagy vastag vonalas téglalaphoz van rendelve. A pontozott vonalas téglalapok ábrázolják az alrégiókat, amelyek a négy gyerekéhez vannak rendelve. Megjegyezzük, hogy az alrégiók nem fedik le az egész régiót, ami elfogadható, amíg a nagy régión fekvő adatrégiók teljesen benne vannak valamelyik kis területben. Továbbá, a kis régiók átfedhetik egymást, bár kívánatos az átfedéseket minimalizálni.



5.19. ábra. *Egy R-fa-csomópont régió és a gyerekeinek az alrégiói*

5.3.8. Műveletek az R-fákon

Egy tipikus lekérdezés, amihez egy R-fa jól használható, az a „hol-vagyok-én" lekérdezés, amely megad egy P pontot, és azt az adatrégiót vagy régiókat kérdezi, amelyikben a pont benne van. A gyökértől indulunk, amihez az egész régió hozzá van rendelve. Megvizsgáljuk a gyökérben található alrégiókat, és meghatározzuk azokat a gyerekeit, amelyek olyan belső régióknak felelnek meg, amik tartalmazzák a P pontot. Ilyen régió lehet: 0, 1 vagy több.

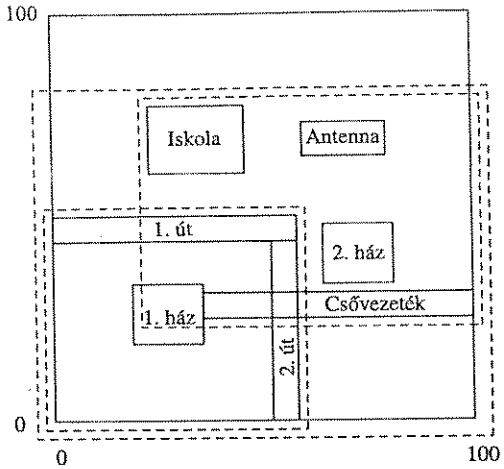
Ha nincs ilyen régió, akkor készen vagyunk, a P nincs benne egyik adatrégióban sem. Ha van legalább egy belső régió, amely tartalmazza a P pontot, akkor minden ilyen régiónak megfelelő gyereknél rekurzívan tovább kell keresnünk. Így eljutva egy vagy több levélhez, találjuk meg a tényleges adatrégiókat, és abban vagy a teljes adatrekordot, vagy csak egy mutatót a keresett adatrekordra.

Amikor egy új R régiót szúrunk be egy R-fába, a gyökértől elindulva próbálunk meg egy olyan alrégiót találni, amibe az R beleillik. Ha egynél több ilyen régió van, kiválasztunk egyet, elmegyünk a neki megfelelő gyerekhez, és ott megismételjük az eljárást. Ha nincs ilyen R-et tartalmazó alrégió, akkor meg kell növelnünk az alrégiók valamelyikét, ennek a kiválasztása azonban általában nem egyszerű. Intuitíven: a régiókat csak a feltétlen szükséges mértékben akarjuk növelni, vagyis a gyerek alrégiói közül azt kell választani, amelyik a legkevésbé fog növekedni; ennek a régiónak változtassuk meg a határait úgy, hogy R-et tartalmazza, majd rekurzívan szúrjuk be R-et a megfelelő gyereknél.

Végül eljutunk egy levélhez, ahová be kell szúrnunk az R régiót. Ha nincs hely a levélben R számára, akkor a levelet szét kell vágnunk. Most is több lehetőség közül választhatunk. Általában azt akarjuk, hogy a két részterület a lehető legkisebb legyen, de még – többek közt – tartalmazzák az eredeti levél összes adatrégióját is. Amikor a levelet szétvágjuk, a felette lévő csomópontban az eredeti levél régió-mutató párját kicseréljük két régió-mutató értékkel, amelyek a két új levélnek felelnek meg. Ha van hely a szülőben, akkor készen vagyunk. Különben – mint a B-fák esetén – rekurzívan szétvágjuk a csomópontokat a fán felfelé haladva.

5.18. példa: Vizsgáljuk meg azt az esetet, amikor az 5.1. ábrán látható térképhez egy új régiót akarunk hozzáadni. Tegyük fel, hogy a levelekben hat régiónak van hely. További feltevés, hogy az 5.1. ábra mind a hat régiója egyetlen levélben van, ennek a régióját a külső (folyamatos vonallal rajzolt) téglalap ábrázolja az 5.20. ábrán.

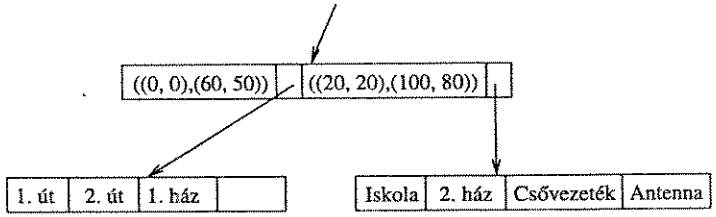
Tegyük fel, hogy a helyi rádiótelefon-társaság egy antennát akar telepíteni az 5.20. ábrán látható helyre. Mivel a hét adatrégió már nem fér el egy levélben, szétvágjuk a levelet, négy régió lesz az egyikben és három a másikban. Több lehetőségünk is van, ezek közül azt a felosztást választottuk, amit a 5.20. ábra mutat (a belső, szaggatott



5.20. ábra. *Objektumhalmaz szétvágása*

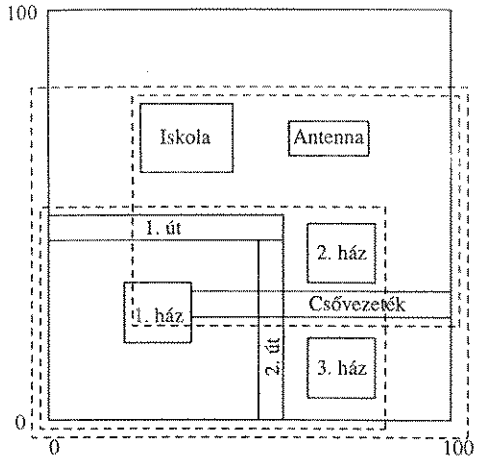
vonallal jelzett téglalapok), amely minimálissá teszi az átfedést, miközben olyan egyenletesen vágja szét a levelet, amennyire csak lehetséges.

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



5.21. ábra. *Egy R-fa*

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



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

5.3.9. Feladatok

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.4. Bittérképindexek

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

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

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

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

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

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

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

5.4.1. Indítékok a bittérképindexekhez

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

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

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

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

```
Film(cím, év, hossz, gyártó)
```

relációt kérdeztük le a következő lekérdezéssel:

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

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

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

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

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

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

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

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

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

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

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

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

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

011110000110 AND 000111100000 = 000110000000

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

5.4.2. Tömörített bittérképek

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

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

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

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

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

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

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

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

A bináris számok nem megfelelőek a szakaszhosszkódoláshoz

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

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

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

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

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

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

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

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

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

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

5.4.3. Műveletek szakaszhosszkódolt bitvektorokon

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

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

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

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

5.4.4. Bittérképindexek kezelése

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

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

Bitvektorok keresése

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

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

Rekordok keresése

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

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

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

Adatállományok módosításának kezelése

Két szempontból jelentenek problémát az adatállomány-módosítások a bittérképindexre:

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

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

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

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

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

5.4.5. Feladatok

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

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

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

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

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

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

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

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

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

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

5.5. Összefoglalás

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

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

Adatállományok módosításának kezelése

Két szempontból jelentenek problémát az adatállomány-módosítások a bittérképindexre:

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

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

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

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

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

5.4.5. Feladatok

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

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

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

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

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

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

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

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

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

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

5.5. Összefoglalás

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

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

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

5.6. Irodalomjegyzék

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

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

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

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

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

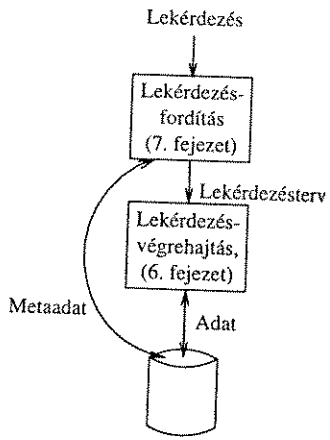
12. R. L. Rivest, „Partial match retrieval algorithms," *SIAM J. Computing* 5:1 (1976), pp. 19–50.
13. J. T. Robinson, „The K-D-B-tree: a search structure for large multidimensional dynamic indexes," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1981), pp. 10–18.
14. J. B. Rothnie Jr. and T.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

6. fejezet

Lekérdezések végrehajtása

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

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



6.1. ábra. *A lekérdezésfeldolgozó fontosabb részei*

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

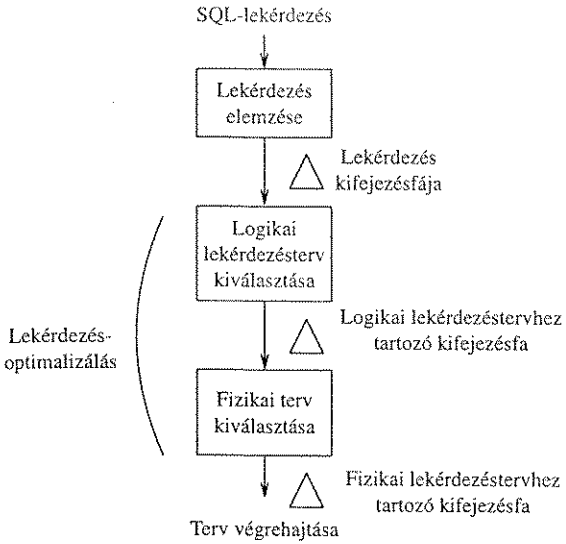
A relációs algebra használatának egyik előnye, hogy megkönnyíti a lekérdezések különböző lehetséges formáinak kifejezését. Az egy lekérdezéshez tartozó különböző algebrai kifejezéseket *logikai lekérdezésterveknek* (logical query plans) nevezzük. Ezeket a terveket gyakran kifejezésfákkal ábrázoljuk.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

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

A lekérdezésfordítás áttekintése

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

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



6.2. ábra. *A lekérdezésfordítás áttekintése*

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

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

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

6.1. Algebrai megközelítés

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

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

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

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

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

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

6.1.1. Egyesítés, metszet és különbség

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.1.2. Kiválasztás

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

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

Alkérdések a `WHERE` záradékban

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

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

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

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

a	b
0	1
2	3
4	5
2	3

A $\sigma_{a \geq 1}(R)$ eredménye:

a	b
2	3
4	5
2	3

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

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

a	b
4	5

□

6.1.3. Vetítés

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

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

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

6.3. példa: Legyen R a következő reláció:

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

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

a	x
0	3
0	3
3	9

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

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

x	y
1	1
1	1
1	1

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

6.1.4. Relációk szorzata

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

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

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

a	b
0	1
2	3
2	3

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

b	c
1	4
1	4
2	5

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

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

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

6.1.5. Összekapcsolások

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

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

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

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

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

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

a	b	c
0	1	4
0	1	4

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

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

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

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

6.4. ábra. *A théta-összekapcsolás eredménye*

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

A „théta-összekapcsolás" jelentése

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

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

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

□

6.1.6. Ismétlődések kiküszöbölése

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

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

a	*b*
0	1
2	3

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

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

6.1.7. Csoportosítás és összesítés

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

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

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

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

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

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

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

δ a γ egy speciális esete

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

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

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

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

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

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

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

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

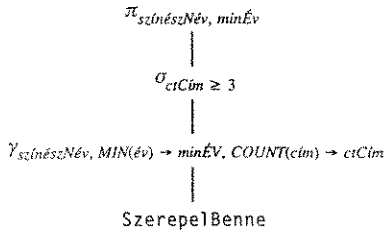
Az ekvivalens algebrai kifejezés csoportosítást fog végezni a `színészNév` attribútumon. Minden bizonnyal ki kell számolnunk minden csoportra a `MIN(év)` összesítést. Ahhoz azonban, hogy meg tudjuk ítélni, hogy melyik csoport teljesíti a `HAVING` záradékot, ki kell számolnunk a `COUNT(cím)` összesítést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

A csoportosító kifejezéssel kezdjük:

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

A kifejezés eredményének első két oszlopára a lekérdezés eredménye miatt van

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



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

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

6.1.8. Rendezés

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

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

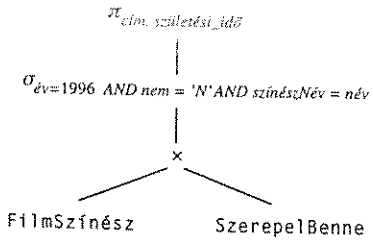
számítanak. Azok a sorok, amelyek megegyeznek az a_1 és a_2 értékeiken, az a_3 értékek szerint kerülnek rendezésre és így tovább. Azok a sorok, amelyek még az a_n attribútumon is megegyeznek, tetszőleges sorrendbe helyezhetők. Éppúgy, mint az SQL esetén, feltételezzük, hogy az alapértelmezett rendezési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

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

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

6.1.9. Kifejezésfák

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



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

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

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

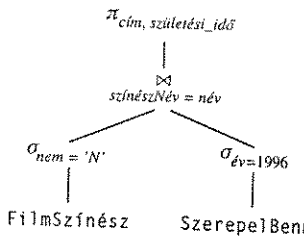
amelyekből szeretnénk kinyerni azokat a filmcímeket, a színésznők születésni idejével együtt, amelyekben az 1996-ban megjelent filmekben szereplő színésznők játszanak:

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

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

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

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



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

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

6.1.10. Feladatok

6.1.1. feladat: Adott két reláció:

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

Számoljuk ki a következőket:

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

6.1.2. feladat: Adott három reláció:

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

Összekapcsolás jellegű operátorok

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

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

Számoljuk ki a következőket:

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

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

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

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

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

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

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

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

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

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

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

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

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


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

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

6.2. Bevezetés a fizikai lekérdezésterv-operátorok világába

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

6.2.1. Táblák átvizsgálása

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

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

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

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

6.2.2. Rendezés a táblák átvizsgálásakor

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

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

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

6.2.3. A fizikai operátorok kiszámításának modellje

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

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

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

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

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

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

6.2.4. A költségbecslés paraméterei

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

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

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

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

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

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

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

6.2.5. Az átvizsgáló operátorok I/O-költsége

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

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

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

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

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

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

6.2.6. Fizikai operátorok megvalósításához használatos iterátorok

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

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

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

Miért éppen iterátorok?

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

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

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

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

CLOSE(R) {
}
```

6.8. ábra. *A táblaátvizsgálás-operátor iterátora*

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

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

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

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

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

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

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

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

6.3. Adatbázis-műveletek egymenetes algoritmusai

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

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

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

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

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

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

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

6.9. ábra. *Az egyesítés iterátor felépítése komponenseiből*

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

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

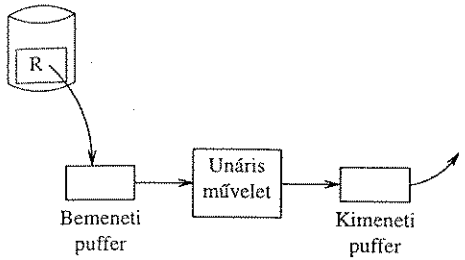
c) Végül vannak olyan módszerek, amelyek az adatok méretétől függetlenül működnek. Az ilyen algoritmusok három vagy annál is több menettel dolgoznak, és tulajdonképpen a kétmenetes algoritmusok természetes, rekurzív általánosításai. A többmenetes algoritmusokkal a 6.9. részben foglalkozunk.

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

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

6.3.1. Soronkénti műveletek egymenetes algoritmusai

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



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

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

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

6.3.2. Unáris, teljes relációs műveletek egymenetes algoritmusai

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

Az extra pufferek felgyorsíthatják a műveleteket

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

Ismétlődések kiküszöbölése

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

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

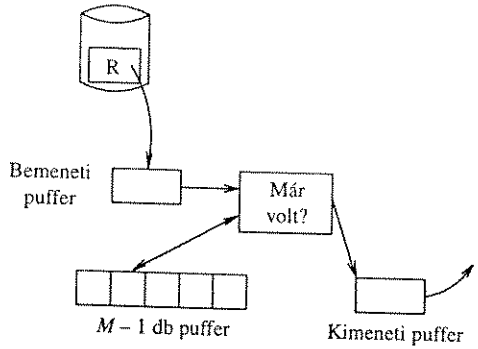
A döntés támogatására kell egy másolatot tartanunk a memóriában az összes általunk már látott sorról, amint azt a 6.11. ábra mutatja. Egy memóriapuffer tartalmazza R sorainak egy blokkját, míg a fennmaradó $M-1$ puffert használhatjuk arra, hogy tartalmazzák a már előfordult minden egyes sor egy másolatát.

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

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

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

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



6.11. ábra. *Memória kezelése ismétlődések egymenetes kiküszöbölésénél*

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

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

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

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

Csoportosítás

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

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

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

Nem nyalábolt adatokkal végzett műveletek

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

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

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

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

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

6.3.3. Bináris műveletek egymenetes algoritmusai

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

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

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

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

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

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

Halmazegyesítés

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

Halmazmetszet

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

Halmazkülönbség

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

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

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

Multihalmazmetszet

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

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

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

Multihalmazkülönbség

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

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

Szorzat

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

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

Természetes összekapcsolás

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

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

Mi van akkor, ha M ismeretlen?

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

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

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

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

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

6.3.4. Feladatok

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

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

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

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

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

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

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

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

6.4. Beágyazott ciklusú összekapcsolások

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

6.4.1. Sor alapú beágyazott ciklusú összekapcsolás

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

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

összekapcsolást a következőképpen számítjuk ki:

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

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

6.4.2. Egy iterátor a sor alapú beágyazott ciklusú összekapcsoláshoz

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

6.4.3. Egy algoritmus a blokk alapú beágyazott ciklusú összekapcsoláshoz

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

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


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

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

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

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

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

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

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

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

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

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

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

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

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

6.4.4. A beágyazott ciklusú összekapcsolás elemzése

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

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

vagy átalakítva

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

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

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

6.4.5. Az eddigi algoritmusok összefoglalása

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

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

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

6.4.6. Feladatok

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

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

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

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

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

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

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

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

6.5. Rendezésen alapuló kétmenetes algoritmusok

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

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

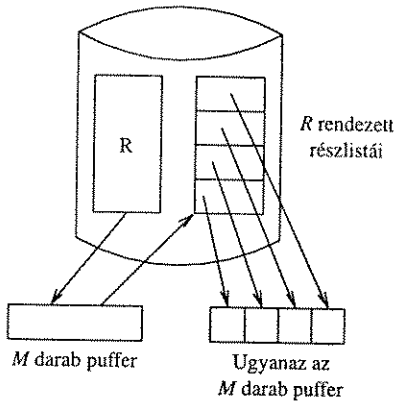
Ebben a részben a rendezést fogjuk vizsgálni, mint a relációs műveletek végrehajtására szolgáló eszközt. Az alapötlet a következő. Tegyük fel, hogy van egy nagy relációnk, R, amelyre $B(R)$ nagyobb M-nél, vagyis a rendelkezésre álló memóriapufferek számánál. Ekkor egymás után többször ismételve a következőket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

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

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

6.5.1. Ismétlődések kiküszöbölése rendezés segítségével

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



6.15. ábra. *Kétmenetes algoritmus az ismétlődések kiküszöbölésére*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.5.2. Csoportosítás és összesítés rendezés segítségével

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

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

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

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

6.5.3. Az egyesítés egy rendezésen alapuló algoritmusa

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

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

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

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

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

6.5.4. A metszet és a különbség rendezésen alapuló algoritmusai

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.5.5. Egy egyszerű rendezésen alapuló összekapcsolási algoritmus

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

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

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

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

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

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

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

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

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

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

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

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

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

6.5.6. Az egyszerű rendezéses összekapcsolás elemzése

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

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

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

6.5.7. Egy hatékonyabb rendezésen alapuló összekapcsolás

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

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

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

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

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

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

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

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

6.5.8. A rendezésen alapuló algoritmusok összefoglalása

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

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

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

6.5.9. Feladatok

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.6. Tördelésen alapuló kétmenetes algoritmusok

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

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

6.6.1. Relációk particionálása tördeléssel

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

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

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

6.6.2. Egy tördelésen alapuló algoritmus az ismétlődések kiküszöbölésére

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

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

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

6.6.3. Egy tördelésen alapuló algoritmus a csoportosításra és az összesítésre

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

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

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

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

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

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

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

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

6.6.5. A tördeléses összekapcsolási algoritmus

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

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

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

A 6.19. példát általánosítva kimondhatjuk, hogy:

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

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

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

6.6.6. Lemez I/O-műveletek megtakarítása

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

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

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

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

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

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

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

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

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

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

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

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

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

a teljes költség pedig

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

képletekkel fejezhető ki.

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

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

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

6.6.7. A tördelésen alapuló algoritmusok összefoglalása

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

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

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

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

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

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

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

6.6.8. Feladatok

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

* a) δ.
b) γ.
c) $\cap_B$.
d) $-_S$.

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

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

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

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

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

6.7. Index alapú algoritmusok

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

6.7.1. Nyalábolt és nem nyalábolt indexek

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

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

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

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

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

6.7.2. Index alapú kiválasztás

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

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

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

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

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

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

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

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

A nyalábolás alapfogalmai

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

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

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

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

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

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

□

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

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

6.7.3. Összekapcsolás index segítségével

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

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

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

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

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

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

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

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

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

6.7.4. Összekapcsolások rendezett index segítségével

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

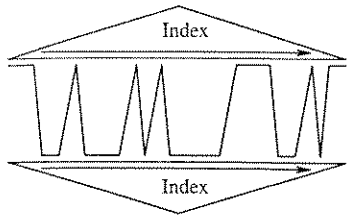
Szélsőséges esetben, amikor az R és az S egyaránt rendelkezik rendezett indexszel az Y attribútumo(ko)n, akkor a 6.5.5. rész egyszeru rendezésen alapuló összekapcsolásából csak a befejező lépést kell végrehajtanunk. Ezt a módszert néha *cikcakk-összekapcsolásnak* is szokták nevezni, mert oda-vissza ugrálunk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

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

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

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

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



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

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

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

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

6.7.5. Feladatok

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

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

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

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

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

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

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

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

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

A fenti lekérdezés a következő „filmes" relációkat használja:

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

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

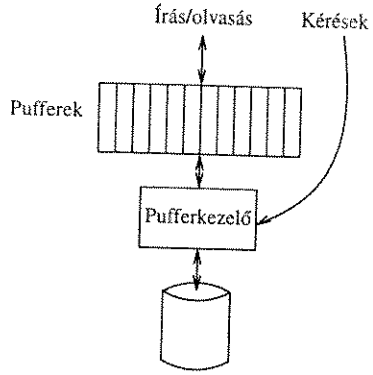
$\sigma_{\text{title} = \text{'King Kong'}}(\text{SzerepelBenne})$

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

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

6.8. Pufferkezelés

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



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

6.8.1. A pufferkezelő működése

A pufferkezelő alapvetően kétféle módon működhet, amelyek a következők:

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

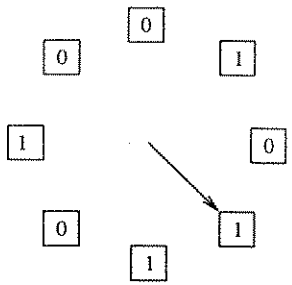
Bármelyik megoldást alkalmazza is az adatbázis-kezelő, mindkét esetben ugyanaz a probléma merül fel. Nevezetesen, a pufferkezelőnek úgy kell korlátoznia a használatban levő pufferek számát, hogy azok beleférjenek a rendelkezésre álló memóriába. Ha a pufferkezelő közvetlenül kezeli a memóriát, és a kérések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

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

6.8.2. Pufferkezelő stratégiák

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

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



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

További lehetőségek az óra algoritmussal kapcsolatban

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

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

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

6.8.3. Kapcsolat a fizikai operátor kiválasztása és a pufferkezelés között

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

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

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

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

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

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

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

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

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

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

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

6.8.4. Feladatok

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

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

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

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

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

6.9. Több mint kétmenetes algoritmusok

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

6.9.1. Többmenetes, rendezésen alapuló algoritmusok

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

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

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

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

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

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

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

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

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

6.9.2. Többmenetes, rendezésen alapuló algoritmusok műveletigénye

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

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

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

Ha a fenti rekurziót tovább folytatjuk, a következőt kapjuk:

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

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

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

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

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

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

6.9.3. Többmenetes, tördelésen alapuló algoritmusok

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

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

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

6.9.4. Többmenetes, tördelésen alapuló algoritmusok műveletigénye

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

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

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

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

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

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

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

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

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

6.9.5. Feladatok

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

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

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

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

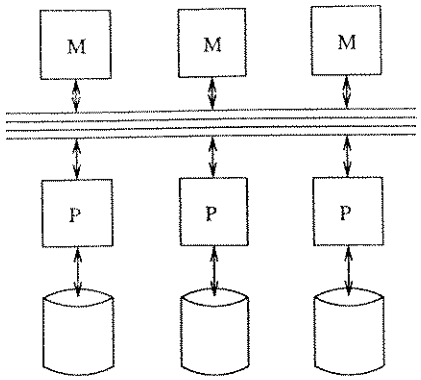
6.10. Párhuzamos algoritmusok relációs műveletekre

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

6.10.1. A párhuzamosság modelljei

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

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

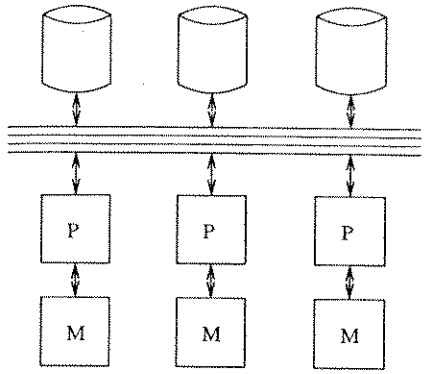


6.23. ábra. *Egy megosztott memóriájú gép*

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

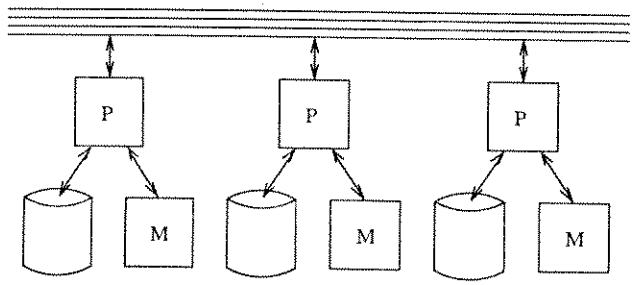
A párhuzamos számítógépek három legfontosabb osztálya a következő.

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



6.24. ábra. *Egy megosztott lemezes gép*

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



6.25. ábra. *Egy megosztás nélküli gép*

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

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

Algoritmusok más párhuzamos architektúrákra

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

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

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

6.10.2. Soronkénti műveletek párhuzamos megvalósítása

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

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

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

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

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

6.10.3. Teljes relációs műveletek párhuzamos algoritmusai

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

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

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

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

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

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

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

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

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

6.10.4. A párhuzamos algoritmusok hatékonysága

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

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

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

Naaagy hiba!

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.10.5. Feladatok

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

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

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

6.11. Összefoglalás

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

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

6.12. Irodalomjegyzék

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

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

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

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

7. fejezet

A lekérdezésfordító

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

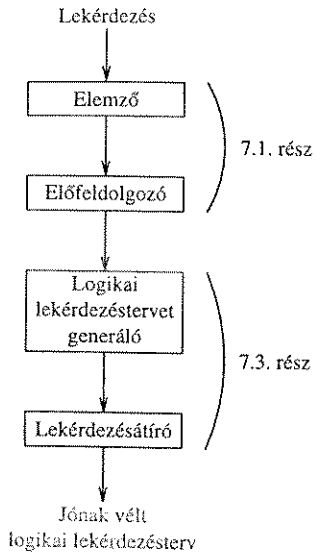
1. Az SQL-szerű nyelven megfogalmazott lekérdezés *elemzése,* azaz átalakítása egy elemzőfává, amely a lekérdezés szerkezetének egy jól használható reprezentációját adja.
2. Az elemzőfa átalakítása egy, a relációs algebrával (vagy más hasonló jelölésrendszerrel) megfogalmazott kifejezésfává, amit *logikai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

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

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

7.1. Elemzés

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



7.1. ábra. *Lekérdezés átalakítása logikai lekérdezéstervvé*

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

7.1.1. Szintaktikus elemzés és elemzőfák

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

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

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

7.1.2. Egy leegyszerűsített SQL-részletet leíró nyelvtan

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

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

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

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

„Select-From-Where" kifejezések

Az <SFW> szintaktikus kategóriát egyetlen szabállyal definiáljuk:

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

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

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

Select-listák

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

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

From-listák

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

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

Feltételek

Az alábbi szabályokat használjuk:

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

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

```
<Sor> ::= <Attribútum>
```

Alap szintaktikus kategóriák

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

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

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

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

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

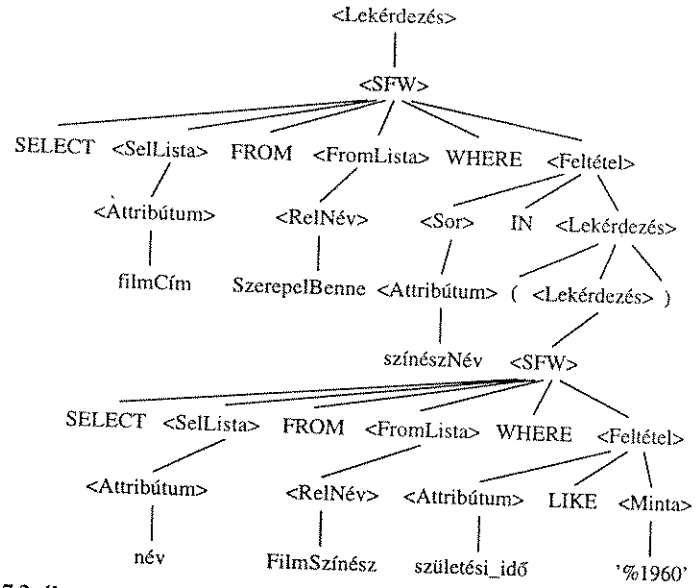
```
SELECT filmCím
FROM SzerepelBenne
WHERE színészNév IN (
   SELECT név
   FROM FilmSzínész
   WHERE születési_idő LIKE '%1960'
);
```

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

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

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

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



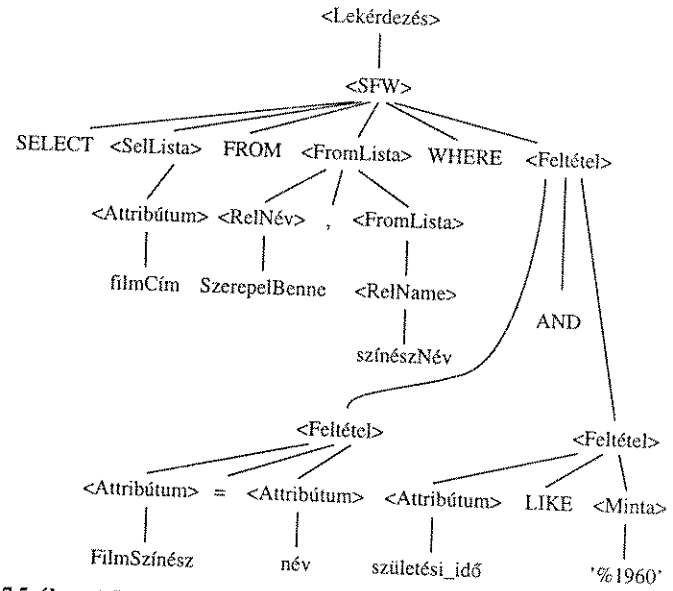
7.3. ábra. *A 7.2. ábrához tartozó elemzőfa*


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

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

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

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



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

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

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

7.1.3. Az előfeldolgozó

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

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

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

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

7.1.4. Feladatok

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

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

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

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

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

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

7.2. Algebrai szabályok lekérdezéstervek javítására

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

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

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

7.2.1. Kommutatív és asszociatív szabályok

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

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

A relációs algebra néhány operátora egyszerre kommutatív és asszociatív:

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

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

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

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

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

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

a	b	c
0	1	2

sor ugyanaz a sor, mint a

b	c	a
1	2	0

vagy mint az oszlopok további négy permutációjával kapott sorok.

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

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

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

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

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

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

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

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

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

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

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

kifejezést egy feltételezett asszociatív szabállyal a következővé alakíthatnánk:

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

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

A multihalmazokra és a halmazokra vonatkozó szabályok különbözhetnek

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

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

7.2.2. Kiválasztással kapcsolatos szabályok

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

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

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

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

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

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

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

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

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

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

Az egyesítésre vonatkozó szabály:

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

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

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

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

- $\sigma_C(R - S) = \sigma_C(R) - \sigma_C(S)$

A soron következő szabályok megengedik, hogy a kiválasztást az egyik vagy mindkét argumentumhoz elvigyük. Ha a kiválasztás σ_C, akkor ezt a kiválasztást csak egy olyan relációhoz tolhatjuk, amely rendelkezik a C-ben említett összes attribútummal, ha van ilyen. Az alábbi szabályokat azzal a feltételezéssel élve fogalmazzuk meg, hogy az R relációban megvan az összes C-ben szereplő attribútum.

- $\sigma_C(R \times S) = \sigma_C(R) \times S$
- $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$
- $\sigma_C(R \underset{D}{\bowtie} S) = \sigma_C(R) \underset{D}{\bowtie} S$
- $\sigma_C(R \cap S) = \sigma_C(R) \cap S$

Ha a C-ben csak S-beli attribútumok szerepelnek, akkor azt írhatjuk, hogy:

- $\sigma_C(R \times S) = R \times \sigma_C(S)$

és hasonlóan írhatók át a $\bowtie$, $\underset{D}{\bowtie}$ és $\cap$ operátorok szabályai. Ha az R és S relációk mindegyikében szerepel az összes C-beli attribútum, akkor használható az alábbi szabály:

- $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie \sigma_C(S)$

Vegyük észre, hogy nem alkalmazható ilyen szabály, ha az operátor $\times$ vagy $\underset{D}{\bowtie}$, ezekben az esetekben ugyanis R-nek és S-nek nincsenek közös attribútumai. A $\cap$ esetében viszont a szabály mindig érvényes lesz, hiszen ekkor az R és S sémája ugyanaz kell hogy legyen.

7.7. példa: Tekintsük az $R(a, b)$ és $S(b, c)$ relációkat és a következő kifejezést:

$\sigma_{(a = 1 \text{ OR } a = 3) \text{ AND } b < c}(R \bowtie S)$

A $b < c$ feltétel egyedül az S-re alkalmazható, az $a = 1$ OR $a = 3$ feltétel pedig csak az R-re alkalmazható. Ezért a két feltétel összekötő AND szétvágásával kezdjük, csakúgy mint a 7.6. példa első változatánál:

$\sigma_{a = 1 \text{ OR } a = 3}(\sigma_{b<c}(R \bowtie S))$

Ezt követően bevihetjük a $\sigma_{b<c}$ kiválasztást az S-hez, ami az alábbi kifejezést adja:

$\sigma_{a = 1 \text{ OR } a = 3}(R \bowtie \sigma_{b<c}(S))$

Végül az első feltételt bevisszük az R-hez: $\sigma_{a = 1 \text{ OR } a = 3}(R) \bowtie \sigma_{b < c}(S)$. Ha akarjuk, szétvághatjuk az OR-ral kapcsolt két feltételt, mint ahogy a 7.6. példában tettük. Ez azonban vagy előnyös, vagy nem. □

Néhány triviális szabály

Nem áll szándékunkban a relációs algebrára érvényes összes szabályt megfogalmazni. Az olvasó legyen különösen óvatos a szabályokkal kapcsolatban, ha speciális esetekről van szó: például amikor egy reláció üres, egy kiválasztás vagy théta-összekapcsolás feltétele mindig igaz vagy hamis, vagy a teljes attribútumlistára történik vetítés. Lássunk néhányat a sok lehetséges speciális esetre vonatkozó szabály közül:

- Üres relációra vonatkozó bármilyen kiválasztás üres relációt ad.
- Ha egy C feltétel mindig igaz (pl. $x > 10$ OR $x \leq 10$ olyan relációra vonatkozóan, amely kizárja, hogy x = NULL), akkor $\sigma_C(R) = R$.
- Ha R üres, akkor $R \cup S = S$.

7.2.3. Kiválasztások tologatása

Amint már említettük, egy kiválasztás tologatása lefelé a fában – azaz a 7.2.2. részben szereplő valamely szabály bal oldalának helyettesítése annak jobb oldalával – a lekérdezésoptimalizáló egyik leghatékonyabb eszköze. Sokáig azt feltételezték, hogy úgy optimalizálhatunk, ha a kiválasztásra vonatkozó szabályokat ebbe az irányba alkalmazzuk. Amikor viszont általánossá vált a nézettáblák támogatása, úgy találták, hogy bizonyos esetekben lényeges volt, hogy egy kiválasztást először olyan fentre felvigyünk a fában, amennyire lehet, és utána tologassuk lefelé a kiválasztásokat a lehetséges ágakon. A kiválasztások tologatásának jó megközelítését egy példával szemléltetjük.

7.8. példa: Tegyük fel, hogy adottak a következő relációk:

```
SzerepelBenne(filmCím, év, színészNév )
Film(cím, év, hossz, stúdióNév)
```

és az alábbi SQL-nézettábla:

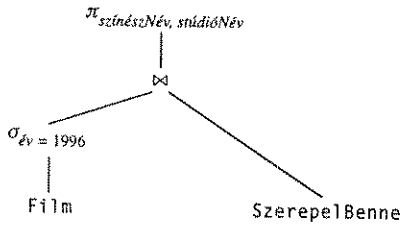
```
CREATE VIEW Filmek1996 AS
   SELECT *
   FROM Film
   WHERE év = 1996;
```

A „mely színészek mely stúdiónak dolgoztak 1996-ban?" kérdést megfogalmazó SQL-lekérdezés:

```
SELECT színészNév, stúdióNév
FROM Filmek1996 NATURAL JOIN SzerepelBenne;
```

A `Filmek1996` nézettáblát egy relációs algebrai kifejezés definiálja:

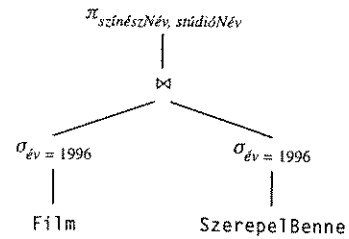
$\sigma_{év = 1996}$(`Film`)



7.6. ábra. *Egy lekérdezés és nézettábla alapján készített logikai lekérdezésterv*

A lekérdezéshez, ami ennek a kifejezésnek a természetes összekapcsolása a `SzerepelBenne` relációval, majd egy vetítés a `színészNév` és `stúdióNév` attribútumokra, a 7.6. ábrán látható kifejezés vagy „logikai lekérdezésterv" tartozik.

A kifejezésben szereplő egyetlen kiválasztás már olyan lent van a fában, amennyire lehet, így nincs lehetőség a „kiválasztás tologatására lefelé a fában". A $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$ szabályt viszont alkalmazhatjuk „visszafelé", hogy a $\sigma_{év = 1996}$ kiválasztást az összekapcsolás fölé vigyük. Ezután, mivel az *év* a `Film` és a `SzerepelBenne` relációknak egyaránt attribútuma, a kiválasztást az összekapcsolás csomópont *mindkét* gyereke felé levihetjük. Az eredményül kapott logikai lekérdezéstervet a 7.7. ábra mutatja. Az új terv valószínűleg javulást jelent, hiszen a `SzerepelBenne` reláció méretét csökkentjük, mielőtt összekapcsoljuk az 1996-os filmekkel. □



7.7. ábra. *A lekérdezésterv javítása a kiválasztás felfelé és lefelé tologatásával*

7.2.4. Vetítéssel kapcsolatos szabályok

A kiválasztáshoz hasonlóan a vetítéseket is „tolhatjuk lefelé", át más operátorokon. A vetítések tologatása abban különbözik a kiválasztások tologatásától, hogy amikor vetítést tolunk, akkor a vetítés általában ott is megmarad, ahol van. Másképpen szólva, vetítés „tolása" valójában egy új vetítés bevezetését jelenti valahol a létező vetítés alatt.

A vetítések eltolása hasznos ugyan, de általában nem annyira, mint a kiválasztás tologatása. Ennek az az oka, hogy míg a kiválasztás gyakran nagymértékben csökkenti egy reláció méretét, a vetítés során a sorok száma ugyanaz marad, csak a sorok hossza csökken. Sőt, amint a 6.1.3. részben megfigyeltük, a vetítés néha növeli a sorok hosszát.

Hogy a transzformációkat a vetítés általános alakjának segítségével írhassuk le, be kell vezetnünk néhány terminológiát. Nézzünk egy $E \rightarrow x$ kifejezést egy vetítési listából, ahol E vagy egy attribútum, vagy egy attribútumokat és konstansokat tartalmazó kifejezés. Azt mondjuk, hogy az E-ben előforduló összes attribútum a vetítés *bemeneti* attribútuma, az x pedig egy *kimeneti* attribútum. Ha a kifejezés egyetlen attribútum, akkor az egyben bemeneti és kimeneti attribútum is. Láthatjuk, hogy a kifejezés nem lehet más, mint egyetlen attribútum nyíl nélkül vagy átnevezés, így az összes esetet lefedtük.

Ha a vetítési lista csak attribútumokból áll, tehát nincs átnevezés vagy olyan kifejezés, ami más, mint egyetlen attribútum, akkor *egyszerű* vetítésről beszélünk. A klaszszikus relációs algebrában minden vetítés egyszerű.

7.9. példa: A $\pi_{a, b, c}(R)$ vetítés egyszerű; a, b és c egyszerre bemeneti attribútumok és kimeneti attribútumok. A $\pi_{a+b \to x, c}(R)$ vetítés viszont nem egyszerű. Ennek bemeneti attribútumai az a, b és a c, kimeneti attribútumai pedig az x és a c. □

A vetítésre vonatkozó szabályok mögött az alábbi alapelv húzódik meg:

- A kifejezésfában bárhol bevezethetünk egy vetítést mindaddig, amíg az csakis olyan attribútumokat tüntet el, amelyeket egyetlen fentebb elhelyezkedő operátor sem használ, valamint a teljes kifejezés eredményében sem szerepelnek.

A szabályok legegyszerűbb alakjaiban a bevezetett vetítések mind egyszerűek:

- $\pi_L(R \bowtie S) = \pi_L(\pi_M(R) \bowtie \pi_N(S))$, ahol M az R azon attribútumainak listája, amelyek vagy összekapcsolási attribútumok (az R és S sémájában egyaránt szerepelnek), vagy bemeneti attribútumai az L-nek, és N az S azon attribútumainak listája, amelyek vagy összekapcsolási attribútumok, vagy bemeneti attribútumai az L-nek.
- $\pi_L(R \underset{C}{\bowtie} S) = \pi_L(\pi_M(R) \underset{C}{\bowtie} \pi_N(S))$, ahol M az R azon attribútumainak listája, amelyek vagy összekapcsolási attribútumok (vagyis a C feltételben előfordulnak), vagy bemeneti attribútumai az L-nek, és N az S azon attribútumainak listája, amelyek vagy összekapcsolási attribútumok, vagy bemeneti attribútumai az L-nek.
- $\pi_L(R \times S) = \pi_L(\pi_M(R) \times \pi_N(S))$, ahol M az R, illetve N az S azon attribútumainak listája, amelyek bemeneti attribútumai az L-nek.

7.10. példa: Legyen $R(a, b, c)$ és $S(c, d, e)$ két reláció. Vegyük a következő kifejezést: $\pi_{a+e \to x, b \to y}(R \bowtie S)$. A vetítés bemeneti attribútumai a, b és e, valamint c az egyetlen összekapcsolási attribútum. A vetítések összekapcsolás alá történő eltolásának szabályát alkalmazva ekvivalens kifejezést kapunk:

$$\pi_{a+e \to x, b \to y}(\pi_{a, b, c}(R) \bowtie \pi_{c, e}(S))$$

Vegyük észre, hogy a $\pi_{a, b, c}(R)$ egy triviális vetítés, ami az R összes attribútumára vetít. Ez a vetítés így kiküszöbölhető, ami egy harmadik ekvivalens kifejezést eredményez:

$$\pi_{a+e \to x, b \to y}(R \bowtie \pi_{c, e}(S))$$

Az egyetlen változás tehát az eredetihez képest az, hogy az összekapcsolás előtt az S-ből elhagyjuk a d attribútumot. □

Egy vetítést teljesen végrehajthatunk egy multihalmaz-egyesítés előtt:

- $\pi_L(R \cup_M S) = \pi_L(R) \cup_M \pi_L(S)$

Nem vihetők azonban a vetítések sem a halmazegyesítések, sem a metszet és különbség halmaz vagy multihalmaz változatai elé.

7.11. példa: Legyenek $R(a, b)$ a $\{(1, 2)\}$, $S(a, b)$ pedig a $\{(1, 3)\}$ relációk. Ekkor $\pi_a(R \cap S) = \pi_a(\emptyset) = \emptyset$. Ugyanakkor $\pi_a(R) \cap \pi_a(S) = \{(1)\} \cap \{(1)\} = \{(1)\}$. □

Ha a vetítés számításokat tartalmaz, és a vetítési lista valamely kifejezésének bemeneti attribútumai teljes egészében egy, a vetítés alatt elhelyezkedő összekapcsolás vagy szorzat egyik argumentumához tartoznak, akkor megvan az a lehetőségünk, bár nem kötelező, hogy az adott számítást közvetlenül azon az argumentumon végezzük el. Egy példával szemléltetjük ezt az esetet.

7.12. példa: Legyen ismét az $R(a, b, c)$ és $S(c, d, e)$ két reláció, és tekintsük a következő összekapcsolást és vetítést: $\pi_{a+b \to x, d+e \to y}(R \bowtie S)$. Az $a + b$ összeadást és annak x-re történő átnevezését közvetlenül az R relációhoz vihetjük, és ugyanezt tehetjük a $d + e$ összeggel az S vonatkozásában. Az így kapott ekvivalens kifejezés: $\pi_{x, y}(\pi_{a+b \to x, c}(R) \bowtie \pi_{d+e \to y, c}(S))$.

Speciálisan kell kezelni azt az esetet, ha x vagy y megegyezik c-vel. Ekkor nem nevezhetnénk át az összeget c-re, mert egy relációnak nem lehet két attribútuma ugyanazzal a c névvel. Be kellene vezetni egy ideiglenes nevet, és az összekapcsolás fölött végre kellene hajtani egy további átnevezést. A $\pi_{a+b \to c, d+e \to y}(R \bowtie S)$ kifejezés például a következő kifejezéssé alakítható át: $\pi_{z \to c, y}(\pi_{a+b \to z, c}(R) \bowtie \pi_{d+e \to y, c}(S))$. □

Egy vetítést be lehet iktatni egy kiválasztás alá is:

- $\pi_L(\sigma_C(R)) = \pi_L(\sigma_C(\pi_M(R)))$, ahol M azoknak az attribútumoknak a listája, amelyek vagy bementi attribútumai az L-nek, vagy szerepelnek a C feltételben.

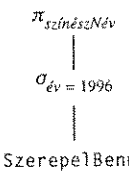
Csakúgy mint a 7.12. példában, lehetséges, hogy az L lista számításait inkább az M-ben végezzük el, feltéve, hogy a C feltétel nem igényli az L azon attribútumait, amelyek valamely számításban érintettek.

Gyakran akkor is lejjebb akarjuk vinni a vetítéseket a kifejezésfában, ha fent ott kell hagyni egy másik vetítést, mert a vetítések általában csökkentik a sorok méretet, és így egy köztes reláció által elfoglalt blokkok számát. Vigyáznunk kell azonban, amikor ezt tesszük, mert vannak tipikus példák, amikor egy vetítés levitele időbe kerül.

7.13. példa: Vegyük azt a `SzerepelBenne(filmCím, év, színészNév)` relációra vonatkozó lekérdezést, amely az 1996-ban dolgozó színészeket keresi:

```
SELECT színészNév
FROM SzerepelBenne
WHERE év = 1996;
```

A 7.8. ábra mutatja ennek a lekérdezésnek a közvetlen átalakítását logikai lekérdezéstervvé.

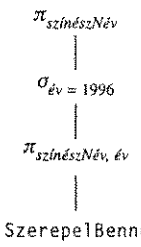


7.8. ábra. *Logikai lekérdezésterv a 7.13. példában szereplő lekérdezéshez*

A kiválasztás alá beilleszthetünk egy vetítést két attribútummal:

1. `színészNév`, ugyanis ez az attribútum kell az eredményhez, és
2. `év`, mert ez az attribútum szükséges a kiválasztási feltételhez.

Az eredmény a 7.9. ábrán látható.



7.9. ábra. *Vetítés bevezetésének az eredménye*

Ha a `SzerepelBenne` nem tárolt reláció lenne, hanem valamilyen művelet – mint például összekapcsolás – által létrehozott reláció, akkor lenne értelme a 7.9. ábra szerinti tervnek. „Futószalagosíthatjuk" a vetítést (lásd a 7.7.3. részt), amint az összekapcsolás sorait előállítjuk, egyszerűen elhagyva a nem használt `filmCím` attribútumot.

A mi esetünkben azonban a `SzerepelBenne` egy tárolt reláció. Az alsó vetítés a 7.9. ábrán valójában nagy időpocsékolást jelent, különösen akkor, ha létezik index az `év` attribútumra. Ekkor a 7.8. ábra logikai lekérdezéstervén alapuló fizikai lekérdezésterv először az indexet használná az olyan `SzerepelBenne` sorok megtalálására, ahol az `év` 1996, ami feltehetően a soroknak csak egy kis hányadát jelenti. Ha a vetítést végezzük el először a 7.9. ábrának megfelelően, akkor a `SzerepelBenne` reláció minden sorát be kell olvasni és vetítetni kell.

Hogy a dolgok még rosszabbul nézzenek ki, az `év`-hez tartozó index valószínűleg nem használható a vetített $\pi_{színészNév,\ év}$(`SzerepelBenne`) relációhoz, így a kiválasztásnak a vetítés eredményeként megkapott összes sort végig kell olvasnia. □

7.2.5. Összekapcsolásra és szorzatra vonatkozó szabályok

A 7.2.1. részben már láttunk több szabályt az összekapcsolással és a szorzattal kapcsolatban: azok kommutatív és asszociatív szabályait. Van azonban néhány további szabály, amelyek közvetlenül az összekapcsolás definíciójából következnek.

- $R \underset{C}{\bowtie} S = \sigma_C(R \times S)$
- $R \bowtie S = \pi_L(\sigma_C(R \times S))$, ahol C az a feltétel, amely az R-ből és S-ből származó azonos nevű attribútumpárok egyenlőségét vizsgálja, az L pedig olyan lista, amely tartalmazza az összes egyenlővé tett attribútumpár egyikét, valamint az R és S minden maradék attribútumát.

Ezeknek a szabályoknak a használatát a 6.1.5. részben adott 6.5. és 6.6. példák szemléltették. A gyakorlatban rendszerint jobbról balra alkalmazzuk ezeket a szabályokat, vagyis egy szorzatot követő kiválasztást azonosítunk összekapcsolásként. Ennek az az oka, hogy az összekapcsolások kiszámításához használt algoritmusok általában sokkal gyorsabbak, mint az olyan algoritmusok, amelyek egy szorzatot és a szorzat (nagyon nagy méretű) eredményére alkalmazott kiválasztást számítanak ki.

7.2.6. Ismétlődések elhagyására vonatkozó szabályok

Az ismétlődéseket eltávolító δ operátort sok operátoron keresztül lehet tolni, de nem mindegyiken. A δ lefelé történő mozgatása a fában csökkenti a köztes relációk méretét, így kifizetődő lehet. Sőt a δ néha olyan helyre vihető, ahol egyszerűen elhagyható, mert olyan relációra vonatkozik, amelyről tudni lehet, hogy nem tartalmaz ismétlődéseket:

- $\delta(R) = R$, ha R-ben nincsenek ismétlődések. Ilyen fontos esetekről van szó például, ha R a következő:

  a) Egy tárolt reláció, amelyhez elsődleges kulcsot deklaráltunk.
  b) Egy γ művelet eredményeként kapott reláció, mivel egy csoportosítás eredménye egy ismétlődések nélküli reláció.

Az alábbi néhány szabály a δ operátort más operátorokon „tolja" keresztül:

- $\delta(R \times S) = \delta(R) \times \delta(S)$
- $\delta(R \bowtie S) = \delta(R) \bowtie \delta(S)$
- $\delta(R \underset{C}{\bowtie} S) = \delta(R) \underset{C}{\bowtie} \delta(S)$
- $\delta(\sigma_C(R)) = \sigma_C(\delta(R))$

A δ odavihető egy metszet egyik vagy mindkét argumentumához is:

- $\delta(R \cap_M S) = \delta(R) \cap_M S = R \cap_M \delta(S) = \delta(R) \cap_M \delta(S)$

Ugyanakkor viszont a δ általában nem vihető át a $\cup_M$, $-_M$ vagy π operátorokon.

7.14. példa: Legyen az R reláció olyan, amelyben a t sor két példányban szerepel, az S pedig olyan, amelyben a t sor egy példányban szerepel. Ekkor a $\delta(R \cup_M S)$ egy példányát, míg a $\delta(R) \cap_M \delta(S)$ két példányát tartalmazza a t sornak. Továbbá, a $\delta(R -_M S)$ tartalmazza a t egy példányát, míg a $\delta(R) -_M \delta(S)$ nem tartalmazza a t sort.

Vegyük most azt a $T(a, b)$ relációt, amely az (1, 2) és (1, 3) sorok egy-egy példányát tartalmazza, és mást nem. Ekkor a $\delta(\pi_a(R))$ eredményében az (1) sor egyszer szerepel, míg a $\pi_a(\delta(R))$ eredményében az (1) sor kétszer fordul elő. □

Végül megjegyezzük, hogy a δ felcserélésének az $\cup_H$, $\cap_H$ és $-_H$ operátorokkal nincs értelme. Ehelyett a δ elhagyható, mivel halmazok előállításakor garantáltan nem kapunk ismétlődéseket. Például:

- $\delta(R \cup_H S) = R \cup_H S$

Vegyük észre azonban, hogy az $\cup_H$ vagy a többi halmazművelet megvalósítása magában foglalja az ismétlődések eltüntetésének folyamatát, ami egyenértékű a δ alkalmazásával; lásd például a 6.3.3. részt.

7.2.7. Csoportosításra és összesítésre vonatkozó szabályok

Ha megnézzük a γ operátort, akkor azt találjuk, hogy sok transzformáció alkalmazhatósága a használt összesítő operátor részleteitől függ. Emiatt nem állíthatunk fel szabályokat olyan általánosságban, mint ahogyan a többi operátor esetében tettük. Kivételt képez az a 7.2.6. részben már említett eset, amikor egy γ elnyel egy δ-t. Egész pontosan:

- $\delta(\gamma_L(R)) = \gamma_L(R)$

Egy másik általános szabály az, hogy ha úgy kívánjuk, akkor a γ operátor alkalmazása előtt az argumentumban nem használt attribútumokat elhagyhatjuk egy vetítés segítségével. Ez a szabály így fogalmazható meg:

- $\gamma_L(R) = \gamma_L(\pi_M(R))$, ahol M az R azon attribútumainak listája, amelyek L-ben előfordulnak.

Annak oka, hogy más transzformációk a γ-ban szereplő összesítésektől függnek, a következőkben áll. Bizonyos összesítéseket – ilyen a `MIN` és a `MAX` – nem befolyásol az ismétlődések jelenléte vagy hiánya. A többi összesítés – `SUM`, `COUNT` és `AVG` – viszont általában más értéket produkál, ha az összesítés alkalmazása előtt megszüntetjük az ismétlődéseket.

Egy γ_L operátort *ismétlődésérzéketlennek* nevezünk, ha L-ben csak `MIN` és/vagy `MAX` összesítések szerepelnek. Ezek után:

- $\gamma_L(R) = \gamma_L(\delta(R))$, feltéve hogy γ_L ismétlődésérzéketlen.

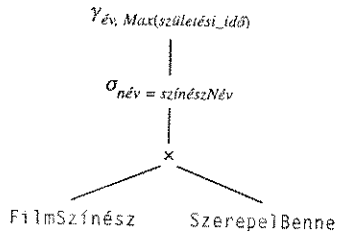
7.15. példa: Tegyük fel, hogy adottak a

```
FilmSzínész(név, cím, nem, születési_idő)
SzerepelBenne(filmCím, év, színészNév)
```

relációk, és hogy minden évhez meg akarjuk keresni az adott évben valamilyen filmben szereplő legfiatalabb színész születési idejét. Ez az alábbi lekérdezéssel fejezhető ki:

```
SELECT év, MAX(születési_idő)
FROM FilmSzínész, SzerepelBenne
WHERE név = színészNév
GROUP BY év;
```

A közvetlenül a lekérdezésből kapott kiindulási logikai lekérdezéstervet a 7.10. ábrán láthatjuk. A `FROM` listát egy szorzat, a `WHERE` záradékot pedig egy fölötte lévő kiválasztás fejezi ki. A csoportosítást és összesítést az ezek fölött elhelyezkedő γ operátor fejezi ki.

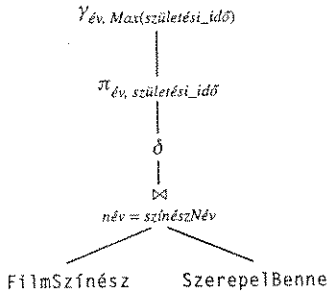


7.10. ábra. *Kiindulási logikai lekérdezésterv a 7.15. példa lekérdezéséhez*

A 7.10. ábrán elvégezhető több átalakítás is:

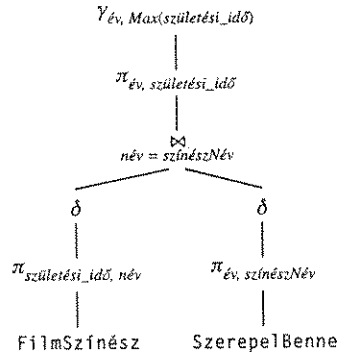
1. A kiválasztás és a szorzat összevonása egy egyenlőségen alapuló összekapcsolássá.
2. Egy δ beillesztése a γ alá, mivel γ ismétlődésérzéketlen.
3. Egy olyan π vetítés beillesztése a γ és az újonnan bevezetett δ közé, ami az `év`-re és a `születési_idő`-re, vagyis a γ szempontjából lényeges attribútumokra terjed ki.

Az eredményül kapott tervet a 7.11. ábra mutatja.



7.11. ábra. *Egy másik lekérdezésterv a 7.15. példa lekérdezéséhez*

Most levihetjük a δ-t az $\bowtie$ alá, és ez alá π-ket vezethetünk be, ha úgy kívánjuk. Az új lekérdezésterv a 7.12. ábrán található. Ha a `név` kulcsa a `FilmSzínész` relációnak, akkor az ehhez a relációhoz vezető ágon a δ elhagyható. □



7.12. ábra. *Egy harmadik lekérdezésterv a 7.15. példa lekérdezéséhez*

7.2.8. Feladatok

* **7.2.1. feladat:** Amikor egy kiválasztást egy bináris operátor mindkét argumentumához be lehet vinni, el kell döntenünk, hogy ezt megtegyük-e. Hogyan befolyásolná a döntésünket az, hogy az egyik argumentumhoz léteznek indexek? Tekintsük például a $\sigma_C(R \cap S)$ kifejezést, ahol az S-hez tartozik egy index.

7.2.2. feladat: Adjunk példákat az alábbiak bizonyítására:

* a) A vetítés nem vihető le a halmazegyesítés alá.
b) A vetítés nem vihető le a halmaz- vagy multihalmaz-különbség alá.
c) Az ismétlődések megszüntetése (δ) nem vihető le a vetítés alá.

d) Az ismétlődések megszüntetése (δ) nem vihető le a multihalmaz-egyesítés vagy különbség alá.

! **7.2.3. feladat:** Bizonyítsuk be, hogy egy vetítés minden esetben levihető egy multihalmaz-egyesítés mindkét ágán.

! **7.2.4. feladat:** A halmazokra vonatkozó szabályok némelyike érvényes multihalmazokra is, míg mások nem. Az alább felsorolt szabályok igazak halmazok esetén. Döntsük el, hogy multihalmazokra is igazak lesznek-e, avagy sem. Vagy bizonyítsuk be, hogy a szabály igaz multihalmazokra, vagy adjunk ellenpéldát.

* a) $R \cup R = R$ (az egyesítés idempotens)
b) $R \cap R = R$ (a metszet idempotens)
c) $R - R = \emptyset$
d) $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$ (az egyesítés metszet feletti disztributivitása)

! **7.2.5. feladat:** Multihalmazokra úgy definiálhatjuk a $\subseteq$ műveletet, hogy $R \subseteq S$ akkor és csak akkor, ha minden x esetén az x R-beli előfordulásainak száma kisebb vagy egyenlő az x S-beli előfordulásainak számával. Döntsük el, hogy a következő állítások (melyek igazak halmazokra) igazak-e multihalmazokra; bizonyítsuk be, vagy adjunk ellenpéldát.

a) Ha $R \subseteq S$, akkor $R \cup S = S$.
b) Ha $R \subseteq S$, akkor $R \cap S = R$.
c) Ha $R \subseteq S$ és $S \subseteq R$, akkor $R = S$.

7.2.6. feladat: Induljunk ki a $\pi_L(R(a, b, c) \bowtie S(b, c, d, e))$ kifejezésből, és tologassuk a vetítést lefelé a fában, amíg csak lehet. L az alábbi:

* a) $b + c \rightarrow x, c + d \rightarrow y$
b) $a, b, a + d \rightarrow z$

! **7.2.7. feladat:** A 7.15. példában említettük, hogy a bemutatott tervek egyike sem feltétlenül a legjobb terv. Tudna esetleg egy jobb tervet adni?

! **7.2.8. feladat:** Vegyük az $R(a, b)$ relációt érintő következő feltételezett egyenlőségeket. Döntsük el, hogy igazak-e; adjunk bizonyítást vagy ellenpéldát.

a) $\gamma_{MIN(a) \rightarrow y, x}(\gamma_{a, SUM(b) \rightarrow x}(R)) = \gamma_{y, SUM(b) \rightarrow x}(\gamma_{MIN(a) \rightarrow y, b}(R))$
b) $\gamma_{MIN(a) \rightarrow y, x}(\gamma_{a, MAX(b) \rightarrow x}(R)) = \gamma_{y, MAX(b) \rightarrow x}(\gamma_{MIN(a) \rightarrow y, b}(R))$

!! **7.2.9. feladat:** A 6.1.3. feladatban szereplő összekapcsolás jellegű operátorok az ismert szabályok némelyikének engedelmeskednek, másoknak viszont nem. Döntsük el, hogy a következő szabályok igazak-e, avagy sem. Bizonyítsuk be a szabályt, vagy adjunk ellenpéldát.

* a) $\sigma_C(R \ltimes S) = \sigma_C(R) \ltimes S$
* b) $\sigma_C(R \overset{\cdot}{\bowtie} S) = \sigma_C(R) \overset{\cdot}{\bowtie} S$
  c) $\sigma_C(R \overset{\cdot}{\bowtie}_L S) = \sigma_C(R) \overset{\cdot}{\bowtie}_L S$ (ahol C-ben az R minden attribútuma szerepel)
  d) $\sigma_C(R \overset{\cdot}{\bowtie}_L S) = R \overset{\cdot}{\bowtie}_L \sigma_C(S)$ (ahol C-ben az R minden attribútuma szerepel)
  e) $\pi_L(R \overline{\ltimes} S) = \pi_L(R) \overline{\ltimes} S$
* f) $(R \overset{\cdot}{\bowtie} S) \overset{\cdot}{\bowtie} T = R \overset{\cdot}{\bowtie} (S \overset{\cdot}{\bowtie} T)$
  g) $R \overset{\cdot}{\bowtie} S = S \overset{\cdot}{\bowtie} R$
  h) $R \overset{\cdot}{\bowtie}_L S = S \overset{\cdot}{\bowtie}_L R$
  i) $R \ltimes S = S \ltimes R$

7.3. Elemzőfák átalakítása logikai lekérdezéstervekké

Most pedig visszatérünk a lekérdezésfordító tárgyalásához. Miután a 7.1. részben létrehoztunk egy elemzőfát, a következő feladat az elemzőfa átalakítása a jónak vélt logikai lekérdezéstervvé. A 7.1. ábrának megfelelően ez két lépésből áll.

Az első lépés az elemzőfa csomópontjait és struktúráit helyettesíti (megfelelő csoportosításban) a relációs algebra egy vagy több operátora segítségével. Néhány ilyen szabályra javaslatot fogunk tenni, néhány továbbit pedig meghagyunk feladatnak. A második lépés veszi az első lépés által előállított relációs algebrai kifejezést, és azt egy olyan kifejezéssé alakítja, amely várhatóan a leghatékonyabb fizikai lekérdezéstervvé konvertálható.

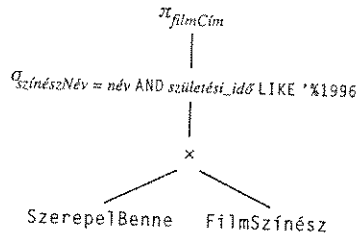
7.3.1. Átfordítás relációs algebrába

Most formalizmusok nélkül bevezetünk néhány olyan szabályt, amely az SQL-elemzőfáknak algebrai logikai lekérdezéstervvé történő transzformálásával kapcsolatos. Az első, talán legfontosabb szabály lehetővé teszi számunkra, hogy minden „egyszerű" „select-from-where" szerkezetet közvetlenül konvertáljunk a relációs algebrába. Informálisan ez a szabály így hangzik:

- Ha adott egy <Lekérdezés>, ami egy <SFW> struktúra, és a <Feltétel> ebben a struktúrában nem tartalmaz alkérdést, akkor a teljes struktúra – a select-lista, a from-lista és a feltétel – egy olyan relációs algebrai kifejezéssel helyettesíthető, amely alulról felfelé az alábbiakból áll:

  1. A <FromLista>-ban szereplő összes reláció szorzata, ami az alábbinak válik az argumentumává:
  2. Egy σ_C kiválasztás, ahol C a helyettesítés alatt álló struktúra <Feltétel> kifejezése, ami viszont az alábbinak lesz az argumentuma:
  3. Egy π_L vetítés, ahol L a <SelLista> attribútumlistája.

7.16. példa: Tekintsük a 7.5. ábrán látható elemzőfát. A fenti „select-from-where" transzformáció a 7.5. ábra teljes elemzőfájára alkalmazható. Vesszük a from-lista két relációjának – `SzerepelBenne` és `FilmSzínész` – szorzatát, kiválasztunk a <Feltétel>-ben gyökerező részfának megfelelő feltétel alapján, és vetítünk a `filmCím`-ből álló select-listára. Az eredményül kapott relációs algebrai kifejezés a 7.13. ábrán látható.



7.13. ábra. *Egy elemzőfa átalakítása algebrai kifejezésfává*

Ugyanez a transzformáció nem alkalmazható a 7.3. ábra külső szintű lekérdezésénél. Ez azért van így, mert a feltétel alkérdést tartalmaz. A 7.3.2. részben fogjuk megtárgyalni, hogy az alkérdéseket tartalmazó feltételeket hogyan lehet kezelni. Tanulmányozzuk a „Kiválasztási feltételek korlátozása" címet viselő bekeretezett részt is,

Kiválasztási feltételek korlátozása

Elcsodálkozhatunk azon, hogy miért nem engedjük meg, hogy egy σ_C kiválasztás C feltétele alkérdést tartalmazzon. A relációs algebrában az a szokás, hogy egy művelet *argumentumai* – a nem indexként szereplő elemek – olyan kifejezések, amelyek relációkat eredményeznek. Másrészt viszont, a paraméterek – az alsó indexként szereplő elemek – nem relációtípusúak. A σ_C-ben például a C paraméter egy logikai típusú feltétel, a π_L-ben pedig az L paraméter egy attribútumokból vagy formulákból álló lista.

Ha követjük ezt a hagyományt, akkor egy paraméter alkalmazható a reláció argumentum(ok) minden egyes sorára, bármilyen számítást jelent is ez. A paraméterek használatára vonatkozó korlátozás egyszerűbbé teszi a lekérdezésoptimalizálást. Tegyük fel most ellenben, hogy egy $\sigma_C(R)$ operátor C feltétele tartalmazhat egy alkérdést. Ekkor a C alkalmazása az R egyes soraira igényli az alkérdés kiszámítását. Kiszámítjuk ezt újra az R minden egyes soránál? Ez szükségtelenül drága volna, kivéve ha az alkérdés *korrelatív*, azaz annak értékei függnek valamitől, ami azon kívül definiált, mint ahogy például a 7.3. ábra alkérdése függ a `színészNév` értékétől. A legtöbb esetben még a korrelatív alkérdéseket is ki lehet értékelni anélkül, hogy azt minden sornál újra ki kellene számítani, feltéve, hogy a kiszámítást helyesen szervezzük meg.

ami arra ad magyarázatot, hogy miért teszünk különbséget az alkérdéseket tartalmazó, illetve nem tartalmazó feltételek között.

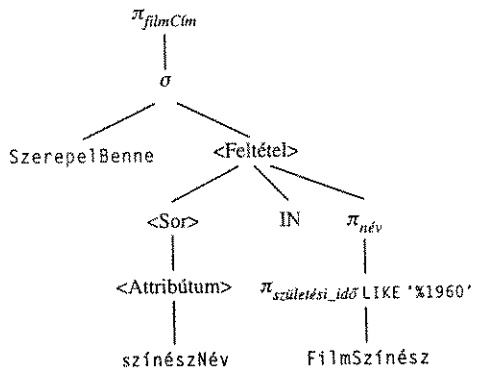
Alkalmazhatjuk azonban a „select-from-where" szabályt a 7.3. ábrán lévő alkérdésre. Az alkérdésből nyert relációs algebrai kifejezés: $\pi_{név}(\sigma_{születési\_idő\ \texttt{LIKE}\ \texttt{'\%1960'}}(\texttt{FilmSzínész}))$. □

7.3.2. Alkérdések eltávolítása feltételekből

Azokhoz az elemzőfákhoz, amelyekben van alkérdést tartalmazó <Feltétel>, bevezetünk egy közbeeső operátort, amely az elemzőfa szintaktikus kategóriái és a relációs algebrai operátorok között helyezkedik el, és relációkra vonatkozik. Ezt az operátort *kétargumentumú kiválasztásnak* nevezzük. A kétargumentumú kiválasztást a transzformált elemzőfában egy csomópont képviseli, amelynek címkéje σ, mégpedig paraméterek nélkül. E csomópont alatt elhelyezkedik egy bal oldali gyerek, amely azt az R relációt képviseli, amelyre a kiválasztás vonatkozik, valamint van egy jobb oldali gyerek, ami az R soraira vonatkozó feltételt megtestesítő kifejezés. Mindkét argumentum ábrázolható mint elemzőfa, mint kifejezésfa és mint a kettő keveréke.

7.17. példa: A 7.14. ábrán láthatjuk a 7.3. ábra elemzőfájának egy olyan átírását, amely használ egy kétargumentumú kiválasztást. Több transzformációt hajtottunk végre, mire a 7.3. ábrától eljutottunk a 7.14. ábrához:

1. A 7.3. ábrán szereplő alkérdést egy relációs algebrai kifejezéssel helyettesítettük a 7.16. példa végén mondottak alapján.
2. A „select-from-where" kifejezésekhez a 7.3.1. részben bevezetett szabálynak megfelelően helyettesítettük a külső szintű lekérdezést. A szükséges kiválasztást azonban egy kétargumentumú kiválasztás segítségével fejeztük ki, és nem a relációs algebra hagyományos σ operátorával. Következésképpen, az elemzőfa felső



7.14. ábra. *Egy σ kétargumentumú kifejezés, középúton az elemzőfa és a relációs algebra között*

<Feltétel> csomópontját nem helyettesítettük, az megmaradt mint a kiválasztás egyik argumentuma, de a hozzá tartozó kifejezés egy részét az (1) szerint helyettesítettük relációs algebrával.

Ez a fa további transzformációt igényel, ezt tárgyaljuk következőként. □

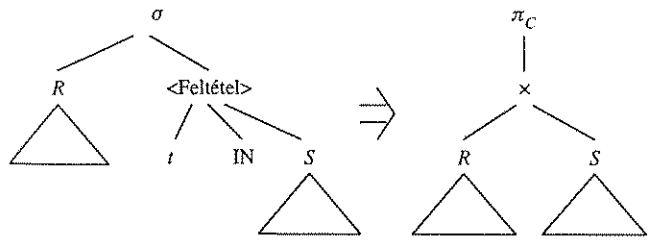
Szükségünk van szabályokra, amelyek lehetővé teszik, hogy egy kétargumentumú kiválasztást egy relációs algebrai egy argumentumú kiválasztással és egy másik relációs algebrai operátorral helyettesítsünk. A feltételek különböző formái külön szabályokat igényelhetnek. A legtöbb helyzetben a kétargumentumú kiválasztás eltávolítható, és tiszta relációs algebrai kifejezéshez juthatunk. Vannak azonban különleges esetek, amikor a kétargumentumú kiválasztást a helyén hagyjuk, és a logikai lekérdezésterv részének tekintjük.

Példaként megadjuk azt a szabályt, amelynek segítségével a 7.14. ábrán szereplő, `IN` operátort tartalmazó feltételt kezelhetjük. Vegyük észre, hogy az alkérdés a feltételben független, azaz a neki megfelelő reláció nem függ az éppen vizsgált sortól (elég egyszer kiszámítani). Az ilyen feltételeket elimináló szabály informálisan így fogalmazható meg:

- Tegyük fel, hogy van egy kétargumentumú kiválasztás, amelynek első argumentuma egy R relációt képvisel, a második argumentuma pedig egy t `IN` S, ahol az S kifejezés egy független alkérdés, t pedig az R bizonyos attribútumaiból összeállított sor. A fa az alábbi módon transzformálható:

  a) Helyettesítsük a <Feltételt>-t azzal a fával, ami nem más, mint az S kifejezés. Ha S-ben lehetnek ismétlődések, akkor egy δ műveletet is be kell iktatni az S-nek megfelelő kifejezés gyökerénél, hogy a kialakuló kifejezés ne állítson elő több sort, mint az eredeti lekérdezés.
  b) A kétargumentumú kiválasztást helyettesítsük egy σ_C egyargumentumú kiválasztással, ahol C az a feltétel, amelyet úgy kapunk, hogy a t sor minden egyes komponensét egyenlővé tesszük az S reláció neki megfelelő attribútumával.
  c) A σ_C argumentumaként az R és S szorzatát adjuk meg.

A 7.15. ábra szemlélteti ezt a transzformációt.



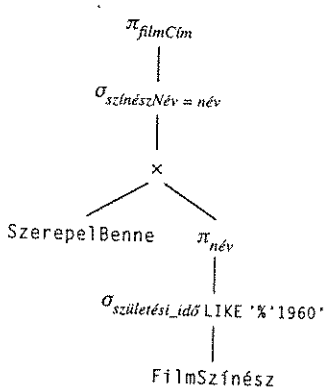
7.15. ábra. `IN`*-t tartalmazó feltétellel rendelkező kétargumentumú kiválasztást kezelő szabály*

7.18. példa: Vegyük a 7.14. ábrán található fát, és alkalmazzuk rá az `IN` feltételekhez megadott fenti szabályt. Ezen az ábrán az *R* a `SzerepelBenne` reláció, az *S* reláció pedig annak a relációs algebrai kifejezésnek az eredménye, amely a $\pi_{név}$ gyökerű részfából áll. A *t* sornak egy komponense van, nevezetesen a `színészNév` attribútum.

A kétargumentumú kiválasztás helyettesítője $\sigma_{színészNév = név}$, amelynek a *C* feltétele a *t* egyetlen tagját egyenlővé teszi az *S* lekérdezés eredményének attribútumával. A σ csomópont gyereke egy × csomópont, és az × csomópont argumentumai a `SzerepelBenne` címkéjű csomópont és az *S*-hez tartozó kifejezés gyökere. Mivel a `név` kulcsa a `FilmSzínész` relációnak, láthatjuk, hogy nincs szükség arra, hogy az *S*-hez tartozó kifejezésben egy ismétlődéseket megszüntető δ operátort bevezessünk. A 7.16. ábra mutatja az új kifejezést, amely teljesen a relációs algebrában van kifejezve, és ekvivalens a 7.13. ábra kifejezésével, habár a szerkezete igencsak különböző. □

Összetettebb az alkérdések relációs algebrába történő átfordítása, ha az alkérdés korrelatív. Mivel a korrelatív alkérdések magukban foglalnak rajtuk kívül definiált ismeretlen értékeket is, nem lehet őket külön átfordítani. Ehelyett az alkérdést úgy transzformáljuk, hogy az egy olyan relációt állít elő, amelyben bizonyos extra attribútumok is megjelennek – attribútumok, amelyeket később a kívül definiált attribútumokkal kell majd összehasonlítani. Az alkérdés attribútumait a külső attribútumokkal összevető feltételt erre a relációra alkalmazzuk, és az ezután már feleslegessé vált extra attribútumokat vetítés segítségével elhagyhatjuk. E folyamat során oda kell figyelnünk az ismétlődő sorok esetleges bevezetésére, amennyiben a lekérdezés a végén nem távolítja el az ismétlődéseket. A következő példa szemlélteti ezt a technikát.

7.19. példa: Használjuk a 7.1. példában bevezetett relációkat, és vegyük az alábbi lekérdezést: „Keressük meg az olyan filmeket, ahol a színészek átlagéletkora legfeljebb 40 év volt, amikor a film készült." A lekérdezés SQL-megfelelőjét a 7.17. ábra mutatja. Az egyszerűség kedvéért a `születési_idő`-t születési évnek vesszük, így ve-



7.16. ábra. *Az* `IN` *feltételre vonatkozó szabály alkalmazása*

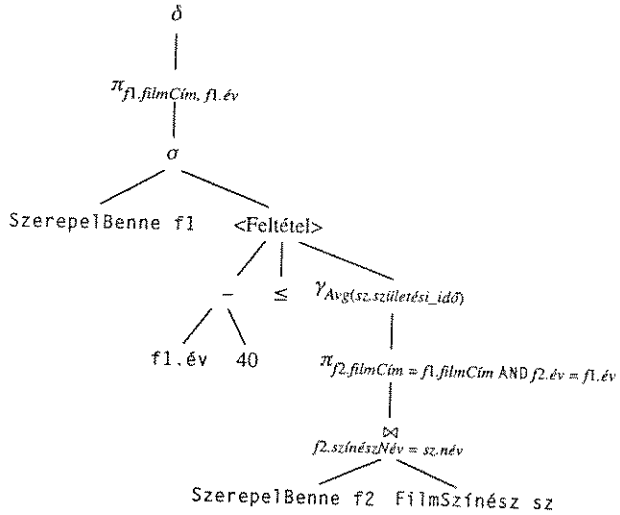
hetjük azok átlagát, amit aztán a `SzerepelBenne` reláció `év` attribútumával össze tudunk hasonlítani. A lekérdezést úgy fogalmaztuk meg, hogy mindhárom hivatkozott reláció rendelkezik a maga saját sorváltozójával, mutatva, hogy a különböző attribútumok honnan származnak.

```
SELECT DISTINCT f1.filmCím, f1.év
FROM SzerepelBenne f1
WHERE f1.év - 40 <= (
   SELECT AVG(születési_idő)
   FROM SzerepelBenne f2, FilmSzínész sz
   WHERE f2.színészNév = sz.név AND
      f1.filmCím = f2.filmCím AND
      f1.év = f2.év
   );
```

7.17. ábra. *Bizonyos átlagéletkorú színészekkel készült filmek megkeresése*

A 7.18. ábrán a lekérdezés elemzésének és a relációs algebrába való részleges átfordítás végrehajtásának az eredménye látható. Ebben a kezdeti transzformációban ketté választottuk az alkérdés `WHERE` záradékát, és az egyik részt úgy használtuk, hogy relációk szorzatából összekapcsolást készítettünk. Az `f1`, `f2` és `sz` sorváltozó neveket megtartottuk a fában is, hogy világos legyen az egyes attribútumok eredete. Megtehettük volna azt is, hogy vetítések segítségével átnevezzük az attribútumokat, de így az eredmény nehezebben lenne követhető.

A <Feltétel> csomópont és a kétargumentumú kiválasztás eltávolításához szükség

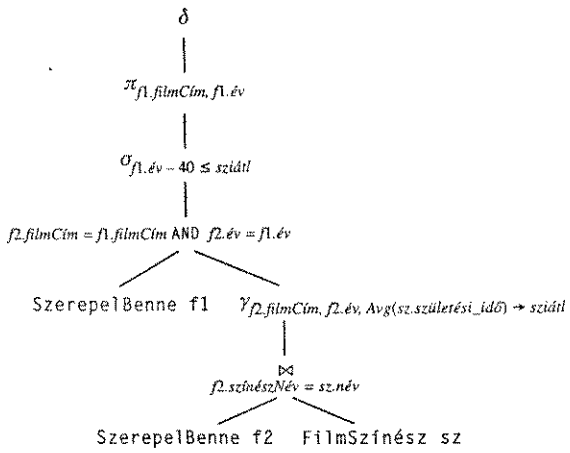


7.18. ábra. *Részlegesen transzformált elemzőfa a 7.17. ábra lekérdezéséhez*

van egy olyan kifejezésre, amely a <Feltétel> jobb oldali ágához tartozó relációt definiálja. Az alkérdés azonban korrelatív, és az `f1.filmCím` és `f1.év` attribútumok nem szerezhetők meg az alkérdésben említett relációkból, amelyek az `f2` sorváltozójú `SzerepelBenne` és a `FilmSzínész`. Emiatt a $\sigma_{f2.filmCím = f1.filCím \text{ AND } f2.év = f1.év}$ kiválasztást akkorra kell elhalasztani, amikor az alkérdés relációját már kombináltuk a `SzerepelBenne` relációnak a lekérdezés külső szintjén megjelenő példányával (az `f1` sorváltozójú példánnyal). A logikai lekérdezésterv ilyen átalakításához a γ operátort módosítani kell, mégpedig úgy, hogy a csoportosítás az `f2.filmCím` és `f2.év` attribútumok szerint történjen, így lesznek ugyanis elérhetők ezek az attribútumok a kiválasztáskor. Ennek hatásaként az alkérdéshez egy filmekből álló relációt számolunk ki, ahol minden egyes filmet annak címe és éve, valamint a filmben szereplő színészek születési évének átlaga képvisel.

A módosított γ operátor a 7.19. ábrán látható. A két csoportosítási attribútum bevezetésén túl az átlagot is átneveztük `sziátl`-ra (születési idők átlaga), hogy később hivatkozhassunk rá. A 7.19. ábra mutatja a relációs algebrába történő teljes átfordítást is. A γ fölött a külső lekérdezésből származó `SzerepelBenne` relációnak és az alkérdés eredményének összekapcsolása szerepel. Az alkérdésben lévő kiválasztást a `SzerepelBenne` relációnak és alkérdés relációjának szorzatára lehet alkalmazni, amit mi már egy théta-összekapcsolásként jelenítettünk meg, amivé ténylegesen azzá válna az algebrai szabályok alkalmazása után. A théta-összekapcsolás fölött egy további kiválasztás szerepel, ami a külső lekérdezés kiválasztásának felel meg, ahol a filmek gyártási évét hasonlítjuk össze a színészek születési évének átlagával. Az algebrai kifejezés a fa tetején úgy végződik, mint a 7.18. ábra kifejezése, vagyis a kívánt attribútumokra történő vetítéssel és az ismétlődések eltávolításával.

A 7.3.3. részben látni fogjuk, hogy egy lekérdezésoptimalizáló sokkal többet is tehet a lekérdezésterv javítása érdekében. A jelenlegi konkrét példánkban teljesül három

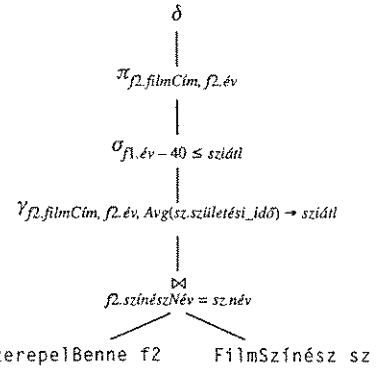


7.19. ábra. *A 7.18. ábra átalakítása egy logikai lekérdezéstervvé*

feltétel, amelyek lehetővé teszik, hogy a terven jelentősen javítsunk. Ezek a feltételek a következők:

1. Az ismétlődések megszüntetése a végén történik.
2. A vetítés a `SzerepelBenne f1` relációból kihagyja a színészek neveit.
3. A `SzerepelBenne f1` és a maradék kifejezés közti összekapcsolás egyenlővé teszi a `SzerepelBenne f1` és `SzerepelBenne f2` relációk `filmCím` és `év` attribútumait.

Mivel ezek a feltételek teljesülnek, az `f1.filmCím` és `f1.év` összes előfordulását helyettesíthetjük `f2.filmCím`-mel, illetve `f2.év`-vel. A 7.19. ábra felső összekapcsolása ezáltal feleslegessé válik, csakúgy mint a `SzerepelBenne f1` argumentum. Az így előálló logikai lekérdezésterv a 7.20. ábrán látható. □



7.20. ábra. *A 7.19. ábra egyszerűsítése*

7.3.3. Logikai lekérdezéstervek javítása

Amikor egy lekérdezést relációs algebrába átfordítunk, egy lehetséges logikai lekérdezéstervet kapunk. Ezután az következik, hogy a 7.2. részben felvázolt algebrai szabályok segítségével átírjuk a tervet. Egy másik megközelítés az lehetne, hogy több lekérdezéstervet generálunk, amelyek az operátorok különböző sorrendjének vagy kombinációjának felelnek meg. Ebben a könyvben abból a feltételezésből indulunk ki, hogy a lekérdezésátíró egyetlen logikai lekérdezéstervet választ ki, amelyet a „legjobbnak" vél, ami azt jelenti, hogy végül a legolcsóbb fizikai tervet eredményezi.

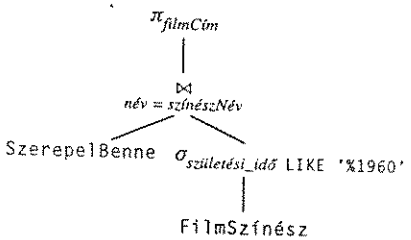
Nyitva hagyjuk viszont az „összekapcsolási sorrend" kérdését, így egy relációk összekapcsolását tartalmazó logikai lekérdezésterv úgy tekinthető, mint tervek egy családja, ami megfelel azoknak a különböző lehetőségeknek, ahogyan egy összekapcsolás sorba rendezhető és csoportosítható. Az összekapcsolási sorrend megválasztását a 7.6. részben tárgyaljuk. Ehhez hasonlóan, ha egy lekérdezésterv három vagy több

relációt tartalmaz a többi asszociatív és kommutatív operátor – mint például az egyesítés – argumentumaiként, akkor feltételezésünk szerint a logikai terv fizikai tervvé történő konvertálásakor átrendezés és átcsoportosítás megengedett. A sorrendet és a fizikai terv kiválasztását taglaló kérdéseket a 7.4. részben kezdjük tárgyalni.

A 7.2. részben számos olyan algebrai szabály szerepelt, amelyek feltehetően javítják a logikai tervet. Az optimalizálókban leggyakrabban használtak a következők:

- A kiválasztásokat mindaddig tologatjuk lefelé a fában, ameddig csak mehetnek. Ha egy kiválasztási feltétel több feltétel ÉS-elése, akkor a feltételt szétvághatjuk, és az egyes darabokat külön-külön vihetjük le a fában. Ez a stratégia valószínűleg a leghatékonyabb javítási technika, de nem árt felidézni a 7.2.3. részben mondottakat, ahol azt láttuk, hogy bizonyos körülmények között a kiválasztást először a fa tetejére kellett felvinni.
- Hasonlóképpen, a vetítéseket is tologathatjuk lefelé a fában, vagy új vetítéseket adhatunk hozzá. Csakúgy mint a kiválasztások esetében, a vetítések tologatásával is óvatosan kell bánni, ahogy ezt a 7.2.4. részben elmondtuk.
- Az ismétlődések megszüntetése néha eltávolítható vagy áthelyezhető alkalmasabb helyre a fában, a 7.2.6. részben mondottaknak megfelelően.
- Bizonyos kiválasztások kombinálhatók egy alatta elhelyezkedő szorzattal úgy, hogy a műveletpár egy egyenlőségen alapuló összekapcsolássá (equijoin) alakul, amit általában sokkal hatékonyabban lehet kiértékelni, mint a két műveletet különkülön. Ezeket a szabályokat a 7.2.5. részben tárgyaltuk.

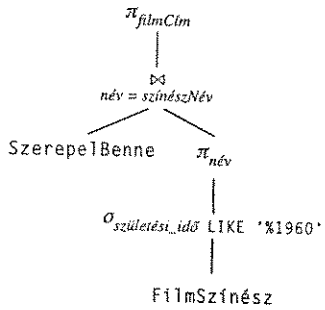
7.20. példa: Vegyük a 7.13. ábra lekérdezését. Először a kiválasztást vágjuk ketté a $\sigma_{színészNév = név}$ és a $\sigma_{születési\_idő\ \texttt{LIKE}\ \texttt{'1960'}}$ operátorokra. Az utóbbi levihető a fában, mivel az egyetlen érintett attribútum (`születési_idő`) a `FilmSzínész` relációból származik. Az első feltétel a szorzat mindkét tagjából tartalmaz egy-egy attribútumot, de azok egyenlővé vannak téve, ezért a szorzat és a kiválasztás együtt valójában egy összekapcsolásnak felel meg. Az átalakítások eredményét a 7.21. ábra mutatja. □



7.21. ábra. *Egy lekérdezés átírásának eredménye*

7.21. példa: A 7.16. ábrán szereplő kifejezésfán szintén lehet javítani. Hasznos transzformációt azonban csak a 7.20. példában is említett szabályok egyike jelent: egy kiválasztás és az alatta elhelyezkedő szorzat helyettesítése egy egyenlőségen alapuló összekapcsolással. A kapott lekérdezésterv a 7.22. ábrán látható, és ez majdnem

ugyanaz, mint a 7.21. ábra, de van benne egy további vetítés a `név` attribútumra vonatkozóan. Amikor az 1960-ban született színészek megkeresésére végrehajtunk egy kiválasztást a `FilmSzínész` reláción, elég, ha csak a `név` komponenst állítjuk elő, mert ez az, amit a későbbi műveletekben használunk. Vegyük észre, hogy a 7.22. ábra tervét a 7.21. ábra tervéből is megkaphatjuk úgy, hogy a vetítést bevisszük a fa jobb ágába (mialatt a $\pi_{filmCím}$ vetítést meghagyjuk a gyökérben). Ugyanakkor viszont a `SzerepelBenne` tárolt reláció vetítése költséges lehet, ha emiatt nem tudunk használni egy indexet a `SzerepelBenne` azon sorainak elérésekor, amelyekre az összekapcsolásnál szükség van. □



7.22. ábra. *A 7.16. ábra egy javítása*

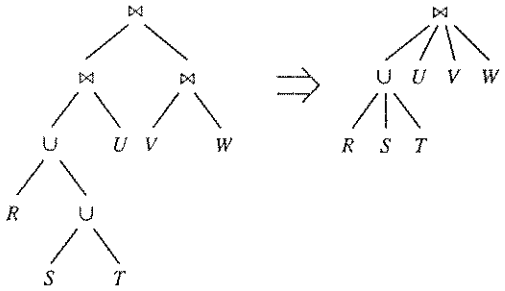
7.3.4. Asszociatív/kommutatív operátorok csoportosítása

A hagyományos elemzők nem állítanak elő olyan fákat, amelyek csomópontjai korlátlan számú gyerekkel rendelkezhetnek. Így az a normális, hogy az operátorok csak unáris vagy bináris formájukban jelennek meg. Asszociatív és kommutatív operátorok azonban felfoghatók úgy, mint amelyeknek tetszőleges számú operandusa van. Sőt, ha egy operátort, mint például az összekapcsolást, úgy tekintjük, mint egy sokoperandusú operátort, akkor lehetőséget kapunk az operandusok sorrendjének átrendezésére. Ez azt eredményezheti, hogy az új sorrendnek megfelelő bináris összekapcsolások sorozata kevesebb idő alatt hajtható végre, mint ha az összekapcsolásokat az elemzőfa által meghatározott sorrendben végeznénk el. A sokoperandusú összekapcsolások rendezését a 7.6. részben tárgyaljuk.

A végső logikai lekérdezésterv előállítása előtt tehát végrehajtunk egy utolsó lépést: ha van egy részfa, amelynek csomópontjaiban ugyanaz az asszociatív és kommutatív operátor szerepel, akkor az azonos operátort tartalmazó csomópontokat egyetlen sok gyerekkel rendelkező csomópontba csoportosítjuk. Emlékezzünk vissza, hogy a szokásos asszociatív/kommutatív operátorok a természetes összekapcsolás, egyesítés és metszet. Természetes összekapcsolások és théta-összekapcsolások is egyesíthetők egymással bizonyos körülmények között:

1. A természetes összekapcsolásokat olyan théta-összekapcsolásokkal kell helyettesíteni, amelyek egyenlővé teszik az azonos nevű attribútumokat.
2. Be kell iktatni egy vetítést az olyan attribútumok ismételt példányainak eltávolítására, amelyek a théta-összekapcsolássá vált természetes összekapcsolásban érintettek.
3. A théta-összekapcsolás feltételeinek asszociatívnak kell lenni. Emlékezzünk a 7.2.1. részben tárgyalt esetekre, ahol a théta-összekapcsolások nem asszociatívak.

Továbbá, a szorzatokat a természetes összekapcsolás speciális eseteként is felfoghatjuk, és egyesíthetjük azokat összekapcsolásokkal, ha a fában egymás szomszédjaiként helyezkednek el. A 7.23. ábra szemlélteti ezt a transzformációt, egy olyan helyzetben, ahol a logikai lekérdezéstervben egy két egyesítésből álló nyaláb, valamint egy három összekapcsolásból álló nyaláb szerepel. A betűk *R*-től *W*-ig kifejezéseket jelölnek, nem feltétlenül tárolt relációkat.



7.23. ábra. *A logikai lekérdezésterv előállításának utolsó lépése: asszociatív és kommutatív operátorok csoportosítása*

7.3.5. Feladatok

7.3.1. feladat: A következő kifejezésekben helyettesítsük a természetes összekapcsolásokat ekvivalens théta-összekapcsolásokkal és vetítésekkel. Döntsük el, hogy az eredményül kapott théta-összekapcsolások egy kommutatív és asszociatív csoportot alkotnak-e.

* a) $(R(a, b) \bowtie S(b, c)) \underset{S.c>T.c}{\bowtie} T(c, d)$

b) $(R(a, b) \bowtie S(b, c)) \bowtie (T(c, d) \bowtie U(d, e))$

c) $(R(a, b) \bowtie S(b, c)) \bowtie (T(c, d) \bowtie U(a, d))$

7.3.2. feladat: Konvertáljuk a 7.1.3.a) és b) feladatok elemzőfáit relációs algebrába. A b)-nél adjuk meg a kétargumentumú kiválasztást használó alakot, valamint annak egyargumentumú (szokásosan σ_C) kiválasztássá átalakított változatát.

! **7.3.3. feladat:** Adjunk egy-egy szabályt a következő alakú <Feltétel>-ek relációs algebrába történő átfordítására. Mindegyik feltételről feltesszük, hogy egy *R* relációra alkalmazzuk (egy kétargumentumú kiválasztás által). Feltételezhető továbbá, hogy az alkérdés nem korrelatív az *R* viszonylatában. Figyeljünk arra, hogy ne vezessünk be és ne szüntessünk meg ismétlődéseket az SQL hivatalos definíciójával szembenálló módon.

* a) `EXISTS(`<Lekérdezés>) alakú feltétel.
b) *a* = `ANY` <Lekérdezés> alakú feltétel, ahol *a* az *R* egy attribútuma.
c) *a* = `ALL` <Lekérdezés> alakú feltétel, ahol *a* az *R* egy attribútuma.

!! **7.3.4. feladat:** Tekintsük újra a 7.3.3. feladatot, megengedve ezúttal, hogy az alkérdés korrelatív legyen *R*-rel. Az egyszerűség kedvéért feltételezhetjük, hogy az alkérdés egy egyszerű „select-from-where" kifejezés, amely nem tartalmaz további alkérdéseket.

!! **7.3.5. feladat:** Hány különböző kifejezésfából adódhat a 7.23. ábra jobb oldali csoportosított fája? Emlékezzünk, hogy a csoportosítás után a gyerekek sorrendje nem feltétlenül tükrözi az eredeti kifejezésfában adott sorrendiséget.

7.4. Műveletek költségének becslése

Tegyük fel, hogy elemeztünk egy lekérdezést, és átalakítottuk egy logikai lekérdezéstervvé. Tegyük fel továbbá, hogy elvégeztük a kiválasztott transzformációkat, és megkaptuk a legjobbnak vélt logikai lekérdezéstervet. Következő lépésként a logikai tervet kell fizikai tervvé alakítani. Ez általában úgy történik, hogy sok különböző fizikai tervet tekintünk, amelyek a logikai tervből származnak, és kiértékeljük vagy becsüljük az ezekhez tartozó költségeket. E kiértékelés után, amit *költségalapú felsorolásnak* nevezünk, kiemeljük a legkisebb költségű fizikai tervet, és azt adjuk tovább a lekérdezés-végrehajtó motornak. Amikor az egy adott logikai lekérdezéstervből levezethető lehetséges fizikai terveket felsoroljuk, az egyes fizikai tervekhez az alábbiakat is kiválasztjuk:

1. Sorrendiség és csoportosítás az asszociatív-kommutatív operátorokra vonatkozóan, mint az összekapcsolás, egyesítés és metszet.
2. Algoritmus a logikai tervben szereplő minden egyes operátorhoz. Például, hogy beágyazott ciklusú összekapcsolást vagy tördelő összekapcsolást használjunk-e.
3. További műveletek – beolvasás, rendezés stb. –, amelyek a fizikai tervhez szükségesek, de amelyek a logikai tervben explicit módon nem voltak jelen.
4. Annak módja, ahogy egy operátor továbbadja az argumentumokat egy másiknak. Például, a közbülső eredmények lemezen történő tárolásával vagy iterátorokat használva, központi memóriapufferként továbbadva az argumentumot.

A továbbiakban megvizsgáljuk mindezeket a kérdéseket. Annak érdekében azonban, hogy az ezekkel a választásokkal kapcsolatban felmerülő kérdéseket megválaszolhassuk, meg kell értenünk, hogy a különböző fizikai tervek költsége mit is jelent. Egy terv pontos költségét nem tudhatjuk meg a terv végrehajtása nélkül, és egy lekérdezéshez nyilván nem akarunk egynél több tervet végrehajtani. Így a tervek költségének becslésére kényszerülünk, anélkül hogy azokat végrehajtanánk.

Mielőtt elkezdjük a fizikai tervek felsorolásának tárgyalását, az ilyen tervek költségbecslésének mikéntjére kell kitérni. Ezek a becslések az adatokkal kapcsolatos paraméterekre épülnek (lásd a „Jelölések áttekintése" c. részt), amelyeket vagy pontosan kiszámítunk az adatokból, vagy a „statisztikai gyűjtés" eljárással becsülünk, amit a 7.5.1. részben ismertetünk. Ha adottak a paramétereknek az értékei, akkor számos elfogadható becslés adható a relációméretekkel kapcsolatban, amelyekkel aztán egy teljes fizikai terv költsége becsülhető.

7.4.1. Közbülső relációk méretének becslése

A fizikai tervet úgy választjuk ki, hogy a lekérdezés kiértékelésének költsége minimális legyen. A legfőbb költségtényező rendszerint a lemez I/O-művelet (input/output = olvasás/írás), de néha fontos a processzoridő és – ha a lekérdezést párhuzamos gépen vagy több egymással összekötött gépen értékeljük ki – a kommunikációs idő is.

Amikor a logikai terv kifejezése több operátort tartalmaz, bizonyos dolgokat tudunk arról, hogy a közbülső relációk hogyan lesznek ábrázolva. Amíg ugyanis a kifejezés argumentumaiként szolgáló tárolt relációk többféleképpen lehetnek tárolva – nyaláboltan vagy nem, indexelve vagy anélkül –, a lekérdezés végrehajtása közben kiszámított valamely reláció, amelyet lemezen tárolunk, tárolható nyaláboltan úgy, hogy minél kevesebb blokkot foglaljon el. Továbbá, egy ilyen relációnak nem lesznek indexei, hacsak nem definiáljuk azokat explicit módon a fizikai lekérdezésterv részeként.

Ezek után azt mondhatjuk, hogy a köztes relációk kezeléséhez szükséges lemez I/O-műveletek száma nem függ mástól, mint a relációk méretétől. Ezt pedig úgy kapjuk meg, hogy a közbülső reláció sorainak számát megszorozzuk a sor tárolásához szükséges bájtok számával. Egy sor által elfoglalt bájtok száma levezethető a közbülső reláció attribútumaiból és azok típusaiból, így csak az marad rejtély, hogy a köztes reláció hány sort tartalmaz. Mivel általában nem tudjuk pontosan megmondani, hogy egy köztes relációnak hány sora lesz, néhány ésszerű szabályt fogunk bevezetni ezeknek a méreteknek a becslésére.

Ideális esetben egy közbülső relációban szereplő sorok számát becslő szabályokra igazak az alábbiak:

1. Elég pontos becslést adnak.
2. Könnyű kiszámolni.
3. Logikailag konzisztensek, azaz egy közbülső reláció méretére vonatkozó becslés nem függ a reláció kiszámításának módjától. Például több reláció összekapcsolására vonatkozó becslés nem függ a relációk összekapcsolásának sorrendjétől.

Jelölések áttekintése

Elevenítsük fel a 6.2.3. részben a relációk méretének jelölésére használt konvenciókat:

- $B(R)$ jelöli az R reláció összes sorának tárolásához szükséges blokkok számát.
- $T(R)$ az R reláció sorainak számát jelöli.
- $V(R, a)$ az R reláció a attribútumához tartozó *értékszámlálót* jelenti, vagyis azoknak a különböző értékeknek a számát, amelyek az R relációban az a attribútum értékeként előfordulnak. Valamint $V(R,[a_1, a_2, \ldots, a_n])$ jelöli azoknak a különböző értékeknek (értékkombinációknak) a számát, amelyek előfordulnak R-ben, amikor az $a_1, a_2, \ldots, a_n$ attribútumokat együtt tekintjük, azaz a $\pi_{a_1, a_2, \ldots, a_n}(R)$-ben szereplő különböző sorok számát jelenti.

Nincs általános egyetértés e feltételek teljesítésére vonatkozóan. Mi bemutatunk néhány egyszerű szabályt, amelyek a legtöbb helyzetben megfelelők. Szerencsére a méret becslésének nem az a célja, hogy a pontos méretet előre kiszámítsuk, hanem az, hogy hozzájáruljon egy fizikai terv kiválasztásához. Még egy pontatlan méretbecslési módszer is szolgálhat erre a célra, ha konzisztens módon hibázik, azaz, ha a méretbecslő a legjobb fizikai tervhez rendeli a legkisebb költséget, még ha annak a tervnek a tényleges költségéről az derül is ki, hogy különbözik az előre kiszámítottól.

7.4.2. Vetítés méretének becslése

A vetítés abban különbözik a többi művelettől, hogy az eredményének a mérete kiszámítható. Mivel egy vetítés minden argumentumsorhoz előállít egy eredménysort, a kimenet méretének változása csak a sorok hosszának megváltozásában jelentkezik. Emlékezzünk, hogy az itt használt vetítés operátor egy multihalmaz operátor, és nem távolítja el az ismétlődéseket. Ha egy vetítés során előálló ismétlődéseket meg akarjuk szüntetni, akkor a δ operátort kell utána alkalmazni.

Normális esetben vetítéskor a sorok összezsugorodnak, hiszen bizonyos komponenseket elhagyunk. A vetítésnek a 6.1.3. részben bevezetett általános formája azonban megengedi új komponensek létrehozását, mint attribútumok kombinációit. Vannak tehát esetek, amikor egy π operátor növeli a reláció méretét.

7.22. példa: Tegyük fel, hogy $R(a, b, c)$ egy reláció, ahol a és b négybájtos egészek, c pedig 100 bájtos karakterlánc. Mondjuk, hogy a sor fejlécek 12 bájtot igényelnek. Ekkor az R minden egyes sorának 120 bájtra van szüksége. Legyenek a blokkok 1024 bájt hosszúak, 24 bájt blokkfejlécekkel. Egyetlen blokkban így 8 sor fér el. Tegyük fel, hogy $T(R) = 10\,000$, vagyis hogy R 10 000 sort tartalmaz. Ekkor $B(R) = 1250$.

Legyen $S = \pi_{a+b, c}(R)$, azaz a-t és b-t az összegükkel helyettesítjük. Az S sorai 116

bájtot igényelnek: 12-t a fejlécnek, 4-et az összegnek és 100-at a karaktersorozatnak. Habár az S sorai valamivel kisebbek, mint az R sorai, még mindig csak 8 sort helyezhetünk be egy blokkba. Tehát: $T(S) = 10\,000$ és $B(S) = 1250$.

Legyen most $U = \pi_{a,\,b}(R)$, amikor is a karakterlánc komponenst elhagyjuk. Az U sorai csak 20 bájt hosszúak. $T(U)$ még mindig 10 000. Most azonban az U-nak 50 sorát pakolhatjuk egy blokkba, vagyis $B(U) = 200$. Ez a vetítés tehát a relációt mintegy 6-od részére zsugorítja. □

7.4.3. Kiválasztás méretének becslése

Amikor egy kiválasztást hajtunk végre, általában csökkentjük a sorok számát, de a sorok mérete ugyanaz marad. A kiválasztás legegyszerűbb esetében, amikor egy attribútumnak egy konstanssal való egyenlőségét vizsgáljuk, létezik egy könnyű módszer az eredmény méretének becslésére, feltéve, hogy tudjuk (vagy becsülni tudjuk) az attribútum által felvett különböző értékek számát. Legyen $S = \sigma_{A = c}(R)$, ahol A az R egy attribútuma és c egy konstans. Ekkor a következő becslést javasoljuk:

- $T(S) = T(R)/V(R, A)$

Ez a szabály biztosan igaz akkor, ha az A atribútum minden értéke egyenlő gyakorisággal fordul elő az adatbázisban. A fenti szabály azonban – az „A Zipfian-eloszlás" c. bekeretezett részben mondottaknak megfelelően – még akkor is a legjobb becslése az átlagnak, ha az A értékei nem mutatnak egyenletes eloszlást az adatbázisban. Elvárjuk viszont, hogy az A minden értéke egyforma valószínűséggel szerepeljen az A értékét meghatározó lekérdezésekben.

Problematikusabb a méret becslése, amikor a kiválasztás egyenlőtlenség-összehasonlítást tartalmaz, például ha $S = \sigma_{a < 10}(R)$. Azt gondolhatnánk, hogy az átlag tekintetében a sorok fele megfelelne az összehasonlításnak, a sorok fele nem, így $T(R)/2$ jó becslése lenne az S méretének. Egy érzés azonban azt súgja, hogy egy ilyen lekérdezés a lehetséges soroknak inkább csak egy kisebb hányadát adná vissza.[3] Egy olyan szabályt javasolunk, amely figyelembe veszi ezt a tendenciát, és azzal a feltételezéssel él, hogy egy tipikus vizsgálat, amely az egyenlőtlenséget vizsgálja, körülbelül a sorok egyharmadát adja vissza, nem a felét. Ha $S = \sigma_{a < c}(R)$, akkor $T(S)$-re a becslésünk:

- $T(S) = T(R)/3$

A „nem egyenlő" összehasonlítások ritkák. Ha azonban egy olyan kiválasztással találkozunk, mint például az $S = \sigma_{a \neq 10}(R)$, akkor javasoljuk annak feltételezését, hogy lényegében minden sor kielégíti majd ezt a feltételt. Vehetjük tehát becslésként a következőt: $T(S) = T(R)$. Egy másik becslés lehet a $T(S) = T(R)(V(R, a) - 1)/V(R, a)$,

[3] Ha például fizetésekről lennének adataink, azt kérdeznénk-e meg nagyobb valószínűséggel, hogy a fizetés *kisebb*, mint 500 000 Ft, vagy azt, hogy *nagyobb*, mint 500 000 Ft?

ami valamivel kevesebbet ad. Ez a megközelítés elismeri, hogy az R sorainak körülbelül $1/V(R, a)$ része elbukik a feltételen, mert azok a értéke egyenlő a konstanssal.

Amikor egy C kiválasztási feltétel több ÉS-sel összekötött egyenlőségvizsgálat vagy más összehasonlítás, akkor a $\sigma_C(R)$ kiválasztást úgy tekinthetjük, mint azoknak az egyszerű kiválasztásoknak egymás utáni alkalmazását, amelyek mindegyike a feltétel egy-egy részét ellenőrzi. Vegyük észre, hogy ezeknek a kiválasztásoknak a sorrendje nem számít. Ennek hatásaként az eredmény méretére vonatkozó becslés az lesz, hogy az eredeti reláció méretét megszorozzuk az egyes feltételekhez tartozó *szelektivitási* tényezőkkel. Ez a tényező 1/3 egyenlőtlenség esetén, 1 ≠ esetén, illetve $1/V(R, A)$ amikor a C feltételben egy A attribútumot hasonlítunk egy konstanshoz.

7.23. példa: Legyen $R(a, b, c)$ egy reláció és $S = \sigma_{a = 10 \text{ AND } b<20}(R)$. Legyen továbbá $T(R) = 10\,000$ és $V(R, a) = 50$. Ekkor a legjobb becslés a $T(S)$-re: $T(R)/(50 \times 3)$, azaz 67. Vagyis az R sorainak az 1/50 része éli túl az $a = 10$ szűrőt, és az 1/3 része éli túl a $b < 20$ szűrőt.

Egy érdekes speciális eset, ami romba dönti az analízisünket, amikor a feltétel ellentmondásos. Nézzük például az $S = \sigma_{a = 10 \text{ AND } a > 20}(R)$ kiválasztást. Ekkor a szabályunk alapján $T(S) = T(R)/3V(R, a)$, azaz 67 sor. Ugyanakkor viszont világos, hogy egyetlen sorra sem teljesülhet az $a = 10$ és az $a > 20$ feltételek mindegyike, tehát a helyes válasz: $T(S) = 0$. A logikai lekérdezésterv átírásakor a lekérdezésoptimalizáló sok speciális esetre vonatkozó szabályt figyelembe tud venni. A fenti esetben az optimalizáló alkalmazhat egy olyan szabályt, amely a kiválasztási feltételt HAMIS-nak találja, és az S-nek megfelelő kifejezést az üres halmazzal helyettesíti. □

Amikor egy kiválasztás VAGY-gyal kapcsolt feltételeket tartalmaz, mondjuk $S = \sigma_{C_1 \text{ OR } C_2}(R)$, kevesebb bizonyosságunk van az eredmény méretét illetően. Egy egyszerű feltételezés az, hogy egyetlen sorra sem teljesül mindkét feltétel, vagyis az eredmény mérete egyenlő az egyes feltételeket kielégítő sorok számának összegével. Ez a becslés általában túlbecslést jelent, és néha valóban ahhoz az abszurd következtetéshez vezethet el minket, hogy az S-ben több sor van, mint az eredeti R relációban. Egy másik egyszerű megközelítés lehet, hogy vesszük a minimumát az R méretének, és annak, amit a C_1-et, illetve a C_2-t kielégítő sorok számának összegeként kapunk.

Egy kevésbé egyszerű, de feltehetően pontosabb becslést kapunk az

$$S = \sigma_{C_1 \text{ OR } C_2}(R)$$

méretére, ha feltesszük, hogy C_1 és C_2 függetlenek. Ekkor, ha R-nek n sora van, amelyek közül m_1-re teljesül a C_1, és m_2-re teljesül a C_2, akkor az S-ben megjelenő sorok számára a következő becslést adhatjuk:

$$n(1 - (1 - m_1/n)(1 - m_2/n))$$

Itt az $1 - m_1/n$ egyenlő a soroknak a C_1-et nem teljesítő a hányadával, $1 - m_2/n$ pedig a soroknak a C_2-et nem teljesítő a hányadát jelenti. Ezek szorzata a R sorainak azon hányadát adja, amelyek *nincsenek* benne az S-ben, és ezt a szorzatotot 1-ből kivonva az S-ben szereplő hányadot kapjuk.

A Zipfian-eloszlás

Amikor feltételezzük, hogy az R reláció $V(R, a)$ sora közül egy fog kielégíteni egy $a = 10$ típusú feltételt, akkor azzal a hallgatólagos feltételezéssel élünk, hogy az a attribútum minden értéke egyforma valószínűséggel szerepel az R egy adott sorában. Azt is feltételezzük, hogy a 10 ezen értékek egyike, de ez egy ésszerű feltételezés, hiszen a legtöbbször olyan dolgokat keresünk egy adatbázisban, amelyek tényleg léteznek. Az a feltételezés azonban, hogy az értékek egyenletesen oszlanak el, többnyire nem tartható fenn, még megközelítőleg sem.

Sok attribútum olyan értékeket tartalmaz, amelyek *Zipfian-eloszlást* mutatnak, ahol az i-edik leggyakoribb érték gyakorisága az $1/\sqrt{i}$ -vel arányos. Ha például a leggyakoribb érték 1000-szer fordul elő, akkor a második leggyakoribb értéktől azt várjuk, hogy körülbelül $1000/\sqrt{2}$ -ször, azaz 707-szer szerepeljen, a harmadik leggyakoribb érték pedig körülbelül $1000/\sqrt{3}$ -szor, azaz 577-szer fordulna elő. Erről az eloszlásról az derült ki, hogy sokféle típusú adatnál fellelhető, jóllehet eredetileg az angol mondatokban előforduló szavak relatív gyakoriságának leírására használták. Az USA-ban például az államok népességei megközelítőleg a Zipfian-eloszlást követik, miszerint a második legnépesebb New York állam népessége körülbelül a 70%-a a legnépesebb Kalifornia állam népességének. Következésképpen, ha az `állam` egy amerikai embereket – mondjuk újság-előfizetőket – leíró reláció egy attribútuma lenne, akkor azt várnánk, hogy az `állam` értékei a Zipfian-eloszlásnak megfelelően oszlanak el, és nem egyenletesen.

Mindaddig, amíg a kiválasztási feltételben a konstanst véletlenszerűen választjuk meg, nem számít, hogy az érintett attribútum egyenletes, Zipfian- vagy más eloszlású-e, az eredmény halmaz *átlagos* mérete még mindig $T(R)/V(R, a)$ lesz. Ha azonban a konstansokat is Zipfian-eloszlásnak megfelelően választjuk, akkor azt várnánk, hogy a kiválasztott halmaz átlagos mérete valamivel nagyobb lesz, mint $T(R)/V(R, a)$.

7.24. példa: Tegyük fel, hogy az $R(a, b)$ relációnak $T(R) = 10\,000$ sora van, és legyen

$$S = \sigma_{a = 10 \text{ OR } b < 20}(R)$$

Legyen $V(R, a) = 50$. Ekkor az $a = 10$ feltételt kielégítő sorok számát, ami $T(R)/V(R, a)$, 200-ra becsüljük. A $b < 20$ feltételt kielégítő sorok számát $T(R)/3$-ra, vagyis 3333-ra becsüljük.

Az S méretére vonatkozó legegyszerűbb becslés ezek összege, azaz 3533. Az $a = 10$ és $b < 20$ feltételek függetlenségére építő bonyolultabb becslés a

$$10\,000(1 - (1 - 200/10\,000)(1 - 3333/10\,000))$$

értéket adja, azaz 3466-ot. A két becslés között kicsi az eltérés, így nagyon valószínűtlen, hogy az egyik választása a másikkal szemben változást jelentene a legjobb fizikai terv kiválasztásában. □

Az utolsó operátor, amely egy kiválasztási feltételben szerepelhet: a `NOT`. Ha egy R relációnak n számú sora van, akkor a `NOT` C feltételt kielégítő sorok becsült számát úgy kapjuk meg, hogy n-ből kivonjuk a C-t kielégítő sorok becsült számát.

7.4.4. Összekapcsolás méretének becslése

Csak a természetes összekapcsolást fogjuk vizsgálni. A többi összekapcsolás az alábbi elveknek megfelelően kezelhető:

1. Egy egyenlőség alapú összekapcsolás (equijoin) eredményében megjelenő sorok száma, miután a változó nevekben bekövetkező változásokkal elszámoltunk, pontosan úgy számítható ki, mint természetes összekapcsolás esetén. Ezt a pontot a 7.26. példa fogja szemléltetni.
2. Más théta-összekapcsolások úgy becsülhetők, mintha szorzatot követő kiválasztások volnának, figyelembe véve a következő további megjegyzéseket:
   a) Egy szorzat sorainak számát úgy kapjuk, hogy a szorzatban részt vevő relációk sorainak számait összeszorozzuk.
   b) Egy egyenlőséget vizsgáló összehasonlítást a természetes összekapcsoláshoz kidolgozott technika segítségével becsülhetünk.
   c) Egy két attribútum egyenlőtlenségét vizsgáló $R.a < S.b$ típusú összehasonlítás úgy kezelhető, mint egy $R.a < 10$ alakú összehasonlítás, amit a 7.4.3. részben tárgyaltunk. Vagyis feltehetjük, hogy ennek a feltételnek a szelektivitási tényezője 1/3 (ha úgy gondoljuk, hogy a feltétel inkább ritkán teljesül), vagy lehet 1/2 (ha nem élünk a feltételezéssel).

Első körben tételezzük fel, hogy két reláció természetes összekapcsolása csak két attribútum egyenlőségét tartalmazza. Ez azt jelenti, hogy az $R(X, Y) \bowtie S(Y, Z)$ összekapcsolást vizsgáljuk, de kezdetben feltesszük, hogy Y egyetlen attribútum, az X és Z viszont tetszőleges attribútum halmazokat jelölhetnek.

Az a probléma, hogy nem tudjuk, hogy az R és S Y értékei milyen viszonyban állnak egymással. Például:

1. A két relációban az Y értékek lehetnek diszjunkt halmazok, amikor is az összekapcsolás üres és $T(R \bowtie S) = 0$.
2. Az Y lehet az S kulcsa és egy idegen kulcs az R-ben. Ilyenkor az R minden egyes sora pontosan egy S-beli sorral kapcsolódik, így tehát $T(R \bowtie S) = T(R)$.
3. Lehet, hogy az R és S majdnem minden sorának ugyanaz az Y értéke, ekkor $T(R \bowtie S)$ körülbelül $T(R)T(S)$ lesz.

A következő két egyszerűsítő feltételezéssel fogunk élni, hogy a leggyakoribb esetekre koncentrálhassunk:

1. *Értékhalmazok tartalmazása.* Ha Y egy több relációban is szereplő attribútum, akkor ez az attribútum mindegyik relációban egy $y_1, y_2, y_3, \ldots$ rögzített értéklistának az elejéről kap értéket, és az összes érték ebből a prefixből származik. Következésképpen, ha R és S két reláció, amelyek tartalmazzák az Y attribútumot, és $V(R, Y) \leq V(S, Y)$, akkor az R minden Y értéke az S-nek Y értéke lesz.
2. *Értékhalmazok megőrzése.* Ha egy R relációt összekapcsolunk egy másik relációval, akkor egy A attribútum, amely nem összekapcsolási attribútum (azaz nem szerepel mindkét relációban), nem veszít el értékeket az értékeinek a lehetséges halmazából. Pontosabban szólva, ha A az R-nek attribútuma, de S-nek nem, akkor $V(R \bowtie S, A) = V(R, A)$. Megjegyezzük, hogy az R és az S összekapcsolásának sorrendje nem lényeges, tehát azt is mondhattuk volna, hogy $V(S \bowtie R, A) = V(R, A)$.

Nyilván előfordulhat, hogy az 1. előfeltevés, értékhalmazok tartalmazása, nem érvényes, de teljesül akkor, ha Y kulcs az S-ben, és idegen kulcs az R-ben. Sok más esetben is megközelítőleg igaz, hiszen intuitíve azt várjuk, hogy ha S-nek sok Y értéke van, akkor egy adott R-ben előforduló Y érték jó eséllyel szerepel S-ben.

A 2. feltételezés, értékhalmazok megőrzése, szintén sérülhet, de igaz a feltevés akkor, ha az $R \bowtie S$ összekapcsolási attribútuma kulcs az S-ben, és idegen kulcs az R-ben. Valójában csak akkor fordulhat elő, hogy a 2. előfeltevés nem teljesül, ha az R-ben „lógó sorok" vannak, vagyis olyan sorok, amelyek az S egyetlen sorával sem kapcsolódnak, de még az ilyen esetekben is érvényes lehet az előfeltétel.

E feltételezések mellett az $R(X, Y) \bowtie S(Y, Z)$ mérete a következőképpen becsülhető. Legyen $V(R, Y) \leq V(S, Y)$. Ekkor $1/V(S, Y)$ az esélye annak, hogy az R egy t sora az S egy adott sorával kapcsolódik. Mivel az S-nek $T(S)$ sora van, azoknak a soroknak a várható száma, amelyekkel t kapcsolódik: $T(S)/V(S, Y)$. Minthogy az R-nek $T(R)$ sora van, az $R \bowtie S$ becsült mérete $T(R)T(S)/V(S, Y)$. Ha $V(R, Y) \geq V(S, Y)$, akkor a szimmetria alapján kapott becslés: $T(R \bowtie S) = T(R)T(S)/V(R, Y)$. Általában a $V(R, Y)$ és a $V(S, Y)$ közül a nagyobbal osztunk, tehát:

- $T(R \bowtie S) = T(R)T(S)/\max(V(R, Y), V(S, Y))$

7.25. példa: Tekintsük a következő három relációt és azok lényeges statisztikáit:

$R(a, b)$	$S(b, c)$	$U(c, d)$
$T(R) = 1000$	$T(S) = 2000$	$T(U) = 5000$
$V(R, b) = 20$	$V(S, b) = 50$	
	$V(S, c) = 100$	$V(U, c) = 500$

Tegyük fel, hogy az $R \bowtie S \bowtie U$ természetes összekapcsolást akarjuk kiszámítani. Ennek egy lehetséges csoportosítási módja $(R \bowtie S) \bowtie U$. Az általunk adott becslés a $T(R \bowtie S)$-re: $T(R)T(S)/\max(V(R, b), V(S, b))$, ami $1000 \times 2000/50$, azaz 40 000.

Ezután jön az $R \bowtie S$ összekapcsolása az U-val. Az eredmény méretére vonatkozó becslésünk a következő: $T(R \bowtie S)T(U)/\max(V(R \bowtie S, c), V(U, c))$. Az előfeltevésünk alapján, miszerint az értékhalmazok megőrződnek, $V(R \bowtie S, c)$ ugyanaz, mint $V(S, c)$-vel, azaz 100, vagyis az összekapcsolás során a c attribútum egyetlen értéke sem tűnik el. Ez esetben az $R \bowtie S \bowtie U$ eredményében lévő sorok számára vonatkozó becslésként a $40\,000 \times 5000/\max(100, 500)$ értéket kapjuk, ami 400 000.

Az S és U összekapcsolásával is kezdhetnénk. Ekkor azt a becslést kapjuk, hogy $T(S \bowtie U) = T(S)T(U)/\max(V(S, c), V(U, c)) = 2000 \times 5000/500 = 20\,000$. Az előfeltevésünk alapján, miszerint az értékhalmazok megmaradnak, $V(S \bowtie U, b) = V(S, b) = 50$, így az eredmény méretének becslése $T(R)T(S \bowtie U)/\max(V(R, b), V(S \bowtie U, b))$, ami $1000 \times 20\,000/50$, azaz 400 000. □

Nem véletlen, hogy a 7.25. példában a $R \bowtie S \bowtie U$ méretére ugyanazt a becslést kapjuk, függetlenül attól, hogy az $R \bowtie S$ vagy az $S \bowtie U$ összekapcsolással kezdjük. Idézzük fel, hogy a 7.4.1. rész egyik kívánalma az, hogy egy kifejezés eredményére vonatkozó becslés ne függjön a kiértékelés sorrendjétől. Belátható, hogy a fenti két előfeltevésünk – értékhalmazok tartalmazása és megőrzése – garantálja, hogy egy természetes összekapcsolásra vonatkozó becslés ugyanaz lesz, függetlenül az összekapcsolások végrehajtási sorrendjétől.

7.4.5. Természetes összekapcsolás több összekapcsolási attribútummal

Most nézzük meg, hogy mi történik akkor, amikor az $R(X, Y) \bowtie S(Y, Z)$ összekapcsolásban az Y több attribútumot jelöl. Tegyük fel, hogy az $R(x, y_1, y_2) \bowtie S(y_1, y_2, z)$ összekapcsolást akarjuk végrehajtani. Vegyük az R egy r sorát. Annak valószínűsége, hogy r az S egy adott s sorával kapcsolódik, a következőképpen számolható ki.

Először is, mi a valószínűsége annak, hogy r és s megegyeznek az y_1 attribútumon? Tegyük fel, hogy $V(R, y_1) \geq V(S, y_1)$. Ekkor, az értékhalmazok tartalmazásáról szóló előfeltevés alapján, az s sor y_1 értéke biztosan az R-ben előforduló y_1 értékek valamelyike. Így annak esélye, hogy r-nek ugyanaz az y_1 értéke, mint s-nek: $1/V(R, y_1)$. Hasonlóképpen, ha $V(R, y_1) < V(S, y_1)$, akkor az r sor y_1 értéke szerepelni fog S-ben, és $1/V(S, y_1)$ a valószínűsége annak, hogy az r és az s y_1-értéke ugyanaz lesz. Általánosan mondhatjuk, hogy $1/\max(V(R, y_1), V(S, y_1))$ az y_1 érték egyezésének valószínűsége.

Hasonló gondolatmenet alapján állíthatjuk, hogy $1/\max(V(R, y_2), V(S, y_2))$ annak a valószínűsége, hogy r és s megegyeznek az y_2 vonatkozásában. Mivel az y_1 és az y_2 értékei függetlenek, annak a valószínűsége, hogy a sorok mind az y_1, mind az y_2 attribútumon megegyeznek, e két tört szorzata lesz. Az R és S soraiból képzett $T(R)T(S)$ darab sorpár közül az y_1 és y_2 attribútumokon egyező párok száma tehát:

$$\frac{T(R)T(S)}{\max(V(R, y_1), V(S, y_1))\max(V(R, y_2), V(S, y_2))}$$

A sorok száma nem elég

Habár a relációk méreteire vonatkozó vizsgálódásaink során az eredményben szereplő sorok számára összpontosítottunk, az egyes sorok méretét is számításba kell venni. Relációk összekapcsolása például az eredeti relációkban előforduló soroknál nagyobb sorokat állít elő. Példának okáért, az $R \bowtie S$ összekapcsolás, ahol mindkét reláció 1000 sort tartalmaz, adhat olyan eredményt, amely szintén 1000 sorból áll. Az eredmény azonban több blokkot foglalna el, mint az R vagy az S.

A 7.26. példa egy érdekes eset ezzel kapcsolatban. Egy théta-összekapcsoláskor kapott sorok számának becslésére használhatunk ugyan természetes összekapcsolásra vonatkozó technikákat, ahogy ott tettük is, de egy théta-összekapcsolás több komponensből álló sorokat állít elő, mint a neki megfelelő természetes összekapcsolás. A konkrét példa esetében, az $R(a, b, c) \underset{R.b = S.d \text{ AND } R.c = S.e}{\bowtie} S(d, e, f)$ théta-összekapcsolás hat komponensből álló sorokat állít elő, egyet minden attribútumhoz a-tól f-ig, míg az $R(a, b, c) \bowtie S(b, c, d)$ természetes összekapcsolás ugyanannyi számú sort állít elő, de minden sornak csak négy komponense van.

Általánosan a következő szabály használható egy természetes összekapcsolás méretének becslésére, ha a két relációnak tetszőleges számú közös attribútuma van:

- Az $R \bowtie S$ méretének becsült értékét úgy számítjuk ki, hogy a $T(R)$-t megszorozzuk $T(S)$-vel, majd elosztjuk a $V(R, y)$ és $V(S, y)$ közül a nagyobbikkal minden közös y attribútum esetén.

7.26. példa: A következő példa a fenti szabályt alkalmazza. Egyben azt is szemlélteti, hogy a természetes összekapcsolásra vonatkozó eddigi eredményeink az egyenlőséget használó összekapcsolásokra is érvényesek. Vegyük az alábbi összekapcsolást:

$$R(a, b, c) \underset{R.b = S.d \text{ AND } R.c = S.e}{\bowtie} S(d, e, f)$$

A méretekkel kapcsolatban a paraméterek a következők legyenek:

$R(a, b, c)$	$S(d, e, f)$
$T(R) = 1000$	$T(S) = 2000$
$V(R, b) = 20$	$V(S, d) = 50$
$V(R, c) = 100$	$V(S, e) = 50$

Ezt az összekapcsolást egy természetes összekapcsolásként is felfoghatjuk, ha az $R.b$ és $S.d$ attribútumokat, illetve az $R.c$ és $S.e$ attribútumokat ugyanazoknak tekintjük. Ekkor a fenti szabály alapján az $R \bowtie S$ méretének becsült értéke az 1000×2000 szorzat, osztva a 20 és az 50 közül a nagyobbal, és tovább osztva a 100 és az 50 közül a nagyobbal. Az összekapcsolás becsült mérete tehát $1000 \times 2000/(50 \times 100) = 400$ sor. □

7.27. példa: Nézzük meg újra a 7.25. példát, de tekintsük most a harmadik lehetséges összekapcsolási sorrendet, amikor is először az $R(a, b) \bowtie U(c, d)$ összekapcsolást vesszük. Ez az összekapcsolás valójában egy szorzat, és az eredményben előálló sorok száma $T(R)T(U) = 1000 \times 5000 = 5\ 000\ 000$. Vegyük észre, hogy a szorzatban a különböző b-k száma $V(R, b) = 20$, a különböző c-k száma pedig $V(U, c) = 500$.

Amikor ezt a szorzatot az $S(b, c)$-vel összekapcsoljuk, akkor összeszorozzuk a sorok számait, majd ezt elosztjuk $\max(V(R, b), V(S, b))$-vel és $\max(V(U, c), V(S, c))$-vel. Az így kapott mennyiség $2000 \times 5\ 000\ 000/(50 \times 500) = 400\ 000$. Vegyük észre, hogy az összekapcsolásnak ez a harmadik módja az eredmény méretére ugyanazt a becslést adja, mint amit a 7.25. példában kaptunk. □

7.4.6. Sok reláció összekapcsolása

Vizsgáljuk meg végül a természetes összekapcsolás általános esetét:

$$S = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$$

Tegyük fel, hogy az A attribútum az R_i-k közül k-ban fordul elő, és hogy ebben a k relációban az A értékek halmazainak méretei (elemszámai) – azaz a $V(R_i, A)$ különböző értékei $i = 1, 2, \ldots, k$ esetén $v_1 \leq v_2 \leq \ldots \leq v_k$, a legkisebbtől a legnagyobb felé haladó sorrendben. Tegyük fel, hogy mindegyik relációból veszünk egy sort. Mennyi a valószínűsége annak, hogy a kiválasztott sorok mindegyike megegyezik az A attribútumon?

Tekintsük azt a t_1 sort, amelyiket abból a relációból választottunk, amelyben az A értékek száma, v_1, a legkisebb. Az értékhalmazok tartalmazására vonatkozó előfeltevés alapján a v_1 számú érték mindegyike az A attribútummal rendelkező összes többi reláció A értékei között megtalálható. Nézzük azt a relációt, amelynek az A attribútuma v_i értékeket vesz fel. Az ebből a relációból vett t_i sor $1/v_i$ valószínűséggel egyezik t_1-gyel az A-n. Mivel ez a kijelentés minden $i = 2, 3, \ldots, k$ esetén igaz, annak valószínűsége, hogy mind a k sor megegyezik az A-n, ezek szorzata lesz, vagyis $1/v_2v_3\ldots v_k$. Ez az eredmény vezet el a tetszőleges összekapcsolás méretének becslésére vonatkozó szabályhoz:

- Vegyük először az egyes relációkban szereplő sorok számainak szorzatát. Ezután minden olyan A attribútumra, amely legalább kétszer előfordul, osszuk el az összes $V(R, A)$-val a legkisebb kivételével.

Az összekapcsolás után az A attribútum értékeként megmaradó értékek számát szintén becsülhetjük. Az értékhalmazok megőrzésére vonatkozó előfeltevés alapján ez ezen $V(R, A)$-k legkisebbike lesz.

7.28. példa: Vegyük az $R(a, b, c) \bowtie S(b, c, d) \bowtie U(b, e)$ összekapcsolást, azokat a lényeges statisztikákat feltételezve, amelyeket a 7.24. ábra mutat. Az eredmény becsléséhez először képezzük a relációk méreteinek szorzatát: $1000 \times 2000 \times 5000$. Ezután megnézzük, hogy mely attribútumok szerepelnek egynél többször, ezek a b, amely háromszor fordul elő, és a c, amely kétszer szerepel. Osztunk a $V(R, b)$, $V(S, b)$ és $V(U, b)$ közül a két legnagyobbal, ami 50 és 200. Végül osztunk a $V(R, c)$ és $V(S, c)$ közül a nagyobbal, ami 200. A kapott becslés tehát $1000 \times 2000 \times 5000/(50 \times 200 \times 200)$, azaz 5000.

Becsülhetjük az összekapcsolás eredményében az egyes attribútumokhoz tartozó értékhalmazok méretét is. Egy-egy ilyen becsült értéket úgy kapunk, hogy vesszük a különböző relációkban – ahol az attribútum megtalálható – az attribútumhoz tartozó értékszámlálók legkisebbikét. Az a, b, c, d és e attribútumok esetében ezek a számok sorra a következők: 100, 20, 100, 400 és 500. □

$R(a, b, c)$	$S(b, c, d)$	$U(b, e)$
$T(R) = 1000$	$T(S) = 2000$	$T(U) = 5000$
$V(R, a) = 100$		
$V(R, b) = 20$	$V(S, b) = 50$	$V(U, b) = 200$
$V(R, c) = 200$	$V(S, c) = 100$	
	$V(S, d) = 400$	
		$V(U, e) = 500$

7.24. ábra. *Paraméterek a 7.28. példához*

A két előfeltevésünkből – értékhalmazok tartalmazása és megőrzése – adódóan a becslés fent megadott szabálya rendelkezik egy meglepő és kellemes jellemzővel.

- Nem számít, hogy egy n relációt magában foglaló természetes összekapcsolást hogyan csoportosítunk és rendezünk, a becslésre vonatkozó szabályokat az egyes összekapcsolásokra egyenként alkalmazva az eredmény méretének ugyanazt a becslését kapjuk. Ez a becslés továbbá megegyezik azzal, amit akkor kapunk, ha az n reláció összekapcsolására, mint az egészre vonatkozó szabályt alkalmazzuk.

A 7.25. és 7.27 példák szemléltetik ezt a szabályt, amennyiben három reláció összekapcsolása történik háromféle csoportosításnak megfelelően, beleértve azt a csoportosítást is, ahol az „összekapcsolások" egyike ténylegesen egy szorzat.

7.4.7. Egyéb műveletek méretének becslése

Láttunk két műveletet, ahol az eredményül kapott sorok száma egy pontos fomulával leírható:

1. A vetítés nem változtatja meg egy relációban szereplő sorok számát.
2. A szorzat olyan eredményt állít elő, amelyben a sorok száma egyenlő az argumentum relációkban lévő sorok számának szorzatával.

Két további műveletre – a kiválasztásra és az összekapcsolásra – elég jó becslési technikákat dolgoztunk ki. A fennmaradó műveletek esetében azonban nem könnyű az eredmény méretének meghatározása. Sorra vesszük a többi relációs algebrai operátort is, és javaslatokat fogunk tenni arra, hogy ez a becslés hogyan végezhető el.

Egyesítés

Ha a multihalmaz-egyesítést vesszük, akkor a méret pontosan az argumentumok méretének összegével egyenlő. Egy halmazegyesítésnél a méret lehet olyan nagy, mint a méretek összege, vagy olyan kicsi, mint a két argumentum mérete közül a nagyobb. Azt ajánljuk, hogy válasszunk valamit a kettő között félúton, például az összeg és a nagyobb átlagát (ami ugyanaz, mint a nagyobb plusz a kisebb fele).

Miért független a sorrendtől az összekapcsolás méretének becslése?

Ezt az állítást az összekapcsolásban szereplő relációk számára vonatkozó indukcióval lehet formálisan bebizonyítani. Ezt a bizonyítást nem közöljük, de a vezérfonalat megadjuk ebben a bekeretezett részben. Tegyük fel, hogy összekapcsolunk néhány relációt, és az utolsó lépés a következő:

$$(R_1 \bowtie \ldots \bowtie R_n) \bowtie (S_1 \bowtie \ldots \bowtie S_m)$$

Feltehető, hogy nem számít, hogy az R-ek összekapcsolását hogyan vesszük. A méret becslése ennek az összekapcsolásnak az esetében az R-ek méreteinek szorzata, osztva az olyan attribútumokhoz tartozó értékszámlálókkal, a legkisebbet kivéve, amely attribútumok az R-ekben többször is előfordulnak. Továbbá, minden egyes attribútumhoz tartozó becsült értékszámláló (az eredményben) az R-ekben az attribútumhoz tartozó értékszámlálók közül a legkisebb. Hasonló kijelentéseket fogalmazhatunk meg az S-ekre vonatkozóan.

Amikor a két reláció összekapcsolásakor kapott eredmény méretének becslésére vonatkozó szabályt (lásd 7.4.4. részt) alkalmazzuk az R-ek összekapcsolásából, illetve az S-ek összekapcsolásából származó két relációra, a becslés a két becsült érték szorzata lesz, osztva minden olyan attribútumhoz tartozó értékszámlálók közül a nagyobbal, amelyek az R-ekben és S-ekben egyaránt szerepelnek. Ez a becslés biztosan tartalmaz egy tényezőt, ami az összes $R_1, \ldots, R_n$, $S_1, \ldots, S_m$ reláció mérete. Ráadásul a becsült értéknek egy olyan osztója lesz minden egyes attribútum esetén, ami *nem* a legkisebb értékszámláló az adott attribútumhoz. Ez az osztó vagy jelen van már az R-ekre vagy S-ekre adott becslésben, vagy az utolsó lépésben kerül be, mert annak A attribútuma mind az R-ben, mind az S-ben szerepel, és ő a nagyobb a két értékszámláló közül, amelyek egyike a $V(R_i, A)$-k legkisebbike, a másik pedig a $V(S_j, A)$-k legkisebbike.

Metszet

Az eredménynek lehet olyan kevés sora, mint például 0, vagy olyan sok sora, mint a két argumentum közül a kisebbnek, függetlenül attól, hogy halmaz- vagy multihalmazmetszetről van szó. Egy lehetséges megközelítés, hogy a szélsőségek közti átlagot vesszük, ami a kisebb felét jelenti.

Egy másik lehetőség, hogy felismerjük azt, hogy a metszet a természetes összekapcsolás egy speciális esete, és a 7.4.4. részben bevezetett formulát használjuk. Halmazmetszet esetén ez a formula garantáltan olyan eredményt ad, ami nem nagyobb, mint a két reláció közül a kisebb. Egy multihalmazmetszet esetében azonban előfordulhatnak rendellenességek, amikor a becslés nagyobb, mint bármelyik argumentum. Nézzük például az $R(a, b) \cap_M S(a, b)$ metszetet, ahol az R a (0, 1) sor két példányából áll, és az S ugyanennek a sornak három példányából áll. Ekkor $V(R, a) = V(S, a) = V(R, b) = V(S, b) = 1$, $T(R) = 2$ és $T(S) = 3$. Az összekapcsolásra vonatkozó szabály alapján a becslés $2 \times 3/(\max(1, 1) \times \max(1, 1)) = 6$, de az eredményben nyilvánvalóan nem lehet több, mint $\min(T(R), T(S)) = 2$ sor.

Különbség

Amikor az $R - S$ különbséget vesszük, akkor az eredményben megkapott sorok száma $T(R)$ és $T(R) - T(S)$ között lehet. Becslésként az átlagot javasoljuk: $T(R) - T(S)/2$.

Ismétlődések megszüntetése

Ha $R(a_1, a_2, \ldots, a_n)$ egy reláció, akkor a $\delta(R)$ mérete $V(R,[a_1, a_2, \ldots, a_n])$. Sokszor azonban nem rendelkezünk ezzel a statisztikai értékkel, ezért közelíteni kell. Mint szélsőségek, a $\delta(R)$ mérete megegyezhet az R méretével (nincsenek ismétlődések, vagy lehet 1 (az R minden sora ugyanaz).[4] Egy másik felső korlát a $\delta(R)$-ben levő sorok számára az elképzelhető különböző sorok száma: a $V(R, a_i)$-k szorzata, ahol $i = 1, 2, \ldots, n$. Ez a szám lehet kisebb, mint a $T(\delta(R))$ más becslései. Több olyan szabály is van, amit használhatnánk a $T(\delta(R))$ becslésére. Az egyik elfogadható az, hogy a $T(R)/2$ és az összes $V(R, a_i)$ szorzata közül vesszük a kisebbiket.

Csoportosítás és összesítés

Tegyük fel, hogy van egy $\gamma_L(R)$ kifejezésünk, és e kifejezés eredményének méretére kell becslést adnunk. Ha rendelkezünk a $V(R,[g_1, g_2, \ldots, g_k])$ statisztikával, ahol a g_i-k az L-ben szereplő csoportosítási attribútumok, akkor az lesz a válaszunk. A statisztika

[4] Szigorúan véve, ha R üres, akkor sem az R-ben, sem a $\delta(R)$-ben nincs sor, vagyis az alsó korlát 0. Csakhogy ritkán érdekel bennünket ez a speciális eset.

azonban esetleg nem elérhető, így szükségünk van egy másik módszerre, amivel a $\gamma_L(R)$ méretét becsülhetjük. A $\gamma_L(R)$ sorainak száma megegyezik a csoportok számával. Az eredményben lehet egy csoport, vagy lehet olyan sok csoport, mint ahány sor van az R-ben. A δ-hoz hasonlóan, a csoportok számára a $V(R, A)$-k szorzatával is adhatunk felső korlátot, de itt az A attribútum csak az L csoportosítási attribútumain fut végig. Újfent azt a becslést javasoljuk, ami veszi a $T(R)/2$ és e szorzat közül a kisebbiket.

7.4.8. Feladatok

7.4.1. feladat: A W, X, Y és Z relációkhoz az alábbi statisztikák tartoznak:

$W(a, b)$	$X(b, c)$	$Y(c, d)$	$Z(d, e)$
$T(W) = 100$	$T(X) = 200$	$T(Y) = 300$	$T(Z) = 400$
$V(W, a) = 20$	$V(X, b) = 50$	$V(Y, c) = 50$	$V(Z, d) = 40$
$V(W, b) = 60$	$V(X, c) = 100$	$V(Y, d) = 50$	$V(Z, e) = 100$

Adjunk becslést a következő kifejezések eredményeként kapott relációk méreteire:

* a) $W \bowtie X \bowtie Y \bowtie Z$
* b) $\sigma_{a=10}(W)$
c) $\sigma_{c=20}(Y)$
d) $\sigma_{c=20}(Y) \bowtie Z$
e) $W \times Y$
f) $\sigma_{d>10}(Z)$
* g) $\sigma_{a=1 \text{ AND } b=2}(W)$
h) $\sigma_{a=1 \text{ AND } b>2}(W)$
i) $X \underset{X.c<Y.c}{\bowtie} Y$

* **7.4.2. feladat:** Az E, F, G és H relációkhoz az alábbi statisztikák tartoznak:

$E(a, b, c)$	$F(a, b, d)$	$G(a, c, d)$	$H(b, c, d)$
$T(E) = 1000$	$T(F) = 2000$	$T(G) = 3000$	$T(H) = 4000$
$V(E, a) = 1000$	$V(F, a) = 50$	$V(G, a) = 50$	$V(H, b) = 40$
$V(E, b) = 50$	$V(F, b) = 100$	$V(G, c) = 300$	$V(H, c) = 100$
$V(E, c) = 20$	$V(F, d) = 200$	$V(G, d) = 500$	$V(H, d) = 400$

Hány sort tartalmaz ezeknek a relációknak az összekapcsolása, ha az ebben a részben bemutatott becslési technikákat alkalmazzuk?

! **7.4.3. feladat:** Hogyan becsülné egy egyoldali összekapcsolás (semijoin) méretét?

!! **7.4.4. feladat:** Vegyük az $R(a, b) \bowtie S(a, c)$ összekapcsolást, ahol R-nek és S-nek egyaránt 1000 sora van. Az a attribútumnak mindkét relációban 100 különböző értéke van, és ez *ugyanaz* a 100 érték. Ha az értékek eloszlása egyenletes lenne, vagyis minden egyes a érték pontosan 10-szer fordulna elő mindkét relációban, akkor az összekapcsolásban 10 000 sor lenne. Tegyük fel ehelyett, hogy a 100 a érték mindkét relációban ugyanazt a Zipfian-eloszlást mutatja. Egész pontosan, legyenek az értékek a_1, a_2, ..., a_{100}. Ekkor az R és az S azon sorainak száma, melyek a értéke a_i, az $1/\sqrt{i}$ -vel arányos. Hány sort tartalmaz az összekapcsolás ezen feltételek mellett? Hagyjuk figyelmen kívül azt a tényt, hogy egy adott a értékkel rendelkező sorok száma nem lehet nem egész szám.

7.5. Bevezetés a költség alapú tervválasztásba

Akár egy logikai terv kiválasztásáról van szó, akár egy fizikai terv logikai tervből történő létrehozásáról, a lekérdezésoptimalizálónak becslést kell végeznie bizonyos kifejezések kiértékelésének a költségére vonatkozóan. A költség alapú tervválasztásban felmerülő kérdéseket itt tárgyaljuk, a 7.6. részben pedig részletesen megvizsgáljuk a költség alapú tervválasztás egyik legfontosabb és legnehezebb problémáját: több reláció összekapcsolási sorrendjének megválasztását.

Akárcsak korábban, most is azzal a feltételezéssel élünk, hogy egy kifejezés kiértékelésének „költségét" a végrehajtott lemez I/O-műveletek száma jól közelíti. A lemez I/O-műveletek számát pedig a következők befolyásolják:

1. A lekérdezés megvalósítására kiválasztott konkrét logikai operátorok, ami akkor dől el, amikor a logikai lekérdezéstervet megválasztjuk.
2. A közbülső relációk méretei, amik becsléseit a 7.4. részben tárgyaltuk.
3. A logikai operátorok megvalósítására használt fizikai operátorok, például hogy egymenetes vagy kétmenetes összekapcsolást választunk, vagy hogy rendezünk vagy nem rendezünk egy adott relációt. Ezt a kérdést a 7.7. részben tárgyaljuk.
4. A hasonló műveletek sorrendje, különös tekintettel az összekapcsolásra, amit a 7.6. rész tárgyal.
5. Az argumentumok átadásának módszere, vagyis ahogy az argumentumok egy fizikai operátortól a következő számára átadódnak. Ennek tárgyalása szintén a 7.7. részben kerül sorra.

Sok kérdést kell megoldanunk ahhoz, hogy hatékony költség alapú tervválasztást valósítsunk meg. Ebben a részben először azt nézzük meg, hogy az adatbázisból hogyan nyerhetjük ki leghatékonyabban a méretre vonatkozó paramétereket, amelyek olyan lényeges voltak, amikor a relációk méretét becsültük a 7.4. részben. Ezután újra elővesszük a jó logikai terv megtalálása érdekében bevezetett algebrai szabályokat. A költségalapú elemzéskor a logikai lekérdezésterv transzformálására való szokásos heurisztikák használata indokolt lehet, mint amilyen például a kiválasztások tologatása lefelé a fában. Végül a kiválasztott logikai tervből származtatható összes fizikai terv felsorolására vonatkozóan nézünk meg különböző megközelítéseket. Különösen fontosak a kiértékelendő tervek számának csökkentésére irányuló módszerek, melyek ugyanakkor valószínűsítik azt is, hogy a legkisebb költségű terv is köztük van.

7.5.1. Méretre vonatkozó paraméterek becslése

A 7.4. rész formuláit arra alapozva fogalmaztuk meg, hogy ismerünk bizonyos fontos paramétereket, különösen a $T(R)$-t, ami egy R reláció sorainak számát jelenti, és a $V(R, a)$-t, ami az R reláció a attribútumában előforduló különböző értékek számát jelöli. Egy modern adatbázis-kezelő rendszer általában lehetővé teszi, hogy a felhasználó vagy a rendszergazda közvetlenül kérje ezeknek a statisztikáknak az összegyűjtését. Ezek a statisztikák ezután használhatók a további lekérdezésoptimalizálások során a műveletek költségének becslésére. Ha ezt követő adatbázis-módosítások hatására a statisztikai értékek megváltoznak, a változásokat a rendszer csak egy újabb, statisztikát gyűjtő parancs után veszi figyelembe.

Ha végigolvasunk egy teljes R relációt, akkor nyilván megszámolható a benne lévő sorok $T(R)$ száma, és minden A attribútumra az általa felvett különböző értékek $V(R, A)$ száma is megkapható. Azoknak a blokkoknak a számát, amelyekben az R elfér – azaz $B(R)$-t – úgy becsülhetjük, hogy vagy megszámoljuk a blokkok tényleges számát (ha R nyalábolatlan tárolt), vagy a $T(R)$-t elosztjuk azon sorok számával, amelyek egy blokkban elférnek (vagy az egy blokkban elhelyezhető sorok átlagos számával, ha a sorok változó hosszúságúak). Vegyük észre, hogy a $B(R)$ e két becslése nem feltétlenül ugyanaz, de azok rendszerint „elég közeliek" a költségek összehasonlítása szempontjából, amíg konzisztens módon az egyik vagy a másik megközelítést választjuk.

Egy adatbázis-kezelő rendszer egy adott attribútum értékeinek *hisztogramját* is ki tudja számítani. Ha $V(R, A)$ nem túl nagy, akkor a hisztogram az A attribútum minden értékéhez tartalmazhatja azok előfordulásának számát (vagy arányát). Ha ennek az attribútumnak nagyon sok értéke van, akkor az is egy lehetőség, hogy csak a leggyakoribb értékeket rögzítjük külön-külön, a többi értéket pedig csoportokba soroljuk. A hisztogramok legjellemzőbb típusai a következők:

1. *Egyenlő szélesség.* Választunk egy w szélességet és egy v_0 konstanst. Meghatározzuk azoknak a soroknak a számát, amely sorokban lévő – v-vel jelölt – értékre: $v_0 \leq v < v_0 + w$. Ugyanezt tesszük akkor is, amikor $v_0 + w \leq v < v_0 + 2w$, és így tovább. A v_0 érték lehet a legkisebb lehetséges érték vagy az aktuális minimum érték. Az utóbbi esetben, ha egy alacsonyabb értékkel találkozunk, csökkentjük a v_0 értékét w-vel és egy új (sor)számlálót adunk a hisztogramhoz.
2. *Egyenlő magasság.* Ezek a szokásos „százalékos arányok". Veszünk egy p törtet, és felsoroljuk a legkisebb értéket, azt az értéket, amelyik p törtnyire van a legkisebbtől, azt amelyik $2p$ törtnyire van a legkisebbtől és így tovább, a legnagyobb értékig.

3. *Leggyakoribb értékek.* Felsorolhatjuk a leggyakoribb értékeket és a hozzájuk tartozó előfordulási számokat. Ezt az információt megadhatjuk úgy, hogy az összes többi értékre úgy számolunk előfordulási gyakoriságot, hogy azokat egyetlen csoportnak tekintjük, de a leggyakoribb értékeket a többi értékre vonatkozó egyenlő szélességű vagy egyenlő magasságú hisztogrammal együtt is rögzíthetjük.

Egy hisztogram használatának az az előnye, hogy az összekapcsolások méreteire a 7.4. részben leírt egyszerűsített módszereknél pontosabb becslést adhatunk. Konkrétabban szólva, ha az összekapcsolási attribútum valamely értéke mindkét összekapcsolandó reláció hisztogramjában közvetlenül megjelenik, akkor pontosan tudjuk, hogy az eredménynek hány sorában fog szerepelni ez az érték. Az összekapcsolási attribútum azon értékei esetén, amelyek valamelyik reláció hisztogramjában nem jelennek meg közvetlenül, az összekapcsolásra gyakorolt hatás a 7.4. résznek megfelelően becsülhető. Ha egyenlő szélességű hisztogramot használunk úgy, hogy a két reláció összekapcsolási attribútumaira ugyanazokat a sávokat alkalmazzuk, akkor becsülhetjük az egymásnak megfelelő sávok összekapcsolásainak méreteit, majd ezeket összeadhatjuk. Az eredmény helyes lesz, mert csak az egymásnak megfelelő sávokba eső sorok kapcsolódhatnak. A következő példákban hisztogram alapú becslésre mutatunk példákat. A későbbiek során nem fogunk hisztogramokat használni a becslésekben.

7.29. példa: Vegyünk olyan hisztogramokat, amelyek a három leggyakoribb értéket és a hozzájuk tartozó számlálókat tartalmazzák, és a maradék értékeket egy csoportba sorolják. Tegyük fel, hogy az $R(a, b) \bowtie S(b, c)$ összekapcsolást akarjuk kiszámítani. Az $R.b$-re vonatkozó hisztogram legyen az alábbi:

1: 200, 0: 150, 5: 100, egyéb: 550

Az R reláció 1000 sorából tehát 200-nak a b értéke 1, 150-nek a b értéke 0 és 100-nak a b értéke 5. 550 sornak van továbbá olyan b értéke, ami nem 0, 1 vagy 5, és ezen egyéb értékek közül egyik sem fordul elő 100-nál többször.

Az $S.b$-re vonatkozó hisztogram a következő legyen:

0: 100, 1: 80, 2: 70, egyéb: 250

Tegyük fel továbbá, hogy $V(R, b) = 14$ és $V(S, b) = 13$. Ez azt jelenti, hogy az R ismeretlen b értéket tartalmazó 550 sora tizenegy érték között van elosztva, vagyis átlagosan 50 sor jut mindegyikre, és az S ismeretlen b értéket tartalmazó 250 sora tíz érték között van elosztva, azaz átlagosan 25 sor jut minden ilyen értékre.

A 0 és 1 értékek explicit módon szerepelnek mindkét hisztogramban, így kiszámolhatjuk, hogy ha az R-nek azt a 150 sorát, amelyekre $b = 0$, összekapcsoljuk az S-nek azzal a 100 sorával, amelyeknek b értéke ugyanez az érték, akkor ez 15 000 sort eredményez. Hasonlóképpen, ha az R-nek azt a 200 sorát, amelyekre $b = 1$, összekapcsoljuk az S-nek azzal a 80 sorával, amelyekre szintén $b = 1$, akkor ez 16 000 további sort eredményez.

A maradék sorok összekapcsolásának becslése összetettebb. Továbbra is fenntartjuk azt a előfeltevést, hogy a kisebb értékhalmazzal rendelkező relációban (jelen esetben S) előforduló minden érték a másik reláció értékhalmazában is szerepel. Az S tizenegy fennmaradó b értéke közül az egyikről tudjuk, hogy az a 2, egy másikról viszont feltesszük, hogy az az 5, hiszen ez az egyik leggyakoribb érték az R-ben. Becslésként azt mondjuk, hogy a 2 az R-ben 50-szer fordul elő, az 5 az S-ben pedig 25-ször. Ezeket a becsléseket úgy kapjuk, hogy azt feltételezzük, az adott érték a megfelelő reláció hisztogramjában említett „egyéb" értékek egyike. A 2 b értékből adódó további sorok száma így $70 \times 50 = 3500$, az 5 b értékből származó további sorok száma pedig $100 \times 25 = 2500$.

Végezetül van még kilenc olyan b érték, ami mindkét relációban szerepel, és az ezekre vonatkozó becslésünk az, hogy mindegyik 50-szer fordul elő R-ben, és 25-ször S-ben. Így mind a kilenc érték $50 \times 25 = 1250$ további sorral járul hozzá az eredményhez. A végső eredmény méretének becsült értéke tehát:

$$15\,000 + 16\,000 + 3500 + 2500 + 9 \times 1250$$

azaz 48 250 sor. Megjegyezzük, hogy a 7.4. részből vett egyszerűbb becslés, ami azon a feltevésen alapul, hogy mindegyik érték ugyanannyiszor fordul elő mindegyik relációban, $1000 \times 500/14$, azaz 35 714 lenne. □

7.30. példa: Ebben a példában egy egyenlő szélesség típusú hisztogramot feltételezünk, és azt demonstráljuk, hogy hogyan befolyásolja egy összekapcsolás méretének becslését az, ha tudjuk, hogy a két reláció értékhalmazai majdnem diszjunktak. A relációk a következők:

```
Jan(nap, hőmérséklet)
Júl(nap, hőmérséklet)
```

A lekérdezés pedig az alábbi:

```
SELECT Jan.nap, Júl.nap
FROM Jan, Júl
WHERE Jan.hőmérséklet = Júl.hőmérséklet;
```

A január és július hónapoknak azokat a napjait keressük tehát, amikor ugyanaz volt a hőmérséklet. A lekérdezésterv az, hogy a `Jan` és `Júl` relációkat a `hőmérséklet` attribútumok egyenlősége alapján összekapcsoljuk, majd vetítünk a két `nap` attribútumra.

A `Jan` és `Júl` relációkhoz tartozó, a hőmérséklet attribútumokra vonatkozó feltételezett hisztogramokat a 7.25. ábra mutatja.[5] Általában, ha mindkét összekapcsolási attribútumhoz egyenlő szélesség típusú hisztogram tartozik ugyanazokkal a sávokkal (amelyik közül némelyik esetleg üres valamelyik reláció esetében), akkor az össze-

[5] Az Egyenlítőtől délre lévő barátaink felcserélhetik a január és július oszlopait.

kapcsolás méretét úgy becsülhetjük, hogy az egyes sávokra leszűkített összekapcsolások méreteit becsüljük külön-kölön, majd az így kapott becsléseket összegezzük.

Sáv	*Jan*	*Júl*
0–9	40	0
10–19	60	0
20–29	80	0
30–39	50	0
40–49	10	5
50–59	5	20
60–69	0	50
70–79	0	100
80–89	0	60
90–99	0	10

7.25. ábra. *Hőmérséklet hisztogramjai*

Ha két megfelelő sávnak T_1, illetve T_2 sora van, és a sávba tartozó értékek száma V, akkor – a 7.4.4. részben lefektetett elveket követve – az ezekre a sávokra vonatkoztatott összekapcsolás eredményének a méretére kapott becslés: T_1T_2/V. A 7.25. ábrán látható hisztogramok esetében ezeknek a szorzatoknak a nagy része 0, mert a T_1 és T_2 közül az egyik vagy mindkettő 0. Csak a 40–49 és 50–59 sávok azok, amelyekre sem a T_1, sem a T_2 nem 0. Mivel egy sáv szélessége $V = 10$, a 40–49 sáv $10 \times 5/10 = 5$ sort, az 50–59 sáv pedig $5 \times 20/10 = 10$ sor jelent.

Következésképpen, az összekapcsolás méretének becslése $5 + 10 = 15$ sor. Ha nem lennének hisztogramok, és csak annyit tudnánk, hogy mindegyik relációnak 245 sora van, amelyek 100 darab 0 és 99 közé eső érték mentén oszlanak el, akkor az összekapcsolás méretének becslése $245 \times 245/100 = 600$ sor lenne. □

7.5.2. Statisztikák növekményes kiszámítása

A lekérdezésoptimalizálókban a statisztikák bizonyos időnkénti kiszámolását részesítik előnyben, mert ezek a statisztikák nem szoktak rövid időn belül radikálisan megváltozni. Továbbá, amint már említettük, még a pontatlan statisztikák is hasznosak, ha azokat mindegyik tervre alkalmazzuk, amelyek a „legjobb" címért versengenek. Teljes relációk időnkénti megvizsgálása azonban drága dolog. A statisztikák időnkénti újra kiszámításának egy alternatívája a *növekményes kiértékelés* (incremental evaluation). Ez a módszer karbantartja és aktualizálja a paraméterekre vonatkozó becsléseket minden alkalommal, amikor az adatbázist módosítják. Íme néhány módja annak, ahogy ezt a rendszer megteheti:

- A $T(R)$ karbantartásához a rendszer 1-et mindig hozzáad, amikor egy sort beszúrnak R-be, illetve 1-et kivon belőle, valahányszor onnan törölnek egy sort. Megjegyezzük, hogy ehhez szükség van a beszúrást és a törlést végző függvények módosítására. Ez növeli minden ilyen művelet költségét, de a többletköltség általában jelentéktelen.
- Ha az R valamely attribútumához létezik B-fa-index, akkor $T(R)$-t úgy becsülhetjük, hogy csak a B-fában lévő blokkok számát számoljuk meg. Feltehetjük például, hogy minden blokk 3/4 részben van tele, és egy blokkban elférő kulcsok és mutatók számát használhatjuk a B-fa levelei által mutatott sorok számának becslésére. Ez a módszer kevésbé pontos, mint a $T(R)$ direkt megszámolása, de csak akkor igényel teendőt, amikor a B-fa szerkezete változik, ami aránylag ritka a beszúrásokhoz és törlésekhez viszonyítva.
- Ha az R reláció a attribútumához létezik index, akkor a $V(R, a)$-t pontosan karbantarthatjuk. Egy R-be történő beszúráskor mindenképpen meg kell találnunk az új sor a értékét az indexben, és ekkor megállapítjuk, hogy létezett-e már sor ugyanezzel az a értékkel. Ha nem, akkor a $V(R, a)$ számlálóhoz egyet hozzáadunk. Hasonlóképpen, amikor törlünk egy sort az R-ből, akkor ellenőrizzük, hogy az utolsó sort töröljük-e R-ből az adott a értékkel, és ha igen, akkor csökkentjük eggyel a $V(R, a)$-t.
- Ha tudjuk, hogy az a kulcsa az R-nek, akkor azt is tudjuk, hogy $V(R, a) = T(R)$, anélkül hogy az indexet vizsgálnánk vagy a $V(R, a)$-t közvetlenül karbantartanánk.
- Ha az $R.a$-ra nincs index, akkor a rendszer létrehozhat egy kezdetleges indexet azáltal, hogy fenntart egy adatstruktúrát (például egy tördelőtáblát vagy B-fát) az a értékeinek tárolására.

Végül az is egy lehetőség a $V(R, A)$ statisztika kiszámítására, hogy akkor adunk erre a mennyiségre statisztikai becslést, amikor szükség van rá, mégpedig az adatok egy

Miért becslünk méretet lemez I/O-műveletek helyett?

Amikor a logikai lekérdezésterveket vizsgáljuk, még nem döntöttük el azt, hogy mely fizikai operátorokat használjuk a relációs algebra operátorainak megvalósításához. Így nem lehetünk biztosak egy adott terv végrehajtásához szükséges lemez I/O-műveletek számában. Követhetjük azonban azt a heurisztikát, miszerint az a terv lesz valószínűleg a legjobb terv, amelyikre a közbülső relációk méreteinek összege a legkisebb. Ezt a heurisztikát az támasztja alá, hogy minél kisebbek a relációk, annál kevesebb lemez I/O-műveletet igényel azok olvasása és írása, és annál hatékonyabbak a relációkra vonatkozó műveleteket megvalósító algoritmusok.

Egy lekérdezés igazi költségének becslését tényleg a lemez I/O-műveletek száma jelenti. Egy részletesebb elemzés figyelembe venné a CPU időt is, és egy még részletesebb elemzés még a lemezfej mozgására is kitérne, számításba véve az elért blokkok helyét a lemezen. A gyakorlatban még a legegyszerűbb becslés – közbülső relációk méretei – is elég jó, hiszen a lekérdezésoptimalizálónak csak lekérdezésterveket kell összehasonlítania, és nem kell pontos végrehajtási időt számolnia.

kis részének mintája alapján. Ez egy bonyolult számítás, és számos feltételtől függ, például attól, hogy egy attribútumban előforduló értékek egyenletesen eloszlást, Zipfian-eloszlást vagy valamilyen más eloszlást mutatnak-e. A vezérfonal azonban a következő. Ha megnézzük az R egy kis mintáját, mondjuk a sorainak 1%-át, és azt találjuk, hogy a legtöbb a érték különböző, akkor $V(R, a)$ valószínűleg közel van $T(R)$-hez. Ha azt tapasztaljuk, hogy az a értékeiben nagyon kevés különböző érték szerepel, akkor az a valószínű, hogy az aktuális relációban létező legtöbb a értékkel találkoztunk.

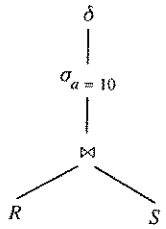
7.5.3. Logikai lekérdezéstervek költségének csökkentésére irányuló heurisztikák

A lekérdezésekre és alkérdésekre vonatkozó költségbecslések jól használhatók a lekérdezések heurisztikus átalakításai során. A 7.3.3. részben már megvizsgáltuk, hogy miként várható el, hogy bizonyos heurisztikák költségbecsléstől független alkalmazása szinte biztosan javítson egy logikai lekérdezésterv költségén. Jó példa erre a kiválasztások tologatása lefelé a fában. Vannak azonban a lekérdezésoptimalizálásnak más pontjai, ahol a költség becslése egy transzformáció előtt és után lehetővé teszi, hogy alkalmazzuk a transzformációt, ha várhatóan csökkenti a költséget, egyébként pedig elkerüljük. A végső logikai terv előállításakor például számba vehetünk számos opcionális transzformációt, és megnézhetjük az azok végrehajtása előtt és után előálló költségeket. Ezeket a kérdéseket szemlélteti a következő példa.

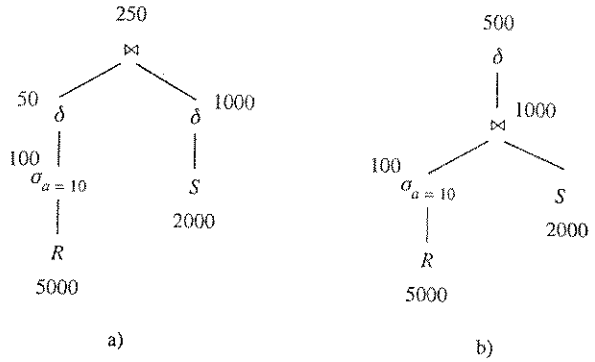
7.31. példa: Vegyük a 7.26. ábrán található kiindulási logikai tervet, valamint legyenek az R és S relációkhoz tartozó statisztikák a következők:

$R(a, b)$	$S(b, c)$
$T(R) = 5000$	$T(S) = 2000$
$V(R, a) = 50$	
$V(R, b) = 100$	$V(S, b) = 200$
	$V(S, c) = 100$

Kiindulva a 7.26. ábrából, a végső logikai lekérdezésterv előállításakor kitartunk amellett, hogy a kiválasztást levisszük a fában ameddig csak lehet. Nem vagyunk vi-



7.26. ábra. *Logikai lekérdezésterv a 7.31. példához*



7.27. ábra. *Két jelölt a legjobb logikai lekérdezéstervre*

szont biztosak abban, hogy a δ operátor összekapcsolás alá történő levitelének van-e értelme vagy sem. A 7.26. ábrából így két lekérdezéstervet generálunk, ezek a 7.27. ábrán találhatók, és abban különböznek, hogy az ismétlődések megszüntetése az összekapcsolás előtt vagy után történik-e. Észrevehetjük, hogy az a) tervben a δ-t levittük a fa mindkét ágán. Ha R és/vagy S nem tartalmaz ismétlődéseket, akkor a megfelelő ágon a δ elhagyható.

A 7.4.3. részből tudjuk, hogy a kiválasztás eredményének méretét hogyan kell becsülni: a $T(R)$-t elosztjuk a $V(R, a)$-val, ahol $V(R, a) = 50$. Azt is tudjuk, hogy az összekapcsolás méretét hogyan becsüljük: az argumentumok méreteinek szorzatát osztjuk $\max(V(R, b), V(S, b))$-vel, ahol ez a maximum érték 200. Azt még nem tudjuk viszont, hogy hogyan becsüljük az ismétlődésektől megszabadított relációk méretét.

Nézzük először a $\delta(\sigma_{a = 10}(R))$ méretének becslését. Mivel a $\sigma_{a = 10}(R)$-ben csak egyetlen értéke lehet az a-nak és mintegy 100 értéke a b-nek, és e reláció becsült mérete 100 sor, a 7.4.7. részben megfogalmazott szabály azt mondja számunkra, hogy az egyes attribútumokhoz tartozó értékszámlálók szorzata nem korlátozó tényező. Ezért a δ eredményének méretét a $\sigma_{a = 10}(R)$-ben lévő sorok felével becsüljük. Ennek megfelelően a 7.27.a) ábrán 50 sor szerepel a $\delta(\sigma_{a = 10}(R))$ méretének becsléseként.

Nézzük most a 7.27.b) ábrán lévő δ eredményére vonatkozó becslést. Az összekapcsolás eredményében egyetlen érték fordul elő az a-ban, becslés alapján a b-ben $\min(V(R, b), V(S, b)) = 100$ különböző érték fordul elő illetve becslés alapján $V(S, c) = 100$ érték szerepel a c-ben. Vagyis ismét arra a következtetésre jutunk, hogy az értékszámlálók szorzata nem korlátozza a δ eredményének méretét. Az eredmény méretére adott becslés az összekapcsolás eredményében lévő sorok fele, azaz 500 sor lesz.

Hogy össze tudjuk hasonlítani a 7.27. ábra két tervét, összeadjuk a csomópontokhoz tartozó becsült méreteket, kivéve a gyökeret és a leveleket. A gyökeret és a levelet azért hagyjuk ki, mert ezek a méretek nem függnek a választott tervtől. Ez a költség, vagyis a közbülső csomópontokhoz tartozó becsült méretek összege, az a) terv esetében: $100 + 50 + 1000 = 1150$, míg a b) terv esetében az összeg: $100 + 1000 = 1100$. Arra a következtetésre jutunk tehát, hogy kis eltéréssel ugyan, de az ismétlődések

Az eredmény méretének becslései nem kell hogy egyformák legyenek

Vegyük észre, hogy a 7.27. ábrán a két fa gyökereihez tartozó becslések különbözőek: 250 az egyik esetben és 500 a másikban. Mivel a becslés egy pontatlan tudomány, ilyen anomáliák előfordulnak. Valójában az a kivétel, amikor a konzisztenciára vonatkozóan garanciát tudunk nyújtani, amint ezt a 7.4.6. részben tettük.

Intuitíve azt mondhatnánk, hogy a b) terv becsült költsége magasabb, mert ha mind az R-ben, mind az S-ben vannak ismétlődések, akkor ezek az ismétlődések sokszorozódnak az összekapcsolás során, azok a sorok például, amelyek háromszor szerepelnek az R-ben és kétszer az S-ben, hatszor fognak megjelenni az $R \bowtie S$ összekapcsolás eredményében. A mi egyszerű formulánk, amellyel egy δ eredményének méretét becsüljük, nem számol azzal a lehetőséggel, hogy korábbi műveletek felerősíthetik az ismétlődések hatását.

megszüntetését a végére halasztva egy jobb tervet kapunk. Az ellenkező következtetésre jutnánk akkor, ha R-nek vagy S-nek kevesebb b értéke lenne. Ekkor nagyobb lenne az összekapcsolás mérete, ami megnövelné a b) terv költségét. □

7.5.4. Fizikai tervek felsorolásának lehetőségei

Most azt vizsgáljuk meg, hogy egy logikai lekérdezésterv fizikai lekérdezéstervvé történő konvertálása során hogyan használjuk a költségbecslést. A – *kimerítőnek* nevezett – legalapvetőbb megközelítésben vesszük a 7.4. rész elején felvázolt pontokra (összekapcsolások sorrendje, operátorok fizikai implementációja stb.) adott lehetséges válaszok összes kombinációját. Mindegyik fizikai tervhez egy becsült költséget rendelünk, és a legkisebb költségű tervet választjuk.

Sok egyéb megközelítés is létezik azonban egy fizikai terv kiválasztására. Ebben a részben különböző használatban lévő megközelítéseket mutatunk be erre vonatkozóan, míg a 7.6. részben az összekapcsolási sorrend megválasztásának problémájával kapcsolatos fő elgondolásokat szemléltetjük. Mielőtt folytatnánk, hadd jegyezzük meg, hogy a lehetséges fizikai tervek tartományának feltárására alapvetően kétféle megközelítés létezik:

- *Felülről lefelé.* Itt a gyökértől indulunk el, és haladunk lefelé a logikai lekérdezésterv fájában. A gyökérben található művelet minden lehetséges megvalósításához megnézzük az argumentum(ok) lehetséges kiértékeléseit, kiszámoljuk az egyes kombinációk költségét, és a legjobbat választjuk.[6]

[6] Emlékezzünk a 7.3.4. részből arra, hogy a logikai lekérdezésterv fájának egy csomópontja egyetlen kommutatív és asszociatív operátor (például összekapcsolás) sokféle használatát képviselheti. Tehát egy adott csomópontra vonatkozó összes lehetséges terv megvizsgálása maga is nagyon sok választás felsorolását foglalhatja magában.

- *Alulról felfelé.* A logikai lekérdezésterv fájának minden részkifejezéséhez kiszámoljuk a részkifejezés lehetséges kiszámítási módjaihoz tartozó költségeket. Egy E részkifejezés kiértékelési lehetőségeit és költségeit úgy számítjuk ki, hogy vesszük az E részkifejezéseire vonatkozó lehetséges választásokat, és az összes lehetséges módon kombináljuk azokat az E gyökér operátorának lehetséges megvalósításaival.

Valójában nincs sok különbség a két megközelítés között, azok legtágabb értelmezését tekintve, hiszen mindegyik esetben tekintetbe vesszük a lekérdezésben szereplő összes operátor megvalósítási módjainak minden lehetséges kombinációját. A keresés korlátozásával kapcsolatban elmondhatjuk, hogy egy felülről lefelé megközelítés esetleg lehetővé teszi, hogy elhagyjunk bizonyos választásokat, amelyeket alulról felfelé megközelítésben nem hagyhatnánk el. Kidolgoztak azonban olyan alulról felfelé stratégiákat is, amelyek jelentősen korlátozzák a választásokat, ezért az elkövetkezőkben az alulról felfelé módszerekre helyezzük a hangsúlyt.

Észrevehettük, hogy az alulról felfelé módszernek létezik egy nyilvánvaló egyszerűsítése, nevezetesen, amikor egy nagyobb részkifejezés kiszámítása során csak a *legjobb* tervet vesszük figyelembe annak minden részkifejezése esetén. Ez a megközelítés, amely a módszerek alábbi listájában *dinamikus programozás* néven szerepel, nem biztos, hogy a legjobb tervet eredményezi, habár ez gyakran megtörténik. A *Selinger-módszer* (vagy *System-R-módszer*) elnevezésű optimalizálás, amely szintén szerepel a listában, egy részkifejezéshez tartozó tervek további tulajdonságait is kihasználja annak érdekében, hogy optimális végső terveket állítson elő olyan tervekből, amelyek bizonyos részkifejezések esetén nem optimálisak.

Heurisztikus választás

Az is egy megoldás, hogy egy fizikai terv kiválasztására ugyanazt a megközelítést alkalmazzuk, mint amit általában egy logikai terv kiválasztására használunk: válasszunk heurisztikák alapján. A 7.6.6. részben egy „mohó" heurisztikát fogunk tárgyalni az összekapcsolási sorrendre vonatkozóan. Eszerint először azt a két relációt kapcsoljuk össze, amelyek eredményének becsült mérete a legkisebb, majd ugyanezt az elvet alkalmazzuk ismételten az így kapott reláció és a többi összekapcsolásra váró reláció összekapcsolása során. Sok egyéb alkalmazható heurisztika is létezik, íme néhány a leggyakrabban használtak közül:

1. Ha a logikai tervben egy $\sigma_{A=c}(R)$ kiválasztás szerepel, és az R tárolt relációnak van egy indexe az A attribútumra vonatkozóan, akkor csak az indexet nézzük végig azoknak az R-beli soroknak a megtalálására, amelyeknél az A értéke egyenlő c-vel.
2. Általánosabban szólva, ha a kiválasztás tartalmaz egy fenti $A = c$ feltételt és más feltételeket is, akkor a kiválasztás megvalósítható egy indexes kereséssel, valamint

egy azt követő, a kapott sorokra vonatkozó további kiválasztással, amit a *szűrés* fizikai szintű operátor (`Filter`) fog képviselni.

3. Ha egy összekapcsolás valamely argumentumának van indexe az összekapcsolási attribútum(ok)ra vonatkozóan, akkor használjunk indexes összekapcsolást azzal a relációval a belső ciklusban.
4. Ha egy összekapcsolás egyik argumentuma rendezett az összekapcsolási attribútum(ok) szerint, akkor egy rendezéses összekapcsolást részesítsünk előnyben egy tördelő összekapcsolással szemben, de nem feltétlenül egy index-összekapcsolással szemben, ha olyan is lehetséges.
5. Három vagy több reláció egyesítése vagy metszete során először a legkisebb relációkat csoportosítsuk.

Elágazás-és-korlát

Ebben a – gyakorlatban gyakran használt – megközelítésben azzal kezdjük, hogy valamilyen heurisztikát alkalmazva egy jó fizikai tervet keresünk a teljes logikai lekérdezéstervhez. Legyen ennek a tervnek a költsége C. Ezután, miközben alkérdésekhez tartozó további terveket vizsgálunk, elvethetünk minden olyan tervet egy alkérdés esetén, amelynek költsége nagyobb C-nél, hiszen egy ilyen – alkérdéshez tartozó – terv nem lehet része egy olyan – a teljes lekérdezéshez tartozó – tervnek, amelytől azt várjuk, hogy jobb, mint amit már ismerünk. Ha egy olyan tervet állítunk elő a teljes lekérdezésre vonatkozóan, amelynek költsége kisebb, mint C, akkor a fizikai lekérdezéstervek tartományának további feltárása során C-t ennek a tervnek a költségével helyettesítjük.

E megközelítésnek egy nagy előnye, hogy megválaszthatjuk, mikor vetünk véget a keresésnek, és vesszük az addig talált legjobb tervet. Ha például a C költség kicsi, akkor még ha esetleg léteznek is sokkal jobb tervek, a megtalálásukra fordított idő meghaladhatja C-t, ezért nincs értelme a keresést tovább folytatni. Ha azonban a C nagy, akkor bölcs dolog még arra időt fordítani, hogy egy jobb tervet keressünk.

Hegymászás

Ebben a módszerben, ahol valójában egy „völgyet" keresünk a fizikai tervek és költségeik tartományában, egy heurisztikusan kiválasztott fizikai tervvel kezdünk. A terven ezután kis változtatásokat hajtunk végre, például egy operátorhoz adott módszert egy másikkal helyettesítünk, vagy a kommutatív és/vagy asszociatív szabályok segítségével átrendezzük az összekapcsolásokat, hogy kisebb költségű „közeli" terveket találjunk. Amikor elérünk egy olyan tervhez, amelynek semmilyen kis változtatása nem eredményez alacsonyabb költségű tervet, a tervet kikiáltjuk a választott fizikai lekérdezéstervnek.

Dinamikus programozás

Az általános alulról felfelé stratégia e változatában minden egyes részkifejezés esetében csak a legkisebb költségű tervet tartjuk meg. Amint haladunk felfelé a fában, megvizsgáljuk az egyes csomópontok lehetséges megvalósításait, mindegyik részkifejezéshez a legjobb tervet feltételezve. A 7.6. részben kimerítően tárgyaljuk ezt a megközelítést.

Selinger-féle optimalizálás

Ez a megközelítés a dinamikus programozás módszeren javít annyiban, hogy az egyes részkifejezésekhez nemcsak a legalacsonyabb költségű tervet tartja meg, hanem bizonyos egyéb terveket is, amelyek magasabb költségűek ugyan, de amelyek a kifejezésfa feljebb eső részeinél jól kihasználható rendezettséggel rendelkező eredményt állítanak elő. Ilyen számunkra érdekes rendezettségre példa, amikor a részkifejezés eredménye az alábbiak valamelyike szerint van rendezve:

1. Egy gyökérben elhelyezkedő rendezés (τ) operátorban megadott attribútum(ok).
2. Egy későbbi csoportosítás (γ) operátor csoportosítási attribútuma(i).
3. Egy későbbi összekapcsolás összekapcsolási attribútuma(i) szerint.

Ha egy terv költségét a közbülső relációk méretei összegének vesszük, akkor egy argumentum rendezettsége nem látszik előnyösnek. Ha azonban az ennél pontosabb becslést – a lemez I/O-műveletek számát – használjuk költségként, akkor egy argumentum rendezettségének előnye világossá válik, amennyiben használhatjuk a 6.5. rész rendezettségen alapuló algoritmusainak valamelyikét, és a már rendezett argumentumra megtakaríthatjuk az első menetet.

7.5.5. Feladatok

7.5.1. feladat: Becsüljük meg az $R(a, b) \bowtie S(b, c)$ összekapcsolás méretét, használva az $R.b$-hez és az $S.b$-hez tartozó hisztogramokat. Tegyük fel, hogy $V(R, b) = V(S, b) = 20$, és a két attribútumhoz tartozó mindkét hisztogram megadja a négy leggyakoribb elem előfordulási gyakoriságát az alábbi táblázatnak megfelelően:

	0	*1*	*2*	*3*	*4*	*egyéb*
R.b	5	6	4	5		32
S.b	10	8	5		7	48

Hogyan viszonyul ez a becslés ahhoz az egyszerűbb becsléshez, amikor azt feltételezzük, hogy mind a 20 érték egyenlő valószínűséggel fordul elő? Legyen $T(R) = 52$ és $T(S) = 78$.

* **7.5.2. feladat:** Becsüljük meg az $R(a, b) \bowtie S(b, c)$ összekapcsolás méretét, ha a következő hisztograminformáció áll rendelkezésünkre:

	$b < 0$	$b = 0$	$b > 0$
R	500	100	400
S	300	200	500

! **7.5.3. feladat:** A 7.31. példában azt sugalltuk, hogy az egyik b attribútumhoz tartozó értékek számának csökkentése a 7.27. ábra a) tervét az ábra b) tervénél jobbá tenné. A

* a) $V(R, b)$
b) $V(S, b)$

mely értékei esetén lesz az a) tervnek alacsonyabb becsült költsége, mint a b) tervnek?

! **7.5.4. feladat:** Vegyünk négy relációt: R, S, T és V. Ezek rendre 200, 300, 400 és 500 sort tartalmaznak, amely sorokat véletlenszerűen és függetlenül választjuk ugyanabból az 1000 sorból álló készletből. Egy adott sor R-beli előfordulásának valószínűsége például 1/5, S-beli valószínűsége pedig 3/10. Annak valószínűsége, hogy egy sor az R-ben és az S-ben is szerepel: 3/50.

* a) Mi az $R \cup S \cup T \cup V$ várható mérete?
b) Mi az $R \cap S \cap T \cap V$ várható mérete?
c) Az egyesítések melyik sorrendjéhez tartozik a legkisebb költség (a közbülső relációk méreteinek összegére adott becslés)?
d) A metszetek melyik sorrendjéhez tartozik a legkisebb költség (a közbülső relációk méreteinek összegére adott becslés)?

! **7.5.5. feladat:** Ismételjük meg a 7.5.4. feladatot azzal a változtatással, hogy mind a négy relációnak 500 sora legyen, véletlenszerűen választva az 1000 sorból.[7]

!! **7.5.6. feladat:** Tegyük fel, hogy ki akarjuk számolni a következő kifejezést:

$\tau_b(R(a, b) \bowtie S(b, c) \bowtie T(c, d))$

Összekapcsolunk tehát három relációt, és az eredményt a b attribútum szerint rendezve állítjuk elő. Éljünk az alábbi egyszerűsítő feltételezésekkel:

i) Nem az R-et és T-t „kapcsoljuk össze" először, mert az egy szorzat.
ii) Bármelyik másik összekapcsolás elvégezhető egy kétmenetes rendezéses összekapcsolással vagy tördelő összekapcsolással, de máshogy nem.
iii) Bármely reláció vagy bármely kifejezés eredménye rendezhető egy kétfázisú, sokágú fésüléses rendezéssel, de máshogy nem.

[7] E feladat megfelelő részeinek megoldása *nincs* a weben közzétéve.

iv) Az első két reláció összekapcsolásának eredményét nem tároljuk ideiglenesen a lemezen, hanem blokkonként adódik át az utolsó összekapcsolásnak, mint annak argumentuma.
v) Mindegyik reláció 1000 blokkot foglal el, és bármelyik két reláció összekapcsolásának eredménye 5000 blokkot foglal el.

Ezen előfeltevések mellett, válaszoljuk meg a következőket:

* a) Melyek azok a részkifejezések és sorrendek, amelyeket egy Selinger-módszerrel történő optimalizálás tekintetbe venne?
b) Lemez I/O-műveletek számát tekintve költségbecslésnek[8], melyik lekérdezésterv jár a legkisebb költséggel.

!! **7.5.7. feladat:** Adjunk egy példát egy $E \bowtie F$ alakú logikai lekérdezéstervre, ahol E és F kifejezések (amiket megválaszthatunk), amikor az E és F kiértékelésére használt legjobb tervek nem engedik meg a végső összekapcsolásra vonatkozóan olyan algoritmus választását, amely a teljes kifejezés kiértékelésének összköltségét minimalizálná. Bármilyen feltevéssel élhetünk a rendelkezésre álló memóriapufferekkel, valamint az E-ben és F-ben említett relációk méreteivel kapcsolatban.

7.6. Összekapcsolások sorrendjének megválasztása

Ebben a részben a költség alapú optimalizálás egyik kritikus problémájára összpontosítunk: az összekapcsolási sorrend megválasztására három vagy több reláció (természetes) összekapcsolása esetén. Hasonló kérdések megfogalmazhatók más bináris műveletek kapcsán is, mint amilyen az egyesítés és metszet, de ezek a műveletek nem annyira jelentősek a gyakorlatban, mivel végrehajtásuk általában kevesebb időt igényel, mint az összekapcsolásé, és ritkábban is szerepelnek három vagy több argumentummal.

7.6.1. Összekapcsolások bal és jobb oldali argumentumainak jelentősége

Egy összekapcsolás sorrendbe állításakor tudnunk kell, hogy a 6. fejezetben tárgyalt összekapcsolási módszerek közül sok aszimmetrikus abban az értelemben, hogy a két argumentum reláció szerepe különböző, és az összekapcsolás költsége függ attól, hogy melyik reláció játssza melyik szerepet. A 6.3.3. rész talán legfontosabb egyme-

[8] Vegyük észre, hogy mivel tettünk néhány nagyon specifikus kitételt a használandó összekapcsolási módszerrel kapcsolatban, becsülhetjük a lemez I/O-műveleteket ahelyett, hogy az egyszerűbb, de pontatlanabb, a sorok számát figyelembe vevő költségbecsléssel dolgoznánk.

netes összekapcsolása beolvassa az egyik relációt – lehetőleg a kisebbet – a központi memóriába, létrehozva mondjuk egy tördelőtábla struktúrát, hogy elősegítse a másik relációból származó sorok illesztését. Ezután beolvassa a másik relációt, egyszerre egy blokkot, és annak sorait összekapcsolja a memóriában tárolt sorokkal.

Tegyük fel, hogy egy fizikai terv kiválasztásakor egy egymenetes összekapcsolás használata mellett döntünk. Ekkor az összekapcsolás bal argumentumát tekintjük annak a (kisebb) relációnak, amelyiket a központi memória adatszerkezetében tárolni fogunk (ezt a relációt nevezzük az *építő relációnak*), míg az összekapcsolás jobb argumentumát (a *vizsgáló relációt*) blokkonként olvassuk be, és illesztjük annak sorait a tárolt reláció soraival. Az argumentumokat a következő összekapcsolási algoritmusok is megkülönböztetik:

1. Beágyazott ciklusú összekapcsolás, ahol azt feltételezzük, hogy a bal argumentum a külső ciklus relációja.
2. Indexes összekapcsolás, ahol azt feltételezzük, hogy a jobb argumentum rendelkezik az indexszel.

7.6.2. Összekapcsolási fák

Ha vesszük két reláció összekapcsolását, sorba kell rendezni az argumentumokat. Megegyezés alapján a kisebb becsült mérettel rendelkezőt tekintjük a bal argumentumnak. Megjegyezzük, hogy a fent említett – egymenetes, beágyazott ciklusú, illetve indexes – algoritmusok mindegyike akkor működik a legjobban, ha a bal argumentum a kisebb. Pontosabban szólva, az egymenetes és a beágyazott ciklusú összekapcsolásban egy speciális szerep jut a kisebb relációnak (építő reláció, illetve külső ciklus), az indexes összekapcsolás pedig csak akkor egy ésszerű választás, ha az egyik reláció kicsi, a másiknak pedig van egy indexe. Eléggé általános ezeknél, hogy jelentős és látható különbség van az argumentumok méreteiben, ugyanis egy összekapcsolásokat tartalmazó lekérdezés nagyon gyakran tartalmaz legalább egy attribútumra vonatkozó kiválasztást is, és az a kiválasztás nagyban csökkenti az egyik reláció becsült méretét.

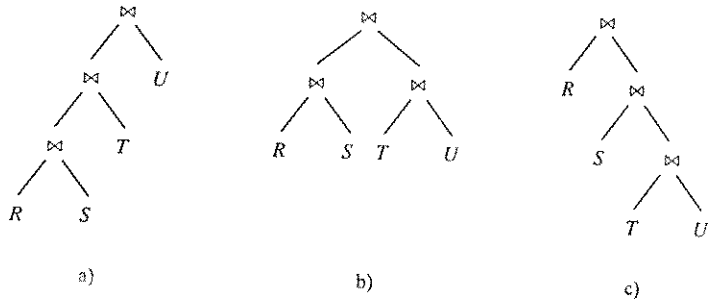
7.32. példa: Vegyük elő újra a 7.4. ábra következő lekérdezését:

```
SELECT filmCím
FROM SzerepelBenne, FilmSzínész
WHERE színészNév = név AND
   születési_idő LIKE '%1960';
```

Az ehhez tartozó, előnyben részesített logikai lekérdezéstervet a 7.21. ábrán látjuk, ahol a `SzerepelBenne` relációnak és a `FilmSzínész` relációra vonatkozó kiválasztás eredményének összekapcsolását vesszük. A `SzerepelBenne` és `FilmSzínész` relációk méreteire nem adtunk becslést, de feltételezhetjük, hogy egy adott évben született színészek kiválasztása a `FilmSzínész` reláció sorainak körülbelül 1/50 ré-

szét állítja elő. Mivel általában több színész szerepel egy-egy filmben, a `SzerepelBenne` reláció várhatóan nagyobb, mint a `FilmSzínész` reláció, elmondható tehát, hogy $\sigma_{születési\_idő\ \texttt{LIKE}\ \texttt{'\%1960'}}(\texttt{FilmSzínész})$, azaz az összekapcsolás második argumentuma sokkal kisebb, mint a `SzerepelBenne` első argumentum. Ebből azt a következtetést vonhatjuk le, hogy a 7.21. ábrán meg kellene fordítani az argumentumok sorrendjét, hogy a `FilmSzínész` reláción végrehajtott kiválasztás legyen a bal argumentum. □

Két reláció esetén az összekapcsolási fára mindössze két lehetséges választás létezik – a két reláció valamelyikét tesszük bal argumentummá. Gyorsan nő a lehetséges összekapcsolási fák száma, ha az összekapcsolás kettőnél több relációt foglal magában. A 7.28. ábra például három lehetséges szerkezetű fát mutat arra az esetre, amikor az *R*, *S*, *T* és *U* négy relációt kapcsoljuk össze. Mind a három bemutatott fában azonban ábécérendben szerepel ez a négy reláció balról jobbra nézve. Mivel számít az argumentumok sorrendje, és *n* dolgot *n*! módon rendezhetünk, a levelek címkézési lehetőségeit tekintve mindegyik fa 4! = 24 különböző fát képvisel.



7.28. ábra. *Változatok négy reláció összekapcsolására*

7.6.3. Bal-mély összekapcsolási fák

A 7.28.a) ábra egy példája annak, amit *bal-mély* fának nevezünk. Általánosan mondva, egy bináris fa bal-mély, ha minden jobb gyerek levél. Hasonlóképpen, egy olyan fát, mint a 7.2.8.c) ábrán látható, amelynek minden bal gyereke levél, *jobb-mély* fának nevezünk. Egy olyan fát, mint a 7.2.8.b) ábrán látható, amely se nem bal-mély, se nem jobb-mély, *bozótszerűnek* (bushy) nevezünk. Kettős előnye is van annak, ha lehetséges összekapcsolási sorrendként csak bal-mély fákat veszünk figyelembe, amint ezt az alábbi érveink alátámasztják:

1. Egy adott számú levéllel rendelkező lehetséges bal-mély fák száma nagy, de közel sem olyan nagy, mint a lehetséges fák teljes száma. Nagyobb lekérdezésekhez tartozó lekérdezésterveket kereshetünk, ha a keresést a bal-mély fákra korlátozzuk.

2. Az összekapcsolásokhoz használt bal-mély fák jól együttműködnek a szokásos összekapcsolási algoritmusokkal – különösen a beágyazott ciklusú összekapcsolásokkal és az egymenetes összekapcsolásokkal. A bal-mély fákon alapuló kereséstervek plusz ezek az algoritmusok várhatóan hatékonyabbak lesznek, mint amikor ugyanezeket az algoritmusokat nem bal-mély fákkal használjuk.

Egy bal- vagy jobb-mély összekapcsolási fa „levelei" valójában lehetnek olyan közbülső csomópontok is, ahol az ott szereplő operátor nem összekapcsolás. Technikai szempontból a 7.21. ábra például egy bal-mély összekapcsolási fa egyetlen összekapcsolás operátorral. Az, hogy az összekapcsolás jobb argumentumára egy kiválasztást alkalmazunk, nem változtat azon a tényen, hogy a fa a bal-mély fák osztályába tartozik.

A bal-mély fák száma közel sem olyan mértékben nő, mint egy adott számú relációt magában foglaló sokargumentumos összekapcsoláshoz tartozó összes fa száma. n relációhoz egyetlen bal-mély fa forma létezik, amelyhez $n!$ módon rendelhetjük hozzá a relációkat. Ugyanennyi jobb-mély fa létezik n relációhoz. Az n relációhoz tartozó fa formák teljes számát viszont, amit $T(n)$-nel jelölünk, az alábbi módon kapjuk meg:

$$T(1) = 1$$
$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i)$$

A második egyenlőséget az magyarázza, hogy egy tetszőleges 1 és $n-1$ közé eső i számnak vehetjük a gyökér bal oldali részfájában szereplő levelek számát, és azok a levelek $T(i)$ különböző módon rendezhetők el. Hasonlóképpen, a jobb oldali részfában szereplő $n-i$ számú levél $T(n-i)$ módon rendezhető el.

Íme a $T(n)$ első néhány értéke: $T(1) = 1$, $T(2) = 1$, $T(3) = 2$, $T(4) = 5$, $T(5) = 14$ és $T(6) = 42$. A fák teljes száma, amikor már a relációkat is hozzárendeljük a levelekhez, a $T(n)$ és az $n!$ szorzataként adódik. A 6 levelet tartalmazó megcímkézett levelű fák száma $42 \times 6! = 30\,240$, amelyek között van 720 bal-mély fa, és másik 720 jobb-mély fa.

Nézzük most a bal-mély összekapcsolási fákkal kapcsolatban említett második előnyt, nevezetesen, hogy várhatóan hatékony terveket állítanak elő. Két példát közlünk erre vonatkozóan:

1. Ha egymenetes összekapcsolásokat használunk, és az építő reláció a bal oldalon van, akkor az egy időben szükséges memóriamennyiség feltehetően kisebb, mint ha egy jobb-mély vagy bozótszerű fát használnánk ugyanazokra relációkra.
2. Ha iterátorokkal megvalósított beágyazott ciklusú összekapcsolásokat használunk, akkor elkerüljük azt, hogy valamely közbülső relációt egynél többször kelljen létrehozni.

7.33. példa: Vegyük a 7.28.a) ábrán látható bal-mély fát, és tegyük fel, hogy mind a három összekapcsoláshoz egy egyszerű egymenetes összekapcsolást fogunk használni, mégpedig úgy, hogy a bal argumentum az építő reláció, vagyis a bal argumentumokat tartjuk a központi memóriában. Az $R \bowtie S$ kiszámításához az R relációt kell a központi memóriában tartanunk, és amint kiszámítottuk az $R \bowtie S$-t, annak eredményét is a központi memóriában kell tartanunk. Tehát $B(R) + B(R \bowtie S)$ számú központi memóriapufferre van szükségünk. Ha R-et vesszük a legkisebb relációnak, és egy kiválasztás eléggé kicsivé tette az R-et, akkor valószínűleg nem lesz probléma, hogy rendelkezésre álljon ennyi puffer.

Az $R \bowtie S$ kiszámítása után az eredményül kapott relációt össze kell kapcsolni a T-vel. Az R tárolásához használt pufferekre már nincs tovább szükség, így azok újra felhasználhatók a $(R \bowtie S) \bowtie T$ eredményének tárolásához. Hasonlóképpen, amikor ezt a relációt kapcsoljuk össze az U-val, az $R \bowtie S$ relációra már nincs tovább szükség, így az ehhez használt pufferek újra felhasználhatók a végső összekapcsolás eredményének tárolásához. Általánosan azt mondhatjuk, hogy egy egymenetes összekapcsolással kiszámított bal-mély összekapcsolási fa feltételezi, hogy legalább két ideiglenes reláció tárolásához szükséges hely mindig rendelkezésre áll a központi memóriában.

Most vizsgáljuk meg a 7.28.c) ábrán található jobb-mély fa egy hasonló megvalósítását. Elsőként az R relációt kell betölteni a központi memória puffereibe, mivel mindig a bal argumentum az építő reláció. Ezután létre kell hozni az $S \bowtie (T \bowtie U)$ relációt, és a gyökérben lévő összekapcsoláskor ezt kell használni vizsgáló relációként. Az $S \bowtie (T \bowtie U)$ kiszámításához be kell vinni az S-et pufferekbe, és ki kell számolni a $T \bowtie U$ összekapcsolást mint a hozzá tartozó vizsgáló relációt. A $T \bowtie U$ kiszámításához viszont először pufferekbe kell betölteni a T-t. Most tehát az R, S és T három relációt tároljuk egy időben a központi memóriában. Általánosan fogalmazva, ha egy n levéllel rendelkező jobb-mély összekapcsolási fát akarunk kiértékelni, akkor egyszerre $n-1$ relációt kell a központi memóriában tárolni.

Természetesen előfordulhat, hogy a $B(R) + B(S) + B(T)$ összméret kisebb, mint a bal-mély fa kiszámítása során a két közbülső lépéshez szükséges helymennyiség, amelyek rendre $B(R) + B(R \bowtie S)$ és $B(R \bowtie S) + B((R \bowtie S) \bowtie T)$.[9] Ahogy azonban a 7.32. példában rámutattunk, a több összekapcsolást tartalmazó lekérdezéseknél gyakran van egy kis reláció, amely lehet a legbalra eső argumentum egy bal-mély fában. Ha R kicsi, akkor az $R \bowtie S$-től azt várhatjuk, hogy lényegesen kisebb, mint az S, az $(R \bowtie S) \bowtie T$-től pedig azt, hogy kisebb, mint a T, ami csak további igazolását jelenti a bal-mély fák használatának. □

7.34. példa: Tegyük fel, hogy a 7.28. ábra négyes összekapcsolását beágyazott ciklusú összekapcsolásokkal szándékozzuk megvalósítani, és hogy az abban szereplő mindhárom összekapcsoláshoz egy-egy iterátort használunk. Az egyszerűség kedvéért azt is tegyük fel, hogy az R, S, T és U relációk mindegyike tárolt reláció, nem pedig kifejezés. Ha a 7.28.a) ábrán szereplő bal-mély fát használjuk, akkor a gyökérhez tartozó iterátor az $(R \bowtie S) \bowtie T$ bal argumentum egy, a központi memória méretének megfelelő darabját kapja meg. Majd ezt a darabot összekapcsolja a teljes U-val, de amennyiben az U egy tárolt reláció, az U-t csak végig kell olvasni, és nem kell előállítani. Amikor megkapja és a memóriába helyezi a bal argumentum következő darabját, újra be kell olvasni az U-t, de a beágyazott ciklusú összekapcsolásnál szükség van erre az ismétlésre, és nem kerülhető el, ha mindkét argumentum nagy.

[9] Vegyük észre, hogy egy kifejezésfa eredményének tárolási költségét nem számoljuk bele a költség becslésébe, ahogy ezt máskor sem tettük.

A pufferkezelő szerepe

Észreveheti az olvasó a különbséget a 6.14. és 6.17. példában látott megközelítések között, ahol az összekapcsoláshoz rendelkezésre álló központi memóriapufferek számára egy rögzített korlátot feltételeztünk, és az itteni rugalmasabb feltételezés között, hogy annyi puffer áll rendelkezésre, amennyi szükséges, de próbálunk nem „túl sokat" használni. Elevenítsük fel a 6.8. részben mondottakat, miszerint a pufferkezelő jelentős rugalmassággal rendelkezik a műveletekhez szükséges pufferek kiosztása ügyében. Ütközések állhatnak azonban elő, ha egyszerre túl sok puffer kerül kiosztásra, ami leronthatja a használt algoritmus feltételezett hatékonyságát.

Ehhez hasonlóan, az $(R \bowtie S) \bowtie T$ egy darabjának megszerzéséhez megkapjuk az $R \bowtie S$ egy darabját a memóriában, és átolvassuk a T-t. A T többszöri átolvasására szükség lehet, hacsak nem elkerülhető. Végül az $R \bowtie S$ egy darabjának megszerzése az R egy darabjának beolvasását és S-sel történő összehasonlítását igényli, esetleg többször. Mindezen tevékenységek során azonban csak tárolt relációkat olvasunk többször, és ez az ismételt olvasás akkor válik a beágyazott ciklusú összekapcsolás működésének részévé, amikor a központi memória nem elég nagy ahhoz, hogy egy teljes reláció elférjen benne.

Hasonlítsuk most össze a bal-mély fához tartozó iterátorok viselkedését a 7.28.c) ábrán szereplő jobb-mély fára vonatkozó iterátorok viselkedésével. A gyökérhez tartozó iterátor az R egy darabjának beolvasásával kezd. Majd meg kell konstruálnia a teljes $S \bowtie (T \bowtie U)$ relációt, és azt össze kell hasonlítania az R megkapott darabjával. Amikor az R következő darabját olvassuk be a memóriába, az $S \bowtie (T \bowtie U)$ relációt újra létre kell hozni. Az R minden egyes további darabjának beolvasásakor újra és újra elő kell állítani ugyanezt a relációt.

Természetesen azt megtehetnénk, hogy az $S \bowtie (T \bowtie U)$ relációt egyszer létrehozzuk és eltároljuk vagy a memóriában, vagy lemezen. Ha lemezen tároljuk el, akkor a bal-mély fának megfelelő tervhez képest extra lemez I/O-műveleteket használunk, ha pedig a memóriában tároljuk, akkor belefutunk a memória túlhasználatának a 7.33. példában tárgyalt problémájába. □

7.6.4. Dinamikus programozás az összekapcsolási sorrend és csoportosítás megválasztására

Több reláció összekapcsolásakor a sorrend megválasztására három lehetőségünk van:

1. Az összes lehetséges sorrendet figyelembe vesszük.
2. Egy részhalmazt veszünk figyelembe.
3. Használunk egy heurisztikát egy sorrend kiválasztására.

Ebben a részben a felsorolás egy ésszerű megközelítését tárgyaljuk, amit *dinamikus programozásnak* nevezünk. Ez vagy arra használható, hogy az összes sorrendet megvizsgáljuk, vagy arra, hogy csak bizonyos részhalmazokat vizsgáljunk meg, mint amilyen például a bal-mély fákra leszűkített sorrendek részhalmaza. A 7.6.6. részben egy olyan elfogadható heurisztikát nézünk majd meg, amely egyetlen sorrend kiválasztására szolgál. A dinamikus programozás egy általános algoritmusminta.[10] A dinamikus programozás mögött az az alapötlet áll, hogy kitöltünk egy táblázatot a költségekről úgy, hogy csak a sikeres következtetéshez szükséges minimális információt őrizzük meg.

Tegyük fel, hogy végre akarjuk hajtani az $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$ összekapcsolást. A dinamikus programozás algoritmusában létrehozunk egy olyan táblázatot, amelyben van egy bejegyzés az n reláció közül egyet vagy többet tartalmazó minden egyes részhalmazhoz. Ebben a táblázatban az alábbiakat helyezzük el:

1. Ezen relációk összekapcsolásának becsült mérete. Ennek meghatározásához a 7.4.6. részben szereplő formulát használhatjuk.
2. Ezeknek a relációknak az összekapcsolásához szükséges legkisebb költség. A példáinkban a közbülső relációk méreteinek összegét fogjuk használni (nem beleértve magukat az R_i relációkat és a táblázat aktuális bejegyzéséhez tartozó összes reláció összekapcsolását). Emlékezzünk vissza, hogy a köztes relációk méretei adják a legegyszerűbb használható becslést a lemez I/O-műveletek, CPU-használat és egyéb tényezők költségére. Más, bonyolultabb becslést is használhatnánk azonban, mint például a teljes lemez I/O-műveletek száma, ha hajlandók és képesek lennénk a szükséges extra számítások elvégzésére. Ha a lemez I/O-műveleteket vagy a futási idő más becslését használjuk, akkor figyelembe kell venni az adott összekapcsoláshoz használt algoritmust is, mivel a különböző algoritmusok különböző költséggel járnak. Ezeket a kérdéseket a dinamikus programozási technikák alapjainak megtanulása után fogjuk megtárgyalni.
3. Az a kifejezés, amelyik a legkisebb költséget adja. Ez a kifejezés a kérdéses relációkat kapcsolja össze, valamilyen csoportosításban. Opcionálisan korlátozhatjuk magunkat a bal-mély kifejezésekre, amikor is a kifejezés csak egy sorrendje a relációknak.

A táblázatot a részhalmaz méretére vonatkozó indukció alapján konstruáljuk meg. Két változat létezik, attól függően, hogy a fák összes lehetséges alakjait figyelembe kívánjuk-e venni, vagy csak a bal-mély fákat. Különbség van ugyanis a táblázat megalkotásának módjában, amely különbséget majd akkor magyarázzuk el, amikor a táblázat megkonstruálásának az indukciós lépését tárgyaljuk.

Alap: Az egyetlen R relációhoz tartozó bejegyzés az R méretéből, egy 0 költségből és abból a formulából áll, ami maga az R. Relációk egy $\{R_i, R_j\}$ párjára is könnyű ki-

[10] A dinamikus programozás általános tárgyalása megtalálható a következő irodalomban is: A. V. Aho, J. E. Hopcroft and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1984.

számítani a bejegyzést. A költség 0, mivel nincsenek közbülső relációk. A méret becslését a 7.4.6. részben szereplő szabály adja meg: az R_i és R_j méreteit összeszorozzuk, majd az R_i és R_j minden közös attribútuma esetén, ha van ilyen, osztunk az attribútumhoz tartozó értékhalmazok méretei közül a nagyobbal. A formula vagy $R_i \bowtie R_j$, vagy $R_j \bowtie R_i$. A 7.6.1. részben bevezetett elvnek megfelelően az R_i és R_j közül a kisebbet vesszük bal argumentumnak.

Indukció: Most építhetjük tovább a táblázatot, kiszámítva a bejegyzéseket minden részhalmazhoz, amelyek mérete 3, 4 és így tovább, amíg el nem jutunk az egyetlen n méretű részhalmaz bejegyzéséhez. Egy ilyen bejegyzés mondja meg, hogy melyik a legjobb módszer az ott szereplő összes reláció összekapcsolásának kiszámítására. Megadja továbbá annak a módszernek a becsült költségét is, amire szükségünk van, amikor a későbbi bejegyzéseket számoljuk. Azt kell megnézni, hogy hogyan számítunk ki egy k relációból álló $\mathcal{R}$ halmazhoz tartozó bejegyzést.

Ha csak bal-mély fákat szándékozunk figyelembe venni, akkor a k relációból álló $\mathcal{R}$ minden egyes R relációja esetén azt a lehetőséget vesszük, amikor az $\mathcal{R}$-hez tartozó összekapcsolást úgy számoljuk ki, hogy először kiszámítjuk az $\mathcal{R} - \{R\}$-hez tartozó összekapcsolást, majd ezt összekapcsoljuk az R-rel. Az $\mathcal{R}$-hez tartozó összekapcsolás költsége az $\mathcal{R} - \{R\}$-hez tartozó költségnek és az utóbbi összekapcsolás méretének összege lesz. A legkisebb költséget eredményező R-et választjuk. Az $\mathcal{R}$-hez tartozó kifejezés bal argumentuma az $\mathcal{R} - \{R\}$-hez tartozó legjobb összekapcsolási kifejezés, a jobb argumentuma pedig az R lesz. Az $\mathcal{R}$-hez tartozó méret az lesz, amit a 7.4.6. rész formulája ad.

Ha az összes fát figyelembe akarjuk venni, akkor a relációk egy $\mathcal{R}$ részhalmazához tartozó bejegyzés kiszámítása valamivel összetettebb. Meg kell vizsgálni az összes lehetséges módját az $\mathcal{R}$ halmaz $\mathcal{R}_1$ és $\mathcal{R}_2$ teljesen elkülönülő részhalmazokra történő particionálásának. Minden ilyen particionálás esetén vesszük az alábbiak összegét:

1. Az $\mathcal{R}_1$ és $\mathcal{R}_2$ legjobb költségei.
2. Az $\mathcal{R}_1$ és $\mathcal{R}_2$ méretei.

A legjobb költséget adó partíció esetén ezt az összeget az $\mathcal{R}$-hez tartozó költségnek vesszük. Az $\mathcal{R}$-hez tartozó formula az $\mathcal{R}_1$-hez és az $\mathcal{R}_2$-höz tartozó legjobb sorrendű összekapcsolások összekapcsolása lesz.

7.35. példa: Tekintsük az R, S, T és U négy reláció összekapcsolását. Az egyszerűség kedvéért feltételezzük, hogy mindegyiknek 1000 sora van. A relációk attribútumait és az egyes relációkban az attribútumokhoz tartozó értékhalmazok becsült méreteit a 7.29. ábra adja meg.

Az egyelemű halmazokra vonatkozó méreteket, költségeket és legjobb terveket a 7.30. ábrán látható táblázat tartalmazza. Minden egyes relációra a méret a megadott 1000, a költség 0, mivel nincsenek közbülső relációk, a legjobb (és az egyetlen) kifejezés pedig maga a reláció.

$R(a, b)$	$S(b, c)$	$T(c, d)$	$U(d, a)$
$V(R, a) = 100$			$V(U, a) = 50$
$V(R, b) = 200$	$V(S, b) = 100$		
	$V(S, c) = 500$	$V(T, c) = 20$	
		$V(T, d) = 50$	$V(U, d) = 1000$

7.29. ábra. *Paraméterek a 7.35. példához*

	$\{R\}$	$\{S\}$	$\{T\}$	$\{U\}$
Méret	1000	1000	1000	1000
Költség	0	0	0	0
Legjobb terv	R	S	T	U

7.30. ábra. *Az egyelemű halmazokhoz tartozó táblázat*

Vegyük sorra most a relációpárokat. Mindegyikhez 0 költség tartozik, hiszen két reláció összekapcsolásakor még mindig nincsenek közbülső relációk. Két lehetséges terv létezik egy-egy párhoz, a két reláció valamelyike lehet a bal argumentum, de mivel mindegyik reláció ugyanolyan méretű, nincs szempont a kezünkben, ami alapján a tervek közül választhatnánk. Az ábécérendnek megfelelő első relációt vesszük hát bal argumentumnak mindegyik esetben. Az eredményül kapott relációk méreteit a szokásos képlet alapján számoljuk ki. A 7.31. ábrán bemutatott táblázat összegzi mindezeket az eredményeket. Megjegyezzük, hogy az 1 M (Mega) jelentése: 1 000 000, az olyan „összekapcsolások" mérete, amelyek valójában szorzatok.

	$\{R, S\}$	$\{R, T\}$	$\{R, U\}$	$\{S, T\}$	$\{S, U\}$	$\{T, U\}$
Méret	5000	1 M	10 000	2000	1M	1000
Költség	0	0	0	0	0	0
Legjobb terv	$R \bowtie S$	$R \bowtie T$	$R \bowtie U$	$S \bowtie T$	$S \bowtie U$	$T \bowtie U$

7.31. ábra. *A relációpárokhoz tartozó táblázat*

Nézzük most a négy reláció közül hármat magában foglaló összekapcsolásokhoz tartozó táblázatot. Három reláció összekapcsolásának kiszámításakor először ki kell választani közülük kettőt, amelyeket összekapcsolunk. Az eredmény méretének becslését a szokásos formulával számoljuk ki, ennek a számításnak a részleteitől most eltekintünk. Emlékezzünk vissza, hogy ugyanazt a méretet kapjuk, függetlenül az összekapcsolás kiszámításának módjától.

Minden relációhármas esetén a költség az egyetlen közbülső relációnak – az elsőként kiválasztott két reláció összekapcsolásának – a mérete. Mivel azt akarjuk, hogy ez a költség a lehető legkisebb legyen, megvizsgáljuk a három relációból származó összes lehetséges relációpárt, és a legkisebb méretet eredményező párt választjuk.

A formula meghatározásakor először a két választott relációt csoportosítjuk egybe, melyek még lehetnek vagy bal, vagy jobb argumentumok. Tegyük fel, hogy csak a bal-mély fákat vesszük figyelembe, vagyis a első két reláció összekapcsolását mindig bal argumentumként használjuk. Mivel két reláció összekapcsolásának becsült mérete

minden esetben legalább 1000 (az egyes relációk mérete), ha megengednénk nem bal-mély fákat, akkor mindig az egyedül álló relációt választanánk bal argumentumnak a példánkban. A hármasokhoz tartozó táblázatot a 7.32. ábra mutatja.

	$\{R, S, T\}$	$\{R, S, U\}$	$\{R, T, U\}$	$\{S, T, U\}$
Méret	10 000	50 000	10 000	2000
Költség	2 000	5 000	1 000	1000
Legjobb terv	$(S \bowtie T) \bowtie R$	$(R \bowtie S) \bowtie U$	$(T \bowtie U) \bowtie R$	$(T \bowtie U) \bowtie S$

7.32. ábra. *A relációhármasokhoz tartozó táblázat*

Példaként vizsgáljuk meg az $\{R, S, T\}$ hármashoz tartozó számításokat. Meg kell néznünk az ezekből képezhető három pár mindegyikét. Ha az $R \bowtie S$-sel kezdünk, akkor a költség ennek a relációnak a mérete, ami 5000, amint ez a 7.31. ábrán látható, a párokra vonatkozó táblázat alapján tudjuk. Az $R \bowtie T$ kezdés 1 000 000-t ad költségként, ha pedig az $S \bowtie T$-vel kezdünk, akkor a költség 2000. A három lehetőség közül a legutóbbi jelenti a legalacsonyabb költséget, így azt a tervet választjuk. Ez a választás nemcsak az $\{R, S, T\}$ oszlop költség rovatában tükröződik, hanem a legjobb terv sorban is, ahol is az a terv jelenik meg, amelyik először az S-et és a T-t csoportosítja össze.

Most azt a helyzetet vizsgáljuk meg, amikor mind a négy relációt összekapcsoljuk. Ennek a relációnak a becsült mérete 100 sor, az igazi költség lényegében a közbülső relációk létrehozásában áll. Emlékezzünk azonban, hogy tervek összehasonlításakor a végeredmény méretét egyébként sem vesszük figyelembe soha.

A mind a négy relációt magában foglaló összekapcsolás kiszámításának két általános módja van:

1. Kiválasztunk és a lehető legjobb módon összekapcsolunk hármat, majd az eredményt összekapcsoljuk a negyedikkel,
2. A négy relációt kétszer két párra osztjuk, összekapcsoljuk a párokat, majd az eredményeket összekapcsoljuk.

Természetesen, ha csak bal-mély fákkal foglalkozunk, akkor a második típusú terveket kizárjuk, hiszen azok bozótszerű fákat eredményeznek. A 7.33. ábrán látható táblázat összefoglalja az összekapcsolások hét lehetséges csoportosítását, amelyek a 7.31. és 7.32. ábrákon szereplő legjobb csoportosításokra épülnek.

Csoportosítás	*Költség*
$((S \bowtie T) \bowtie R) \bowtie U$	12 000
$((R \bowtie S) \bowtie U) \bowtie T$	55 000
$((T \bowtie U) \bowtie R) \bowtie S$	11 000
$((T \bowtie U) \bowtie S) \bowtie R$	3 000
$(T \bowtie U) \bowtie (R \bowtie S)$	6 000
$(R \bowtie T) \bowtie (S \bowtie U)$	2 000 000
$(S \bowtie T) \bowtie (R \bowtie U)$	12 000

7.33. ábra. *Összekapcsolások csoportosításai és azok költségei*

Nézzük például a 7.33. ábrán található első formulát, amely azt ábrázolja, hogy először összekapcsoljuk az R, S és T relációkat, majd az így kapott eredményt összekapcsoljuk az U-val. A 7.32. ábra táblázatából tudjuk, hogy az R, S és T összekapcsolásának a legjobb módja az, ha először az S-et és a T-t kapcsoljuk össze. Ennek a kifejezésnek a bal-mély alakját használtuk, és hogy a bal-mély formát folytassuk, az U-t jobb oldalról kapcsoltuk hozzá. Ha csak bal-mély fákat engedünk meg, akkor ez a kifejezés és reláció sorrend az egyetlen választási lehetőség. Ha megengednénk bozótszerű fákat is, akkor az U-t bal oldalról kapcsolnánk, ugyanis kisebb, mint a másik három összekapcsolása. Az általunk kapott összekapcsolás költsége 12 000, ami az $(S \bowtie T) \bowtie R$ költségének és méretének az összege, amelyek rendre 2000 és 10 000.

A 7.33. ábra utolsó három kifejezése további lehetőségeket ad, amennyiben bozótszerű fákat is figyelembe veszünk. Ezeket úgy kapjuk, hogy először két relációpárt kapcsolunk össze. Az utolsó sor például azt a megoldást mutatja, hogy először elvégezzük az $R \bowtie U$ és az $S \bowtie T$ összekapcsolásokat, majd összekapcsoljuk azok eredményeit. Ennek a kifejezésnek a költsége a két pár méretének és költségének az összegeként adódik. Mindkét költség 0, amint ez minden pár esetében így kell hogy legyen, a méretek pedig rendre 10 000, illetve 2000. Mivel általában a kisebb relációt választjuk bal argumentumnak, végül az $(S \bowtie T) \bowtie (R \bowtie U)$ kifejezést kapjuk.

Azt látjuk ebben a példában, hogy a legkisebb költség a negyedik formulához tartozik: $((T \bowtie U) \bowtie S) \bowtie R$. Ezt a kifejezést választjuk ki az összekapcsolás kiszámítására, és ennek költsége 3000. Mivel ez egy bal-mély fa, ez a logikai lekérdezésterv kerül kiválasztásra, függetlenül attól, hogy a mi dinamikus programozási stratégiánk mindenféle típusú tervet figyelembe vesz, vagy csak a bal-mély terveket. □

7.6.5. Dinamikus programozás részletesebb költségfüggvényekkel

Egy dinamikus programozási algoritmusban leegyszerűsíti a számításokat az, ha költségbecslésként reláció méreteket használunk. Ennek az egyszerűsítésnek egy hátránya azonban az, hogy a számolás során az összekapcsolásoknak nem a tényleges költségeit vesszük figyelembe. Erre egy szélsőséges példa, amikor egy $R(a, b) \bowtie S(b, c)$ összekapcsolásban az R relációnak egyetlen sora van, az S relációnak pedig van egy indexe a b összekapcsolási attribútumra, ekkor ugyanis az összekapcsolás szinte 0 időbe kerül. Másrészről viszont, ha az S-nek nincs indexe, akkor végig kell azt olvasni, ami $B(S)$ lemez I/O-műveletet igényel, még akkor is, ha az R egyelemű. Egy olyan költségbecslés, ami csak az R, S és $R \bowtie S$ méreteit veszi figyelembe, nem tud különbséget tenni e két eset között, vagyis a csoportosításban az $R \bowtie S$ használatának költségét vagy túlbecsüljük, vagy alulbecsüljük.

Nem nehéz azonban a dinamikus programozási algoritmust úgy módosítani, hogy az összekapcsolási algoritmusokat is számításba vegye. Először is, a használt költségbecslés a lemez I/O-műveletek száma lesz, vagy valamilyen más általunk előnyben részesített futási idő egység. Az $\mathcal{R}_1 \bowtie \mathcal{R}_2$ költségének kiszámításakor összeadjuk az $\mathcal{R}_1$ költségét, az $\mathcal{R}_2$ költségét és e két reláció összekapcsolásának legkisebb költségét, a legjobb rendelkezésre álló algoritmust használva. Mivel az utóbbi költség rendszerint

függ az $\mathcal{R}_1$ és $\mathcal{R}_2$ méreteitől, ezeknek a méreteknek a becslését szintén ki kell számolni, ahogy ezt a 7.35. példában tettük.

A dinamikus programozás egy még hatékonyabb változata a 7.5.4. részben említett Selinger-féle optimalizálásra épül. Ekkor az egyes relációhalmazok esetén, amelyek összekapcsolásra számot tarthatnak, nemcsak egy költséget tartunk meg, hanem több költséget. Emlékezzünk vissza, hogy a Selinger-féle optimalizálás az összekapcsolás eredményének előállításakor nemcsak annak legkisebb költségét veszi figyelembe, hanem azokat a legkisebb költségeket is, amelyek a reláció különféle „érdekes" rendezettségű változatainak előállításához szükségesek. Ezek az érdekes rendezettségek lehetnek olyanok, amelyek egy későbbi rendezéses összekapcsolás során előnyösen használhatók, vagy olyanok, amelyek felhasználhatók a teljes lekérdezés végeredményének előállításakor, ha a felhasználó valamilyen sorrendnek megfelelő rendezettségben várja a végeredményt. Amikor rendezett relációkat kell előállítani, rendezéses összekapcsolás – legyen az egymenetes vagy többmenetes – használatát figyelembe kell venni mint egy lehetőséget, ha viszont egy eredmény rendezettsége nem értékes szempont, akkor a tördelő összekapcsolások minden esetben legalább olyan jók, mint a megfelelő rendezéses összekapcsolások.

7.6.6. Egy mohó algoritmus az összekapcsolási sorrend kiválasztására

Ahogy a 7.35. példa is mutatja, még a dinamikus programozás gondosan korlátozott keresése is exponenciális számú számításhoz vezet, mint az összekapcsolandó relációk számának egy függvénye. Öt vagy hat reláció összekapcsolásakor az optimális sorrend megtalálásához indokolt egy olyan alapos módszert használni, mint amilyen a dinamikus programozás vagy az elágazás-és-korlátozás keresés. Amikor azonban az összekapcsolások száma túlnő ezen, vagy ha az alapos kereséshez szükséges időt nem akarjuk ráfordítani, akkor használhatunk heurisztikus összekapcsolási sorrendet a lekérdezésoptimalizálóban.

A leggyakrabban választott heurisztika valamilyen *mohó* (greedy) algoritmus, ahol egy adott ponton döntést hozunk az összekapcsolási sorrendre vonatkozóan, és soha nem lépünk vissza vagy vizsgáljuk felül az egyszer már meghozott döntéseket. Egy olyan mohó algoritmust vizsgálunk meg, amely csak bal-mély fákat választ ki. A „mohóság" alapja az az elv, hogy a köztes relációk méretét a lehető legalacsonyabban akarjuk tartani a fa minden szintjén.

Alap: Kezdjük azzal a relációpárral, amelyek összekapcsolásának becsült mérete a legkisebb. Ezeknek a relációknak az összekapcsolás lesz az *aktuális fa* (current tree).

Indukció: A aktuális fában még nem szereplő relációk közül keressük meg azt, amelyiket az aktuális fával összekapcsolva a legkisebb becsült méretű relációt kapjuk. Az új aktuális fa bal argumentuma a régi aktuális fa, jobb argumentuma pedig a kiválasztott reláció lesz.

Összekapcsolás szelektivitása

Hasznos lehet úgy szemlélni az olyan heurisztikákat, mint amilyen a bal-mély fát kiválasztó mohó algoritmus, hogy minden R reláció rendelkezik egy *szelektivitással*, amikor összekapcsoljuk az aktuális fával, ami a következő hányados:

$$\frac{\text{az összekapcsolás eredményének mérete}}{\text{az aktuális fa eredményének mérete}}$$

Mivel rendszerint egyik relációnak sem ismerjük a pontos méretét, ezeket a méreteket becsüljük, mint ahogy ezt az előzőekben is tettük. Az összekapcsolási sorrend egy mohó megközelítése az, hogy a legkisebb szelektivitással rendelkező relációt választjuk ki.

Ha például egy összekapcsolási attribútum kulcsa az R-nek, akkor a szelektivitás legfeljebb 1, ami általában egy kedvező szituáció. Megjegyezzük, hogy a 7.29. ábra statisztikájából ítélve, a d attribútum kulcs az U-ban, és a többi relációnak nincs kulcsa, ami megmagyarázza, hogy miért az a legjobb kezdés, hogy a T-t összekapcsoljuk az U-val.

7.36. példa: Alkalmazzuk a mohó algoritmust a 7.35. példa relációira. Az alaplépés az, hogy megkeressük azt a relációpárt, amelyeknek az összekapcsolása a legkisebb méretű. A 7.31. ábrát megnézve azt látjuk, hogy ez a $T \bowtie U$ összekapcsolásra igaz, amelynek költsége 1000. Vagyis $T \bowtie U$ az „aktuális fa".

Most azt nézzük meg, hogy az R vagy az S relációt kapcsoljuk-e össze a fával következőként. Összehasonlítjuk tehát a $(T \bowtie U) \bowtie R$, illetve a $(T \bowtie U) \bowtie S$ méretét. A 7.32. ábra azt mondja, hogy a második, amelynek mérete 2000, jobb, mint az első, amelynek mérete 10 000. Az új aktuális fa ennek megfelelően a $(T \bowtie U) \bowtie S$ lesz.

Nincs más lehetőségünk, mint hogy az utolsó lépésben az R-t kapcsoljuk össze az eddigivel, aminek során a teljes költség 3000 lesz, ami egyenlő a két közbülső reláció méretének összegével. Vegyük észre, hogy a mohó algoritmus ugyanazt a fát adja eredményül, mint amit a 7.35. példában a dinamikus programozási algoritmus kiválasztott. Léteznek azonban olyan példák, amikor a mohó algoritmus nem találja meg a legjobb megoldást, a dinamikus programozási algoritmus viszont garantáltan megtalálja a legjobbat; lásd 7.6.4. feladat. □

7.6.7. Feladatok

7.6.1. feladat: Vegyük a 7.4.1. feladatban szereplő relációkat, és adjuk meg a dinamikus programozási táblázat bejegyzéseit, amelyek a lehetséges összekapcsolási sorrendek kiértékeléséhez tartoznak, figyelembe véve:

a) Csak bal-mély fákat engedünk meg.
b) Minden fa megengedett.

Melyek a legjobb választások az egyes esetekben?

7.6.2. feladat: Ismételjük meg a 7.6.1. feladatot az alábbi változtatásokkal:

i) A Z sémája $Z(d, a)$-ra változik.
ii) $V(Z, a) = 100$.

7.6.3. feladat: Ismételjük meg a 7.6.1. feladatot a 7.4.2. feladat relációival.

* **7.6.4. feladat:** Vegyük az $R(a, b)$, $S(b, c)$, $T(c, d)$ és $U(a, d)$ relációk összekapcsolását, ahol az R-nek és az U-nak 1000 sora van, az S-nek és a T-nek pedig 100 sora van. Továbbá, minden reláció minden attribútumának 100 értéke van, kivéve a c attribútumot, amelyre $V(S, c) = V(T, c) = 10$.

a) Milyen sorrendet választ ki a mohó algoritmus? Mi a hozzá tartozó költség?
b) Melyik az optimális összekapcsolási sorrend, és mi a hozzá tartozó költség?

7.6.5. feladat: Hány fa létezik

a) hét reláció,
b) nyolc reláció

összekapcsolása esetén? Ezek közül hány se nem bal-mély, se nem jobb-mély?

! 7.6.6. feladat: Tegyük fel, hogy össze akarjuk kapcsolni az R, S, T és U relációkat a 7.28. ábrán látható fastruktúrák valamelyikének megfelelően, és minden közbülső relációt a memóriában akarunk tartani mindaddig, ameddig szükség van rájuk. A szokásos feltevésünket követve, a négy reláció összekapcsolásának eredményét valamilyen további folyamat felhasználja amint azt előállítottuk, így ehhez a relációhoz nincs szükség memóriára. A tárolt relációkhoz és a köztes relációkhoz szükséges blokkok számával [pl. $B(R)$ vagy $B(R \bowtie S)$] kifejezve adjunk alsó korlátot M-re, a szükséges memóriablokkok számára, ha a kiértékelés az alábbi alapján történik:

* a) 7.28.a) ábrán szereplő bal-mély fa,
b) 7.28.b) ábrán szereplő bozótszerű fa,
c) 7.28.c) ábrán szereplő jobb-mély fa.

Milyen feltételezések alapján következtethetünk arra, hogy egy fa biztosan kevesebb memóriát használ, mint valamelyik másik?

***! 7.6.7. feladat:** A táblázat hány bejegyzését kell kitöltenünk, ha k reláció összekapcsolásakor dinamikus programozást használunk az összekapcsolási sorrend kiválasztására?

7.7. A fizikai lekérdezésterv kiválasztásának befejezése

A lekérdezést elemeztük, átalakítottuk egy kiindulási logikai lekérdezéstervvé, és a 7.3. részben leírt transzformációk segítségével feljavítottuk a logikai lekérdezéstervet. A fizikai lekérdezésterv kiválasztási folyamatának részeként felsoroljuk a lehetséges választásokat, és becsüljük a hozzájuk tartozó költségeket, amit a 7.5. részben tárgyaltunk. A 7.6. rész központi kérdése a sok relációt magában foglaló összekapcsolások felsorolása, költségbecslése és összekapcsolási sorrendje volt. Mintegy ennek kiterjesztéseként, hasonló technikákat használhatunk egyesítések, metszetek vagy bármilyen más kommutatív/asszociatív művelet esetén.

Hátravan még néhány lépés ahhoz, hogy a logikai tervet egy teljes fizikai lekérdezéstervvé alakítsuk. Az alapvető témakörök, amelyekre még ki kell térnünk ebben a részben, a következők:

1. A lekérdezésterv műveleteit megvalósító algoritmusok kiválasztása, feltéve hogy az algoritmus kiválasztása még nem történt meg valamilyen korábbi lépésben. Utóbbira példa egy összekapcsolási sorrend megválasztása dinamikus programozással.
2. Annak eldöntése, hogy a köztes eredményeket mikor *materializáljuk* (előállítjuk teljes egészében és lemezen tároljuk), illetve mikor *„futószalagosítjuk"* (csak a központi memóriában állítjuk elő, és nem feltétlenül tartjuk egyben az egészet egyszerre).
3. Jelölésrendszer a fizikai lekérdezésterv operátorai számára. Ez magában foglalja a tárolt relációk elérési módszereivel, illetve a relációs algebra operátorait megvalósító algoritmusokkal kapcsolatos részleteket.

Az operátorok algoritmusainak kiválasztását nem fogjuk teljes egészében megtárgyalni. Ehelyett a két legfontosabb operátor ide vonatkozó kérdéseit nézzük meg: a kiválasztást a 7.7.1. részben és az összekapcsolásokat a 7.7.2. részben. Ezután, a 7.7.3. résztől a 7.7.5. részig, a futószalagos technika és a materializáció közötti választás kérdését vesszük szemügyre. A fizikai lekérdezéstervekkel kapcsolatos jelölésrendszert a 7.7.6. részben mutatjuk be.

7.7.1. Kiválasztási eljárás megválasztása

Amikor fizikai lekérdezéstervet választunk, az egyik legfontosabb lépés az, hogy minden egyes kiválasztás operátorhoz algoritmust jelölünk ki. A 6.3.1. részben beszéltünk a $\sigma_C(R)$ operátor triviális megvalósításáról, amikor is a teljes R relációhoz hozzáférünk, és megnézzük, hogy mely sorok elégítik ki a C feltételt. Majd a 6.7.2. részben azt a lehetőséget vizsgáltuk, amikor a C feltétel „attribútum egyenlő konstans" alakú volt, és volt egy index az adott attribútumra. Ha ez adott, akkor a feltétel-

nek megfelelő sorokat anélkül is megtalálhatjuk, hogy az R relációt egyáltalán megnéznénk. Tekintsük most ennek a problémának az általánosítását, nevezetesen amikor van egy olyan kiválasztási feltételünk, amely több feltétel konjunkciója (AND-je), amelyek között vannak „attribútum egyenlő konstans" alakúak és más összehasonlítások attribútumok és konstansok között, mint például a $<$.

Ha nincsenek több attribútumra vonatkozó multidimenzionális indexek, akkor a használandó stratégia egy vagy több olyan attribútum választását foglalja magában, amely

a) Rendelkezik indexszel, és
b) A kiválasztásban szereplő részfeltételek valamelyikében egy konstanssal van összehasonlítva.

Ekkor ezeket az indexeket használjuk az egyes feltételeket kielégítő sorok halmazainak a beazonosítására. A 4.2.3. és 5.1.5. rész tárgyalta azt, hogy az indexekből megkapott sormutatókat hogyan használhatjuk arra, hogy csak az olyan sorokat találjuk meg, amelyek az összes feltételnek eleget tesznek, még mielőtt azokat a sorokat a lemezről beolvasnánk.

Az egyszerűség kedvéért, több indexnek az ilyen módon történő használatát nem vizsgáljuk. Ehelyett inkább csak az alábbi típusú algoritmusokra korlátozzuk vizsgálódásunkat:

1. Egy $A\theta\ c$ alakú összehasonlítást használ, ahol A egy indexszel rendelkező attribútum, c egy konstans és θ egy összehasonlító operátor (= vagy $<$).
2. Visszaadja az 1.-ben szereplő összehasonlítást kielégítő sorokat, használva a 6.2.1. részben tárgyalt indexolvasás operátort.
3. A 2.-ben kapott minden sorra megvizsgálja, hogy kielégíti-e a kiválasztási feltétel fennmaradó részét. Az ezt a lépést végrehajtó fizikai operátort `Filter`nek fogjuk nevezni. A sorok kiválasztásához használt feltétel paramétere ennek az operátornak, nagy hasonlóságot mutatva így a relációs algebra σ operátorával.

Az ilyen típusú algoritmusokon túl szükség van egy olyan algoritmusra is, amely nem használ indexet, hanem a teljes relációt végigolvassa (a táblaolvasás fizikai operátor segítségével), és az egyes sorokat átadja a `Filter` operátornak a kiválasztási feltétel teljesülésének ellenőrzése céljából.

Hogy egy adott kiválasztást az algoritmusok közül melyikkel valósítjuk meg, azt úgy döntjük el, hogy becsüljük az adatok olvasásának költségét az egyes megoldások esetén. A lehetséges algoritmusok költségeinek összehasonlításakor már nem használható tovább a közbülső relációk méretein alapuló egyszerűsített költségbecslés. Ennek az az oka, hogy most a logikai lekérdezésterv egyetlen lépésének megvalósításával foglalkozunk, és a közbülső relációk függetlenek a megvalósítástól.

Újra a lemez I/O-műveletek számával fogunk tehát dolgozni, csakúgy mint a 6. fejezetben, amikor szintén algoritmusokat és hozzájuk tartozó költségeket tárgyaltunk. A korábbiakhoz hasonlóan azzal az egyszerűsítéssel élünk, hogy csak az adatblokkok

elérési költségét vesszük figyelembe, az indexblokkét nem. Ne feledjük el, hogy a szükséges indexblokkok száma általában sokkal kisebb, mint a szükséges adatblokkok száma, így a lemez I/O-költség e közelítése elég pontos.

Most pedig felvázoljuk, hogy hogyan lehet becsülni a különböző algoritmusok költségét. Feltesszük, hogy adott a $\sigma_C(R)$ művelet, ahol a C feltétel egy vagy több ÉS-sel összekötött összehasonlításból áll. Példaként az $a = 10$ és $b < 20$ összehasonlításokat használjuk mint az egyenlőségi feltétel, illetve az egyenlőtlenségi feltétel reprezentánsait.

1. A táblaolvasási algoritmus és egy hozzá társított szűrési lépés együttes költsége:

   a) $B(R)$, ha az R nyalábolt, és
   b) $T(R)$, ha az R nem nyalábolt.

2. Egy algoritmus, amely egy egyenlőséget vizsgáló összehasonlítást vesz, mint például $a = 10$, ahol az a attribútumhoz létezik index, először indexolvasást használ az illeszkedő sorok megtalálására, majd egy szűrést hajt végre a megkapott sorokon, hogy ellenőrizze, hogy azok a teljes C feltételt kielégítik-e. Az ilyen algoritmus költsége:

   a) $B(R)/V(R, a)$, ha az index nyalábolt, és
   b) $T(R)/V(R, a)$, ha az index nem nyalábolt.

3. Egy algoritmus, amely egy egyenlőtlenségi összehasonlítást vesz, mint például $b < 20$, ahol a b attribútumhoz létezik index, először indexolvasást használ az illeszkedő sorok megtalálására, majd egy szűrést hajt végre a megkapott sorokon, hogy ellenőrizze, hogy azok a teljes C feltételt kielégítik-e. Az ilyen algoritmus költsége:

   a) $B(R)/3$, ha az index nyalábolt,[11] és
   b) $T(R)/3$, ha az index nem nyalábolt.

7.37. példa: Tekintsük a $\sigma_{x = 1\ \text{AND}\ y = 2\ \text{AND}\ z < 5}(R)$ kiválasztást, ahol az $R(x, y, z)$ a következő paraméterekkel rendelkezik: $T(R) = 5000$, $B(R) = 200$, $V(R, x) = 100$ és $V(R, y) = 500$. Tegyük fel továbbá, hogy az R nyalábolt, és az x, y és z attribútumok mindegyikéhez létezik index, de csak a z-hez tartozó index nyalábolt. Ennek a kiválasztásnak a megvalósítását a következő lehetőségek közül választhatjuk ki:

1. Táblaolvasás és azt követő szűrés. A költség $B(R)$, azaz 200 lemez I/O-művelet, mivel az R nyalábolt.
2. Használjuk az x-hez tartozó indexet és az indexolvasás operátort az $x = 1$ egyenlőséget kielégítő sorok megtalálására, majd használjuk a szűrés operátort annak

[11] Emlékezzünk a feltételezésünkre, miszerint a tipikus egyenlőtlenség a sorok 1/3-át adja vissza, a 7.4.3. részben tárgyalt indokoknak megfelelően.

ellenőrzésére, hogy az $y = 2$ és $z < 5$ összehasonlítások is teljesülnek-e ezekre a sorokra. Mivel körülbelül $T(R)/V(R, x) = 50$ olyan sor van, amelyre $x = 1$, és az index nem nyalábolt, körülbelül 50 lemez I/O-műveletre van szükség.

3. Használjuk az y-hoz tartozó indexet és az indexolvasás operátort az $y = 2$ egyenlőséget kielégítő sorok megtalálására, majd használjuk a szűrés operátort annak ellenőrzésére, hogy az $x = 1$ és $z < 5$ összehasonlítások is teljesülnek-e ezekre a sorokra. Ennek a nem nyalábolt indexnek a használata esetén a költség $T(R)/V(R, y)$, azaz 10 lemez I/O-művelet.
4. Használjuk a z-hez tartozó nyalábolt indexet és az indexolvasás operátort a $z < 5$ egyenlőtlenséget kielégítő sorok megtalálására, majd ezekre a sorokra alkalmazzuk a szűrés operátort, hogy lássuk, hogy az $x = 1$ és $y = 2$ feltételek is teljesülnek-e. A lemez I/O-műveletek száma körülbelül $B(R)/3 = 67$.

Azt látjuk, hogy a legkisebb költségű algoritmus a harmadik, és ennek a becsült költsége 10 lemez I/O-művelet. A kiválasztáshoz tartozó legjobb fizikai terv tehát először megkeresi az összes sort, amelyre $y = 2$, majd szűri azokat a másik két feltételnek megfelelően. □

7.7.2. Összekapcsolási eljárás megválasztása

A 6. fejezetben láttuk a különböző összekapcsolási algoritmusokhoz tartozó költségeket. Ha építünk arra a feltételezésre, hogy ismerjük (vagy becsüljük) az összekapcsolás végrehajtásához rendelkezésre álló pufferek számát, akkor alkalmazhatjuk a 6.5.8. részben szereplő képleteket a rendezéses összekapcsolásokra, a 6.6.7. részben szereplőket a tördelő összekapcsolásokra, valamint a 6.7.3. és 6.7.3. részben szereplőket az indexes összekapcsolásokra.

Ha viszont nem vagyunk biztosak benne vagy nem ismerjük a lekérdezés végrehajtása során rendelkezésre álló pufferek számát (mert nem tudjuk, hogy mi egyebet végez még az adatbázis-kezelő rendszer ugyanabban az időben), vagy nem vagyunk birtokában a méretre vonatkozó fontos paraméterek becsült értékeinek, mint amilyenek például a $V(R, a)$-k, akkor még mindig van néhány elv, amelyet egy összekapcsolási módszer megválasztásakor alkalmazhatunk. Hasonló gondolatmenet érvényes más bináris operátorokra – mint amilyen az egyesítés –, valamint a teljes relációra vonatkozó, a γ és a δ unáris operátorokra is.

- Az egyik megközelítés az, hogy az egymenetes összekapcsolást alkalmazzuk, azt remélve, hogy a pufferkezelő elég puffert tud az összekapcsolásnak szentelni, vagy legalábbis közel jár ehhez, így az ütközések nem eredményeznek jelentős költséget. Egy másik lehetőség az (csak összekapcsolásoknál, más bináris operátoroknál nem), hogy beágyazott ciklusú összekapcsolást választunk, azt remélve, hogy még ha a bal argumentum nem is kap elég puffert ahhoz, hogy egyszerre elférjen a memóriában, nem kell majd túl sok darabra osztani azt, és a kapott összekapcsolás még mindig elfogadható hatékonyságú lesz.

- Egy rendezéses összekapcsolás akkor jó választás, ha az alábbiak valamelyike teljesül:
  1. Az egyik vagy mindkét argumentum már rendezett az összekapcsolási attribútum(ok) szerint.
  2. Kettő vagy több összekapcsolás van ugyanazzal az összekapcsolási attribútummal, mint például az $(R(a, b) \bowtie S(a, c)) \bowtie T(a, d)$ összekapcsolás esetében, ahol az R és az S relációk a szerinti rendezése maga után vonja azt, hogy az $R \bowtie S$ eredménye az a szerint rendezett lesz, így az közvetlenül használható egy második rendezéses összekapcsolásban.

- Ha van alkalmunk indexet használni, mint például az $R(a, b) \bowtie S(b, c)$ összekapcsolásnál, ahol az R várhatóan kicsi (esetleg egy kulcs alapján történő kiválasztás eredménye, ami csak egy sort eredményez), és az $S.b$ összekapcsolási attribútumra létezik index, akkor indexes összekapcsolást válasszunk.
- Ha nem áll módunkban már rendezett relációkat vagy indexeket használni, és többmenetes összekapcsolásra van szükség, akkor valószínűleg a tördelő módszer a legjobb választás, mert a szükséges menetek száma a kisebb argumentum méretétől függ, és nem mindkét argumentumtól.

7.7.3. Futószalagosítás és materializáció

Az utolsó fő téma, amit egy fizikai lekérdezésterv megválasztása kapcsán meg kell beszélni, az az eredmények futószalagosítása. Egy lekérdezésterv végrehajtásának naiv módja az, hogy kialakítjuk a műveletek megfelelő sorrendjét (tehát egy művelet nem kerül végrehajtásra mindaddig, amíg az alatta szereplő argumentumok végrehajtása meg nem történt), és mindegyik művelet eredményét lemezen tároljuk mindaddig, ameddig egy másik művetnek szüksége van rá. Ezt a stratégiát *materializációnak* nevezzük, mivel minden közbülső reláció lemezen létrejön („testet ölt").

Egy lekérdezésterv végrehajtásának kifinomultabb és általában hatékonyabb módja

Materializáció a memóriában

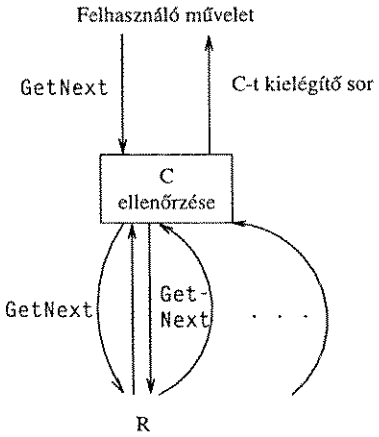
Bárki el tudja képzelni, hogy létezik egy közbülső megközelítés a futószalagosítás és a materializáció között, amikor egy művelet teljes eredményét a központi memória puffereiben (nem lemezen) tároljuk, mielőtt az azt felhasználó műveletnek átadódik. Az ilyen műveletmódot futószalagosításnak tekintjük, ahol is első teendőként a felhasználó művelet elrendezi a memóriában a teljes relációt vagy annak egy nagy darabját. Az ilyen viselkedésre példa egy olyan kiválasztás, amelynek eredménye tovább adódik, mint bizonyos összekapcsolási algoritmusok – ilyen az egyszerű egymenetes összekapcsolás, a többmenetes tördelő összekapcsolás vagy a rendezéses összekapcsolás – valamelyikének bal (építő) argumentuma.

az, amikor több művelet fut egyszerre. Egy adott művelet által előállított sorok közvetlenül átadódnak annak a műveletnek, amelyik használja azokat anélkül, hogy köztes sorokat bármikor is tárolnánk lemezen. Ezt a módszert *futószalagosításnak* nevezzük, és általában iterátorok egy hálózatával valósítjuk meg (lásd a 6.2.6. részt), amelyek függvényei a megfelelő időben hívják egymást. A futószalagosításnak nyilván megvan a maga előnye, hiszen lemez I/O-műveleteket takarít meg, van azonban egy hátránya is. Mivel egyszerre több műveletnek kell a központi memórián osztozni, megeshet, hogy magas lemez I/O-művelet számú algoritmusokat kell választani, vagy ütközések fordulhatnak elő, ami a futószalagosítás által nyert lemez I/O-művelet megtakarításokat visszaveszi, vagy esetleg még többet használ fel.

7.7.4. Unáris műveletek futószalagosítása

Az unáris műveletek – kiválasztás és vetítés – kiváló jelöltek arra, hogy a futószalagosítást alkalmazzuk rajtuk. Mivel ezek a műveletek egyszerre egy sort dolgoznak fel, soha nincs szükség egy blokknál többre a bemenet számára, és egy blokknál többre a kimenet számára. A 6.10. ábrán szemléltettük ezt a műveletmódot.

Egy futószalagosított unáris műveletet megvalósíthatunk iterátorok segítségével, ahogy ezt a 6.2.6. részben megtárgyaltuk. A futószalagra kerülő eredményt felhasználó művelet a `GetNext()` függvényt hívja, amikor egy újabb sorra van szüksége. Egy vetítés esetében egyszer meg kell hívni a `GetNext()`-et a forrássorokra, megfelelően vetíteni kell az adott sort, és visszaadni az eredményt a felhasználó művelet számára. Egy σ_C vetítésnél (ami technikailag a `Filter(C)` fizikai operátor) viszont előfordulhat, hogy többször kell a forrásra meghívni a `GetNext()`-et amíg talál olyan sort, amely kielégíti a C feltételt. Ezt az utóbbi folyamatot a 7.34. ábra szemlélteti.



7.34. ábra. *Egy futószalagosított kiválasztás végrehajtása iterátorok használatával*

7.7.5. Bináris műveletek futószalagosítása

A bináris műveletekből származó eredményeket is lehet futószalagosítani. Egy puffert használunk arra a célra, hogy az eredményt a felhasználó műveletnek átadjuk, mégpedig blokkonként. Az eredmény kiszámításához és az eredmény felhasználásához szükséges további pufferek száma azonban az eredmény méretétől és a lekérdezésben szereplő további relációk méreteitől függően változik. Az idevonatkozó problémákat és lehetőségeket egy összetett példán keresztül szemléltetjük.

7.38. példa: Nézzünk fizikai lekérdezésterveket az alábbi kifejezéshez:

$$(R(w, x) \bowtie S(x, y)) \bowtie U(y, z)$$

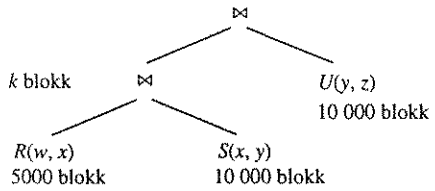
A következő feltételezésekkel élünk:

1. Az R 5000 blokkot foglal el, az S és az U pedig egyaránt 10 000 blokkot foglalnak el.
2. Az $R \bowtie S$ közbülső eredmény által elfoglalt blokkok számát k-val jelöljük. A k-t becsülhetnénk az R-ben és S-ben előforduló x-értékek száma, valamint a (w, x, y) soroknak az R-beli (w, x) sorokhoz és az S-beli (x, y) sorokhoz viszonyított mérete alapján. Látni szeretnénk azonban, hogy mi történik, ha k változik, így nyitva hagyjuk ezt a konstanst.
3. Mindkét összekapcsolást tördelő összekapcsolással valósítjuk meg, mégpedig k-tól függően egymenetessel vagy kétmenetessel.
4. 101 puffer áll rendelkezésre. Ezt a számot, mint eddig is, mesterkélten alacsonyra vettük.

A kifejezést és a paramétereket a 7.35. ábra mutatja.

Nézzük először az $R \bowtie S$ összekapcsolást. Egyik reláció sem fér el a központi memóriában, ezért egy kétmenetes tördelő összekapcsolásra van szükségünk. Ahhoz, hogy a kisebb R reláció egyes kosarait 100 blokkra korlátozzuk, legalább 50 kosárra szükség van.[12] Ha pontosan 50 kosarat használunk, akkor az $R \bowtie S$ tördelő összekapcsolás második menete 51 puffert használ, így szabadon marad 50 puffer, amit az $R \bowtie S$ eredményének az U-val való összekapcsolásához lehet használni.

Tegyük most fel, hogy $k \leq 49$, vagyis az $R \bowtie S$ eredménye legfeljebb 49 blokkot foglal el. Ekkor az $R \bowtie S$ eredményét 49 pufferbe futószalagosíthatjuk, a kikeresés



7.35. ábra. *Logikai lekérdezésterv és paraméterek a 7.38. példához*

céljából tördelőtáblába rendezhetjük, és marad egy blokk arra, hogy az U egyes blokkjait sorra beolvassuk. A második összekapcsolást tehát egy egymenetes összekapcsolással végrehajthatjuk. A lemez I/O-műveletek száma:

a) Az R és S kétmenetes tördelő összekapcsolásának végrehajtásához: 45 000.
b) Az $(R \bowtie S) \bowtie U$ egymenetes tördelő összekapcsolásban az U beolvasásához: 10 000.

Ez összesen 55 000 lemez I/O-művelet.

Most azt tegyük fel, hogy $k > 49$, de $k \leq 5000$. Az $R \bowtie S$ eredményét még mindig futószalagosíthatjuk, de egy másik stratégiát kell használni, amikor is ezt a relációt az U-val egy 50 kosaras, kétmenetes összekapcsolással kapcsoljuk össze.

1. Mielőtt elkezdjük az $R \bowtie S$-t, az U-t széttördeljük 50 darab 200 blokkos kosarakba.
2. Következő lépésként elvégezzük az R és az S kétmenetes tördelő összekapcsolását 51 kosarat használva, csakúgy mint az előbb. Most azonban amint előállítjuk az $R \bowtie S$ egy sorát, elhelyezzük azt a fennmaradó 50 puffer valamelyikében, amelyek az $R \bowtie S$-nek az U-val történő összekapcsolásához szükséges 50 kosár kialakítására valók. Ezeket a puffereket lemezre írjuk, amikor megtelnek, ahogy ez a kétmenetes tördelő összekapcsolások esetében szokásos.
3. Végezetül kosaranként összekapcsoljuk az $R \bowtie S$-t az U-val. Minthogy $k \leq 5000$, az $R \bowtie S$ kosarai legfeljebb 100 blokk méretűek lesznek, ez az összekapcsolás tehát megvalósítható. Az a tény, hogy az U kosarai 200 blokk méretűek, nem jelent problémát, hiszen a kosarak egymenetes összekapcsolásában az $R \bowtie S$ kosarait használjuk építő relációként, és az U kosarait vizsgáló relációként.

Az ehhez a futószalagosított összekapcsoláshoz tartozó lemez I/O-műveletek száma:

a) az U beolvasásához és sorainak kosarakba történő írásához: 20 000,
b) az $R \bowtie S$ kétmenetes tördelő összekapcsolás végrehajtásához: 45 000,
c) az $R \bowtie S$ kosarainak kiírásához: k.
d) az $R \bowtie S$ és az U kosarainak beolvasásához a végső összekapcsolásban: $k + 10\,000$.

Az összköltség tehát $75\,000 + 2k$. Vegyük észre, hogy a folytonosság nyilvánvalóan megszakad, amikor a k 49-ről 50-re nő, az utolsó összekapcsolást ugyanis egymenetesről kétmenetesre kellett változtatni. A gyakorlatban nem változna meg a költség ilyen hirtelen, mivel akkor is használhatnánk az egymenetes összekapcsolást, ha nem volna elég puffer, és némi ütközés előfordulna.

Végül nézzük meg, hogy mi a helyzet, ha $k > 5000$. Most a rendelkezésre álló 50 pufferben nem végezhetünk egy kétmenetes összekapcsolást, ha az $R \bowtie S$ eredményét futószalagosítjuk. Használhatnánk egy hárommenetes összekapcsolást, de ez mindkét argumentumnál blokkonként 2 extra lemez I/O-műveletet igényelne, ami $20\,000 + 2k$-val több lemez I/O-műveletet jelentene. Jobban tesszük, ha inkább eltekintünk az $R \bowtie S$ futószalagosításától. Az összekapcsolások kiszámításának váza ebben az esetben így néz ki:

1. Kiszámítjuk az $R \bowtie S$-t egy kétmenetes tördelő összekapcsolást használva, és az eredményt eltároljuk lemezen.
2. Az $R \bowtie S$-t összekapcsoljuk az U-val, szintén egy kétmenetes tördelő összekapcsolást használva. Vegyük észre, hogy mivel $B(U) = 10\,000$, a 100 blokktárolót használó kétmenetes tördelő összekapcsolás végrehajtható, függetlenül attól, hogy milyen nagy a k. Technikailag az U-nak a megfelelő összekapcsolás bal argumentumaként kellene szerepelnie a 7.35. ábrán, ha úgy döntünk, hogy az U-t választjuk építő relációnak a tördelő összekapcsolásban.

Az ehhez az algoritmushoz szükséges lemez I/O-műveletek száma:

a) Az R és az S kétmenetes összekapcsolásához: 45 000.
b) Az $R \bowtie S$ eredményének eltárolásához: k.
c) Az U-nak és az $R \bowtie S$-nek a kétmenetes tördelő összekapcsolásához: $30\,000 + 3k$.

A k tartománya	*Futószalagosítás vagy materializáció*	*Az utolsó összekapcsolás algoritmusa*	*Lemez I/O-műveletek száma*
$k \leq 49$	Futószalagosítás	Egymenetes	55 000
$50 \leq k \leq 5000$	Futószalagosítás	50 kosaras, kétmenetes	$75\,000 + 2k$
$5000 < k$	Materializáció	100 kosaras, kétmenetes	$75\,000 + 4k$

7.36 ábra. Összekapcsolási algoritmusok költségei az $R \bowtie S$ méretének függvényében

A teljes költség így $75\,000 + 4k$, ami kevesebb, mint annak a költsége, ha az utolsó lépesben egy hárommenetes összekapcsolást használunk. A három algoritmust a 7.36. ábrán található táblázatban foglaltuk össze. □

7.7.6. Fizikai lekérdezéstervekkel kapcsolatos jelölések

Sok példát láttunk olyan operátorokra, amelyek arra használhatók, hogy fizikai lekérdezésterveket képezzenek. Általában a logikai terv minden operátora a fizikai terv egy vagy több operátorává alakul át, és a logikai terv minden egyes leveléből (tárolt relációk) a fizikai tervben valamilyen olvasási operátor lesz, amely az adott relációra vonatkozik. A materializációt pedig az eltárolandó köztes eredményre alkalmazott valamilyen `Store` operátor jelezné, amit egy megfelelő olvasási művelet követne (rendszerint `TableScan`, mivel a köztes relációkhoz nem létezik index, hacsak explicit módon létre nem hozunk egyet), amint az eltárolt eredményhez az azt felhasználó művelet hozzáfér. Az egyszerűség kedvéért azonban a fizikai lekérdezésterveknél mi úgy fogjuk jelezni egy bizonyos közbülső reláció materializálásának tényét, hogy a fában egy kettős vonallal áthúzzuk azt az élt, amelyik az adott relációt és azt felhasználó műveletet összeköti. Az összes többi élnél feltételezzük, hogy futószalagos technikát alkalmazunk a sorok előállítója és felhasználója között.

Most pedig felsoroljuk a fizikai lekérdezéstervekben általában előforduló különféle operátorokat. A relációs algebrával ellentétben, amely eléggé szabványos jelölésrendszerrel rendelkezik, a fizikai lekérdezéstervekhez az egyes adatbázis-kezelő rendszerek saját belső jelölésrendszert használnak.

Operátorok a levelekhez

Minden egyes R relációt, amely a logikai lekérdezésterv fájában egy levél operandus, valamilyen olvasás operátorra cserélünk. A választási lehetőségek a következők:

1. `TableScan(R)`: Az R sorait tartalmazó összes blokkot beolvassa tetszőleges sorrendben.
2. `SortScan(R,L)`: Az R sorait az L listában szereplő attribútum(ok) szerint rendezve olvassa be.
3. `IndexScan(R,C)`: Itt a C egy $A\theta\ c$ alakú feltétel, ahol az A egy attribútum, a θ az = vagy < összehasonlító operátor, és a c egy konstans. Az R sorait egy az A attribútumhoz létező indexen keresztül érjük el. Ha a θ összehasonlítás nem =, akkor az indexnek olyannak kell lennie, amelyik támogatja a tartományra vonatkozó kérdéseket, mint amilyen például egy B-fa.
4. `IndexScan(R,A)`: Itt az A az R egy attribútuma. A teljes R relációt egy az $R.A$-hoz tartozó indexen keresztül kapjuk meg. Ez az operátor úgy viselkedik, mint a `TableScan`, de hatékonyabb lehet bizonyos körülmények között, ha az R nem nyalábolt és/vagy a blokkjait nem könnyű megtalálni.

Fizikai operátorok a kiválasztáshoz

Ha az R egy tárolt reláció, akkor egy $\sigma_C(R)$ logikai operátor gyakran vegyül, vagy részben vegyül, az R relációhoz történő hozzáférési eljárással. Más kiválasztásokat, ahol az argumentum nem tárolt reláció vagy nem áll rendelkezésre alkalmas index, megfelelő fizikai `Filter` operátorokkal fogunk helyettesíteni. Idézzük fel a kiválasztás megvalósításának megválasztására szolgáló stratégiát, amit a 7.7.1. részben tárgyaltunk. A kiválasztás különböző megvalósításaihoz a következő jelöléseket fogjuk használni:

1. A $\sigma_C(R)$-t egyszerűen helyettesíthetjük a `Filter(C)` operátorral. Ennek a választásnak akkor van értelme, ha az R-hez nem létezik index, pontosabban ha egyetlen C-ben előforduló attribútumra sincs index. Ha R – a kiválasztás argumentuma – egy közbülső reláció, amit a futószalagos technikán keresztül kap meg a kiválasztás, akkor a `Filter` mellett nincs más operátorra szükség. Ha R egy tárolt vagy materializált reláció, akkor az R eléréséhez kell egy operátor, `TableScan` vagy `SortScan`. A `SortScan` mellett maradhatunk, ha a $\sigma_C(R)$ eredménye később egy olyan operátornak adódik át, amely rendezett argumentumot vár.

2. Ha a C feltétel $A\theta\ c$ `AND` D alakú, ahol D egy másik feltétel, és az $R.A$-ra létezik index, akkor a következőket tehetjük:

   a) Az R eléréséhez használjuk az `IndexScan(R,Aθc)` operátort, és
   b) A $\sigma_C(R)$ helyett használjuk a `Filter(D)` operátort.

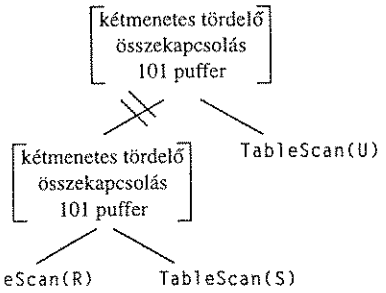
Fizikai operátorok a rendezéshez

A fizikai lekérdezéstervben bárhol rendezhetünk egy relációt. Bevezettük már a `SortScan(R,L)` operátort, amely beolvas egy R relációt, és előállítja annak rendezett formáját az L attribútumlistának megfelelően. Ha rendezéses algoritmust használunk egy művelethez, mondjuk összekapcsoláshoz vagy csoportosításhoz, akkor van egy kiindulási fázis, amikor az argumentumot valamilyen attribútumlistának megfelelően rendezzük. Nem tárolt operandusok esetén általában egy explicit `Sort(L)` operátort használunk a rendezés elvégzésére. Ugyanez az operátor a fizikai lekérdezésterv fájának tetején is használható, amennyiben az eredeti lekérdezés `ORDER BY` záradékot tartalmaz, és rendezni kell az eredményt, megfelelve ezzel a relációs algebránk τ operátorának.

Egyéb relációs algebrai műveletek

Az összes többi művelet helyettesíthető megfelelő fizikai operátorral. Adhatunk olyan megnevezéseket ezeknek az operátoroknak, amelyek mutatják az alábbiakat:

1. A végrehajtandó műveletet, például összekapcsolást vagy csoportosítást.
2. A szükséges paramétereket, például egy théta-összekapcsolásban a feltételt, vagy egy csoportosításban az attribútumlistát.
3. Az algoritmushoz használt általános stratégiát: rendezéses, tördelő, vagy index alapú bizonyos összekapcsolásoknál.
4. Menetek számára vonatkozó döntést: egymenetes, kétmenetes vagy sokmenetes



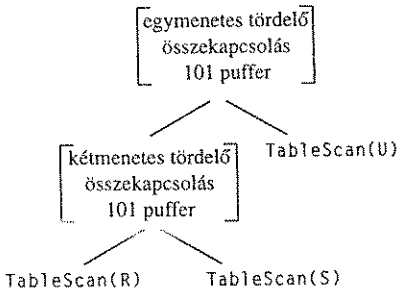
7.37. ábra. *Egy fizikai terv a 7.38. példából*

(rekurzív, ami annyi menetet használ, amennyi az aktuális adatok esetén szükséges). Azt is megtehetjük, hogy ezt a választást a futási időre halasztjuk.

5. A művelethez szükséges pufferek várható száma.

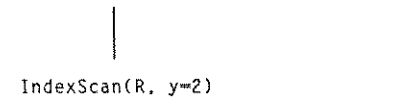
7.39. példa: A 7.38. példában a $k > 5000$ esetre kidolgozott fizikai tervet a 7.37. ábra mutatja. Ebben a tervben mindhárom relációt a `TableScan` operátor segítségével érjük el. Használunk egy kétmenetes tördelő összekapcsolást az első összekapcsoláshoz, materializáljuk az eredményt, majd használunk egy kétmenetes tördelő összekapcsolást a második összekapcsoláshoz. A materializációt jelző kettős vonalból következik, hogy a felső összekapcsolás bal argumentumát is egy `TableScan` operátor segítségével szerezzük meg, valamint hogy az első összekapcsolás eredményét a `Store` operátort használva tároljuk el.

A 7.38. példában a $k \leq 49$ esetre adott fizikai terv a 7.38. ábrán látható. Vegyük észre, hogy a második összekapcsolás az eddigitől eltérő számú menetet és puffert használ, továbbá nem egy materializált, hanem egy futószalagosított bal argumentumot használ. □



7.38. ábra. *Egy másik fizikai terv arra az esetre, amikor az $R \bowtie S$ várhatóan nagyon kicsi*

7.40. példa: Tekintsük a 7.37. példa kiválasztási műveletét, ahol úgy döntöttünk, hogy az a legjobb megoldás, hogy az y szerinti indexet használjuk az $y = 2$ feltételt kielégítő sorok megtalálására, majd ezekre a sorokra ellenőrizzük az $x = 1$ és $z < 5$ további feltételeket. A fizikai lekérdezésterv a 7.39. ábrán látható. A levél mutatja azt, hogy az R-et az ő y szerinti indexén keresztül érjük el, és az $y = 2$ összehasonlításnak eleget tevő sorokat keressük vissza. A szűrési operátor (`Filter`) mondja meg azt, hogy a kiválasztást azzal fejezzük be, hogy a megkapott sorok közül kiválasztjuk azokat, amelyekre $x = 1$ és $z < 5$. □



7.39. ábra. *Egy kiválasztás kifejtése úgy, hogy a legmegfelelőbb indexet használja*

7.7.7. Fizikai operátorok sorrendbe állítása

A fizikai lekérdezéstervekkel kapcsolatos utolsó téma a műveletek sorrendje. A fizikai lekérdezéstervet általában egy fával ábrázoljuk, és a fák mondanak valamit a műveletek sorrendjéről, hiszen az adatoknak alulról felfelé kell haladni a fában. A bozótszerű fákban azonban lehetnek olyan belső csomópontok, amelyek sem nem ősei, sem nem leszármazottjai egymásnak, ezért a belső csomópontok kiértékelésének sorrendje esetleg nem mindig világos. Ráadásul, mivel iterátorokat használhatunk műveletek futószalagtechnikával történő megvalósítására, elképzelhető, hogy különböző csomópontok futási idői átfedik egymást, és nincs is értelme a csomópontok „sorrendjéről" beszélni.

Ha a materializációt a kézenfekvő tárol-és-később-visszakeres módon valósítjuk meg, és a futószalagosítást iterátorokkal valósítjuk meg, akkor kialakíthatunk egy előre rögzített eseménysorozatot, amely mentén a fizikai lekérdezéstervek egyes műveleteit végrehajtjuk. Az események sorrendjét, ami egy fizikai lekérdezésterv fájában közvetett módon benne van, a következő szabályok foglalják össze:

1. Vágjuk szét a fát részfáira az olyan éleknél, amelyek materializációt képviselnek. A részfákat egyenként hajtjuk végre.
2. A részfák közötti sorrendet alulról felfelé, balról jobbra állapítsuk meg. Pontosan szólva, végezzük el a teljes fa egy előrendezéses bejárását. A részfákat annak a sorrendnek megfelelően rendezzük, ahogyan az előrendezéses bejárás a részfákat elhagyja.
3. Hajtsuk végre az egyes részfák összes csomópontját iterátorok hálózatát használva. Így az egy részfán belül szereplő összes csomópontot egyidejűleg hajtjuk végre, `GetNext` hívásokkal a bennük szereplő operátorok között, meghatározva ezáltal az események pontos sorrendjét.

Ezt a stratégiát követve a lekérdezésoptimalizáló futtatható kódot tud generálni a lekérdezéshez, ami várhatóan függvényhívások egy sorozata lesz.

7.7.8. Feladatok

7.7.1. feladat: Vegyünk egy $R(a, b, c, d)$ relációt, amelynek van egy nyalábolt indexe az a-ra, és egy-egy nem nyalábolt indexe az összes többi attribútumra. A lényeges paraméterek a következők: $B(R) = 1000$, $T(R) = 5000$, $V(R, a) = 20$, $V(R, b) = 1000$, $V(R, c) = 5000$ és $V(R, d) = 500$. Adjuk meg az alábbi kiválasztásokhoz tartozó legjobb lekérdezéstervet (index- vagy táblaolvasás és azt követő szűrés) a lemez I/O-művelet költséggel együtt:

* a) $\sigma_{a=1 \text{ AND } b=2 \text{ AND } d=3}(R)$

b) $\sigma_{a=1 \text{ AND } b=2 \text{ AND } c\geq 3}(R)$

c) $\sigma_{a=1 \text{ AND } b\leq 2 \text{ AND } c\geq 3}(R)$

! **7.7.2. feladat:** A $B(R)$, $T(R)$, $V(R, x)$ és $V(R, y)$ segítségével fejezzük ki az alábbi feltételeket, amelyek egy R-en végrehajtott kiválasztás költségére vonatkoznak:

* a) Jobb, ha indexolvasást egy x-re létező nem nyalábolt indexszel és egy x-et egy konstanssal egyenlővé tévő összehasonlítással használunk, mint ha egy y-ra létező nem nyalábolt indexszel és egy y-t egy konstanssal egyenlővé tévő összehasonlítással tennénk.

b) Jobb, ha indexolvasást egy x-re létező nem nyalábolt indexszel és egy x-et egy konstanssal egyenlővé tévő összehasonlítással használunk, mint ha egy y-ra létező nyalábolt indexszel és egy y-t egy konstanssal egyenlővé tévő összehasonlítással végeznénk el.

c) Jobb, ha indexolvasást egy x-re létező nem nyalábolt indexszel és egy x-et egy konstanssal egyenlővé tévő összehasonlítással használunk, mint ha egy y-ra létező nem nyalábolt indexszel és egy $y > C$ alakú összehasonlítással tennénk, ahol C egy konstans.

7.7.3. feladat: Hogyan változnának meg az arra vonatkozó következtetések, hogy mikor alkalmazzunk futószalagosítást a 7.38. példában, ha az R reláció mérete nem 5000 blokk lenne, hanem:

a) 2000 blokk,

! b) 10 000 blokk,

! c) 100 blokk.

! **7.7.4. feladat:** Tegyük fel, hogy az $(R(a, b) \bowtie S(a, c)) \bowtie T(a, d)$ kifejezést a jelzett sorrendnek megfelelően akarjuk kiszámítani. $M = 101$ központi memóriapufferrel rendelkezünk, és $B(R) = 2000$. Mivel az a összekapcsolási attribútum ugyanaz mindegyik összekapcsolásnál, úgy döntünk, hogy az első $R \bowtie S$ összekapcsolást egy kétmenetes rendezéses összekapcsolással valósítjuk meg. A második összekapcsoláshoz annyi menetet fogunk használni, amennyi szükséges, szétdarabolva a T-t bizonyos számú a szerint rendezett részlistára, és összefésülve azokat az $R \bowtie S$ összekapcsolásból származó rendezett és futószalagosított soraival. A $B(T)$ milyen értékei esetén kellene a T-nek az $R \bowtie S$-sel történő összekapcsolásához a következőt választani:

* a) Egy egymenetes összekapcsolást: a T-t beolvassuk a memóriába, és sorait összehasonlítjuk az $R \bowtie S$ soraival, amint azok előállnak.

b) Egy kétmenetes összekapcsolást: T-hez rendezett részlistákat hozunk létre, és mindegyik rendezett részlistához fenntartunk egy puffert a memóriában, mialatt az $R \bowtie S$ sorait előállítjuk.

7.8. Összefoglalás

- *Lekérdezések fordítása:* A fordítás egy lekérdezést fizikai lekérdezéstervvé alakít át, ami egy műveletsorozat, amelyet a lekérdezés-végrehajtó motorral lehet megvalósítani. A lekérdezésfordítás alapvető lépései: elemzés, szemantikus ellenőrzés, az előnyben részesített logikai lekérdezésterv (algebrai kifejezés) kiválasztása és abból a legjobb fizikai terv generálása.
- *Az elemző:* Egy SQL-lekérdezés feldolgozása során az első lépés a lekérdezés elemzése, csakúgy mint bármilyen programozási nyelvben írt kód esetén. Az elemzés eredménye egy elemzőfa, amelynek csomópontjai az SQL elemeinek felelnek meg.
- *Szemantikus ellenőrzés:* Az előfeldolgozó megvizsgálja az elemzőfát, ellenőrzi, hogy az attribútumok, relációnevek és típusok értelmesek-e, és feloldja az attribútumhivatkozásokat, ha ugyanaz az attribútum több relációban is előfordul.
- *Átalakítás logikai lekérdezéstervvé:* A lekérdezésfeldolgozónak a szemantikai szempontból ellenőrzött elemzőfát át kell alakítania egy algebrai kifejezéssé. A relációs algebrába történő átfordítás túlnyomó része kézenfekvő, az alkérdések azonban problémát jelentenek. A szokásos megközelítésben egy kétargumentumú kiválasztást vezetünk be, ami az alkérdést a kiválasztás feltételébe teszi, és azután megfelelő transzformációkat alkalmazunk, amelyek lefedik a szokásos speciális eseteket.
- *Algebrai transzformációk:* Egy logikai lekérdezéstervet sokféle módon átalakíthatunk egy jobb tervvé algebrai szabályok felhasználásával. A 7.2. rész felsorolja a alapvető szabályokat.
- *A logikai lekérdezésterv kiválasztása:* A lekérdezésfeldolgozónak ki kell választania azt a lekérdezéstervet, amely leginkább esélyes arra, hogy egy hatékony fizikai tervhez vezessen. Az algebrai transzformációk alkalmazásán túl, érdemes az asszociatív és kommutatív operátorokat csoportosítani, különösen az összekapcsolásokat. Ezáltal a fizikai lekérdezésterv a legjobb sorrendet és csoportosítást tudja megválasztani ezekhez a műveletekhez.
- *Relációk méreteinek becslése:* Amikor kiválasztjuk a legjobb logikai tervet, vagy amikor meghatározzuk az összekapcsolások vagy más asszociatív-kommutatív műveletek sorrendjét, a közbülső relációk becsült méreteit használjuk a legvégén kiválasztott fizikai terv tényleges futási idő vagy lemez I/O-műveletek költségének helyettesítésére. Ha ismerjük – vagy becsüljük – a relációk méreteit (sorok száma), valamint a különböző értékek számát minden reláció minden attribútumára vonatkozóan, akkor ez segít abban, hogy a köztes relációk méreteire jó becsléseket kapjunk.
- *Hisztogramok:* Néhány rendszer hisztogramokat tart fenn bizonyos attribútumok értékeiről. Ez az információ arra használható, hogy a közbülső relációk méreteire így jobb becsléseket kapjunk, mint a fejezetben hangsúlyozott egyszerű módszerekkel.
- *Költség alapú optimalizálás:* A legjobb fizikai terv kiválasztásakor szükség van arra, hogy minden egyes lehetséges terv költségét becsülni tudjuk. Különböző stratégiákat lehet arra használni, hogy előállítsunk minden vagy néhány lehetséges fizikai tervet, ami egy adott logikai tervet valósít meg.

- *Tervek felsorolására szolgáló stratégiák:* A fizikai tervek tartományában a legjobb megtalálására irányuló keresés szokásos megközelítései között szerepelnek: a dinamikus programozás (táblázatban összegyűjti az adott logikai terv egyes részkifejezéseihez tartozó legjobb terveket), a Selinger-féle dinamikus programozás (a nem rendezett eredmény mellett rendezett eredményt adó terveket is megtart a táblázatban), mohó megközelítések (lokális értelemben optimális döntéseket hoz a fizikai tervvel kapcsolatos addigi választások alapján), valamint az elágazás-és-korlát (csak azokat a terveket sorolja fel, amelyekről nem lehet azonnal tudni, hogy az addig talált legjobb tervnél rosszabb).
- *Bal-mély összekapcsolási fák:* Amikor több reláció összekapcsolásakor kiválasztunk egy csoportosítást és sorrendet, bevett gyakorlat, hogy a keresést a bal-mély fákra korlátozzuk. Ezek bináris fák egyetlen gerinccel, ami a bal oldalon fut lefelé, és csak olyan levelekkel, amelyek jobb gyerekek. Az összekapcsolási kifejezéseknek ez az alakja várhatóan hatékony tervet eredményez, és a megvizsgálandó fizikai tervek számát is korlátozza.
- *Fizikai terv a kiválasztáshoz:* Ha lehetséges, akkor egy kiválasztást ketté kell vágni a kiválasztás alapjául szolgáló reláció egy alkalmas indexének olvasására (ahol általában egy olyan feltételt használunk, amelyben az indexelt attribútumot egy konstanssal tesszük egyenlővé), valamint egy azt követő szűrési műveletre. A szűrés megvizsgálja az indexolvasás által visszaadott sorokat, és csak azokat engedi tovább, amelyek a kiválasztási feltétel többi részét (ami nem az indexolvasás alapja) is kielégítik.
- *Futószalag-technika és materializáció:* Ideális esetben minden egyes fizikai operátor eredményét felhasználja egy másik operátor, és az eredmény átadása a központi memóriában történik („futószalag-technika"), esetleg egy iterátor használatával a kettő közötti adatáramlás vezérlésére. Néha azonban annak van előnye, hogy egy operátor eredményét eltároljuk („materializáljuk"), helyet takarítva meg ezáltal a központi memóriában más operátorok számára. A fizikai lekérdezéstervet generálónak tehát a közbülső eredmények futószalagosításával és materializációjával is számolni kell.

7.9. Irodalomjegyzék

A 6. fejezet irodalomjegyzékében említett áttekintő tanulmányok a lekérdezésfordítás szempontjából lényeges anyagokat is tartalmaznak. Javasoljuk még az [1] áttekintést, amely kereskedelmi forgalomban lévő rendszerek lekérdezésoptimalizálóit vizsgálja.

Korai tanulmányok a lekérdezésoptimalizálásról a [4], [5] és [3]. A [7] – szintén korai tanulmány – egyesíti a kiválasztások – a fában lefelé történő – tologatásának elvét az összekapcsolási sorrend megválasztására szolgáló mohó algoritmussal. A [2] cikk a „Selinger-féle optimalizálás" forrása, valamint leírja a System R-optimalizálót is, amely a maga idejében a legigényesebb kísérlet volt lekérdezésoptimalizálásra.

Az SQL2 teljes nyelvtanát a [6]-ban találhatja meg az olvasó.

1. G. Graefe (ed.), *Data Engineering* **16**:4 (1993), special issue on query processing in commercial database management systems, IEEE.
2. P. Griffiths-Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price, „Access path selection in a relational database system", *Proc. ACM SIGMOD Intl.Conf. on Management of Data* (1979), pp. 23–34.
3. P. A. V. Hall, „Optimization of a single relational expression in a relational database system", *IBM J. Research and Development* **20**:3 (1976), pp. 244–257.
4. F. P. Palermo, „A database search problem", in: J. T. Tou (ed.) *Information Systems COINS IV*, Plenum, New York, 1974.
5. J. M. Smith and P. Y. Chang, „Optimizing the performance of a relational algebra database interface", *Comm. ACM* **18**:10 (1975), pp. 568–579.
6. `ftp://jerry.ece.umassd.edu/isowg3/dbl/BASEdocs/public/sql-92.bnf`
7. E. Wong and K. Youssefi, „Decomposition – a strategy for query processing", *ACM Trans. on Database Systems* **1**:3 (1976), pp. 223–241.

8. fejezet

A rendszerhibák kezelése

E fejezetben figyelmünket az adatbázis-kezelő rendszereknek az adatok meghibásodásával foglalkozó részére fordítjuk. Két fő témakör, melyeket tanulmányozni kell:

1. Az adatokat meg kell védeni a rendszerhibáktól. E fejezet azon technikákkal foglalkozik, melyek célja a *helyreállíthatóság*, azaz az adatok integritásának (épségének és összefüggéseinek) megőrzése a rendszerhibák előfordulásakor.
2. Az adatoknak nem szabad sérülniük több hibamentes lekérdezés vagy adatbázis-módosítások egyszerre való végrehajtásakor sem. Ezzel a 9. és 10. fejezet foglalkozik.

A helyreállíthatóság biztosítására az elsődleges technika a *naplózás* (log), mely valamilyen biztonságos módszerrel rögzíti az adatbázisban végrehajtott módosítások történetét. Három különböző – a „semmisségi" (undo), a „helyrehozó" (redo) és a „semmisségi/helyrehozó" (undo/redo) – naplózási módszert tanulmányozunk. Foglalkozunk továbbá a *helyreállítással*, azzal az eljárással, amikor a naplót felhasználva rekonstruáljuk az adatbázis hiba bekövetkezése előtti állapotát. A naplózás és a helyreállítás egy nagyon fontos vonatkozása az olyan helyzetek elkerülése, amikor a naplót a távoli múltra vonatkozóan kellene tanulmányozni. Így meg fogunk ismerni egy fontos technikát, az „ellenőrzőpont" használatát, mellyel korlátozzuk a helyreállítás során elemzendő napló(rész) hosszát.

Végül az „archiválással" foglalkozunk, mellyel biztosíthatjuk, hogy az adatbázis nemcsak az ideiglenes rendszerhibákat, de a teljes adatbázis elveszítését is „túlélje". Ezzel a módszerrel az adatbázis legfrissebb másolatára (az archivált adatbázisra) és a naplózott információkra támaszkodva rekonstruáljuk az adatbázis valamely korábbi állapotát.

8.1. A helyreállítható beavatkozások példái és modelljei

A vizsgálódásunkat azzal kezdjük, hogy áttekintjük a hibafajtákat, és azt, hogy az adatbázis-kezelő rendszerek mit tud(hat)nak tenni velük. Először a „rendszerhibákat" vagy „katasztrófákat" vesszük szemügyre. Ezen hibafajták elhárítására tervezték a

naplózási és helyreállítási módszereket. A 8.1.4. részben bemutatjuk a pufferkezelés modelljét is, mely a rendszerhibákból való helyreállítás minden meggondolásának alapja. Ugyanez a modell szükséges a következő fejezetben is, amikor az adatbázisok több tranzakcióval történő egyidejű (konkurens) elérését vizsgáljuk.

8.1.1. A hibák fajtái

Az adatbázis lekérdezése vagy módosítása során számos dolog hibát okozhat. A problémák felsorolása a billentyűzeten történt adatbeviteli hibáktól kezdve az adatbázist tároló lemez elhelyezésére szolgáló helyiségben történő robbanásig folytatható. A következő pontokban a legfontosabb hibaokokat csoportosítjuk, valamint összefoglaljuk, hogy az adatbázis-kezelő rendszerek mit tehetnek ezek előfordulásakor.

Hibás adatbevitel

Az adatok tartalmi hibáit sokszor nem tudjuk észrevenni. Ha például a hivatalnok elüt egy számot az ön telefonszámán, akkor az adat még úgy néz ki, mint egy telefonszám, mely az öné is *lehetne*. Másrészről, ha a hivatalnok kifelejt egy számot az ön telefonszámából, akkor az adat nyilvánvalóan hibás, mert nem felel meg a telefonszám formai követelményeinek[1].

A modern adatbázis-kezelő rendszerek számos szoftverelemet biztosítanak a fentihez hasonló adatbeviteli hibák felismerésére. Például az SQL2- és SQL3-szabványokban, s az SQL összes közismert megvalósításaiban az adatbázis tervezője megadhat előírásokat, mint például kulcsra, idegen kulcsra vagy értékekre vonatkozó megszorításokat (hogy például a telefonszámnak 10 számjegyből kell állnia). A triggerek azok a programok, melyek bizonyos típusú módosítások (például az R relációba való sor beszúrása) esetén hajtódnak végre, annak ellenőrzésére, hogy a frissen bevitt adatok megfelelnek-e az adatbázis-tervező által megszabott előírásoknak.

Készülékhibák

A lemezegységek olyan helyi hibái, melyek egy vagy néhány bit megváltozását okozzák, a lemez szektoraihoz rendelt paritás-ellenőrzéssel megbízhatóan felismerhetők, amint erről a 2.2.5. részben már szó volt. A lemezegységek jelentős sérülése, elsősorban a fejek (író-olvasó fejek) katasztrófái, az egész lemez olvashatatlanná válását okozhatják. A katasztrofális hibákat általában az alábbi megoldások segítségével kezelik:

1. A 2.6. részben már megismert RAID-módszerek valamelyikének használatával az elveszett lemez tartalma visszatölthető.

[1] Az egyszerűség kedvéért tegyük most fel, hogy a telefonszámok egyetlen formai előírásnak kell hogy megfeleljenek.

2. *Archiválás* (mentés) használatával az adatbázisról másolatot készítünk valamely eszközre pl. szalagra vagy optikai lemezre. A mentést rendszeresen kell végezni vagy teljes vagy növekményes (csak a változások archiválása) mentést használva. A mentett anyagot az adatbázistól biztonságos távolságban kell tárolnunk. Az archiválást a 8.5. részben tárgyaljuk.
3. Az archiválás helyett az adatbázisról fenntarthatunk elosztott, on-line másolatokat is. Ezen másolatok konzisztenciáját (összhangját az eredetivel) biztosító mechanizmusokat a 10.6. részben tanulmányozzuk.

Katasztrofális hibák

Ebbe a kategóriába soroljuk azokat a helyzeteket, amikor az adatbázist tartalmazó eszköz teljesen tönkremegy. A példák közé tartoznak a robbanás, a tűz, a vandalizmus és a vírusok is. A RAID ekkor nem segít, mert az összes adatlemezek és a paritás-ellenőrző lemezeik is egyszerre használhatatlanná válnak. Ugyanakkor más biztonsági megoldások – mint az archiválás és a redundáns osztott másolatok – használata az ilyen típusú katasztrófák elleni védekezésre is alkalmas.

Rendszerhibák

A lekérdező- és az adatbázis-módosító eljárásokat *tranzakcióknak* nevezzük. A tranzakció, hasonlóan más programokhoz, lépések sorozatát hajtja végre, gyakori esetben ezen lépések közül néhány az adatbázist fogja módosítani. Minden tranzakciónak van *állapota*, mely azt képviseli, hogy mi történt eddig a tranzakcióban. Az állapot tartalmazza a tranzakció kódjában a végrehajtás pillanatnyi helyét, és a tranzakció összes lokális változójának értékét, melyekre később még szükség lehet.

A *rendszerhibák* azok a problémák, melyek a tranzakció állapotának elvesztését okozzák. Tipikus rendszerhibák az áram kimaradásból és a szoftverhibákból eredők. Azért, hogy átlássuk, miért okozza az állapot elvesztését az olyan probléma, mint az áramkimaradás, vegyük figyelembe, hogy – más programokhoz hasonlóan – a tranzakció lépései is elsődlegesen a memóriában fordulnak elő. Eltérően a lemeztől, a memória tartalma „illékony", amint erről a 2.1.6. részben szó volt. Ez azt jelenti, hogy az áramkimaradás a memória tartalmának elvesztését okozza, amíg a lemezeken tárolt adatok sértetlenek maradnak. Hasonlóan, egy szoftverhiba a memória egy részének felülírását okozhatja, előfordulhat, hogy éppen a programunk állapotában szereplő értékeket is felülírja.

Ha a memória tartalma elveszett, a tranzakció állapota is elveszett, innentől kezdve nem tudjuk, hogy a tranzakció mely részei kerültek már végrehajtásra, beleértve az adatbázis-módosításokat is. A tranzakció ismételt futtatásával nem biztos, hogy a problémát korrigálni tudjuk. Például, ha a tranzakció az adatbázisban valamely értéket 1-gyel kell hogy növeljen, nem tudhatjuk, hogy az ismétléskor szükséges-e megismételni az 1-gyel való növelést, vagy sem. A rendszerhibákból származó problémák legfontosabb ellenszere: minden adatbázis-változtatás naplózása egy elkülönült, nem illékony naplófájlban, lehetővé téve ezzel a visszaállítást, ha az szükséges. Az a mechanizmus, ahogy a naplózás módszerét hibavédetté tesszük, meglepően bonyolult, amint azt a 8.2. rész elején látni fogjuk.

Tranzakciók és triggerek

A tranzakció kibővíthető triggerek használatával vagy más, az adatbázissémában előforduló aktív elemekkel. Ha a tranzakció módosítási tevékenységet is tartalmaz, és ez egy vagy több triggert is aktivizál, akkor a triggerakciók is a tranzakció részei lesznek. Egyes rendszerekben a triggerek kiválthatják újabb triggerek működését. Ha így van, akkor az összes kiváltott akciók a triggersorozatot aktivizáló tranzakció részének számítanak.

8.1.2. Részletesebben a tranzakciókról

Mielőtt folytatnánk az adatbázisok hibából adódó helyreállítási lehetőségeinek tanulmányozását, részletesebben tisztáznunk kell a tranzakciókra vonatkozó alapelgondolásokat. A tranzakció az adatbázis-műveletek végrehajtási egysége. Például, ha egy ad hoc utasítást adunk az SQL-rendszernek, akkor minden lekérdezés vagy adatbázis-módosító utasítás egy tranzakció. Amennyiben beágyazott SQL-interface-t használva a programozó készíti el a tranzakciót, akkor *egy* tranzakcióban – a használt programozási nyelv utasításaiként – több SQL-lekérdezés és -módosítás is szerepelhet. Tipikus beágyazott SQL- rendszerben a tranzakció adatbázis-akciók végrehajtásával kezdődik, és egy `COMMIT` vagy `ROLLBACK` („abort") paranccsal fejeződik be.

Amint a 8.1.3. részben látni fogjuk, a tranzakciót atomosan kell végrehajtani, ami azt jelenti, hogy mindent-vagy-semmit módon és időben egy egységként kell működnie. A tranzakciók korrekt végrehajtásának biztosítása a *tranzakciókezelő* feladata. A tranzakciókezelő részrendszer egy sor feladatot lát el, közöttük:

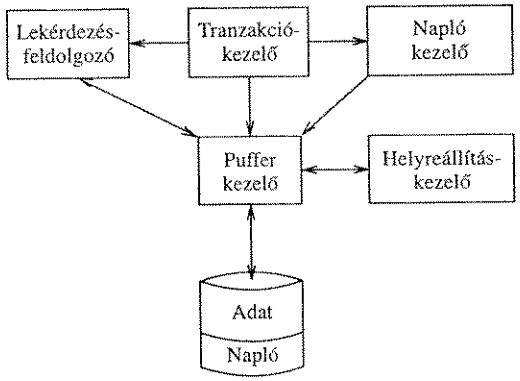
1. Jelzéseket ad át a naplókezelőnek (alább részletezzük) úgy, hogy a szükséges információ „naplóbejegyzés" formában a naplóban tárolható legyen.
2. Biztosítja, hogy a párhuzamosan végrehajtott tranzakciók ne zavarhassák egymás működését („ütemezés"; lásd a 9.1. részben).

A tranzakciókezelőt és kapcsolatait a 8.1. ábra mutatja. A tranzakciókezelő a tranzakció tevékenységeiről üzeneteket küld a naplókezelőnek, üzen a pufferkezelőnek arra vonatkozóan, hogy a pufferek tartalmát szabad-e vagy kell-e lemezre másolni, és üzen a lekérdezésfeldolgozónak arról, hogy a tranzakcióban előírt lekérdezéseket vagy más adatbázis-műveleteket kell végrehajtania.

A naplókezelő a naplót tartja karban. Együtt kell működnie a pufferkezelővel, hiszen a naplózandó információ elsődlegesen a memóriapufferekben jelenik meg, és bi-

zonyos időnként a pufferek tartalmát lemezre kell másolni. A napló, adat lévén, a lemezen területet foglal el, ahogy ezt a 8.1. ábrán is jelezzük.

Végezetül a 8.1. ábrán látható helyreállítás-kezelő szerepéről: ha baj van, akkor aktivizálódik. Megvizsgálja a naplót, és ha szükséges, a naplót használva helyreállítja az adatokat. Mint mindig, a lemez elérése a pufferkezelőn át történik.



8.1. ábra. *A naplókezelő és a tranzakciókezelő*

8.1.3. A tranzakciók korrekt végrehajtása

Mielőtt a rendszerhibák javításával foglalkoznánk, meg kell értenünk, hogy mit is jelent a tranzakciók „korrekt" végrehajtása. Először is tegyük fel, hogy az adatbázis „elemekből" áll. Azt, hogy mi az „elem", nem fogjuk precízen meghatározni, de úgy tekintjük, hogy az adatbázis elemeinek van valamilyen értékük, és ezt az értéket tranzakciókkal lehet elérni (kiolvasni) vagy módosítani. Más-más adatbázisrendszerek más-más megnevezést használnak az elemekre, de többnyire az alábbiak közül választanak:

1. Relációk vagy az objektumorientált megfelelője: az osztály kiterjedése.
2. Lemezblokkok vagy -lapok.
3. A relációk sorai vagy az objektumorientált megfelelői: objektumok.

A példáinkban az adatbázis elemeit tekinthetjük soroknak vagy sok példában egyszerűen egész számoknak. Ugyanakkor a gyakorlatban számos jó ok van arra is, hogy a 2. választást – lemezblokkokat vagy -lapokat – tekintsünk az adatbázis elemeinek. Ekkor a puffer tartalma egyszerű elemekké válik, s ezzel elkerülhető a naplózás és a tranzakciók néhány súlyosabb problémája, amelyeket majd periodikusan kifejtünk a különféle technikák tanulása során. Nem használva a lemezblokkméreténél nagyobb adatbáziselemeket, megelőzzük az olyan helyzeteket is, amikor a hiba fellépésekor az adatbázis valamely elemének egy része, de nem az egész van csak a nem illékony memóriában.

Hihető-e a korrektség alapelve?

Legyen adott egy olyan adatbázis-tranzakció, mely lehetővé teszi ad hoc módosító parancs kiadását a terminálról (a felhasználó készülékéről) esetleg olyan valaki számára, aki nem ismeri az adatbázis tervezője által elgondolt összefüggéseket. Nyilvánvaló-e ekkor, hogy az adatbázist konzisztens állapotából az összes lehetséges tranzakciók ismét konzisztens állapotba viszik? Az explicit megszorítások betartását az adatbázisrendszer kényszeríteni tudja azzal, hogy az olyan tranzakciókat, melyek megsértik az előírt összefüggéseket, a rendszer visszautasítja, s így az adatbázisban semmilyen változtatás nem történik. Az implicit megszorítások azok, melyeket nem tudunk egzakt módon jellemezni. Az egyetlen lehetőségünk a korrektség alapelvének érvényesítésére annak feltételezése, hogy ha valaki jogot kap az adatbázisban módosítani, akkor neki legyen joga annak eldöntésére is, hogy melyek az elvárt implicit megszorítások.

Az adatbázis összes elemeinek pillanatnyi értékét az adatbázis-*állapotának* nevezzük[2]. Bizonyos adatbázis-állapotokat konzisztensnek tekintünk, míg a többi adatbázis-állapotot inkonzisztensnek minősítjük. A konzisztens állapotok kielégítik az adatbázissémára vonatkozó összes megszorításokat, mint például a kulcsokra és az elemek értékeire vonatkozó előírásokat. Túl ezen, a konzisztens állapotnak ki kell elégítenie az implicit megszorításokat is, melyek az adatbázis tervezőjének elgondolásaiban szerepelnek. Az implicit megszorításokat részben az adatbázisséma részének tekintett triggerek alkalmazásával lehet biztosítani, de kikényszeríthetjük az adatbázisra vonatkozó rendtartási előírásokkal is. Használhatunk a felhasználónak szóló figyelmeztető üzeneteket is, amikor módosítja az adatbázist.

8.1. példa: Tegyük fel, hogy adatbázisunk a következő relációkból áll

```
Szerepel(filmCím, év, színészNév)
FilmSzínész(név, cím, nem, születési_idő)
```

Előírhatjuk a következő idegen kulcsra vonatkozó megszorítást: minden `színészNév` értéknek meg kell jelennie a `FilmSzínész név` értékeként; vagy a következő értékelőírást tehetjük: a `nem` értéke csak `'F'` vagy `'N'` lehet. Az adatbázis állapota akkor és csakis akkor konzisztens, ha az összes előírásokat kielégítik a két reláció pillanatnyi értékei. □

A tranzakciókra vonatkozó alapvető feltételezésünk:

- *A korrektség alapelve:* Ha a tranzakciót minden más tranzakciótól függetlenül („egyedül") és rendszerhiba nélkül végrehajtjuk, és ha indulásakor az adatbázis

[2] Ne keverjük össze az adatbázis-állapotot a tranzakció állapotával; utóbbiak a tranzakció lokális változóinak értékei, s ezek nem adatbáziselemek.

konzisztens állapotban volt, akkor a tranzakció befejezése után ismét konzisztens állapotban lesz.

A korrektség alapelvéhez kapcsolódik a naplózás technikája, melyet e fejezetben tárgyalunk, és a konkurencia vezérlési mechanizmus, melyet a 9. fejezetben tárgyalunk. Ebből két dolog következik:

1. A tranzakció *atomi*, azaz teljes egészében vagy végrehajtandó, vagy egyáltalán nem. Ha a tranzakciónak csak egy részét sikerült végrehajtani, akkor nagy esélyünk van arra, hogy az általa előállított adatbázis-állapot nem lesz konzisztens állapot.
2. A párhuzamosan végrehajtott tranzakciók nagy eséllyel inkonzisztens állapothoz vezethetnek, hacsak meg nem tesszük a 9. fejezetben tárgyalt megelőző lépéseket.

8.1.4. A tranzakciók alaptevékenységei

Vizsgáljuk meg részletesen a tranzakció és adatbázis kölcsönhatását. A kölcsönhatásoknak három fontos színhelye van:

1. Az adatbázis elemeit tartalmazó lemezblokkok területe.
2. A pufferkezelő által használt virtuális vagy valós memóriaterület.
3. A tranzakció memóriaterülete.

Ahhoz, hogy a tranzakció egy adatbáziselemet beolvashasson, azt előbb memória-puffer(ek)be kell behozni, ha még nincs ott. Ezt követően tudja a puffer(ek) tartalmát a tranzakció saját memóriaterületére beolvasni. Az adatbáziselem új értékének kiírása fordított sorrendben történik. Az új értéket a tranzakció alakítja ki saját memóriaterületén, majd ez az új érték másolódik át a megfelelő puffer(ek)be.

A pufferek tartalmát vagy azonnal lemezre lehet írni, vagy nem; az erre vonatkozó döntés általában a pufferkezelő joga. Amint már korábban láthattuk, a naplózó rendszer használatának egyik legfőbb lépése a rendszerhibákból való helyreállíthatóság biztosítása érdekében a pufferkezelő ösztönzése a pufferbeli blokkok megfelelő időpontban történő lemezre írására. Ugyanakkor a lemez I/O-műveletek számának csökkentésére az adatbázisrendszerek megengedik/megengedhetik a módosításoknak csak az illékony memóriában történő végrehajtását, legalábbis bizonyos ideig és arra megfelelő feltételek teljesülése esetén.

A naplózási algoritmusoknak és más tranzakciókezelő algoritmusoknak részletes tanulmányozása során megfelelő jelölésekre lesz szükségünk, melyekkel a különböző területek közötti adatmozgást tudjuk leírni. A következő alapműveleteket fogjuk használni:

1. `INPUT(X)`: Az X adatbáziselemet tartalmazó lemezblokk másolása a memóriapufferbe.
2. `READ(X,t)`: Az X adatbáziselem bemásolása a tranzakció t lokális változójába. Részletesebben, ha az X adatbáziselemet tartalmazó blokk nincs a memóriapufferben, akkor előbb végrehajtódik `INPUT(X)`. Ezután kapja meg a t lokális változó az X értékét.
3. `WRITE(X,t)`: A t lokális változó tartalma az X adatbáziselem memóriapufferbeli tartalmába másolódik. Részletesebben: ha az X adatbáziselemet tartalmazó blokk nincs a memóriapufferben, akkor előbb végrehajtódik `INPUT(X)`. Ezután másolódik át a t lokális változó értéke a pufferbeli X-be.
4. `OUTPUT(X)`: Az X adatbáziselemet tartalmazó puffer kimásolása lemezre.

Pufferek szerepe a lekérdezések feldolgozásában és a tranzakciókban

Ha visszagondolunk a lekérdezésfeldolgozással foglalkozó fejezet pufferhasználati elemzésére, akkor a jelenlegi nézőpontunkban változás tapasztalható. A 6. és 7. fejezetekben a puffereket elsősorban a lekérdezés kiértékelése közben szükséges ideiglenes táblák elhelyezésére használtuk. Ez a pufferek egyik fontos alkalmazása ekkor, mivel senki nem igényli az ideiglenes értékek megőrzését, így ezen pufferek tartalmát általában nem kell naplózni. Másrészről, azon pufferek tartalmát, melyek az adatbázisból beolvasott elemeket tartalmazzák, meg *kell* őrizni, különösen ha a tranzakció módosítja őket.

A fenti műveleteknek addig van értelmük, amíg az adatbáziselemek elférnek egy-egy lemezblokkban és így egy-egy pufferben is. Ezt az esetet úgy is tekinthetjük, hogy az adatbáziselemek *pontosan* a blokkok. Adatbáziselem lehet az adatbázis egy-egy sora is. Mindaddig így tekinthetjük, amíg a relációséma nem engedi meg nagyobb („hosszabb") sorok előfordulását, mint amennyi hely egy blokkban rendelkezésre áll. Ha az adatbáziselem több blokkot foglal el, akkor úgy is tekinthetjük, hogy az adatbáziselem minden blokkméretű része önmagában egy adatbáziselem. A naplózási mechanizmus, melyet arra használunk, hogy a tranzakció ne fejeződhessen be az X kiírása nélkül, atomos; azaz X összes blokkját vagy lemezre írja, vagy semmit sem ír ki. A továbbiakban a naplózási meggondolásokban úgy tekintjük, hogy:

- Az adatbáziselem nem nagyobb egy blokknál.

Fontos figyelembe venni, hogy ezen parancsokat kiadó komponensek különbözőek. A `READ` és `WRITE` utasításokat a tranzakciók használják, az `INPUT` és `OUTPUT` utasításokat a pufferkezelő alkalmazza, ezen túl, ahogy már láttuk, bizonyos feltételek esetén az `OUTPUT` utasítást a naplózási rendszer is használja.

8.2. példa: Annak bemutatására, hogy a tranzakció mikor és hogyan használja a fenti alapműveleteket, tegyük fel, hogy az adatbázis két, A és B eleme tartalmának, az adatbázis minden konzisztens állapotában meg kell egyeznie[3].

[3] Természetesen feltehető a kérdés, hogy miért használnánk két különböző elemet, melyek tartalma mindig megegyezik ahelyett, hogy egyetlen elemet alkalmaznánk. Mindazonáltal, ennek az egyszerű numerikus megszorításnak a teljesítése jól jellemez nagyon sok valóságos megszorítást, mint például amikor előírják, hogy a repülőn az eladott helyek száma 10%-nál

A T tranzakció tartalmazza a következő két lépést:

```
A := A*2;
B := B*2;
```

Vegyük figyelembe, hogyha a tranzakcióra az egyetlen konzisztenciaelvárás az $A = B$, továbbá ha T korrekt adatbázis-állapotban indul, és tevékenységét rendszerhiba, valamint a párhuzamosan működő tranzakciókkal való kölcsönhatás nélkül be tudja fejezni, akkor az adatbázis befejezéskori állapotának is konzisztensnek kell lennie. Ekkor T megduplázva két azonos tartalmú elem értékét, kap két új, azonos értékű elemet.

T végrehajtása maga után vonja A és B lemezről való beolvasását, az aritmetikai műveletek a T lokális memória változóiban kerülnek végrehajtásra, végül A és B új értékei visszaírásra kerülnek a puffereikbe. T-t hat lényeges lépésből állónak tekinthetjük:

```
READ(A,t); t := t*2;  WRITE(A,t);
READ(B,t); t := t*2;  WRITE(B,t);
```

Ehhez még hozzáadódik az, hogy a pufferkezelő önállóan végrehajt OUTPUT lépéseket a pufferek tartalmának lemezre történő visszaírása végett. A 8.2. ábra a T elemi lépéseit és az őket követő, a pufferkezelő által végrehajtott OUTPUT utasításokat szemlélteti. Tegyük fel, hogy kezdetben $A = B = 8$. Az A és B pufferbeli és lemezen tárolt értékei és a T tranzakció t lokális változójának értékei lépésenként a következők:

Tevékenység	t	*Mem* A	*Mem* B	*Lemez* A	*Lemez* B
READ(A,t)	8	8		8	8
t := t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
READ(B,t)	8	16	8	8	8
t := t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

8.2. ábra. *A tranzakció lépései, és hatásuk a memóriában és a lemezen*

T első lépésében beolvassa A-t, mely igény a pufferkezelőben kiváltja az INPUT(A) utasítást, ha A még nincs a pufferben. A értéke a READ utasítás hatására a T tranzakció memóriaterületére a t változóba is bemásolódik. A következő lépés megduplázza t tartalmát, ennek nincs hatása sem A pufferbeli, sem A lemezen tárolt értékére. A harmadik lépés írja t-t A pufferébe, s ennek nincs hatása A lemezen tárolt értékére. A következő három lépés ugyanez, csak B-re vonatkozóan. Végül az utolsó két lépésben másolódik A és B lemezre.

Figyeljük meg, hogy ezen lépések összességének végrehajtása alatt az adatbázis konzisztenciája megőrződik. Ha OUTPUT(A) végrehajtása előtt rendszerhiba fordul elő, akkor ennek nincs hatása a lemezen tárolt adatbázisra, az még olyan, mintha T egyáltalán nem is működött volna, s így a konzisztencia megőrződött. Ha rendszerhiba áll elő OUTPUT(A) végrehajtása után, de még OUTPUT(B) végrehajtása előtt, akkor az adatbázis inkonzisztens állapotban marad. Azt nem tudjuk megelőzni, hogy ilyen szituáció soha elő ne forduljon, de lépéseket tehetünk azért, hogy amikor mégis bekövetkezik, akkor a problémát elháríthassuk – vagy A és B értékének 8-ra való visszaállításával vagy mindkettő 16-ra növelésével. □

---

többel nem haladhatja meg a fedélzeten lévő ülések számát, vagy amikor előírják, hogy a bank kölcsönei összegének meg kell egyeznie a bank követelései összegével.

8.1.5. Feladatok

8.1.1. feladat: Tegyük fel, hogy az adatbázisra vonatkozó konzisztenciamegszorítás: $0 \leq A \leq B$. Állapítsuk meg, hogy következő tranzakciók megőrzik-e az adatbázis konzisztenciáját?

* a) `A := A + B; B := A + B;`
b) `B := A + B; A: = A + B;`
c) `A := B + 1; B := A + 1;`

8.1.2. feladat: A 8.1.1. feladat mindegyik tranzakciójához a számításokon kívül tegyük hozzá a beolvasó-kiíró tevékenységeket is, és mutassuk be a tranzakciók lépésenkénti hatását a memóriában és a lemezen tárolt adatokra. Tegyük fel, hogy kezdetben $A = 5$ és $B = 10$. Mondjunk valamit arról is, hogy lehetséges-e megfelelő OUTPUT műveletekkel biztosítani az adatbázis konzisztenciáját, a tranzakciók végrehajtása közben fellépő hibák esetében is.

8.2. Semmisségi (undo) naplózás

A naplózás tanulmányozását annak elemzésével kezdjük, hogy milyen úton biztosítható a tranzakciók atomossága – ami az adatbázisra nézve abban mutatkozik meg, hogy a tranzakciót vagy teljes egészében végrehajtjuk, vagy egyáltalán nem. A *napló* nem más, mint a *naplóbejegyzések* (log records) sorozata, melyek mindegyike arról tartalmaz valami információt, hogy mit tett egy tranzakció. A tranzakciók tevékenysége nyomon követhető azáltal, hogy a tranzakció működésének hatása lépésenként naplózódik, ugyanez történik az összes tranzakcióval. A tranzakciók nyomkövetése bonyolultabbá teszi a naplózást; nem elegendő egyszerűen a tranzakció végén a tranzakció történetének naplózása.

Ha rendszerhiba fordul elő, akkor a napló segítségével rekonstruálható, hogy a tranzakció mit tett a hiba fellépéséig. A naplót – az archívmentéssel együtt – használhatjuk akkor is, amikor eszközhiba keletkezik a naplót nem tároló lemezen. Általános-

Miért oly erős a tranzakciók abortálási hajlama?

Elgondolkodhatunk azon, hogy miért abortálnak (fejeződnek be a normálisnál korábban) a tranzakciók ahelyett, hogy teljesen rendesen befejeződnének. Ennek számos oka van. A legegyszerűbb ok, amikor magában a tranzakció kódjában hiba van, például egy zérussal való osztás fordul elő, melyet a tranzakció „kilövésével" (cancel) kezel a rendszer. Az adatbázis-kezelő rendszer is számos okkal abortáltathatja a tranzakciót. Példaként a tranzakció holtponti helyzetbe (deadlock) kerülhet, ha egy vagy több másik tranzakció lekötve tart olyan erőforrásokat (például ugyanazon adatbáziselembe új érték beírásának joga), melyeket mások is használni kívánnak. A 10.3. részben látni fogjuk, hogy ehhez hasonló szituációkban a rendszer kénytelen egy vagy több tranzakciót abortálni.

ságban a katasztrófák hatásának kijavítását követően a tranzakciókat meg kell ismételni, és az általuk adatbázisba írt új értékeket ismételten ki kell írni. Egyes tranzakciók a munkájukat vissza kívánják vonni, azaz kérik az adatbázis visszaállítását olyan állapotba, mintha a tekintett tranzakció nem is működött volna.

Az általunk vizsgált első naplózási stílus, melyet *semmisségi* (undo) *naplózásnak*[4] neveznek, csak az utóbbi típusú helyreállításra alkalmas. Ha nem teljesen biztos, hogy a tranzakció hatásai teljesen befejeződtek és lemezen tárolódtak, akkor minden olyan változtatást, melyet a tranzakció tehetett az adatbázisban, semmissé kell tenni, azaz az adatbázist vissza kell állítani a tranzakció működése előtti állapotába.

Ebben a részben a naplóbejegyzések alapelgondolását kívánjuk bemutatni, beleértve a tranzakció teljes és hibátlan befejezését, a *véglegesítési* (commit) tevékenységet, és ennek hatását az adatbázis állapotára és a naplózásra. Áttekintjük azt is, hogy maga a napló hogyan keletkezik a memóriában és hogyan íródik ki a lemezre a „flush-log" (naplókiírás) művelettel. Végül megvizsgáljuk konkrétan a semmisségi naplózást, és megtanuljuk, hogyan használhatjuk a katasztrófákból való helyreállításra. Elkerülendő azt, hogy helyreállítás során a teljes naplót át kelljen vizsgálni, bemutatjuk az „ellenőrzőpont-képzés" (checkpointing) ötletét, mely lehetővé teszi, hogy a napló régi részét eldobjuk[5]. Az ellenőrzőpont-képzés módszerét a semmisségi naplózáshoz kapcsolódóan ebben a fejezetben tanulmányozzuk.

[4] Az undo naplózást semmisségi naplózásnak, vagy visszavonási naplózásként is fordítják. *A fordító megjegyzése.*

[5] Ha „csak" a helyreállításra használjuk a naplót, akkor az utolsó ellenőrzőpont-képzésnél korábban keletkezett naplórészt valóban eldobhatjuk, de ha a naplót a korábban történt akciók utólagos elemzésére is fel kívánjuk használni, akkor meg kell tartanunk. Itt figyelembe kell vennünk, hogy a napló állandóan és jelentősen növekedik, de szerencsére a háttértárolók kapacitása is nő, a fajlagos tárolási költség pedig csökken. *A fordító megjegyzése.*

8.2.1. Naplóbejegyzések

Úgy kell tekintenünk, hogy a napló mint fájl, kizárólag bővítésre van megnyitva. Tranzakció végrehajtásakor a naplókezelőé a feladat, hogy minden fontos eseményt a naplóban rögzítsen. A napló blokkjai mindenkor naplóbejegyzésekkel vannak feltöltve, mindegyik bejegyzés egy-egy naplózandó eseményre vonatkozik. A naplóblokkokat elsődlegesen a memóriában hozza létre a rendszer, és a pufferkezelő az adatbázisrendszer többi blokkjaihoz hasonlóan kezeli őket. A naplóblokkokat, amint csak lehetséges, a nem illékony tárolóra írja a rendszer, erről bővebben a 8.2.2. részben szólunk.

A naplózás minden típusa a naplóbejegyzésnek számos formáját használja. E részben a következőkkel foglalkozunk:

1. <START *T*>: Ez a bejegyzés jelzi a *T* tranzakció (végrehajtásának) elkezdődését.
2. <COMMIT *T*>: A *T* tranzakció rendben befejeződött, az adatbázis elemein már semmi további módosítást nem kíván végrehajtani. A *T* által végrehajtott összes adatbázis-módosítás már megtörtént a lemezen. Minthogy azt nem tudjuk felügyelni, hogy a pufferkezelő mikor dönt a memóriablokkok lemezre másolásáról, így általában nem lehetünk biztosak abban, hogyha meglátjuk a <COMMIT *T*> naplóbejegyzést, akkor a változtatások a lemezen már megtörténtek. Ha ragaszkodunk ahhoz, hogy a módosítások már a lemezen is megtörténjenek, ezt az igényt a naplókezelőnek kell kikényszerítenie (mint például a semmisségi naplózás esetében).
3. <ABORT *T*>: A *T* tranzakció nem tudott sikeresen befejeződni. Ha a *T* tranzakció abortált, az általa tett változtatásokat nem kell a lemezre másolni. A tranzakciókezelő feladata annak biztosítása, hogy az ilyen változtatások ne jelenjenek meg a lemezen, vagy ha volt valami hatásuk a lemezen, akkor az törlődjék. Az abortált tranzakció hatásainak helyreállításával a 10.1.1. részben foglalkozunk.

A semmisségi (undo) naplózáshoz csak egyetlen további naplóbejegyzésre van

Milyen nagy a módosítást leíró naplóbejegyzés?

Ha az adatbáziselemek lemezblokkok, és a módosítást leíró naplóbejegyzés tartalmazza az adatbáziselem régi (módosítás előtti) értékét (vagy mind a régi, mind az új értékét, amint a 8.4. részben a semmisségi/helyrehozó naplózásnál látni fogjuk), akkor előfordulhat, hogy a naplóbejegyzés a blokknál nagyobb méretű lesz. Ez nem feltétlen probléma, mert minden hagyományos fájlhoz hasonlóan, a naplót lemezblokkok sorozatának tekinthetjük, mely bájtsorozatot tartalmaz, a (technikai) blokkhatároktól függetlenül. Ezáltal mód nyílik a napló tömörítésére is. Például bizonyos körülmények között csak a módosításokat kell naplóznunk, azaz csak a tranzakció által módosított sor érintett attribútumainak neveit és azok régi értékeit. A változtatások „logikai naplózása" témájával a 10.1.5. részben foglalkozunk.

szükségünk, a *módosítási bejegyzésre* (update record), mely a <*T*,*X*,*v*> hármas. Ezen bejegyzés jelentése: a *T* tranzakció módosította az *X* adatbáziselemet, melynek módosítás előtti értéke *v* volt. A módosítási bejegyzés által leírt változtatás rendesen csak a memóriában történt meg, a lemezen nem; azaz a naplóbejegyzés a `WRITE` tevékenységre vonatkozik, nem pedig az `OUTPUT`ra! (Emlékeztetünk a két művelet közötti különbségre, amit a 8.1.4. részben már láttunk.) Megjegyezzük még, hogy a semmisségi naplózás nem rögzíti az adatbáziselem új értékét, csak a módosítás előtti értéket. Amint látni fogjuk, a semmisségi naplózást alkalmazó rendszerekben a helyreállításkezelő feladata a tranzakció lehetséges hatásainak semmissé tétele, amelyhez elegendő csak a régi értékek tárolása.

8.2.2. A semmisségi naplózás szabályai

Ahhoz, hogy a rendszerhibák utáni helyreállításra a semmisségi naplózást használhassuk, a tranzakcióknak két előírást kell kielégíteniük. Ezek a szabályok arra vonatkoznak, hogy a pufferkezelőnek hogyan kell működnie, valamint előírnak bizonyos, a tranzakció szabályos befejezésekor elvégzendő tevékenységeket. Ezeket itt összefoglalva:

U_1: Ha a *T* tranzakció módosítja az *X* adatbáziselemet, akkor a <*T*,*X*,*v*> típusú naplóbejegyzést *azt megelőzően* kell lemezre kiírni, mielőtt *X* új értékét a lemezre írná a rendszer.

U_2: Ha a tranzakció hibamentesen teljesen befejeződött, akkor a `COMMIT` naplóbejegyzést csak *azt követően* szabad lemezre írni, hogy a tranzakció által módosított összes adatbáziselem már lemezre íródott, de ezután viszont a lehető leggyorsabban.

Összefoglalva az U_1 és U_2 szabályokat, az egy tranzakcióhoz tartozó lemezre írásokat a következő sorrendben kell megtenni:

a) Az adatbáziselemek módosítására vonatkozó naplóbejegyzések kiírása.
b) Maguknak a módosított adatbáziselemeknek a kiírása.
c) A `COMMIT` naplóbejegyzés kiírása.

Az a) és b) lépések minden módosított adatbáziselemre vonatkozóan önmagukban, külön-külön végrehajtandók (nem lehet a tranzakció több módosítására csoportosan megtenni)!

A naplóbejegyzések lemezre írásának kikényszerítésére a naplókezelőnek szüksége van a *flush-log* parancsra, mely felszólítja a pufferkezelőt az összes korábban még ki nem írt naplóblokkoknak a lemezre való kiírására, valamint azon pufferek kiírására, amelyek tartalma utolsó kiírásuk óta megváltozott. A `FLUSH LOG` parancsot a tevékenységek közé fogjuk iktatni. A tranzakciókezelőnek szüksége van arra is, hogy a pufferkezelőt az adatbáziselemekre vonatkozó `OUTPUT` akció végrehajtására felszólíthassa. A folytatásban be fogjuk mutatni a tranzakció lépései közé illesztett `OUTPUT` tevékenységet is.

8.3. példa: A semmisségi naplózás fényében vizsgáljuk meg újra a 8.2. példában már megismert tranzakciót. A 8.3. ábra a 8.2. ábra kibővítése, bemutatván a naplóbejegyzéseket is, és a naplókiírási tevékenységet is a *T* tranzakció végrehajtása során. Megjegyezzük, hogy a fejlécben rövidítéseket kellett használnunk; M-*A* rövidítést használjuk „*A*-nak memóriapufferbe másolása", D-*B*-t pedig „*B*-nek lemezre másolása" jelentéssel, és hasonlóan a többi rövidítésben is.

A 8.3. ábra 1) sorában a *T* tranzakció elkezdődik. Az első, ami történik, az a <`START` *T*> bejegyzés naplóba írása. A 2) sor *A*-nak *T* általi beolvasását jelenti. A 3) sor *t* módosítása, melynek nincs semmilyen hatása sem a lemezen tárolt adatbázisra, sem annak memóriapufferben található egyetlen részére sem. Sem a 2), sem a 3) sor nem igényel naplóbejegyzést, mert nincs hatásuk az adatbázisra.

A 4) sor *A* új értékének pufferbe írása. *A* ezen módosítására vonatkozik a <*T*,*A*,8> naplóbejegyzés, mely azt rögzíti, hogy *A* korábbi értékét, 8-at *T* megváltoztatta. Megjegyezzük, hogy az új érték, 16, nincs megemlítve a semmisségi naplózás naplójában.

Lépés	*Tevékenység*	*t*	*M*-A	*M*-B	*D*-A	*D*-B	*Napló*
1)							<`START` *T*>
2)	`READ(A,t)`	8	8		8	8	
3)	`t := t*2`	16	8		8	8	
4)	`WRITE(A,t)`	16	16		8	8	<*T*,*A*,8>
5)	`READ(B,t)`	8	16	8	8	8	
6)	`t := t*2`	16	16	8	8	8	
7)	`WRITE(B,t)`	16	16	16	8	8	<*T*,*B*,8>
8)	`FLUSH LOG`						
9)	`OUTPUT(A)`	16	16	16	16	8	
10)	`OUTPUT(B)`	16	16	16	16	16	
11)							<`COMMIT` *T*>
12)	`FLUSH LOG`						

8.3. ábra. *Tevékenységek és naplóbejegyzéseik*

Más naplózási módszerek áttekintése

A „helyrehozó" (redo) naplózás (8.3. részben tárgyaljuk), a katasztrófát követő helyreállítás során helyrehozza az összes olyan tranzakció hatását, melyek már elindultak, de még nem fejeződtek be. A helyrehozó naplózás szabályai biztosítják, hogy ne legyen szükséges az olyan tranzakciók helyrehozása, melyekre vonatkozó `COMMIT` bejegyzést a naplóban megtaláljuk. A „semmisségi/helyrehozó" (undo/redo) naplózás (8.4. részben tárgyaljuk) a katasztrófát követő helyreállítás során semmissé teszi az összes olyan tranzakció hatását, amely még nem fejeződött be, és helyre kívánja hozni azokat a tranzakciókat, melyek már befejeződtek. Ezenfelül a napló és a pufferek kezelési szabályai biztosítani kívánják, hogy ezek a lépések az adatbázis minden sérülését helyreállítsák.

Az 5)-től 7)-ig sorokban a B-re vonatkozóan ugyanazon lépések hajtódnak végre, mint korábban A-ra. E ponton a T rendben befejeződött, tevékenységét véglegesíteni kell. A megváltozott A és B értékét lemezre kell másolni, betartva a semmisségi (undo) naplózás két szabályát, a következő lépéseknek kötött sorrendben kell megtörténnie.

Első, hogy A és B addig nem másolható lemezre, amíg a módosítást leíró naplóbejegyzések lemezre nem kerülnek. Ezt a 8) lépéssel biztosítjuk, a `FLUSH LOG` hatására az eddigi összes naplóbejegyzés lemezre íródik. E kiírást követően a 9) és 10) lépések A-t és B-t lemezre másolják. Ezeket a lépéseket a T teljes befejeződése, a véglegesítés érdekében a tranzakciókezelő igénye szerint a pufferkezelő valósítja meg.

S ekkor lehetséges a T teljes és sikeres befejezése, ezt jelzendő a 11) lépésben <COMMIT T> bejegyzés a naplóban íródik. Végül a 12) lépésben ismét ki kell adni a `FLUSH LOG` utasítást azért, hogy biztosítsuk a <COMMIT T> naplóbejegyzés lemezre való kiírását. Ezen naplóbejegyzés lemezre való kiírása nélkül, bár olyan helyzetben vagyunk, hogy a tranzakció teljesen és hibamentesen befejeződött, ennek a napló későbbi elemzésekor nem fogjuk nyomát találni. Az ilyen szituációk olyan furcsa viselkedést eredményezhetnek, hogy hiba esetén, amint a 8.2.3. részben látni fogjuk, a felhasználó azt tapasztalja, hogy a tranzakció hibamentesen rendesen befejeződött, a lemezre kiírt módosítások mégis semmissé váltak, a tranzakció ténylegesen abortált[6]. □

8.2.3. Helyreállítás a semmisségi naplózás használatával

Tételezzük fel, hogy rendszerhiba fordult elő. Előfordulhat, hogy valamely adott tranzakció által végzett adatbázis-módosítások közül bizonyosak lehet, hogy már lemezre íródtak, míg más módosítások – melyeket ugyanezen tranzakció hajtott végre – nem jutottak el a lemezre. Ha így történt, a tranzakció nem atomosan hajtódott végre, ennek következtében az adatbázis inkonzisztens állapotba kerülhetett. A *helyreállítás-kezelő* (recovery manager) feladata – a napló használatával – az adatbázist konzisztens állapotba visszaállítani.

Ebben a részben csak a legegyszerűbb helyreállítás-kezelő módszerrel foglalkozunk, mely a teljes naplót látja, függetlenül annak méretétől, és a napló vizsgálatával hajtja végre az adatbázis módosításait. A 8.2.4. részben egy sokkal finomabb megközelítést mutatunk be, amikor ellenőrzőpont periodikus készítésével a rendszer korlátozza azt a távolságot, ameddig a helyreállítás-kezelőnek a korábbi történéseket (a naplóban) vizsgálnia kell.

A helyreállítás-kezelő első feladata a tranzakciók felosztása sikeresen befejezett és nem befejezett tranzakciókra. Ha található <COMMIT T> naplóbejegyzés, akkor az U_2 szabálynak megfelelően a T tranzakció által végrehajtott összes adatbázis-módosítások már korábban lemezre íródtak. Így a T tranzakció önmagában, a hiba fellépésekor, nem hagyhatta az adatbázist inkonzisztens állapotban.

[6] A tranzakció valójában nem abortált, de egy később fellépett hiba elemzésekor, mivel a rendszer nem talál <COMMIT T> naplóbejegyzést, úgy tekinti, hogy a tranzakció nem tudott teljesen és rendesen befejeződni, ezért a helyreállító rendszer a tranzakció hatásait semmissé teszi, végeredményben az történik, mintha T abortált volna. *A fordító megjegyzése.*

Háttértevékenységek, naplózás, pufferkezelés

A 8.3. ábrán látottaknak megfelelően a tranzakció tevékenységei és a naplóbejegyzések sorozata azt az elképzelést sugallják, mintha ezek a tevékenységek elkülönülten következnének be. Ugyanakkor az adatbázis-kezelő rendszer számos tranzakció szimultán kezelését kell hogy megoldja. Így egy T tranzakció négy naplóbejegyzése a naplóban más tranzakciók naplóbejegyzéseivel keveredhet. Ezenfelül, ha a másik tranzakciók valamelyike is a napló lemezre írását kezdeményezi (`FLUSH LOG` kiadásával), akkor a T-re vonatkozó naplóbejegyzések esetleg már korábban lemezre kerülnek, mint ahogy azt a 8.3. ábrán látható `FLUSH LOG` utasítások okoznák. Abból nem származik probléma, ha az adatbázis módosítására vonatkozó naplóbejegyzések a szükségesnél korábban jelennek a naplóban. A <COMMIT T> naplóbejegyzést úgysem fogjuk a T `OUTPUT` utasításai végrehajtásának befejezésénél korábban kiírni, ezzel biztosítani tudjuk, hogy a módosított adatbázisértékek korábban megjelenjenek a lemezen, mint a `COMMIT` naplóbejegyzés.

Kényes helyzet áll elő, ha az A és B, két adatbáziselem, közös blokkban található. Akkor egyikük lemezre írása maga után vonja a másikuk kiírását is. Legrosszabb esetben az egyik adatbáziselem túl korai kiírásával megsértjük az U_1 szabályt. Ez szükségessé tehet további előírásokat a tranzakcióra nézve azért, hogy a semmisségi naplózási módszer használható legyen. Például a 9.3. részben ismertetett zárolási módszert kell használnunk annak megelőzésére, nehogy két tranzakció, egyszerre, ugyanazon blokkot használja (e példában lemezblokkokat tekintünk adatbáziselemeknek). Ilyen és hasonló problémák akkor jelentkeznek, amikor az adatbáziselemek blokkok részei, ez motiválja azt a javaslatunkat, hogy a blokkokat adatbáziselemeknek tekintsük.

Amennyiben feltételezzük, hogy a naplóban találunk <START T> bejegyzést, de nem találunk <COMMIT T> bejegyzést, akkor a T végrehajthatott az adatbázisban olyan módosításokat, melyek még a hiba fellépése előtt lemezre íródtak, amíg más változtatások a memóriapufferekben sem történtek meg, vagy a memóriapufferben megtörténtek ugyan, de a lemezre már nem íródtak ki. Ilyen esetben a *T nem komplett tranzakció*, és hatását semmissé kell tenni, azaz a T által módosított adatbáziselemek értékét vissza kell állítani korábbi értékeikre. Szerencsére az U_1 szabály betartása biztosítja, hogy ha T, a hiba jelentkezése előtt, az X értékét módosította, akkor a hiba jelentkezése előtt már a lemezen lévő naplóba kellett kiíródni egy <T,X,v> bejegyzésnek. S így a helyreállítás során módunkban áll a v értéket az X adatbáziselembe visszaírni. Megjegyezzük, hogy ez a szabály bizonyítottnak tekinti, hogy X korábbi értéke v volt, de ennek tényleges ellenőrzésére alkalmatlan. (A <T,X,v> naplóbejegyzésnek hinnünk kell, annak helyességét önmagában nem tudjuk ellenőrizni.)

Minthogy a naplóban számos, rendesen befejezett és teljesen be nem fejezett tranzakció nyomát találhatjuk, s ezek közül több tranzakció módosíthatta az X adatbázis-

elemet is, így nagyon ügyelnünk kell arra, hogy milyen sorrendben állítjuk vissza X korábbi tartalmát. Ezért a helyreállítás-kezelő a naplót a végéről kezdi átvizsgálni (tehát az utoljára felírt bejegyzéstől a korábban felírottak irányába). Amint halad a napló átvizsgálásával, megjegyzi mindazon T tranzakciókat, melyekre vonatkozóan a naplóban <COMMIT T> vagy <ABORT T> bejegyzést talált. Amint halad visszafelé, amikor $<T,X,v>$ bejegyzést lát, akkor:

1. Ha ugyanerre a T tranzakcióra vonatkozó COMMIT bejegyzéssel már találkozott, akkor nincs teendője, T rendesen és teljesen befejeződött, hatásait nem kell tehát semmissé tenni.
2. Minden más esetben T nem teljes vagy abortált tranzakció. A helyreállítás-kezelő X értékét v-re kell hogy cserélje.[7]

Miután a helyreállítás-kezelő végrehajtotta a fenti változtatásokat, minden, korábban abortált, nem teljes T tranzakcióra vonatkozóan <ABORT T> naplóbejegyzést ír a naplóba, és kiváltja annak naplófájlba való kiírását is (FLUSH LOG). Ekkor az adatbázis normál használata folytatódhat, új tranzakciók végrehajtása kezdődhet.

8.4. példa: Tekintsük a 8.3. ábrán és a 8.3. példában látott tevékenységeket. Rendszerhiba számos különböző időpontban felléphet; tekintsük át az összes lényeges, különböző esetet:

1. A hiba a 12) lépést követően jelentkezett. Tudjuk, hogy ekkor a <COMMIT T> bejegyzést már lemezre írta a rendszer. A hiba kezelése során a T tranzakció hatásait már nem kell visszaállítani, s a T-re vonatkozó összes naplóbejegyzést a helyreállítás-kezelő figyelmen kívül hagyhatja.
2. A hiba a 11) és 12) lépések között keletkezett. Ekkor előfordulhat, hogy a COMMIT bejegyzést tartalmazó naplóbejegyzés már lemezre íródott, például, ha a naplóbejegyzés kiírását másik tranzakció már kérte a pufferkezelőtől. Ha így történt, akkor T-re vonatkozólag a hiba kezelése az 1) esethez hasonló. Ha azonban a COMMIT bejegyzés a lemezen nem található, akkor a helyreállítás-kezelő a T tranzakciót befejezetlennek tekinti. Ahogy olvassa a naplót visszafelé, először a $<T,B,8>$ bejegyzést fogja megtalálni (a T tranzakcióra vonatkozólag). Ennek megfelelően a lemezen a B tartalmába a 8-at állítja vissza. Majd a $<T,A,8>$ naplóbejegyzés miatt A tartalmába kerül 8. Végezetül <ABORT T> bejegyzést ír a naplóba és a lemezre.
3. Ha a hiba a 10) és 11) lépések között lépett fel, akkor a COMMIT bejegyzés még biztosan nem történt meg, tehát T befejezetlen, hatásainak semmissé tétele a 2) esetnek megfelelően történik.
4. A 8) és 10) lépések között bekövetkező hiba fellépésekor az előző 3) esethez hasonlóan T hatásait semmissé kell tenni. Az egyetlen különbség, hogy az A és/vagy B módosítása még nem jelent meg a lemezen. Ettől függetlenül mindkét adatbázis-elem korábbi értékét, 8-at, állítja vissza a rendszer.

[7] Ha T abortált, akkor az összes hatását az adatbázisban mindenképpen vissza kell állítani.

A helyreállítás közben bekövetkező (újabb) katasztrófákról

Tegyük fel, hogy egy korábbi hiba utáni helyreállítás közben ismét rendszerhiba lép fel. A semmisségi (undo) naplózás oly módon van megtervezve, hogy a korábbi érték változtatás előtti tárolása következtében a helyreállító lépések idempotensek; ami azt jelenti, hogy a helyreállító tevékenység többszöri végrehajtása pontosan ugyanolyan hatású, mint egyszeri végrehajtása. Arra koncentrálunk csak, hogyha találunk $<T,X,v>$ naplóbejegyzést, akkor nem számít, hogy X értéke már v, X értékét (esetleg ismételten) v-re állíthatjuk. Hasonlóan semmi problémát nem okoz, ha a helyreállítási folyamat egészét (vagy félbemaradt részét) többször megismételjük, az esetleg már visszaállított értékeket ismételten visszaállítjuk. Ugyanezt fogjuk látni az e fejezetben tárgyalt többi naplózási módszerek esetében is. Minthogy a helyreállító tevékenység idempotens, másodszor (többször is) probléma nélkül megismételhetjük, függetlenül a korábbi helyreállító akció által végrehajtott módosításoktól.

5. Amennyiben a hiba a 8) lépésnél korábban jelentkezik, akkor még az sem biztos, hogy a T tranzakcióra vonatkozó naplóbejegyzések közül bármi is lemezre (a naplóba) került-e, de nem is fontos, mivel az U_1 szabály miatt tudjuk, hogy mielőtt az A és/vagy B adatbáziselemek a lemezen módosulnának, a megfelelő módosítási naplóbejegyzésnek a naplóban meg kell jelennie. Esetünkben ez nem történt meg, következésképp a módosítások sem történtek meg, tehát nincs is visszaállítási feladat. □

8.2.4. Az ellenőrzőpont-képzés

Mint láttuk, a helyreállítás elvben a teljes napló átvizsgálását igényli. Ha a naplózásban (az eddig bemutatott) semmisségi (undo) naplózás módszerét követjük, akkor, ha egy tranzakció a COMMIT naplóbejegyzést már kiírta a naplóba, akkor az ezen tranzakcióra vonatkozó naplóbejegyzésekre a helyreállítás során nincs már szükség[8]. Gondolhatnánk arra, hogy a tranzakcióra vonatkozó, a COMMITot megelőző naplóbejegyzéseket törölhetnénk a naplóból, de ezt nem mindig tehetjük meg. Ennek oka az, hogy gyakran sok tranzakció működik egyszerre, ha a naplót egy tranzakció befejezése után csonkítanánk, esetleg elveszítenénk más, még aktív tranzakciókra vonatkozó bejegyzéseket, s így – ha szükség lenne rá –, nem tudnánk a naplót a helyreállításra használni.

E lehetséges probléma megoldására a legegyszerűbb mód, a naplóra vonatkozóan, ismétlődően *ellenőrzőpontot* képezni. Az egyszerű ellenőrzőpont képzése:

[8] A tranzakció teljesen és helyesen befejeződött, nem kell tehát semmissé tenni hatásait, a napló bejegyzéseire ez okból már valóban nincs szükség, a tevékenységek utólagos elemzése miatt azonban még e naplóbejegyzések is fontosak lehetnek. *A fordító megjegyzése.*

<START T_1>
<T_1,A,5>
<START T_2>
<T_2,B,10>
<T_2,C,15>
<T_1,D,20>
<COMMIT T_1>
<COMMIT T_2>
<CKPT>
<START T_3>
<T_3,E,25>
<T_3,F,30>

8.4. ábra. *Napló egyszerű ellenőrzőpont-képzéssel*

1. Új tranzakcióindítási kérések kiszolgálásának leállítása.
2. A még aktív tranzakciók helyes és teljes befejezésének vagy abortálásának és a COMMIT vagy az ABORT bejegyzés naplóba írásának kivárása.
3. A napló lemezre kiírása (FLUSH).
4. <CKPT>[9] naplóbejegyzés képzése és kiírása a naplóba, és ismételt FLUSH.
5. Tranzakcióindítási kérések kiszolgálása.

Az ellenőrzőpont kiírását megelőzően végrehajtott tranzakciók mind befejeződtek, s az U_2 szabálynak megfelelően módosításaik már lemezre kerültek. Ennek megfelelően – ezen tranzakciók tevékenységére nézve – egy esetleges későbbi hiba elhárításakor már nem igényel a rendszer helyreállítást. A helyreállítás során a naplót a végétől visszafelé csak a <CKPT> bejegyzésig kell elemezni azért, hogy a nem befejezett tranzakciókat azonosítsuk. Amikor a <CKPT> bejegyzést megtaláljuk, ebből tudjuk, hogy már láttuk az összes befejezetlen tranzakciót. Mivel az ellenőrzőpont-képzés alatt újabb tranzakció nem indulhatott, látnunk kellett a befejezetlen tranzakciókhoz tartozó összes naplóbejegyzést. Ezért nem szükséges a <CKPT> bejegyzésnél korábbi naplórészt elemeznünk, s – ha más okból már nincs szükségünk rá – biztonsággal törölhetjük vagy felülírhatjuk.

8.5. példa: Tegyük fel, hogy a napló így kezdődik:

<START T_1>
<T_1,A,5>
<START T_2>
<T_2,B,10>

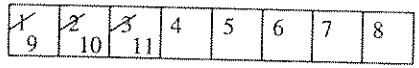
S ekkor döntünk ellenőrzőpont létrehozásáról. Minthogy T_1 és T_2 aktív (nem befejezett) tranzakciók, meg kell várnunk befejeződésüket, mielőtt a <CKPT> bejegyzést a naplóba írnánk.

[9] CKPT – Checkpoint (ellenőrzőpont) rövidítése. *A lektor megjegyzése.*

Az utolsó naplóbejegyzés megtalálása

A napló lényegében egy fájl, melynek blokkjai tartalmazzák a naplóbejegyzéseket. A blokk még ki nem töltött részeit „üres"-ként jelölik. Ha a bejegyzéseket soha nem írjuk felül, akkor a helyreállítás-kezelő az utolsó bejegyzést úgy keresi meg, hogy megkeresi az első üres bejegyzést, és az ezt megelőző bejegyzés a fájl utolsó érvényes bejegyzése.

Ha a régi naplóbejegyzéseket felülírjuk, akkor a naplóbejegyzéseket az alábbi módon:



egyedi, növekvő sorszámmal kell ellátnunk. Ekkor azt a bejegyzést kell megtalálnunk, melynek nagyobb a sorszáma, mint a következőé; ez a bejegyzés a napló pillanatnyi vége, s a naplóbejegyzései sorszámuk szerinti sorban keletkeztek.

A gyakorlatban a nagyméretű napló több fájl egyesítése is lehet. Logikailag ekkor is egy fájlnak tekintjük, s a végét a megfelelő részfájlban keressük.

A napló egy lehetséges folytatását a 8.4. ábra mutatja. Tegyük fel, hogy e ponton lép fel rendszerhiba. A naplót a végétől visszafelé elemezve, T_3-at fogjuk az egyetlen be nem fejezett tranzakciónak találni, és így E és F korábbi értékeit, 25-öt és 30-at kell csak visszaállítanunk. Amikor megtaláljuk a <CKPT> bejegyzést, tudjuk, hogy nem kell a korábbi naplóbejegyzéseket elemeznünk, és tudjuk, hogy az adatbázis állapotának helyrehozásával végeztünk.
□

8.2.5. Ellenőrzőpont-képzés a rendszer működése közben[10]

A 8.2.4. részben bemutatott ellenőrzőpont-képzési technika problémája, hogy gyakorlatilag le kell állítani a rendszer működését (nem engedni új tranzakciók indítását) az ellenőrzőpont elkészültéig. Minthogy az aktív tranzakciók még hosszabb időt igényelhetnek a normális vagy abnormális befejeződésükig, így a felhasználó számára a rendszer leállítottnak tűnhet. Egy jóval bonyolultabb módszerrel, a *működés közbeni ellenőrzőpont-képzéssel* elérjük, hogy az ellenőrzőpont-képzés alatt új tranzakciók indítását ne kelljen szüneteltetni. E módszer lépései:

1. <START CKPT (T_1, ..., T_k)> naplóbejegyzés készítése és a naplóbejegyzés lemezre írása (FLUSH). T_1, ..., T_k az éppen *aktív* tranzakciók nevei.

[10] Az eredeti műben „nonquiescent checkpointing", azaz „nem nyugalmi ellenőrzőpont-képzés"-ként szerepel. *A fordító megjegyzése.*

2. Meg kell várni a T_1, ..., T_k tranzakciók mindegyikének normális vagy abnormális befejeződését, nem tiltva közben újabb tranzakciók indítását.
3. Ha a T_1, ..., T_k tranzakciók mindegyike befejeződött, akkor <END CKPT> naplóbejegyzés elkészítése és a naplóbejegyzés lemezre írása (FLUSH).

Az ilyen típusú napló felhasználásával a következőképpen tudunk rendszerhiba után helyreállítani: a naplót a végétől visszafelé elemezve megtaláljuk az összes nem befejezett tranzakciót, régi értékére visszaállítjuk az ezen tranzakciók által megváltoztatott adatbáziselemek tartalmát. Két eset fordulhat elő aszerint, hogy visszafelé olvasván a naplót, az <END CKPT> naplóbejegyzést vagy a <START CKPT (T_1, ..., T_k)> naplóbejegyzést találjuk előbb.

- Ha előbb az <END CKPT> naplóbejegyzéssel találkozunk, akkor tudjuk, hogy az összes még be nem fejezett tranzakcióra vonatkozó naplóbejegyzést a legközelebbi korábbi <START CKPT (T_1, ..., T_k)> naplóbejegyzésig megtaláljuk. Ennél a <START CKPT (T_1, ..., T_k)> naplóbejegyzésnél megállhatunk, a még korábbiakat már nem kell használnunk, azokat el is dobhatjuk.
- Amennyiben a <START CKPT (T_1, ..., T_k)> naplóbejegyzéssel találkozunk előbb, az azt jelenti, hogy a katasztrófa az ellenőrzőpont-képzés közben fordult elő. Ennek következtében T_1, ..., T_k tranzakciók nem fejeződtek be (legalábbis nem tudtuk a befejeződést regisztrálni) a hiba fellépéséig. Ekkor a be nem fejeződött tranzakciók közül a legkorábban kezdődött tranzakció indulásáig kell a naplóban visszakeresnünk, annál korábbra nem. Az ezt megelőző START CKPT bejegyzés biztosan megelőzi a keresett összes tranzakció indítását leíró bejegyzéseket.[11] Ezenfelül, ha ugyanazon tranzakció naplóbejegyzéseire nézve láncokat is használunk, akkor nem kell a napló minden bejegyzését átnéznünk ahhoz, hogy megtaláljuk a még be nem fejezett tranzakciókra vonatkozó bejegyzéseket, elegendő csak az adott tranzakció bejegyzései láncán visszafelé haladnunk.

Általános szabályként, ha egy <END CKPT> naplóbejegyzést kiírunk lemezre, akkor a megelőző START CKPT bejegyzésnél korábbi naplóbejegyzéseket törölhetjük.

8.6. példa: Tegyük fel, hogy a napló, mint a 8.5. példában is, így kezdődik:

<START T_1>
<T_1,A,5>
<START T_2>
<T_2,B,10>

S most úgy döntünk, hogy működés közbeni ellenőrzőpontot hozunk létre. Mint-

[11] Mivel működés közbeni ellenőrzőpont-képzéssel dolgozunk, előfordulhat, hogy a be nem fejeződött tranzakciók némelyike az előző ellenőrzőpont-képzés kezdete és befejezése között indult el.

hogy e pillanatban T_1 és T_2 aktív (nem befejezett) tranzakciók, ezért a következő naplóbejegyzést kell felírnunk:

<START CKPT (T_1, T_2)>

Tegyük fel, hogy amíg T_1 és T_2 befejeződésére várunk, azalatt egy másik tranzakció, T_3 elkezdődik. A napló egy lehetséges folytatását a 8.5. ábrán mutatjuk be.

Tételezzük fel, hogy most lépett fel valamilyen hiba. A naplót a végétől visszafelé vizsgálva, úgy fogjuk találni, hogy T_3 egy be nem fejezett tranzakció, s ezért hatásait semmissé kell tenni. Az utolsó naplóbejegyzés arról informál bennünket, hogy az F adatbáziselembe a 30 értéket kell visszaállítani. Amikor az <END CKPT> naplóbejegyzést találjuk, tudjuk, hogy az összes be nem fejezett tranzakciók a megelőző START CKPT naplóbejegyzés után indulhattak csak el. Tovább visszafelé elemezve, megtaláljuk a <T_3,E,25> bejegyzést, mely megmondja nekünk, hogy az E adatbáziselem értékét 25-re kell visszaállítani. Ezen bejegyzés és a START CKPT naplóbejegyzés között további elindult, de be nem fejeződött tranzakcióra vonatkozó bejegyzést, és további adatbázis-módosításra vonatkozó bejegyzést nem találunk, így az adatbázison mást már nem kell megváltoztatnunk.

Tegyük fel most, hogy az ellenőrzőpont képzése közben történt katasztrófa, s a

<START T_1>
<T_1,A,5>
<START T_2>
<T_2,B,10>
<START CKPT (T_1, T_2)>
<T_2,C,15>
<START T_3>
<T_1,D,20>
<COMMIT T_1>
<T_3,E,25>
<COMMIT T_2>
<END CKPT>
<T_3,F,30>

8.5. ábra. *Napló működés közbeni ellenőrzőpont-képzéssel*

<START T_1>
<T_1,A,5>
<START T_2>
<T_2,B,10>
<START CKPT (T_1, T_2)>
<T_2,C,15>
<START T_3>
<T_1,D,20>
<COMMIT T_1>
<T_3,E,25>

8.6. ábra. *Napló ellenőrzőpont-képzés közben történt rendszerkatasztrófa során*

napló vége a 8.6. ábrán bemutatott. Visszafelé elemezve a naplót, azonosítjuk a T_3, majd a T_2 tranzakciókat, melyek nincsenek befejezve, s helyreállító módosításokat kell tennünk. Amikor megtaláljuk a <START CKPT (T_1, T_2)> naplóbejegyzést, megtudjuk, hogy az egyetlen további olyan tranzakció, mely lehetséges, hogy nincs befejezve, a T_1. Minthogy azonban a <COMMIT T_1> bejegyzést már láttuk, ebből tudjuk, hogy T_1 *nem* be nem fejezett tranzakció. Láttuk már továbbá a <START T_3> bejegyzést is, s így már tudjuk, hogy csak addig kell folytatnunk a napló visszafelé elemzését, amíg T_2 START bejegyzését meg nem találjuk. Eközben még a B adatbáziselem értékét is visszaállítjuk 10-re. □

8.2.6. Feladatok

8.2.1. feladat: Adjuk meg a 8.1.1. feladatban szereplő tranzakciók (nevezzük mindet T-nek) semmisségi (undo) naplóbejegyzéseit. Tegyük fel, hogy kezdetben $A = 5$ és $B = 10$.

8.2.2. feladat: Az alábbi naplóbejegyzés-sorozatok valamely T tranzakció tevékenységeit tükrözik. Állapítsa meg a semmisségi (undo) naplózás szabályainak megfelelően a naplóbejegyzések és az adatbáziselemeket tartalmazó blokkok lemezre írási lehetőségeit, figyelembe véve, hogy naplóbejegyzést nem lehet addig a lemezre írni, amíg a megelőző bejegyzés nem került lemezre.

* a) <START T>; <T,A,10>; <T,B,20>; <COMMIT T>;
b) <START T>; <T,A,10>; <T,B,20>; <T,C,30>; <COMMIT T>;

! **8.2.3. feladat:** A 8.2.2. feladatban szereplő naplórészleteket bővítsük ki olyan tranzakciók naplóivá, melyek n számú adatbáziselemnek új értéket adnak. Mennyi szabályos, naplózott esemény lesz a tranzakciókban, ha betartjuk a semmisségi naplózás szabályait?

8.2.4. feladat: A következő naplóbejegyzés-sorozat a T és U két tranzakcióra vonatkozik: <START T>; <T,A,10>; <START U>; <U,B,20>; <T,C,30>; <U,D,40>; <COMMIT U>; <T,E,50>; <COMMIT T>. Adjuk meg a helyreállítás-kezelő tevékenységeit, beleértve a lemezen és a naplóban tett módosításait, ha katasztrófa lépett fel, és az utolsó lemezre került naplóbejegyzés:

a) <START U>.
* b) <COMMIT U>.
c) <T,E,50>.
d) <COMMIT T>.

8.2.5. feladat: A 8.2.4. feladatban leírt helyzetek mindegyikére adjuk meg, hogy a T és U által lemezre írott értékek közül melyeknek *kell* megjelenni a lemezen, és melyek jelen*het*nek meg a lemezen?

*! **8.2.6. feladat:** Tegyük fel, hogy a 8.2.4. feladatban szereplő U tranzakciót úgy módosítjuk, hogy az <U,D,40> bejegyzés helyett az <U,A,40> keletkezzen. Mi annak a hatása az A lemezen tárolt értékére, ha a 8.2.4. feladatban megadott pillanatokban lép fel katasztrófa? Mit mutat ez a példa a naplózás lehetőségeiről a tranzakciók atomossága megőrzésében?

8.2.7. feladat: Tegyük fel, hogy a napló a következő bejegyzéssorozatot tartalmazza: <START S>; <S,A,60>; <COMMIT S>; <START T>; <T,A,10>; <START U>; <U,B,20>; <T,C,30>; <START V>; <U,D,40>; <V,F,70>; <COMMIT U>; <T,E,50>; <COMMIT T>; <V,B,80>; <COMMIT V>. Tegyük fel továbbá, hogy a működés közbeni ellenőrzőpontképzést kezdjük alkalmazni, közvetlenül az alábbi naplóbejegyzések (memóriában való) megjelenésétől kezdve:

a) <S,A,60>.
* b) <T,A,10>.
c) <U,B,20>.
d) <U,D,40>.
e) <T,E,50>.

Mindegyik fenti esetre adjuk meg, hogy:

i) Mikor íródik fel az <END CKPT> naplóbejegyzés, és
ii) Bármelyik lehetséges pillanatban, ha hiba lép fel, meddig kell a naplóban visszafelé tekinteni ahhoz, hogy minden befejezetlen tranzakciókra vonatkozó bejegyzést megtaláljunk.

8.3. Helyrehozó naplózás (redo logging)

A semmisségi naplózás (undo logging) természetes és egyszerű stratégiát valósít meg a napló kezelésére és a rendszerhibák esetén való visszaállításra, de a probléma megoldásának nem ez az egyetlen lehetséges megközelítése. A semmisségi naplózás potenciális problémája az, hogy csak azután tudjuk befejezni a tranzakciót, ha az összes adatbázis-módosításai már lemezre íródtak. Olykor a lemezműveletekkel tudnánk takarékoskodni, ha megengednénk, hogy az adatbázis-módosításokat csak a memóriában végezzék a tranzakciók, miközben a napló az eseményeket rögzíti, azért, hogy katasztrófa esetében is biztonságban legyen az adatbázis.

Az adatbáziselemek lemezre való azonnali visszaírásának kényszerét elkerülhetjük, ha a *helyrehozó naplózás* (redo logging) módszerét választjuk. Az alapvető különbségek a semmisségi és a helyrehozó naplózás között az alábbiak:

1. Amíg a semmisségi naplózás a helyreállítás során a be nem fejezett tranzakciók hatásait semmissé teszi, a befejezett tranzakciók hatásait pedig nem módosítja, ad-

dig a helyrehozó naplózás figyelmen kívül hagyja a be nem fejezett tranzakciókat, és megismétli a normálisan befejezettek által végrehajtott változtatásokat.

2. A semmisségi naplózás megkívánja az adatbáziselemek lemezen való módosítását a COMMIT naplóbejegyzés lemezre írása előtt, addig a helyrehozó naplózás a COMMIT naplóbejegyzés lemezre írását várja el, mielőtt bármit is változtatna a lemezen lévő adatbázisban.
3. A semmisségi naplózás U_1 és U_2 szabályainak betartása mellett csak a módosított adatbáziselemek régi tartalmát kell megőriznünk az esetleges visszaállítás biztosításához, a helyrehozó naplózással történő helyreállításhoz a módosított elemek új értékére van szükség. Emiatt a helyrehozó naplózás naplóbejegyzései ugyanolyan formájúak, de más a jelentésük, mint a semmisségi naplózásnál alkalmazottaké.

8.3.1. A helyrehozó naplózás szabályai

A helyrehozó naplózás az adatbáziselemek módosítását a naplóbejegyzésben az új értékkel képviseli (nem pedig a régivel, mint a semmisségi naplózásnál). Ez a bejegyzés ugyanúgy néz ki, mint a semmisségi naplózásnál használt: <*T*,*X*,*v*>, a jelentése azonban más. E bejegyzés jelentése: „a *T* tranzakció az *X* adatbáziselemnek a *v* értéket adta". E bejegyzésben az *X* régi értékét nem jelzi semmi. Mindig, ha a *T* tranzakció módosítja az *X* adatbáziselem értékét, akkor egy <*T*,*X*,*v*> bejegyzést kell a naplóba írni.

Annak sorrendjét, hogy az adat- és naplóbejegyzések hogyan kell hogy lemezre kerüljenek, az alábbi egyszerű „helyrehozó naplózási szabály", az úgynevezett *írj korábban naplózási szabály* írja le.

R_1: Mielőtt az adatbázis bármely *X* elemét a lemezen módosítanánk, szükséges, hogy az *X* ezen módosítására vonatkozó összes naplóbejegyzése, azaz <*T*,*X*,*v*> és <COMMIT *T*>, a lemezre kerüljenek.

Minthogy a COMMIT bejegyzést csak akkor írhatjuk a naplóba, ha a tranzakció teljesen és hibamentesen befejeződött, így a COMMIT bejegyzés csak a módosításokat leíró bejegyzések után állhat, ezért úgy is összegezhetjük az R_1 szabálya hatását, hogy: ha helyrehozó naplózást használunk, akkor az egy tranzakcióra vonatkozó lemezre írásoknak a következő sorrendben kell megtörténniük:

1. Az adatbáziselemek módosítását leíró naplóbejegyzések lemezre írása.
2. A COMMIT naplóbejegyzés lemezre írása.
3. Az adatbáziselemek értékének tényleges cseréje a lemezen.

8.7. példa: Tanulmányozzuk ugyanazt a tranzakciót, amelyiket a 8.3. példában is elemeztünk. A 8.7. ábrán látható ezen tranzakcióra vonatkozó események lehetséges sorrendje.

A főbb különbségek a 8.7. és 8.3. ábrák között a következők: először nézzük a 8.7. ábra 4) és 7) sorait, ezekben a módosítást leíró naplóbejegyzésben az *A* és *B* adatbáziselemek új értéke szerepel (s nem a régi, mint a 8.3. ábrán). A másik különbség, hogy a COMMIT bejegyzés korábbra került, a 8) lépésbe. Ezt követően a napló lemezre írását kiváltó FLUSH LOG következik, s így a *T* tranzakció által végrehajtott módosításokat leíró összes naplóbejegyzés lemezre íródik. Csak ezt követően kerül lemezre az *A* és *B* új, módosított értéke. Az ábrán ezen új értékek kiírását a közvetlenül következő 10) és 11) sorokban láthatjuk, bár a gyakorlatban ezekre esetleg csak később kerül sor. □

Lépés	*Tevékenység*	*t*	*M*-A	*M*-B	*D*-A	*D*-B	*Napló*
1)							<START *T*>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<*T*,*A*,16>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<*T*,*B*,16>
8)							<COMMIT *T*>
9)	FLUSH LOG						
10)	OUTPUT(A)	16	16	16	16	8	
11)	OUTPUT(B)	16	16	16	16	16	

8.7. ábra. *Tevékenységek és naplóbejegyzéseik helyrehozó naplózás használatakor*

8.3.2. Helyreállítás a helyrehozó naplózás használatával

A helyrehozó naplózás R_1 szabályának fontos következménye, hogy ha a naplóban nincs <COMMIT *T*> bejegyzés, akkor tudjuk, hogy a *T* tranzakció nem hajtott végre az adatbázisban módosítást a lemezen. Így a be nem fejezett (nem teljes) tranzakciók a helyreállítás során úgy tekinthetők, mintha meg sem történtek volna. Problémát a befejezett (COMMIT) tranzakciók jelenthetnek, mert nem tudjuk, hogy az általuk elvégzett adatbázis-változtatások közül melyik íródott már lemezre. Szerencsére a helyrehozó naplózás naplója éppen azon információkat – az új értékeket – tartalmazza, melyekre szükségünk van a helyreállításhoz. Ezen új értékeket kell lemezre írnunk, attól függetlenül, hogy esetleg már korábban is kiírodtak. A rendszerkatasztrófa bekövetkezése után a helyrehozó naplózással történő helyreállításhoz a következőket kell tennünk:

1. Meghatározni a befejezett (COMMIT) tranzakciókat.
2. Elemezni a naplót az elejétől kezdve. Minden <*T*,*X*,*v*> naplóbejegyzés megtalálásakor:

   a) Ha *T* nem befejezett tranzakció, akkor nem kell tenni semmit.
   b) Ha *T* befejezett tranzakció, akkor *v* értéket kell az *X* adatbáziselembe írni.

3. Minden *T* be nem fejezett tranzakcióra vonatkozóan <ABORT *T*> naplóbejegyzést kell a naplóba írni, és a naplót ki kell írni lemezre (FLUSH LOG).

8.8. példa: Tegyük fel, hogy a napló a 8.7. ábrának megfelelő, nézzük meg hogyan lehet a helyreállítást elvégezni a különböző pillanatokban bekövetkező katasztrófák esetében.

1. Ha a katasztrófa a 9) lépés után bármikor következik be, akkor a <COMMIT T> bejegyzés már lemezen van. A helyreállító rendszer T-t befejezett tranzakcióként azonosítja. Amikor a naplót az elejétől kezdve elemzi, a <T,A,16> és a <T,B,16> bejegyzések hatására a helyreállítás-kezelő az A és B adatbáziselemekbe a 16 értéket írja. Megjegyezzük, hogy ha a katasztrófa a 10) és 11) lépések között következett be, akkor A újraírása redundáns ugyan, de B írása (korában nem történt meg) lényeges lépés az adatbázis konzisztens állapotának eléréséhez. Amennyiben a hiba a 11) lépést követően keletkezett, akkor mindkét adatbáziselem új értékének lemezre írása redundáns ugyan, de semmi gondot nem okoz.
2. Ha a hiba a 8) és 9) lépések között jelentkezik, akkor bár a <COMMIT T> bejegyzés már a naplóba került, de nem biztos, hogy lemezre íródott (ez attól függ, hogy esetleg valami más okból sor került-e a napló lemezre írására). Ha lemezre került, akkor a helyreállítási eljárás az 1) esetnek megfelelően történik. Ha pedig a napló még nem került lemezre, akkor a helyreállítás a következő, 3) esettel megegyező.
3. Ha a katasztrófa a 8) lépést megelőzően keletkezik, akkor <COMMIT T> naplóbejegyzés még biztosan nem került lemezre, így T be nem fejezett tranzakciónak tekintendő. Ennek megfelelően A és B értékeit a lemezen még nem változtatta meg a T tranzakció, nincs mit helyreállítani, s végül egy <ABORT T> bejegyzést írunk a naplóba.

□

8.3.3. Helyrehozó naplózás ellenőrzőpont-képzés használatával

A semmisségi naplózásnál látottakhoz hasonlóan a helyrehozó naplózás naplójába is illeszthetünk ellenőrzőpontokat. A helyrehozó naplózásnál azonban új probléma jelentkezik: Minthogy a befejeződött tranzakciók módosításainak lemezre írása a befejeződés után sokkal később is történhet, így az e vonatkozásban ugyanazon pillanatban aktív tranzakciók számát nem tudjuk korlátozni, azon pillanatban sem, amikor az ellenőrzőpont létrehozásáról döntünk. Tekintet nélkül arra, hogy az ellenőrzőpont-képzés alatt tranzakciók indulását megengedjük vagy sem, a kulcsfeladat – amit meg kell tennünk az ellenőrzőpont-készítés kezdete és befejezése közötti időben – azon összes adatbáziselem lemezre való kiírása, melyeket befejezett tranzakciók módosítottak, és még nem voltak lemezre kiírva. Ennek megvalósításához a pufferkezelőnek nyilván kell tartania a *piszkos* puffereket, melyekben már végrehajtott, de lemezre még ki nem írt módosításokat tárol. Azt is tudnunk kell, mely tranzakciók mely puffereket módosították.

Más oldalról viszont, be tudjuk fejezni az ellenőrzőpont-képzést az aktív tranzakciók (normális vagy abnormális) befejezésének kivárása nélkül, mert ők ekkor még amúgy sem engedélyezik lapjaik lemezre írását. A helyrehozó naplózásban a működés közbeni ellenőrzőpont-képzés a következőkből áll:

A helyrehozó naplózás eseményeinek sorrendje

Mivel sok befejezett tranzakció is adhatott új értéket ugyanazon X adatbázis-elemnek, ezért a helyrehozó naplózás alkalmazásakor a naplót a korábbi bejegyzésektől a későbbiek felé időrendben haladva kell elemeznünk. Így érhető el, hogy X adatbázisbeli végső értéke – ahogy kell – a normálisan befejeződött tranzakciók által utoljára adott legyen. Ugyanazt az állapotot érjük el tehát, mint ami a semmisségi naplózásnál a napló visszafelé elemzésével volt elérhető.

Ha az adatbázisrendszerünk az atomosságot követeli meg, akkor a semmisségi naplózásnál nem tudtuk pontosan megállapítani, két be nem fejeződött tranzakció esetében, hogy azok módosították-e ugyanazon adatbáziselemet. Ezzel szemben a helyrehozó naplózás alkalmazásával a befejeződött tranzakciókra figyelünk, ha szükséges, ezek módosításait megismételve állítjuk helyre az adatbázis konzisztens állapotát. Ez teljesen rendben van, két rendben befejezett (COMMIT) tranzakció esetében akkor is, ha mindkettő ugyanazon adatbáziselemet módosította különböző pillanatokban. A helyreállítás helyes sorrendje itt mindig fontos, nem úgy, mint a semmisségi naplózás esetében volt (amennyiben a konkurenciafelügyelet megfelelő formája működött).

1. <START CKPT (T_1, ..., T_k)> naplóbejegyzés elkészítése és kiírása lemezre, ahol T_1, ..., T_k az összes éppen aktív (még be nem fejezett) tranzakció.
2. Az összes olyan adatbáziselem kiírása lemezre, melyeket olyan tranzakciók írtak pufferekbe, melyek a START CKPT naplóba írásakor már befejeződtek, de puffereik lemezre még nem kerültek.
3. <END CKPT> bejegyzés naplóba írása és a napló lemezre írása (FLUSH LOG).

<START T_1>
<T_1,A,5>
<START T_2>
<COMMIT T_1>
<T_2,B,10>
<START CKPT (T_2)>
<T_2,C,15>
<START T_3>
<T_3,D,20>
<END CKPT>
<COMMIT T_2>
<COMMIT T_3>

8.8. ábra. *A helyrehozó naplózás naplója*

8.9. példa: A 8.8. ábra egy lehetséges naplót mutat, melynek közepén ellenőrzőpont található. Amikor az ellenőrzőpont-képzés elkezdődött, csak T_2 volt aktív, de a T_1 által A-ba írott érték még csak esetleg került lemezre. Ha még nem, akkor A-t lemezre

kell másolnunk, mielőtt az ellenőrzőpont-képzést befejezhetnénk. A napló érzékelteti, hogy az ellenőrzőpont-képzés befejezéséig más események is bekövetkezhetnek: T_2 a C adatbáziselem tartalmát módosítja, elindul T_3 új tranzakció, és módosítja D értékét. Az ellenőrzőpont-képzés befejezése után már csak T_2 és T_3 tranzakciók befejeződése történt meg. □

8.3.4. Visszaállítás az ellenőrzőponttal kiegészített helyrehozó típusú naplózással

Mint a semmisségi naplózásnál, most is, az ellenőrzőpontok naplóba illesztése segít a naplóátvizsgálás korlátozásában, amikor adatbázis-helyreállítás szükséges. Szintén a semmisségi naplózáshoz hasonlóan két eset fordulhat elő, attól függően, hogy az utolsó ellenőrzőpont-bejegyzés a START vagy az END.

- Tegyük fel először, hogy a katasztrófa előtt a naplóba feljegyzett utolsó ellenőrzőpont-bejegyzés <END CKPT>. Ekkor tudjuk, hogy az olyan értékek, melyeket olyan tranzakciók írtak, melyek a <START CKPT (T_1, ..., T_k)> naplóbejegyzés megtétele előtt befejeződtek, már biztosan lemezre kerültek, s így nem kell velük foglalkoznunk helyreállítandó ezen tranzakciók hatását. Foglalkoznunk kell viszont a T_i-k közé tartozó, valamint az ellenőrzőpont kialakításának megkezdése után induló tranzakciókkal, ezeknek lehetnek olyan adatbázis-módosításaik, melyek még nem kerültek lemezre, pedig a tranzakció már befejeződött. Ekkor olyan visszaállítást kell tennünk, amilyenről a 8.3.2. részben már szó volt, azzal a különbséggel, hogy figyelmünket azon tranzakciókra korlátozhatjuk, melyek az utolsó <START CKPT (T_1, ..., T_k)> naplóbejegyzésben a T_i-k között szerepelnek, vagy ezen naplóbejegyzést követően indultak el. A naplóban való keresés során a legkorábbi <START T_i> naplóbejegyzésig kell visszamennünk, annál korábbra már nem. Megjegyezzük, hogy ezek a START naplóbejegyzések akárhány korábbi ellenőrzőpontnál korábban is felbukkanhatnak. Ahogy a semmisségi naplózásnál is láttuk, az adott tranzakcióra vonatkozó naplóbejegyzések visszafelé keresése segít megtalálni a számunkra éppen fontos bejegyzéseket.
- Tegyük fel most, hogy a naplóba feljegyzett utolsó ellenőrzőpont-bejegyzés a <START CKPT (T_1,..., T_k)> naplóbejegyzés. Nem lehetünk abban biztosak, hogy az ezt megelőzően befejezett tranzakciók által módosított adatbáziselemek már lemezre íródtak. Ezért az előző <END CKPT> bejegyzéshez tartozó <START CKPT (S_1, ..., S_m)> naplóbejegyzésig[12] vissza kell keresnünk, és helyre kell állítanunk az olyan befejeződött tranzakciók tevékenységének eredményeit, melyek ez utóbbi <START CKPT (S_1, ..., S_m)> naplóbejegyzés után indultak, vagy az S_i-k közül valók.

8.10. példa: Tekintsük ismét a 8.8. ábrán bemutatott naplót. Ha a katasztrófa a végén lép fel, akkor az <END CKPT> bejegyzésig kell visszakeresnünk. Ekkor tudjuk, hogy a helyreállítás szempontjából elegendő csak azon tranzakciókat figyelembe venni, melyek egyrészt a <START CKPT (T_2)> bejegyzés felírását követően indultak, vagy szerepelnek e bejegyzés listájában (most csak T_2). Így a vizsgálandó tranzakcióhalmazunk {T_2, T_3}. <COMMIT T_2> és <COMMIT T_3> bejegyzéseket találunk, s ebből tudjuk, hogy mindkettő tranzakció hatását helyre kell állítanunk. A naplóban visszafelé meg kell keresnünk a <START T_2> bejegyzést, s innen már időrendben haladva a naplóban a következő – T_2, T_3 befejezett tranzakciókra vonatkozó – módosítást leíró bejegyzéseket találjuk: <T_2,B,10>, <T_2,C,15> és <T_3,D,20>. Mivel azt nem tudjuk, hogy ezen változtatások a lemezen már megtörténtek-e, ezért most a lemezre újraírjuk a B, C és D tartalmát, megfelelően 10, 15 és 20 értékeket adva nekik.

Tegyük fel most, hogy a katasztrófa a <COMMIT T_2> és <COMMIT T_3> bejegyzések között történt. A helyreállítás az előbbi esethez hasonló, azzal a különbséggel, hogy T_3 nem befejezett tranzakció, ennek megfelelően a <T_3,D,20> helyreállítást *nem* kell végrehajtani. D értékét a helyreállítás során nem változtatjuk meg, hacsak a vizsgált naplórészben található, más tranzakció bejegyzése miatt meg nem kell változtatnunk. A helyreállítást követően egy <ABORT T_3> bejegyzést írunk a naplóba.

Végül, ha a hiba az <END CKPT> bejegyzést megelőzően lépett fel, akkor az utolsó előtti START CKPT bejegyzést kell megkeresnünk (melynek már van <END CKPT> párja), és annak listájából tudjuk meg, melyek az aktív tranzakciók. Ha nem találunk korábbi ellenőrzőpont-bejegyzést, akkor mindenképpen a napló elejére kell mennünk. Így esetünkben az egyedüli befejezett tranzakciónak T_1-et fogjuk találni, s ezért a <T_1,A,5> tevékenységét helyreállítjuk. A helyreállítást követően <ABORT T_2> és <ABORT T_3> bejegyzéseket írunk a naplóba. □

Minthogy a tranzakciók több ellenőrzőpont készítésekor is aktívak lehetnek, célszerű lehet, hogy a <START CKPT (T_1, ..., T_k)> naplóbejegyzésbe nem csak az aktív tranzakciók neveit, hanem olyan mutatókat is elhelyezzünk, melyek az aktív tranzakciók indulását leíró bejegyzések naplóbeli helyét adják meg. Így eljárva, biztonsággal meg tudjuk állapítani, hogy a napló mely korábbi részeit törölhetjük. Amikor <END CKPT> bejegyzést írunk a naplóba, akkor tudjuk, hogy a naplóban már sosem kell korábbra visszatekintenünk, mint ahol a T_i aktív tranzakcióra vonatkozó, legkorábbi <START T_i> bejegyzést találjuk. Következésképpen az ezen START bejegyzést megelőző bejegyzések mindegyike törölhető.

[12] Előzetes hiba miatt előfordulhat, hogy a START CKPT bejegyzésnek nincs <END CKPT> párja. Ezért kell úgy eljárnunk, hogy nem csak a korábbi START CKPT bejegyzést keressük, hanem előbb egy <END CKPT>-t, majd az ezt megelőző START CKPT-t.

8.3.5. Feladatok

8.3.1. feladat: Adjuk meg a 8.1.1. feladatban szereplő tranzakciók (nevezzük mindet T-nek) helyreállítási típusú naplóbejegyzéseit. Tegyük fel, hogy kezdetben $A = 5$ és $B = 10$.

8.3.2. feladat: Ismételjük meg a 8.2.2. feladatot, helyreállítási típusú naplózást használva.

8.3.3. feladat: Ismételjük meg a 8.2.4. feladatot, helyreállítási típusú naplózást használva.

8.3.4. feladat: Ismételjük meg a 8.2.5. feladatot, helyreállítási típusú naplózást használva.

8.3.5. feladat: A 8.2.7. feladat adatait használva az a)–e) helyzetek mindegyikére válaszoljuk meg az alábbi kérdéseket:

i) Mely pontokban fordulhat elő az <END CKPT> felírása, és
ii) Minden lehetséges hibabekövetkezési ponthoz adjuk meg, hogy a naplóban meddig kell visszatekintenünk ahhoz, hogy megtaláljuk az összes befejezetlen tranzakciót. Vegyük figyelembe mindkét lehetőséget, azt is, hogy a hibát megelőzően az <END CKPT> felíródott a naplóba és azt is, ha nem.

8.4. A semmisségi/helyrehozó (undo/redo) naplózás

Láthattuk, hogy a naplózás két különböző megközelítése abban mutat eltérést, hogy a napló az adatbáziselemek értékének módosítása esetén a régi (módosítás előtti) vagy az új (módosítás utáni) érteket tartalmazza. Mindkét módszernek vannak bizonyos hátrányai is:

- A semmisségi (undo) naplózás alkalmazása megköveteli, hogy az adatokat a tranzakció befejezésekor nyomban lemezre írjuk, ezzel (esetleg jelentősen) növeljük a végrehajtandó lemezműveletek számát.
- Másik oldalról, a helyrehozó (redo) naplózás minden módosított adatbázisblokk pufferben tartását igényli, egészen a tranzakció rendes és teljes befejezéséig (commit), a napló kezelésével együtt (esetleg jelentősen) növeli a tranzakciók átlagos pufferigényét.
- Mindkét naplózási módszer az ellenőrzőpont képzése közben ellentétes igényeket támaszt a pufferek lemezre írása szempontjából, kivéve, ha az adatbáziselemek teljes blokkok vagy blokkok sokasága. Például, ha a puffer tartalmaz egy *A* adatbáziselemet, melyet egy rendesen és teljesen befejezett tranzakció módosított, és tartalmaz egy *B* adatbáziselemet is, melyet olyan tranzakció módosított, melyre vonatkozóan a COMMIT bejegyzés még nem került lemezre, akkor az R_1 szabálynak megfelelően, a puffer lemezre másolását igényeljük *A* miatt, viszont tiltjuk ennek megtételét *B* miatt.

Most a *semmisségi/helyrehozó* (undo/redo)-nak nevezett naplózást vizsgáljuk meg. Ez a módszer a tevékenységek elvégzési sorrendjének rugalmasságát növeli azáltal, hogy bővíti a naplózott információk körét.

8.4.1. A semmisségi/helyrehozó (undo/redo) naplózás szabályai

A semmisségi/helyrehozó naplózás, egyetlen különbséggel, ugyanolyan típusú naplóbejegyzéseket használ, mint a naplózás többi módszere. E módszerben az adatbáziselem értékének módosítását leíró naplóbejegyzés négykomponensű. A <*T*,*X*,*v*,*w*> naplóbejegyzés azt jelenti, hogy a *T* tranzakció az adatbázis *X* elemének korábbi *v* értékét *w*-re módosította. A semmisségi/helyrehozó naplózást alkalmazó rendszer a következő előírást kell hogy betartsa:

UR_1: Mielőtt az adatbázis bármely *X* elemének értékét – valamely *T* tranzakció által végzett módosítás miatt – a lemezen módosítanánk, ezt megelőzően a <*T*,*X*,*v*,*w*> módosítást leíró naplóbejegyzésnek lemezre kell kerülnie.

A semmisségi/helyrehozó naplózás, UR_1 szabálya csak azokat a feltételeket kényszeríti, amelyek a semmisségi és a helyrehozó naplózási szabályok mindegyikében szerepelnek. Speciálisan, a <COMMIT *T*> bejegyzés megelőzheti és követheti is az adatbáziselemek lemezen történő bármilyen megváltoztatását.

8.11. példa: A 8.9. ábra, az utoljára a 8.7. példában látott, *T* tranzakcióhoz tartozó naplóbejegyzések sorrendjének egy változatát mutatja. Megjegyezzük, hogy a módosítást leíró naplóbejegyzések már az *A* és *B* adatbáziselemeknek mind a régi, mind az új értékét tartalmazzák. Ebben a sorozatban a <COMMIT *T*> naplóbejegyzés kiírását az *A* és *B* adatbáziselemek lemezre való írása közé tettük. A 10) lépés kerülhetett volna a 9) lépés elé vagy a 11) lépés mögé is. □

Lépés	*Tevékenység*	*t*	*M*-A	*M*-B	*D*-A	*D*-B	*Napló*
1)							<START *T*>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<*T*,*A*,8,16>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<*T*,*B*,8,16>
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)							<COMMIT *T*>
11)	OUTPUT(B)	16	16	16	16	16	

8.9. ábra. *Tevékenységek és naplóbejegyzéseik lehetséges sorrendje semmisségi/helyrehozó naplózás használatakor*

8.4.2. Helyreállítás a semmisségi/helyrehozó (undo/redo) naplózás használatakor

Amikor a semmisségi/helyrehozó naplózást használjuk, és helyreállításra kényszerülünk, akkor a módosítást leíró naplóbejegyzésben megtaláljuk mind a *T* tranzakció hatásainak semmissé tételéhez szükséges régi, mind a *T* tranzakció hatásainak helyreállításához szükséges új adatbáziselem-értékeket. A semmisségi/helyrehozó módszer alapelvei:

A késleltetett véglegesítés problémája

A semmisségi naplózáshoz hasonlóan a semmisségi/helyrehozó naplózás is olyan viselkedést mutat, hogy a tranzakció a felhasználó számára korrekten befejezettnek tűnik (például: az ügyfél számítógép-hálózaton vásárolt egy repülőjegyet, majd levált a hálózatról), s még a <COMMIT *T*> naplóbejegyzés lemezre kerülése előtt fellépett hiba utáni helyreállítás során a rendszer a tranzakció hatásait semmissé teszi ahelyett, hogy helyreállította volna. Amennyiben ez a lehetőség problémát jelent, akkor a semmisségi/helyrehozó naplózás során egy további szabály használatát javasoljuk:

UR_2 A <COMMIT *T*> naplóbejegyzést nyomban lemezre kell írni, amint megjelenik a naplóban.

Ennek teljesítéséért a 8.9. példánkban a 10) lépés után egy FLUSH LOG lépést kell beiktatnunk.

1. A legkorábbitól kezdve állítsuk helyre minden befejezett tranzakció hatásait.
2. A legutolsótól kezdve tegyük semmissé minden be nem fejezett tranzakció cselekedeteit.

Megjegyezzük, hogy mindkét eljárásra szükségünk van. A rugalmasság lehetővé teszi, hogy a COMMIT bejegyzés és a lemezen végrehajtott adatbázis-módosítások egymáshoz viszonyított sorrendje kötetlen legyen, így előfordulhat az is, hogy egy befejezett tranzakció néhány vagy összes változtatása még nem került lemezre, és az is, hogy egy be nem fejezett tranzakció néhány vagy összes változtatása már lemezen is megtörtént.

8.12. példa: Tegyük fel, hogy az események a 8.9. ábrán látható sorrendben történtek. A hiba fellépésének időpontja függvényében különböző helyreállítási lehetőségeink vannak.

1. Feltéve, hogy a katasztrófa a <COMMIT *T*> naplóbejegyzés lemezre írását követően fordul elő, ekkor *T*-t befejezett tranzakciónak tekintjük. 16-ot írunk mind az *A*, mind *B* adatbáziselemekbe. Az események jelenlegi sorrendjében *A*-nak már 16 a tartalma, de *B*-nek lehet, hogy nem, aszerint, hogy a hiba a 11) lépés előtt vagy után következett be.
2. Ha a katasztrófa a <COMMIT *T*> naplóbejegyzés lemezre írását megelőzően következett be, akkor *T* befejezetlen tranzakciónak számít. Ez esetben az *A* és *B* adatbáziselemek korábbi értéke, 8 íródik lemezre. Ha a hiba a 9) és 10) lépések között következett be, akkor *A* értéke már 16 volt a lemezen, és emiatt a 8-ra való visszaállítás feltétlenül szükséges. Ebben a konkrét példában a *B* értéke nem igényelne visszaállítást (mert még meg sem változott), ha pedig a hiba a 9) lépés előtt következik be, akkor *A* sem igényelné a visszaállítást. Mivel általában nem lehetünk biztosak abban, vajon a visszaállítás szükséges-e vagy sem, így (a biztonság kedvéért) mindig végre kell hajtanunk a visszaállítást.
□

8.4.3. Semmisségi/helyrehozó naplózás ellenőrzőpont-képzéssel

A működés közbeni ellenőrzőpont-képzés valamivel egyszerűbb a semmisségi/helyrehozó naplózás alkalmazásakor, mint más naplózási módszereknél volt. Csak a következőket kell tennünk:

1. Írjunk a naplóba <START CKPT (T_1, ..., T_k)> naplóbejegyzést, ahol T_1, ..., T_k-k az összes éppen aktív tranzakciók, majd írjuk a naplót lemezre.
2. Írjuk lemezre az összes *piszkos* puffert, tehát azokat, melyek egy vagy több módosított adatbáziselemet tartalmaznak. A helyrehozó naplózástól eltérően itt az összes piszkos puffert lemezre írjuk, nemcsak a már befejezett tranzakciók által módosítottakat.
3. Írjunk <END CKPT> naplóbejegyzést a naplóba, majd írjuk a naplót lemezre.

A 2) ponttal kapcsolatban megjegyezzük, hogy a semmisségi/helyrehozó naplózás által, a lemezre írások sorrendjére vonatkozóan biztosított rugalmasság miatt, megengedhetjük a be nem fejezett tranzakciók adatainak lemezre való kiírását. Így megen-

A tranzakciók különös viselkedése a helyreállítás alatt

A figyelmes olvasó észrevehette, hogy nem adtuk meg azt, hogy a semmisségi/helyrehozó (undo/redo) naplózás alkalmazásakor a helyreállítás során a semmisségi (undo) vagy a helyrehozó (redo) lépést tesszük meg előbb. Valóban, azt, hogy a semmisségi vagy a helyrehozó lépéseket tesszük-e meg előbb, nyitva hagytuk a következő szituáció miatt: előfordulhat, hogy a *T* tranzakció rendben és teljesen befejeződött, s emiatt helyreállítása során az általa kialakított *X* értéket rekonstruáljuk, melyet viszont egy be nem fejezett, és ezért visszaállítandó *U* tranzakció korábban módosított. A probléma nem az, hogy először helyreállítjuk *X* értékét, és aztán visszaállítjuk *U* előttire, vagy pedig előbb visszaállítjuk, és utána a *T* által írottra rekonstruáljuk. E szituációban egyik út sem helyes, mert a végső adatbázis-állapot nem felel meg egyik – atomosnak elvárt – tranzakció hatásának sem.

A gyakorlatban az adatbázisrendszereknek a módosítások naplózásánál többet kell tenniük. Biztosítaniuk kell, hogy ilyen szituációk ne fordulhassanak elő. A konkurencia kérdéseivel foglalkozó fejezetben vizsgáljuk azt is, mit jelent a *T* és *U* tranzakciók elkülönítése, amivel az ugyanazon *X* adatbáziselemen való kölcsönhatásuk előfordulása elkerülhető. A 10.1. részben kifejezetten az olyan helyzetek megelőzésével foglalkozunk, amikor a *T* tranzakció egy piszkos – más tranzakció által módosított, de még nem véglegesített – *X* adatbáziselemet használna.

gedhetjük a teljes blokknál kisebb adatbáziselemek használatát is, melyek közös pufferbe kerülnek. A tranzakciókra vonatkozóan egyetlen előírást kell tennünk:

- A tranzakció semmilyen értéket nem írhat (még a memóriapufferbe sem), amíg biztosak nem vagyunk abban, hogy nem abortál.

Amint a 10.1. részben látni fogjuk, ezt a megszorítást szinte mindig be kell tartani ahhoz, hogy elkerülhessük a tranzakciók közötti inkonzisztens kölcsönhatást. Megjegyezzük, hogy a helyrehozó naplózás használatakor a fenti feltétel nem elégséges, éppen ezért írja elő az R_1 szabály, hogy ha egy tranzakció B-t módosítja, akkor a tranzakcióra vonatkozó COMMIT naplóbejegyzésnek előbb kell lemezre íródnia, s csak azután írhatjuk B-t lemezre.

8.13. példa: A 8.10. ábra a semmisségi/helyrehozó naplózás alkalmazását mutatja egy, a 8.8. ábrán (helyrehozó naplózás) látottal megegyező esetre. Csak a módosításokat leíró naplóbejegyzéseket cseréltük, megadva bennük a régi és új értékeket. Az egyszerűség kedvéért feltételeztük, hogy a régi érték mindig eggyel kisebb az új értéknél.

Amint a 8.9. példában is, az ellenőrzőpont képzésének kezdetekor T_2 az egyetlen aktív tranzakció. Minthogy ez a napló semmisségi/helyrehozó napló, így lehetséges, hogy T_2 által B-nek adott új érték, 10, lemezre íródik, ami nem volt megengedett a helyrehozó naplózásban. Most lényegtelen, hogy ez a lemezre írás mikor történik meg. Az ellenőrzőpont képzése alatt biztosan lemezre írjuk B-t (ha még nem került oda), mivel minden piszkos (változásban érintett) puffert kiírunk lemezre. Hasonlóan A-t – melyet a befejezett T_1 tranzakció alakított ki – is lemezre fogjuk írni, ha még nem került oda.

Ha a katasztrófa ezen eseménysorozat végén jelentkezik, akkor a T_2-t és T_3-at teljesen és rendesen befejezett (COMMIT) tranzakciónak tekintjük. T_1 tranzakció az ellenőrzőpontnál korábbi. Minthogy <END CKPT> bejegyzést találunk a naplóban, így T_1-ről biztosan tudjuk, hogy teljesen és rendesen befejeződött, valamint az általa okozott módosítások lemezre íródtak. Ezért, mint a 8.9. példában is, a T_2 és T_3 által végzett módosítások helyreállítandók, T_1 pedig figyelmen kívül hagyható. Amikor olyan tranzakció hatásait állítjuk helyre, mint amilyen a T_2 is, akkor a naplóban nem kell a

<START T_1>
<T_1,A,4,5>
<START T_2>
<COMMIT T_1>
<T_2,B,9,10>
<START CKPT (T_2)>
<T_2,C,14,15>
<START T_3>
<T_3,D,19,20>
<END CKPT>
<COMMIT T_2>
<COMMIT T_3>

8.10. ábra. *A semmisségi/helyrehozó (undo/redo) naplózás naplója*

<START CKPT (T_2)> bejegyzésnél korábbra visszatekinteni, mert tudjuk, hogy az ellenőrzőpont-képzést megelőzően T_2 által végzett módosítások az ellenőrzőpont képzése alatt lemezre íródtak.

Másik példaként, tegyük fel, hogy a katasztrófa éppen <COMMIT T_3> bejegyzés lemezre írását megelőzően fordult elő. Ekkor T_2-t befejezett, T_3-at pedig befejezetlen tranzakciónak kell tekintenünk. T_2 tevékenységét helyreállítandó C értékét a lemezen 15-re írjuk; B-t már nem kell 10-re írnunk a lemezen, mert tudjuk, hogy ez már lemezre került az <END CKPT> előtt. A helyreállító naplózástól eltérő módon pedig a T_3 hatásait semmissé tesszük, azaz a lemezen D tartalmát 19-re írjuk. Ha T_3 az ellenőrzőpont-képzés kezdetekor már aktív tranzakció lett volna, akkor a naplóban a megelőző START CKPT bejegyzésig kellene visszakeresnünk, azért hogy megtaláljuk T_3 semmissé teendő tevékenységeit (az azokat leíró naplóbejegyzéseket). □

8.4.4. Feladatok

8.4.1. feladat: Adjuk meg a 8.1.1. feladatban szereplő tranzakciók (nevezzük mindet T-nek), semmisségi/helyrehozó (undo/redo) típusú naplóbejegyzéseit. Tegyük fel, hogy kezdetben $A = 5$ és $B = 10$.

8.4.2. feladat: Az alábbi naplóbejegyzés-sorozatok valamely T tranzakció tevékenységeit tükrözik. Állapítsuk meg a semmisségi/helyrehozó (undo/redo) naplózás szabályainak megfelelően a naplóbejegyzések és az adatbáziselemeket tartalmazó blokkok lemezre írási lehetőségeit, figyelembe véve, hogy naplóbejegyzést nem lehet addig a lemezre írni, amíg a megelőző bejegyzés nem került lemezre.

* a) <START T>; <T,A,10,11>; <T,B,20,21>; <COMMIT T>;
b) <START T>; <T,A,10,21>; <T,B,20,21>; <T,C,30,31>; <COMMIT T>;

8.4.3. feladat: A következő semmisségi/helyrehozó naplóbejegyzés-sorozat a T és U két tranzakcióra vonatkozik: <START T>; <T,A,10,11>; <START U>; <U,B,20,21>; <T,C,30,31>; <U,D,40,41>; <COMMIT U>; <T,E,50,51>; <COMMIT T>. Adjuk meg a helyreállítás-kezelő tevékenységeit, beleértve a lemezen és a naplóban tett módosításait, ha katasztrófa lépett fel, és az utolsó lemezre került naplóbejegyzés:

a) <START U>.
* b) <COMMIT U>.
c) <T,E,50,51>.
d) <COMMIT T>.

8.4.4. feladat: A 8.4.3 feladatban leírt helyzetek mindegyikére adjuk meg, hogy a T és U által lemezre írott értékek közül melyeknek *kell* megjelenni a lemezen, és melyek jelen*het*nek meg a lemezen?

8.4.5. feladat: Tegyük fel, hogy a napló a következő bejegyzéssorozatot tartalmazza: <START *S*>; <*S*,*A*,60,61>; <COMMIT *S*>; <START *T*>; <*T*,*A*,61,62>; <START *U*>; <*U*,*B*,20,21>; <*T*,*C*,30,31>; <START *V*>; <*U*,*D*,40,41>; <*V*,*F*,70,71>; <COMMIT *U*>; <*T*,*E*,50,51>; <COMMIT *T*>; <*V*,*B*,21,22>; <COMMIT *V*>. Tegyük fel továbbá, hogy a működés közbeni ellenőrzőpont-képzést kezdjük alkalmazni, közvetlenül az alábbi bejegyzések (memóriában való) megjelenésétől kezdve:

a) <*S*,*A*,60,61>.
* b) <*T*,*A*,61,62>.
c) <*U*,*B*,20,21>.
d) <*U*,*D*,40,41>.
e) <*T*,*E*,50,51>.

Mindegyik fenti esetre adjuk meg, hogy:

i) Mikor írható fel az <END CKPT> naplóbejegyzés, és
ii) Bármelyik lehetséges pillanatban, ha hiba lép fel, meddig kell a naplóban visszafelé tekinteni, ahhoz, hogy minden, be nem fejezett tranzakciókra vonatkozó bejegyzést megtaláljunk. Fontoljuk meg mindkét lehetőséget is, hogy a hiba az <END CKPT> naplóbejegyzés felírása előtt vagy az után jelentkezik.

8.5. Az eszközök meghibásodása elleni védekezés

A naplózással a rendszerhibák ellen védekezhetünk. Gondoskodhatunk arról, hogy rendszerhiba következtében legfeljebb csak a memóriában tárolt ideiglenes adatok vesznek el, de a lemezről semmi nem veszhet el. Ugyanakkor, amint a 8.1.1. részben már foglalkoztunk vele, sok komoly hibát okoz egy vagy több lemez elvesztése (tönkremenetele). Az adatbázist a naplóból elméletileg akkor tudjuk rekonstruálni, ha:

a) A naplót tároló lemez különbözik az adatokat (adatbázist) tartalmazó lemez(ek)től.
b) A naplót sosem dobjuk el az ellenőrzőpont-képzést követően, és
c) A napló helyrehozó (redo) vagy semmisségi/helyrehozó (undo/redo) típusú, s így az új értékeket tárolja.

Ugyanakkor, amint már említettük, a napló esetleg az adatbázisnál is gyorsabban növekedhet, s így nem praktikus a naplót örökre megőrizni.

8.5.1. Az archívmentés

Az eszközök meghibásodása elleni védekezés egyik megoldása az *archiválás* – az adatbázis másolatának elkészítése egy (több), az adatbázisétól különböző adathordozón. Ha lehetséges, lezárjuk az adatbázist addig, amíg elkészítjük a biztonsági másolatot (backup) valamely tárolóeszközön (például optikai lemezen vagy mágnesszalagon), majd a biztonsági másolatot az adatbázistól távol, biztonságos helyen tároljuk. A biztonsági másolat megőrzi az adatbázis mentéskori állapotát, s ha eszközhiba lép fel, akkor a mentésből az adatbázis ezen (mentéskori) állapotát vissza tudjuk állítani.

Miért nem csak a naplót mentjük?

Felmerülhet bennünk a kérdés, milyen gyakran kell elkészíteni a biztonsági mentést, hiszen a napló használatával – ha nem akadunk el – egy régi mentésből is helyreállíthatnánk az adatbázist. A válasz nem nyilvánvaló, az adatbázis méretén és tipikus módosítási fokán múlik. Amíg az adatbázisnak naponta esetleg csak kis része változik, addig a naplózandó módosítások tömege egy egész év folyamán sokkal nagyobb lehet, mint maga az adatbázis. Ha soha nem archiválunk, akkor a napló soha nem csonkolható, és a napló tárolási/kezelési költsége hamar túllépheti az adatbázis másolatának tárolási költségét.

A napló használatával sokkal frissebb állapotot tudunk rekonstruálni. Ha a biztonsági másolat készítéséről keletkező naplót megőrizzük, és az túlélte az eszköz meghibásodását, akkor a hiba után (esetleg másik lemezen) visszaállítva a biztonsági másolatból, a napló felhasználásával a mentés óta történt adatbázis-változásokat is át tudjuk vezetni az adatbázison. A napló keletkezése közben, amilyen gyorsan csak lehet, távoli másolatot készítünk róla. Ezzel a napló elvesztése ellen védekezhetünk. Így, ha a napló, az adatokkal együtt elveszik is, akkor még mindig használhatjuk az adatbázis mentését és a napló távoli másolatát az adatbázis visszaállításra egészen addig a pillanatig, amikor a napló utolsó átvitele történt a távoli másolatára.

Ha az adatbázis nagy, akkor a biztonsági mentés elkészítése (kiírása) hosszas folyamat, általánosan bevált, hogy nem mentik a teljes adatbázist minden archiváló alkalommal. Ezért a mentésnek két szintjét különböztetjük meg:

1. *Teljes mentés* (full dump), amikor az egész adatbázisról másolat készül.
2. *Növekményes mentés* (incremental dump), amikor az adatbázisnak csak azon elemeiről készítünk másolatot, melyek az utolsó teljes vagy növekményes mentés óta megváltoztak.

Lehetséges a mentésnek több szintjét is használni, a teljes mentést „0-dik szintűnek" tekintve, az „i-edik szintű" mentésen pedig azt értve, mely mentés az előző „i-edik szintű", vagy alacsonyabb szintű mentések óta megváltozott elemek másolatát tartalmazza.

Az adatbázist a teljes mentésből és a megfelelő növekményes mentésekből (a helyreállító vagy a semmisségi/helyreállító naplók rendszerhiba utáni visszaállítási folyamatához hasonló) módszerrel tudjuk rekonstruálni. Visszamásoljuk a teljes mentést, majd az ezt követő, legkorábbi növekményes mentéstől kezdve végrehajtjuk a növekményes mentésekben tárolt változtatásokat. A növekményes mentések az adatok-

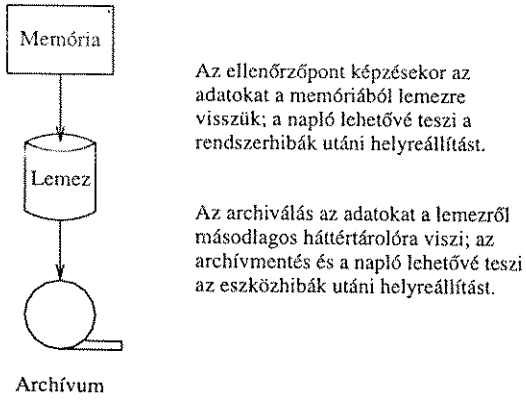
nak (a teljes adatbázishoz viszonyítva) csak azt a kis részét érintik, amely az utolsó mentés óta változott meg, s így ezek kevesebb helyet igényelnek és gyorsabban menthetők, mint a teljes mentés hely- és időigénye.

8.5.2. Archiválás működés közben

A 8.5.1. részben bemutatott, egyszerűnek látszó archiválással az a probléma, hogy sok adatbázist nem lehet lezárni arra az időre (lehet, hogy órákra), amíg a biztonsági mentését elkészítjük. Így, a működés közbeni ellenőrzőpont-képzéshez hasonlóan, meg kell oldanunk a *működés közbeni archiválást* is. Visszagondolunk arra, ahogy a működés közbeni ellenőrzőpont-képzés megkísérli az indulásakori adatbázis-állapot (megközelítő) másolatát létrehozni a lemezen. Az ellenőrzőpont létrehozásának környékén keletkezett kis naplórészletre támaszkodva az adatbázis állapotában történt minden olyan eltérést rendbe tudunk hozni, melyet az okozott, hogy az ellenőrzőpont képzése alatt új tranzakciók indulhattak és lemezírások történhettek.

Ehhez hasonlóan a működés közbeni archiválás megbízhatóan tud az adatbázisról olyan másolatot készíteni, ami az archiválás kezdetének megfelelő adatbázis-állapotot rögzíti, ugyanakkor a mentés alatti percekben vagy órákban az adatbázis működése sok adatbáziselemet cserélhet. Ha az adatbázis mentésből való visszaállítása szükséges, akkor a mentés alatt keletkezett naplóbejegyzések felhasználásával az adatbázis konzisztens állapota állítható elő. A hasonlóságot a 8.11. ábrán mutatjuk be.

A működés közbeni archiválás az adatbázis elemeit valamely fix sorrendben másolja, mialatt megeshet, hogy ezen elemeket az éppen végrehajtott tranzakciók módosítják. Ennek eredményeként megtörténhet, hogy a biztonsági mentésbe másolt adatbáziselem értéke nem ugyanaz, mint a mentés megkezdésekor volt. Amíg a mentés alatt keletkezett naplót megőrizzük, addig az eltérések a napló felhasználásával korrigálhatók.



8.11. ábra. *Az ellenőrzőpont-képzés és az archiválás közötti hasonlóság*

8.14. példa: Nagyon egyszerű példaként, tegyük fel, hogy adatbázisunk 4 elemből, *A*, *B*, *C* és *D*-ből áll. Ezek értéke a biztonsági mentés (archiválás) kezdetekor rendre 1, 2, 3, 4. A mentés közben *A* értéke 5-re, *C* értéke 6-ra, *B* értéke 7-re módosul. Az adatbáziselemeket a mentéskor sorban másoljuk az archívumba, az események sorrendje pedig legyen a 8.12. ábrának megfelelő. Ekkor noha az adatbázis tartalma a mentés kezdetekor 1, 2, 3, 4 volt, a mentés végére 5, 7, 6, 4 lett, a mentett archívumba 1, 2, 6, 4 került, jóllehet ilyen adatbázis-állapot a mentés ideje alatt nem is fordult elő. □

Lemez	*Mentés*
	A másolása az archívumba
A := 5	
	B másolása az archívumba
C := 6	
	C másolása az archívumba
B := 7	
	D másolása az archívumba

8.12. ábra. *Események a működés közbeni archiválás alatt*

Részletesebben a biztonsági mentés (archívum) elkészítése a következő lépésekből áll. Feltételezzük, hogy az alkalmazott naplózási módszer a helyrehozó (redo) vagy a semmisségi/helyrehozó (undo/redo) módszerek valamelyike; a semmisségi (undo) naplózás nem alkalmas a működés közbeni archiválással való használatra.

1. A <START DUMP> bejegyzés naplóba írása.
2. Az alkalmazott naplózási módnak megfelelő ellenőrzőpont kialakítása.
3. A kívánt adatlemez(ek) teljes vagy növekményes mentésének végrehajtása, arra ügyelve, hogy az adatok másolata (a mentés) biztonságos távoli helyre kerüljön.
4. Gondoskodjunk arról is, hogy a napló szükséges részéről is másolat készüljön, és az is biztonságos, távoli helyre kerüljön. A mentett naplórész tartalmazza legalább a 2. pontbeli ellenőrzőpont-képzés közben keletkezett naplóbejegyzéseket, melyeknek túl kell élniük az adatbázist hordozó eszköz meghibásodását.
5. <END DUMP> bejegyzés naplóba írása.

A mentés befejezésekor biztonsággal eldobhatjuk a napló 2. pontban végrehajtott ellenőrzőpont-képzést *megelőzően keletkezett* részét.

8.15. példa: Tegyük fel, hogy a 8.14. példabeli egyszerű adatbázis mentés közbeni módosításait két tranzakció, T_1 (mely *A*-t és *B*-t módosította) és T_2 (amely *C*-t módosította) végezte, melyek a mentés kezdetekor aktívak voltak. A 8.13 ábrán látjuk a mentés alatti események lehetséges naplóbejegyzéseit, semmisségi/helyrehozó (undo/redo) naplózási módszert alkalmazva.

Megjegyezzük, hogy nem tüntettük fel T_1 befejezését. Az eléggé valószínűtlen, hogy egy tranzakció a teljes mentés egész ideje alatt aktív maradjon, de ez a lehetőség nem befolyásolja a következőként bemutatandó helyreállítási módszer helyességét. □

<START DUMP>
<START CKPT (T_1, T_2)>
<T_1,A,1,5>
<T_2,C,3,6>
<COMMIT T_2>
<T_1,B,2,7>
<END CKPT>
mentés befejezése
<END DUMP>

8.13. ábra. *A mentés közben keletkező napló*

8.5.3. Helyreállítás az archívmentés és a napló használatával

Tegyük fel, hogy készülékhiba lépett fel, s az adatbázist rekonstruálnunk kell. A helyreállítást a legutolsó biztonsági mentés (archívmentés), és a napló – katasztrófa során el nem veszett – távoli mentése felhasználásával végezzük. A következő lépéseket hajtjuk végre:

1. Az adatbázis visszaállítása a biztonsági (archív) mentésből.

   a) Meg kell keresni a legutolsó teljes mentést, belőle rekonstruálni az adatbázist. (Azaz a mentést az adatbázisba másoljuk.)
   b) Ha van(nak) későbbi növekményes mentés(ek), akkor ezeket időrendi sorrendben használva, módosítjuk az adatbázist.

2. Módosítjuk az adatbázist a napló katasztrófát túlélt részével. (Természetesen a naplózási módszernek megfelelő helyreállítási eljárást kell alkalmaznunk.)

8.16. példa: Tegyük fel, hogy a 8.15. példában szerelő biztonsági mentés elkészítését követően történik eszközmeghibásodás, és a 8.13 ábrán látott napló ezt túlélte. Azért, hogy az eljárást érdekesebbé tegyük – a 8.13. ábrának megfelelően – tekintsük úgy, hogy a napló katasztrófát túlélt részében nincs <COMMIT T_1> bejegyzés, jóllehet <COMMIT T_2> bejegyzés van. Az adatbázist először a biztonsági mentésből visszatöltjük, s így A, B, C, D elemei rendre az 1, 2, 6, 4 értékeket kapják.

Ezután a naplót vesszük elő. Minthogy T_2 befejezett tranzakció, helyreállítjuk (redo) azon lépés hatását, mely C értékét 6-ra módosította. Példánkban C értéke már 6, de előfordulhatna, hogy:

a) C mentése azt megelőzően történt, hogy C értékét T_2 tranzakció módosította volna, vagy
b) A mentésben C-nek később kapott értéke van, mely értéket olyan tranzakció állított be, melyre vonatkozó COMMIT bejegyzést a napló – katasztrófát túlélt részében – vagy találunk, vagy nem. C értékét a mentésben talált értékre akkor állítjuk, *ha* az ezt beállító tranzakció COMMIT bejegyzését megtaláljuk.

Minthogy T_1 gyaníthatóan nem befejezett tranzakció (mert COMMIT bejegyzését nem találjuk), így T_1 hatásait semmissé kell tennünk (undo). A T_1-re vonatkozó naplóbejegyzések használatával meg tudjuk állapítani, hogy A értékét 1-re, B értékét 2-re kell visszaállítanunk. Előfordulhat persze, hogy a mentésen ez az értékük, de a pillanatnyi mentésben ettől eltérő értékeik is lehetnek, ha A és/vagy B módosított értéke archiválódott. (Ez a módosításnak és a mentésnek az időbeli sorrendjétől függ.) □

8.5.4. Feladatok

8.5.1. feladat: Ha semmisségi/helyrehozó (undo/redo) naplózás helyett a helyrehozó (undo) naplózást használjuk a 8.15. és 8.16. példákban, akkor:

a) Hogyan fog kinézni a napló?
*! b) Ha a mentést és a naplót használjuk a helyreállításhoz, mi lesz annak következménye, hogy T_1 nem befejezett?
c) Mi lesz az adatbázis állapota a helyreállítás után?

8.6. Összefoglalás

- *Tranzakciókezelés:* A tranzakciókezelő két tipikus feladata: naplózással biztosítani az adatbázisban végrehajtott tevékenységek hatásainak helyreállíthatóságát, és az ütemezőn keresztül biztosítani tranzakciók korrekt párhuzamos működését. (Utóbbit e fejezetben nem tárgyaltuk.)
- *Adatbáziselemek:* Az adatbázist elemekre osztottaknak tekintjük. Az adatbáziselemek tipikusan lemezblokkok, de lehetnek sorok, osztályok extentjei, vagy más egységek. Az adatbáziselemek a naplózás és ütemezés egységei.
- *Naplózás:* A naplóban a tranzakciók összes fontosabb ténykedéseit – a működés megkezdése, adatbáziselemek módosítása, normális vagy abnormális befejeződés – leíró naplóbejegyzéseket tároljuk. A naplót – az általa leírt adatbázis-módosítások lemezre mentése környékén lemezre kell menteni. A napló lemezre mentésének pontos ideje az alkalmazott naplózási módszer függvénye.
- *Helyreállítás:* Rendszerhiba fellépésekor, a naplót használva, az adatbázis konzisztens állapota helyreállítható.
- *Naplózási módszerek:* A naplózás három tipikus módszere a semmisségi (undo), a helyreállító (redo) és a semmisségi/helyreállító (undo/redo), nevüket az alkalmazott helyreállítási eljárásnak megfelelően kapták.
- *Semmisségi (undo) naplózás:* Ez a naplózási módszer az adatbáziselemek értékének megváltoztatásakor csak a régi értéket tárolja a naplóbejegyzésekben. A semmisségi naplózás alkalmazásakor az adatbáziselem új értéke csak azt követően íródik lemezre, miután a változást leíró naplóbejegyzés már lemezre került, ugyanakkor az adatbáziselem lemezre írásának meg kell előznie a tranzakció normál befe-

jezését leíró COMMIT naplóbejegyzés lemezre írását. A helyreállítás a befejezetlen tranzakciók által módosított adatbáziselemek régi értékének visszaállításával történik, azaz a be nem fejezett tranzakciók esetleges hatásainak semmissé tételével.

- *Helyreállító (redo) naplózás:* Ebben a naplózási módszerben a módosított adatbáziselemeknek csak az új értékét tároljuk a módosítást leíró naplóbejegyzésekben. A naplózás eme módszerében az adatbáziselemek értéke csak azt követően íródik lemezre, miután a módosítást végző tranzakció összes változtatást leíró, valamint a véglegesítést jelentő naplóbejegyzése már lemezre került. A helyreállítás a befejezett tranzakciók által módosított adatbáziselemek új értékének újra (adatbázisba) írásával történik.
- *Semmisségi/helyreállító (undo/redo) naplózás:* Ezzel a naplózási módszerrel mind a régi, mind az új értékek naplózódnak. A semmisségi/helyreállító naplózás a többinél sokkal rugalmasabb módszer abban, hogy csak annyit követel meg, hogy a változást leíró naplóbejegyzés a tényleges adatbázisbeli módosítást megelőzően kerüljön lemezre. Arra vonatkozóan nincs kikötése, hogy a tranzakció befejezését leíró bejegyzés mikor kerül lemezre. A helyreállítás a befejezett tranzakciókra a „helyreállító", a be nem fejezett tranzakciókra a „semmisségi" módszer szerint történik.
- *Ellenőrzőpont-képzés:* Az összes helyreállítási módszer elvileg a teljes napló visszamenőleges elemzését igényli. Az adatbázisrendszerek a naplóban alkalmankénti ellenőrzőpont-képzéssel biztosítani tudják, hogy a helyreállítás során az ellenőrzőpontnál korábban felírt naplóbejegyzésekre már ne legyen szükség. Így a régi naplórész törölhető, s a lemez területe újra felhasználható.
- *Működés közbeni ellenőrzőpont-képzés:* Az ellenőrzőpont képzése közben az adatbázisrendszer más működését (esetleg hosszú időre) le kellene állítani. Ennek elkerülésére az összes naplózási módszerhez megalkották a működés közbeni ellenőrzőpont-képzés technikáját. Ez lehetővé teszi, hogy az ellenőrzőpont képzése közben a rendszer működjön, és adatbázis-módosítások is megtörténjenek. Ennek egyetlen pluszköltsége az, hogy a helyreállítás során néhány, az ellenőrzőpont-képzést megelőzően keletkezett, naplóbejegyzést is át kell vizsgálni.
- *Archiválás – biztonsági mentés:* A naplózás csak a memóriatartalom elvesztésével járó rendszerhibák utáni helyreállítást teszi lehetővé. Az archiválással tudunk védekezni az adatbázist hordozó lemez tartalmának sérülése ellen. Az archívum az adatbázis biztonságos helyen tárolt másolata.
- *Növekményes mentés:* A teljes adatbázis rendszeres másolása-mentése helyett, a teljes másolatot követően néhány növekményes mentést készíthetünk. A növekményes mentés során csak az utolsó mentés óta megváltozott adatokat mentjük, ezzel mentési időt és helyet takarítunk meg.
- *Működés közbeni archiválás:* Az a módszer, amikor az adatbázis a biztonsági mentés közben működésben van. E módszer használata megköveteli az archiválás közben a napló használatát és ellenőrzőpont-képzését is.
- *Készülék meghibásodás utáni helyreállítás:* Ha a lemez megy tönkre, akkor az adatbázis konzisztens állapotát egy teljes biztonsági mentés visszatöltése, majd a későbbi növekményes mentések, végül a napló mentett másolatának használatával helyreállíthatjuk.

8.7. Irodalomjegyzék

A legjelentősebb mű, mely a tranzakciók összes vonatkozásaival, közte a naplózással és a helyreállítással is foglalkozik, Gray és Reuter [5] könyve. Ez a könyv a tranzakciókra vonatkozó meggondolásokat részben Jim Gray [3] széles körben használt cikkére alapozza. Később a naplózási és helyreállítási módszerek elsődleges forrásai [4] és [8].

[2] a tranzakciófeldolgozás egy korai, nagyon tömör leírása. [7] a téma egy sokkal újabb feldolgozása.

Két korai áttekintés, [1] és [6] a helyreállítással foglalkozó alapművek, a naplózás azon három alaptípusának rendszerezői, mely felosztást mi is követtünk.

1. P. A. Bernstein, N. Goodman, and V. Hadzilacos, „Recovery algorithms for database systems", *Proc. 1983 IFIP Congress,* North Holland, Amsterdam, pp. 799–807.
2. P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems,* Addison-Wesley, Reading MA, 1987.
3. J. N. Gray, „Notes on database-operating systems", in *Operating Systems: an Advanced Course,* pp. 393–481, Springer-Verlag, 1978.
4. J. N. Gray, P. R. McJones, and M. W. Blasgen, „The recovery manager of the System R database manager", *Computing Surveys* **13**:2 (1981), pp. 223–242.
5. J. N. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques,* Morgan-Kaufmann, San Francisco, 1993.
6. T. Haerder and A. Reuter, „Principles of transaction-oriented database recovery – a taxonomy", *Computing Surveys* **15**:4 (1983), pp. 287–317.
7. V. Kumar and M. Hsu, *Recovery Mechanisms in Database Systems,* Prentice-Hall, Englewood Cliffs NJ, 1998.
8. C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. Schwarz, „ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging", *ACM Trans. On Database Systems* **17**:1 (1992), pp. 94–162.

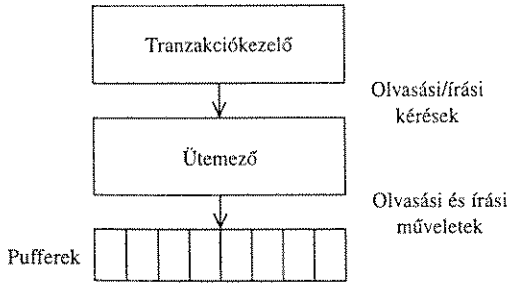
9. fejezet

Konkurenciavezérlés

A tranzakciók közötti egymásra hatás az adatbázis-állapot inkonzisztensé válását okozhatja, még akkor is, amikor a tranzakciók külön-külön megőrzik az állapot helyességét, és rendszerhiba sem történt. Ezért valamiképpen szabályoznunk kell, hogy a különböző tranzakciók egyes lépései milyen sorrendben következzenek egymás után. A lépések szabályozásának a feladatát az adatbázis-kezelő rendszer *ütemező* (scheduler) része végzi. Azt az általános folyamatot, amely biztosítja, hogy a tranzakciók egyidejű végrehajtása során megőrizzék a konzisztenciát, *konkurenciavezérlésnek* (concurrency control) nevezzük. Az ütemező szerepét a 9.1. ábrán láthatjuk.

Amint a tranzakciók az adatbáziselemek olvasását és írását kérik, ezek a kérések az ütemezőhöz kerülnek. Legtöbbször az ütemező közvetlenül végrehajtja az olvasásokat és írásokat, mégpedig először a pufferkezelőt hívja meg, amennyiben a szükséges adatbáziselem nincs a pufferben. Bizonyos esetekben azonban nem biztonságos azonnal végrehajtania a kéréseket. Az ütemezőnek késleltetnie kell a kérést, sőt valamilyen konkurenciavezérlési technikában az ütemező abortálhatja (leállíthatja a befejezés előtt, vagyis sikertelenül befejezheti) a kérést kiadó tranzakciót.

Először azt tanulmányozzuk hogyan biztosítható, hogy a konkurensen végrehajtott tranzakciók megőrizzék az adatbázis-állapot helyességét. Az elméleti követelményt *sorbarendezhetőségnek* (serializability) nevezzük, melynél van egy fontosabb, erősebb feltétel, amelyet *konfliktus-sorbarendezhetőségnek* (conflict-serializability) hívunk, és a legtöbb ütemező valójában ezt alkalmazza. Vizsgálni fogjuk az ütemezők legfontosabb megvalósítási technikáit: a zárolást, az időbélyegzést és az érvényesítést.



9.1. ábra. *Az ütemező fogadja a tranzakcióktól az olvasási/írási kéréseket, és vagy azonnal végrehajtja ezeket a pufferben, vagy késlelteti őket*

A zároláson alapuló ütemezésekről szóló rész tartalmazza a fontos „kétfázisú zárolás" fogalmát, amely a legelterjedtebb követelmény annak érdekében, hogy biztosítsuk az ütemezések sorbarendezhetőségét. A zármódok számos különböző halmazával ismerkedünk meg, amelyeket az ütemező a különféle alkalmazásokhoz alkalmazhat. A zárolási sémák közül azokat tanulmányozzuk, amelyekben a zárolható elemek beágyazott, illetve faszerkezetűek.

9.1. Soros és sorba rendezhető ütemezések

A konkurenciavezérlés tanulmányozását azzal kezdjük, hogy megvizsgáljuk, a konkurensen végrehajtott tranzakciók milyen feltételekkel tudják megőrizni az adatbázis-állapot konzisztenciáját. Az alapfeltevésünk, amelyet „helyességi elv"-nek (correctness principle) neveztünk, a 8.1.3. részben az volt, hogy ha minden egyes tranzakciót elkülönítve hajtunk végre (anélkül, hogy más tranzakció konkurensen futna), akkor az adatbázist konzisztens állapotból konzisztens állapotba alakítjuk. A gyakorlatban azonban a tranzakciók általában más tranzakciókkal egyidejűleg konkurensen futnak, emiatt a helyességi elvet közvetlenül nem alkalmazhatjuk. Így olyan „ütemezéseket" kell tekintenünk, amelyek biztosítják, hogy ugyanazt az eredményt állítják elő, mintha a tranzakciókat egyesével hajtottuk volna végre. Az egész fejezet fő témáját adják azok a módszerek, amelyek biztosítják, hogy a tranzakciók csak olyan módon legyenek konkurensen végrehajtva, mintha sorban egyesével futottak volna le.

9.1.1. Ütemezések

Az *ütemezés* (schedule) egy vagy több tranzakció által végrehajtott lényeges műveletek időrendben vett sorozata. Amikor a konkurenciavezérlést tanulmányozzuk, a lényeges olvasási és írási műveletek a központi memória puffereiben történnek, nem pedig lemezen. Vagyis egy A adatbáziselemet, amelyet valamelyik T tranzakció hozott be a pufferbe, ebben a pufferben nemcsak a T tudja olvasni vagy írni, hanem más tranzakciók is hozzáférhetnek az A-hoz. Idézzük fel a 8.1.4. részből, hogy a `READ (OLVASÁS)` és a `WRITE (ÍRÁS)` műveletek először meghívnak egy `INPUT` utasítást, hogy az adatbáziselemet a lemezről betöltsék, ha még nincs a pufferben, egyébként pedig a `READ` és `WRITE` műveletek közvetlenül a pufferben hozzáférnek az elemhez. Ezért csupán a `READ` és `WRITE` műveletek és a sorrendjük számít, amikor a konkurenciával foglalkozunk, és az `INPUT`, illetve `OUTPUT` műveleteket figyelmen kívül fogjuk hagyni.

9.1. példa: Tekintsünk két tranzakciót és az adatbázison való hatásukat, amikor egy meghatározott sorrendben hajtjuk végre a műveleteiket. A T_1 és T_2 tranzakciók fő

T_1	T_2
READ(A,t)	READ(A,s)
t := t+100	s := s*2
WRITE(A,t)	WRITE(A,s)
READ(B,t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)

9.2. ábra. *Két tranzakció*

műveletei a 9.2. ábrán találhatók. A t és s változók a T_1-nek, illetve T_2-nek megfelelő helyi változók, és *nem* adatbáziselemek.

Tételezzük fel, hogy az adatbázis-állapoton az egyetlen konzisztenciamegszorítás az $A = B$. Mivel a T_1 az A-hoz és a B-hez is hozzáad 100-at, és a T_2 az A-t és a B-t is megszorozza 2-vel, tudjuk, hogy az egyes tranzakciók egymástól elkülönítve futva megőrzik a konzisztenciát. □

9.1.2. Soros ütemezések

Azt mondjuk, hogy egy ütemezés *soros* (serial schedule), ha úgy épül fel a tranzakciós műveletekből, hogy először az egyik tranzakció összes műveletét tartalmazza, majd azután egy másik tranzakció összes műveletét stb., miközben nem cseréli fel a műveleteket. Pontosabban kifejezve, egy S ütemezés soros, ha bármely két T és T' tranzakcióra, ha T-nek van olyan művelete, amely megelőzi a T' valamelyik műveletét, akkor a T összes művelete megelőzi a T' valamennyi műveletét.

9.2. példa: A 9.2. ábrán szereplő tranzakcióknak két soros ütemezése van, az egyikben T_1 megelőzi T_2-t, a másikban T_2 előzi meg T_1-et. A 9.3. ábra azt az eseménysoro-

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250

9.3. ábra. *Soros ütemezés, amelyben T_1 megelőzi T_2-t*

T_1	T_2	A	B
		25	25
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	50	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		50
READ(A,t)			
t := t+100			
WRITE(A,t)		150	
READ(B,t)			
t := t+100			
WRITE(B,t)			150

9.4. ábra. *Soros ütemezés, amelyben T_2 megelőzi T_1-t*

zatot mutatja, amikor T_1 megelőzi T_2-t, és a kezdeti állapot $A = B = 25$. Azt a megállapodást követjük, hogy időrendi sorrendben függőlegesen lefelé írunk. Továbbá a megjelenített A és B értékek a központi memória pufferbeli értékeire utalnak, nem szükségképpen a lemezen tárolt értékeire.

A 9.4. ábrán látjuk a másik soros ütemezést, amelyben T_2 megelőzi T_1-et. A kezdeti állapot legyen megint $A = B = 25$. Megjegyezzük, hogy A és B végső értéke különböző a két ütemezésben, mégpedig mindkettő értéke 250, ha a T_1 fut először, és 150, ha a T_2 fut előbb. De nem is a végeredmény a központi kérdés addig, amíg a konzisztenciát megőrizzük. Általában nem várjuk el, hogy az adatbázis végső állapota független legyen a tranzakciók sorrendjétől. □

A soros ütemezést úgy ábrázolhatjuk, mint ahogyan a 9.3. ábrán vagy a 9.4. ábrán látható, a műveleteket az előfordulásuk sorrendjében soroljuk fel. Másrészt, mivel a soros ütemezésben a műveletek sorrendje csak magától a tranzakciók sorrendjétől függ, ezért a soros ütemezést néha a tranzakciók felsorolásával fogjuk megadni. Így a 9.3. ábra ütemezését (T_1, T_2) reprezentálja, a 9.4. ábráét pedig (T_2, T_1).

9.1.3. Sorba rendezhető ütemezések

A tranzakciókra vonatkozó helyességi elv szerint minden soros ütemezés megőrzi az adatbázis-állapot konzisztenciáját. Vajon van-e más ütemezés is, amely szintén biztosítja a konzisztencia megmaradását? Igen, ilyen létezik, ahogyan ezt a következő példa mutatja. Általában azt mondjuk, hogy egy ütemezés *sorba rendezhető* (serializable schedule), ha ugyanolyan hatással van az adatbázis állapotára, mint valamelyik soros ütemezés, függetlenül attól, hogy mi volt az adatbázis kezdeti állapota.

9.3. példa: A 9.5. ábrán látjuk a 9.1. példában szereplő két tranzakciónak egy sorba rendezhető, ám nem soros ütemezését. Ebben az ütemezésben T_2 azután van hatással

az A-ra, miután a T_1 volt, de mielőtt a T_1 hatással lenne a B-re. Mégis azt látjuk, hogy ebben az ütemezésben a két tranzakció hatása megegyezik a 9.3. ábrán látható (T_1, T_2) soros ütemezés hatásával. Ahhoz, hogy meggyőződjünk az állítás igazságáról, nemcsak azt az esetet kell megnéznünk, amely a 9.5. ábrán látható, amikor az adatbázis-állapot $A = B = 25$-ről indul, hanem bármely konzisztens adatbázis kiindulási állapotból kiindulva. Mivel minden konzisztens adatbázis-állapotban az $A = B = c$ valamely c konstanssal, nem nehéz levezetnünk, hogy a 9.5. ábra ütemezésében az A-nak is és a B-nek is $2(c + 100)$ lesz az értéke, és így bármelyik konzisztens állapotból indulunk ki, a konzisztenciát megőrizzük.

Másrészt tekintsük a 9.6. ábrán található ütemezést. Világos, hogy ez nem soros, de ami lényegesebb, nem is sorba rendezhető. Meggyőződhetünk arról, hogy nem sorba rendezhető, ugyanis legyen a kiindulási konzisztens állapotban $A = B = 25$, és az adatbázis inkonzisztens állapotba kerül, amikor $A = 250$ és $B = 150$ lesz. Megjegyezzük,

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250

9.5. ábra. *Sorba rendezhető, de nem soros ütemezés*

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		50
READ(B,t)			
t := t+100			
WRITE(B,t)			150

9.6. ábra. *Nem sorba rendezhető ütemezés*

hogy ebben a műveleti sorrendben, a T_1 dolgozik előbb az A-val, viszont T_2 dolgozik előbb a B-vel, ennek hatásaként másképpen kell kiszámolnunk A-t és B-t, vagyis $A := 2(A + 100)$, szemben $B := 2B + 100$-zal. A 9.6. ábrán található ütemezés olyan viselkedést mutat, amelyet a konkurenciavezérlési működésekkel el kell kerülnünk. □

9.1.4. A tranzakció szemantikájának hatása

A sorbarendezhetőségi vizsgálatainkban eddig a tranzakciók által végrehajtott műveleteket néztük meg részletesen annak érdekében, hogy meghatározzuk sorba rendezhető-e az ütemezés. Azonban a tranzakciók részletei is számítanak, ahogyan ezt a következő példából láthatjuk.

9.4. példa: Tekintsük a 9.7. ábrán látható ütemezést, amely csak a T_2 által végrehajtott számításokban különbözik a 9.6. ábrától, mégpedig abban, hogy a T_2 nem 2-vel szorozza meg A-t és B-t, hanem 1-gyel.[1] Ekkor A és B értéke az ütemezés végén megegyezik, és könnyen ellenőrizhetjük, hogy a konzisztens kezdeti állapottól függetlenül a végállapot is konzisztens lesz. Valójában az egyetlen végállapot az, amelyet vagy a (T_1, T_2) vagy a (T_2, T_1) soros ütemezés eredményez. □

Sajnos, az ütemező számára nem reális a tranzakciós számítások részleteinek figyelembevétele. Mivel a tranzakciók gyakran tartalmaznak általános célú programozási

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*1		
	WRITE(A,s)	125	
	READ(B,s)		
	s := s*1		
	WRITE(B,s)		25
READ(B,t)			
t := t+100			
WRITE(B,t)			125

9.7. ábra. *Egy olyan ütemezés, amely csak a tranzakciók részletezett viselkedése miatt sorba rendezhető*

[1] Valaki jogosan kérdezheti, hogy miért is viselkedik így egy tranzakció? Mégis a példa kedvéért ezt most hagyjuk figyelmen kívül. Valójában több elfogadható tranzakciót is helyettesíthetnénk a T_2 helyére, amely az A-t és B-t változatlanul hagyná. Például amikor a T_2 csak egyszerűen beolvassa az A-t és B-t, és kiíratja az értéküket. Vagy T_2 a felhasználótól kérhet be adatokat, hogy kiszámoljon egy F tényezőt, amivel megszorozza az A-t és a B-t, és előfordulhat olyan felhasználói input, amelyre az $F = 1$.

nyelven írt kódokat éppúgy, mint SQL vagy más magas szintű nyelv utasításait, néha nagyon nehéz megválaszolni azokat a kérdéseket, mint pl. „ez a tranzakció az A-t egy 1-től különböző konstanssal szorozta-e meg?". Az ütemezőnek azonban látnia kell a tranzakciók olvasási és írási kéréseit, így tudhatja, hogy az egyes tranzakciók mely adatbáziselemeket olvasták be, és mely elemek *változhattak* meg. Az ütemező feladatának az egyszerűsítésére megszokott az a feltételezés, hogy:

- Bármely A adatbáziselemnek egy T tranzakció olyan értéket ír be, amely az adatbázis-állapottól függ oly módon, hogy ne forduljon elő aritmetikai egybeesés.

Más szóval kifejezve, ha a T tudna az A-ra olyan hatással lenni, hogy az adatbázis-állapot inkonzisztenssé váljék, akkor a T ezt meg is teszi. Ezt a feltevést a 9.2. részben pontosítjuk, amikor a sorbarendezhetőség biztosítására adunk meg elégséges feltételeket.

9.1.5. A tranzakciók és ütemezések jelölése

Ha elfogadjuk, hogy egy tranzakció által végrehajtott pontos számítások tetszőlegesek lehetnek, akkor nem szükséges a helyi számítási lépések részleteit néznünk, mint amilyen a `t := t+100`. Csak a tranzakciók által végrehajtott olvasások és írások számítanak. Így a tranzakciókat és az ütemezéseket rövidebben jelölhetjük. Ekkor $r_T(X)$ és $w_T(X)$ tranzakcióműveletek, és azt jelentik, hogy a T tranzakció olvassa (r, az angol *read* = olvasás rövidítése), illetve írja (w, az angol *write* = írás rövidítése) az X adatbáziselemet. Továbbá, mivel a tranzakcióinkat T_1, T_2, ...-vel fogjuk általában jelölni, így megállapodunk abban, hogy $r_i(X)$ és $w_i(X)$ ugyanazt jelöli, mint $r_{T_i}(X)$, illetve $w_{T_i}(X)$.

9.5. példa: A 9.2. ábrán látható tranzakciók az alábbi módon írhatók fel:

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$;
T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$;

Megjegyezzük, hogy nem említettük sehol a t és az s helyi változókat, és nem jelöltük azt sem, hogy mi történt a beolvasás után az A-val és B-vel. Intuíció alapján ezt úgy értelmezzük, hogy az adatbáziselemek megváltozásában a „legrosszabbat fogjuk feltételezni".

Egy másik példaként nézzük meg a T_1 és T_2-nek a 9.5. ábrán látható sorba rendezhető ütemezését. Ezt az ütemezést átírva:

$r_1(A)$; $w_1(A)$; $r_2(A)$; $w_2(A)$; $r_1(B)$; $w_1(B)$; $r_2(B)$; $w_2(B)$;
□

Pontosítva a jelölést:

1. Egy *tranzakció műveletét* (action of transaction) $r_i(X)$ vagy $w_i(X)$ formában fejezünk ki, amely azt jelenti, hogy a T_i tranzakció olvassa *(read)*, illetve írja *(write)* az X adatbáziselemet.

2. Egy T_i *tranzakció* az i indexű műveletekből álló sorozat.
3. A $\mathcal{T}$ tranzakciók halmazának egy S *ütemezése* olyan műveletek sorozata, amelyben minden $\mathcal{T}$ halmazbeli T_i tranzakcióra teljesül, hogy T_i műveletei ugyanabban a sorrendben fordulnak elő az S-ben, mint ahogy magában a T_i definíciójában szerepeltek. Azt mondjuk, hogy az S az őt alkotó tranzakciók műveleteinek *átlapolása* (interleaving).

Például a 9.5. példában található ütemezésben az összes 1-es indexű művelet ugyanabban a sorrendben szerepel, mint ahogy a T_1 definíciójában volt, és az összes 2-es indexű művelet ugyanabban a sorrendben fordul elő, mint ahogy a T_2 definíciójában szerepelt.

9.1.6. Feladatok

* **9.1.1. feladat:** Egy repülő-helyfoglalási rendszer végzi a T_1 tranzakciót, és az alábbi lépéseket hajtja végre:

i) A vevőtől lekérdezzük a keresett járat idejét és városait. A keresett járatokról az információ az A és B adatbáziselemekben található (valószínűleg lemezblokkokban), amelyeket a rendszer a lemezről érhet el.
ii) A vevőnek elmondjuk a feltételeket, és kiválasztjuk a járatot, amelynek adatai, beleértve a járatra való foglalás számát is, a B-ben található. A járatra való helyfoglalást a vevő végzi el.
iii) A vevő kiválasztja a járatra az ülőhelyet, a járat ülőhelyadatait a C adatbáziselem tartalmazza.
iv) A rendszer megkapja a vevő hitelkártyaszámát, és hozzáfűzi a számlát a számlák jegyzékéhez, mely jegyzék a járat D adatbáziselemében található.
v) A vevő telefonszámát és a járat adatait hozzáadjuk egy másik jegyzékhez az E adatbáziselemben, hogy egy faxot tudjunk küldeni a járatra való érvényesítéshez.

Fejezzük ki a T_1 tranzakciót r és w műveletek sorozataként!

*! **9.1.2. feladat:** Ha van két tranzakciónk, az egyik 4, a másik pedig 6 műveletből áll, akkor ezeknek a tranzakcióknak mennyi átlapolása (ütemezése) lehetséges?

9.2. Konfliktus-sorbarendezhetőség

Most egy olyan elégséges feltételt adunk meg, mely biztosítja egy ütemezés sorbarendezhetőségét. A piaci rendszerek ütemezői a tranzakciók sorbarendezhetőségére általában ezt az erősebb feltételt biztosítják, amelyet „konfliktus-sorbarendezhetőségnek" nevezünk. Ez a *konfliktus* (conflict) fogalmon alapul: amely olyan egymást követő műveletpár az ütemezésben, amelynek ha a sorrendjét felcseréljük, akkor legalább az egyik tranzakció viselkedése megváltozhat.

9.2.1. Konfliktusok

Azzal kezdjük, hogy vegyük észre a legtöbb műveletpár *nincs* konfliktusban a fenti értelemben. Ugyanis, tételezzük fel, hogy T_i és T_j különböző tranzakciók, vagyis $i \neq j$.

1. $r_i(X)$; $r_j(Y)$ sohasem konfliktus, még akkor sem, ha $X = Y$. Ennek az az oka, hogy egyik lépés sem változtatja meg az értékeket.
2. $r_i(X)$; $w_j(Y)$ nincs konfliktusban, feltéve, ha $X \neq Y$. Ennek az az oka, hogy a T_j írhatja az Y-t, mielőtt a T_i beolvasta az X-et, az X értéke ugyanis ettől nem változik. Annak sincs hatása a T_j-re, hogy a T_i olvassa az X-et, ugyanis ez nincs hatással arra, hogy milyen értéket ír be a T_j az Y-ba.
3. $w_i(X)$; $r_j(Y)$ nincs konfliktusban, ha $X \neq Y$, ugyanazért, mint a 2.
4. Szintén hasonlóan $w_i(X)$; $w_j(Y)$ sincs konfliktusban mindaddig, amíg $X \neq Y$.

Másrészt három esetben nem cserélhetjük fel a műveletek sorrendjét:

a) Ugyanannak a tranzakciónak két művelete, pl. $r_i(X)$; $w_i(Y)$ konfliktus. Ennek az az oka, hogy egyetlen tranzakción belül a műveletek sorrendje rögzített, és az adatbázis-kezelő rendszer ezt a sorrendet nem rendezheti át újra.
b) Különböző tranzakciók ugyanarra az adatbáziselemre való írása konfliktus. Vagyis $w_i(X)$; $w_j(X)$ konfliktus. Ennek az az oka, mint már írtuk, hogy az X értéke az marad, amelyet a T_j számolt ki. Ha felcseréljük a sorrendjüket, hogy $w_j(X)$; $w_i(X)$, akkor az X-nek a T_i által kiszámított értéke marad meg. Az a feltevésünk, hogy „nincs egybeesés", azt adja, hogy a T_i és a T_j által írt értékek lehetnek különbözőek, és ezért az adatbázis valamelyik kezdeti állapotára különbözni fognak.
c) Különböző tranzakcióknak ugyanabból az adatbáziselemből való olvasása és írása is konfliktus. Vagyis $r_i(X)$; $w_j(X)$ konfliktus, és $w_i(X)$; $r_j(X)$ is konfliktus. Ha átvisszük $w_j(X)$-et $r_i(X)$ elé, akkor a T_i által olvasott X-beli érték az lesz, amelyet a T_j írt, amiről pedig feltételeztük, hogy nem szükségképpen egyezik meg az X korábbi értékével. Tehát $r_i(X)$ és $w_j(X)$ sorrendjének cseréje befolyásolja, hogy T_i milyen értéket olvas X-ből, ez pedig befolyásolja a T_i működését.

Levonhatjuk a következtetést, hogy különböző tranzakciók bármely két műveletének sorrendje felcserélhető, hacsak nem

1. Ugyanarra az adatbáziselemre vonatkoznak, és
2. Legalább az egyik művelet írás.

Ezt az elvet kiterjesztve tetszőleges ütemezést véve annyi nem konfliktusos cserét készíthetünk, amennyit csak kívánunk, abból a célból, hogy az ütemezést soros ütemezéssé alakítsuk át. Ha ezt meg tudjuk tenni, akkor az eredeti ütemezés sorba rendezhető volt, ugyanis az adatbázis állapotára való hatása változatlan marad minden nem konfliktusos cserével.

Azt mondjuk, hogy két ütemezés *konfliktusekvivalens* (conflict-equivalent), ha szomszédos műveletek nem konfliktusos cseréinek sorozatával az egyiket átalakíthatjuk a másikká. Azt mondjuk, hogy egy ütemezés *konfliktus-sorbarendezhető* (conflict-serializable schedule), ha konfliktusekvivalens valamely soros ütemezéssel. Megjegyezzük, hogy a konfliktus-sorbarendezhetőség elégséges feltétele a sorbarendezhetőségnek, vagyis egy konfliktus-sorbarendezhető ütemezés sorba rendezhető ütemezés is egyben. Azonban a konfliktus-sorbarendezhetőség nem szükséges ahhoz, hogy egy ütemezés sorba rendezhető legyen, mégis általában ezt a feltételt ellenőrzik a piaci rendszerek ütemezői, amikor a sorbarendezhetőséget kell biztosítaniuk.

9.6. példa: Legyen az ütemezés

$r_1(A)$; $w_1(A)$; $r_2(A)$; $w_2(A)$; $r_1(B)$; $w_1(B)$; $r_2(B)$; $w_2(B)$;

a 9.5. példából. Azt állítjuk, hogy ez az ütemezés konfliktus-sorbarendezhető. A 9.8. ábrán látható a cserék sorozata, amellyel ez az ütemezés átalakítható a (T_1, T_2) soros ütemezéssé, ahol az összes T_1-beli művelet megelőzi az összes T_2-beli műveletet. Aláhúztuk azokat a szomszédos műveletpárokat, amelyeket felcserélünk az egyes lépésekben. □

$r_1(A)$; $w_1(A)$; $r_2(A)$; $\underline{w_2(A); r_1(B)}$; $w_1(B)$; $r_2(B)$; $w_2(B)$;
$r_1(A)$; $w_1(A)$; $\underline{r_2(A); r_1(B)}$; $w_2(A)$; $w_1(B)$; $r_2(B)$; $w_2(B)$;
$r_1(A)$; $w_1(A)$; $r_1(B)$; $r_2(A)$; $\underline{w_2(A); w_1(B)}$; $r_2(B)$; $w_2(B)$;
$r_1(A)$; $w_1(A)$; $r_1(B)$; $\underline{r_2(A); w_1(B)}$; $w_2(A)$; $r_2(B)$; $w_2(B)$;
$r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$; $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$;

9.8. ábra. *Egy konfliktus-sorbarendezhető ütemezés szomszédos műveletek felcserélésével való átalakítása soros ütemezéssé*

9.2.2. Megelőzési gráfok és teszt a konfliktus-sorbarendezhetőségre

Viszonylag könnyű megvizsgálnunk egy S ütemezést, és eldöntenünk, hogy konfliktus-sorbarendezhető-e vagy nem. Az az alapötlet, hogy ha valahol konfliktusban álló műveletek szerepelnek az S-ben, ezeket a műveleteket végrehajtó tranzakcióknak ugyanabban a sorrendben kell előfordulniuk a konfliktusekvivalens soros ütemezésekben, mint ahogyan az S-ben voltak. Tehát a konfliktusban álló műveletpárok megszorítást adnak a feltételezett konfliktusekvivalens soros ütemezésben a tranzakciók sorrendjére. Ha ezek a megszorítások nem mondanak egymásnak ellent, akkor találhatunk konfliktusekvivalens soros ütemezést. Ha pedig ellentmondanak egymásnak, akkor tudjuk, hogy nincs ilyen soros ütemezés.

Adott a T_1 és T_2 tranzakcióknak, esetleg további tranzakcióknak, egy S ütemezése. Azt mondjuk, hogy T_1 *megelőzi* T_2-t, és $T_1 <_S T_2$-vel jelöljük, ha van a T_1-ben olyan A_1 művelet, és a T_2-ben olyan A_2, hogy

1. A_1 megelőzi A_2-t az S-ben,
2. A_1 és A_2 ugyanarra az adatbáziselemre vonatkoznak, és
3. A_1 és A_2 közül legalább az egyik írás művelet.

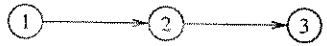
Megjegyezzük, hogy ezek pontosan azok a feltételek, amikor nem lehet felcserélni az A_1 és A_2 sorrendjét. Tehát, A_1 az A_2 előtt szerepel bármely az S-sel konfliktusekvivalens ütemezésben. Ennek eredményeként, ha ezek közül az ütemezések közül az egyik soros ütemezés, akkor ebben T_1-nek meg kell előznie T_2-t.

Ezeket a megelőzéseket a *megelőzési gráfban* (precedence graph) összegezhetjük. A megelőzési gráf csomópontjai az S ütemezés tranzakciói. Ha a tranzakciókat T_i-vel jelöljük az i függvényében, akkor a T_i-nek megfelelő csomópontot az i egésszel jelöljük. Az i csomópontból a j csomópontba vezet irányított él, ha $T_i <_S T_j$.

9.7. példa: A következő S ütemezés a T_1, T_2, T_3 három tranzakciót tartalmazza:

S: $r_2(A)$; $r_1(B)$; $w_2(A)$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $r_2(B)$; $w_2(B)$;

Ha az A-val kapcsolatos műveleteket nézzük meg, akkor több okot találunk, hogy miért igaz a $T_2 <_S T_3$. Például $r_2(A)$ az S-ben $w_3(A)$ előtt áll, és $w_2(A)$ az $r_3(A)$ és a $w_3(A)$ előtt is áll. A három észrevételünk közül bármelyik elegendő, hogy igazoljuk, valóban vezet él a 2-ből 3-ba a 9.9. ábrán szereplő megelőzési gráfban.



9.9. ábra. *A 9.7. példa S ütemezéséhez tartozó megelőzési gráf*

Hasonló módon ha megnézzük a B-vel kapcsolatos műveleteket, akkor szintén több okot találunk, hogy miért igaz a $T_1 <_S T_2$. Például $r_1(B)$ művelet a $w_2(B)$ művelet előtt áll. Tehát az S megelőzési gráfjában az 1-ből 2-be szintén vezet él. Tulajdonképpen ezek és csak ezek az élek azok, amelyeket az S ütemezésben szereplő műveletek sorrendjéből tudunk ellenőrizni. □

Van egy egyszerű szabály, amivel megmondhatjuk, hogy egy S ütemezés konfliktus-sorbarendezhető-e:

- Rajzoljuk fel az S megelőzési gráfját, és nézzük meg tartalmaz-e kört!

Ha igen, akkor S nem konfliktus-sorbarendezhető. Ha pedig a gráf körmentes, akkor S konfliktus-sorbarendezhető, továbbá a csomópontok bármelyik topologikus sorrendje[2] megadja a konfliktusekvivalens soros sorrendet.

9.8. példa: A 9.9. ábra megelőzési gráfja körmentes, így a 9.7. példa S ütemezése konfliktus-sorbarendezhető. A csomópontoknak, azaz a tranzakcióknak csak egyetlen sorrendje van, amely konzisztens a gráf éleivel, és ez: (T_1, T_2, T_3). Megjegyezzük,

[2] Egy körmentes gráf *topologikus sorrendje* a csomópontok bármely olyan rendezése, amelyben minden $a \to b$ élre, az a csomópont megelőzi a b csomópontot a topologikus rendezésben. Bármely körmentes gráfnak találhatunk topologikus rendezettséget úgy, hogy többszörösen ismételve töröljük azokat a csomópontokat, amelyeknek nincs megelőzője a maradék csomópontok között.

Miért nem szükséges konfliktus-sorbarendezhetőség a sorbarendezhetőséghez?

Egy példát már láttunk a 9.7. ábrán. Akkor megnéztük, hogy a T_2 által végrehajtott speciális számítások miatt hogyan vált az ütemezés sorba rendezhetővé. Pedig a 9.7. ábra ütemezése nem konfliktus-sorbarendezhető, ugyanis az A-t T_1 írja előbb, a B-t pedig a T_2. Mivel sem az A írását, sem a B írását nem lehet átrendezni, semmilyen módon nem kerülhet T_1 összes művelete a T_2 összes művelete elé, sem fordítva.

Azonban vannak olyan sorba rendezhető, de nem konfliktus-sorbarendezhető ütemezések is, amelyek nem függnek a tranzakciók által végrehajtott számításoktól. Például tekintsük a T_1, T_2, és T_3 három tranzakciót, amelyek mindegyike X értékét írja. A T_1 és T_2 az Y-nak is ír értéket, mielőtt az X-nek írnának értéket. Az egyik lehetséges ütemezés, amely itt éppen soros is, a következő:

S_1: $w_1(Y)$; $w_1(X)$; $w_2(Y)$; $w_2(X)$; $w_3(X)$.

Az S_1 ütemezés X értékének a T_3 által írt értéket, Y értékének pedig a T_2 által írt értéket adja. Ugyanezt végzi a következő ütemezés is:

S_2: $w_1(Y)$; $w_2(Y)$; $w_2(X)$; $w_1(X)$; $w_3(X)$.

Intuíció alapján átgondolva annak, hogy a T_1 és a T_2 milyen értéket ír az X-be, nincs hatása, ugyanis a T_3 felülírja az értékeket. Emiatt S_1 és S_2 az X-nek is és az Y-nak is ugyanazt az értéket adja. Mivel az S_1 soros, és az S_2-nek bármely adatbázis-állapotra ugyanaz a hatása, mint az S_1-nek, tehát S_2 sorba rendezhető. Ugyanakkor mivel nem tudjuk felcserélni $w_1(Y)$-t $w_2(Y)$-nal, és nem tudjuk felcserélni $w_1(X)$-et $w_2(X)$-szel, így cseréken keresztül nem lehet az S_2-t valamelyik soros ütemezéssé átalakítani. Tehát S_2 sorba rendezhető, de nem konfliktus-sorbarendezhető.

hogy az S-et valóban át lehet alakítani olyan ütemezéssé, amelyben a három tranzakció mindegyikének az összes művelete ugyanebben a sorrendben van, és ez a soros ütemezés:

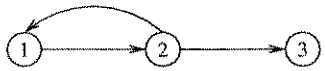
S'': $r_1(B)$; $w_1(B)$; $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$; $r_3(A)$; $w_3(A)$;

Ahhoz, hogy belássuk, megkaphatjuk az S-ből az S'-t szomszédos elemek cseréjével, az első észrevételünk, hogy az $r_1(B)$-t konfliktus nélkül az $r_2(A)$ elé hozhatjuk. Ezután három cserével a $w_1(B)$-t közvetlenül az $r_1(B)$ utánra tudjuk cserélni, ugyanis mindegyik közbeeső művelet az A-ra vonatkozik, és a B-re nem. Ezután az $r_2(B)$-t és a $w_2(B)$-t csak az A-ra vonatkozó műveleteken keresztül át tudjuk vinni pontosan a $w_2(A)$ utáni helyzetbe, amivel megkapjuk az S'-t. □

9.9. példa: Tekintsük az alábbi ütemezést:

S_1: $r_2(A)$; $r_1(B)$; $w_2(A)$; $r_2(B)$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $w_2(B)$;

amely csak abban különbözik az S-től, hogy az $r_2(B)$ művelet három hellyel előbb szerepel. Az A-ra vonatkozó műveleteket megvizsgálva most is csak a $T_2 <_{S_1} T_3$ megelőzési kapcsolathoz jutunk. De, ha a B-t vizsgáljuk, akkor nemcsak $T_1 <_{S_1} T_2$ teljesül (ugyanis $r_1(B)$ és $w_1(B)$ a $w_2(B)$ előtt szerepel), hanem $T_2 <_{S_1} T_1$ is (ugyanis $r_2(B)$ a $w_1(B)$ előtt fordul elő). Emiatt az S_1 ütemezéshez tartozó megelőzési gráf az, amely a 9.10. ábrán látható.



9.10. ábra. *A 9.9. példa S_1 ütemezéséhez tartozó ciklikus megelőzési gráf, ez az ütemezés nem konfliktus-sorbarendezhető*

Ez a gráf nyilvánvalóan tartalmaz kört. Arra következtethetünk, hogy S_1 nem konfliktus-sorbarendezhető, ugyanis intuíció alapján láthatjuk, hogy bármely konfliktusekvivalens soros ütemezésben a T_1-nek T_2 előtt is és után is kellene állnia, így emiatt nem létezik ilyen ütemezés. □

9.2.3. Miért működik a megelőzési gráfon alapuló tesztelés?

Láttuk, hogy a megelőzési gráfban a kör túl sok megszorítást jelent a feltételezett konfliktusekvivalens soros ütemezésre nézve. Azaz, ha létezik $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$ n darab tranzakcióból álló kör, akkor a feltételezett soros sorrendben T_1 műveleteinek meg kell előznie a T_2-ben szereplő műveleteket, amelyek megelőzik a T_3-belieket és így tovább egészen T_n-ig. De a T_n műveletei emiatt a T_1-beliek mögött vannak, ugyanakkor meg is kellene előzniük a T_1-belieket a $T_n \rightarrow T_1$ él miatt. Ebből következik, hogyha a megelőzési gráf tartalmaz kört, akkor az ütemezés nem konfliktus-sorbarendezhető.

A másik irányt egy kicsit nehezebb belátnunk. Azt kell megmutatnunk, hogy amikor a megelőzési gráf körmentes, akkor az ütemezés műveletei átrendezhetők szomszédos műveletek szabályos cseréivel úgy, hogy az ütemezés egy soros ütemezéssé váljon. Ha ezt meg tudjuk tenni, akkor bebizonyítottuk, hogy minden körmentes megelőzési gráffal rendelkező ütemezés egyben konfliktus-sorbarendezhető. A bizonyítás az ütemezésben részt vevő tranzakciók száma szerinti indukcióval történik.

Alapeset: Ha $n = 1$, vagyis csak egyetlen tranzakcióból áll az ütemezés, akkor az ütemezés már önmagában soros, emiatt biztosan konfliktus-sorbarendezhető.

Indukció: Legyen S

$T_1, T_2, \ldots, T_n$

n darab tranzakció műveleteiből álló ütemezés. Tételezzük fel, hogy S-nek körmentes megelőzési gráfja van. Ha a véges gráf körmentes, akkor van legalább egy olyan csomópontja, amelybe nem vezet él. Legyen a T_i tranzakciónak megfelelő i csomópont egy ilyen csomópont. Mivel nem vezet az i csomópontba él, nincs az S-ben olyan A művelet, hogy:

1. Valamelyik T_j, a T_i-től különböző tranzakcióra vonatkozzon,
2. T_i valamely műveletét megelőzi, és
3. Ezzel a művelettel konfliktusban van.

Ugyanis, ha lenne ilyen, akkor a megelőzési gráfban be kellene rajzolnunk egy élt a j csomópontból az i csomópontba.

Így lehetséges, hogy a T_i minden műveletét az S legelejére mozgatjuk át, miközben megtartjuk a sorrendjüket. Az ütemezés most a következő alakú:

(T_i műveletei) (a többi $n - 1$ tranzakció műveletei)

Most tekintsük az S második részét, vagyis a T_i-től különböző összes tranzakciónak a műveleteit. Mivel ezek a műveletek egymáshoz viszonyítva ugyanabban a sorrendben vannak, mint ahogyan az S-ben voltak, ennek a második résznek a megelőzési gráfja megegyezik az S olyan megelőzési gráfjával, amelyből elhagyjuk a T_i csomópontot és az ebből a csomópontból kimenő éleket.

Mivel az eredeti megelőzési gráf körmentes volt, és csomópontok, illetve élek törlésével nem válhatott ciklikussá, következik, hogy a második rész megelőzési gráfja is körmentes. Továbbá, mivel a második része $n - 1$ tranzakciót tartalmaz, alkalmazzuk rá az indukciós feltevést. Így tudjuk, hogy a második rész műveletei szomszédos műveletek szabályos cseréivel átrendezhetők soros ütemezéssé. Ily módon magát az S-et alakítottuk át olyan soros ütemezéssé, amelyben a T_i műveletei állnak legelöl, és a többi tranzakció műveletei ezután következnek valamilyen soros sorrendben. Az indukciót beláttuk, és így következik, hogy minden olyan ütemezés, amelynek körmentes a megelőzési gráfja, egyben konfliktus-sorbarendezhető.

9.2.4. Feladatok

9.2.1. feladat: Az alábbiakban úgy adunk meg két tranzakciót, hogy leírjuk az A és B két adatbáziselemre vonatkozó hatásukat, amelyekről feltehetjük, hogy egészek.

T_1: `READ(A,t); t := t+2; WRITE(A,t); READ(B,t); t := t*3; WRITE(B,t);`

T_2: `READ(B,s); s := s*2; WRITE(B,s); READ(A,s); s := s+3; WRITE(A,s);`

Tételezzük fel, hogy bármilyen, az adatbázis konzisztenciájára vonatkozó megszorításokat megőrzik ezek a tranzakciók, ha egymástól elkülönítve hajtjuk végre őket. Megjegyezzük, hogy $A = B$ *nem* konzisztenciára vonatkozó megszorítás.

a) Igazoljuk, hogy mindkét soros ütemezésnek az adatbázison való hatása megegyezik; azaz (T_1, T_2) és (T_2, T_1) ekvivalensek! Mutassuk be ezt a tényt úgy, hogy a két tranzakció hatását tetszőleges kezdeti adatbázis-állapotból vizsgáljuk meg!
b) Adjunk példákat a fenti 12 művelet sorba rendezhető és nem sorsorba rendezhető ütemezésére!
c) Mennyi soros ütemezése van a 12 műveletnek?
*!! d) Mennyi sorba rendezhető ütemezése van a 12 műveletnek?

9.2.2. feladat: A 9.2.1. feladat két tranzakcióját átírva olyanná, amely csak az olvasási és írási műveleteket tartalmazza, a következőt kapjuk:

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$;
T_2: $r_2(B)$; $w_2(B)$; $r_2(A)$; $w_2(A)$;

Válaszoljunk a következő kérdésekre:

*! a) A fenti nyolc művelet lehetséges ütemezései közül hány darab konfliktusekvivalens a (T_1, T_2) soros sorrenddel?
b) A nyolc művelet hány darab ütemezése ekvivalens a (T_2, T_1) soros sorrenddel?
!! c) A nyolc művelet hány ütemezése ekvivalens (nem feltétlenül konfliktusekvivalens) a (T_1, T_2) soros sorrenddel, feltéve, hogy a tranzakciók hatása az adatbázis-állapotra ugyanaz, mint amelyet a 9.2.1. feladatban leírtunk?
! d) Miért különbözik a fenti c) és a 9.2.1.d) feladat válasza?

! **9.2.3. feladat:** Tegyük fel, hogy a 9.2.2. feladat tranzakcióit megváltoztatjuk:

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$;
T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$;

Azaz a tranzakciók szemantikája ugyanaz marad, mint a 9.2.1. feladatban volt, de a T_2 annyiban változik, hogy az A-t előbb dolgozza fel, mint a B-t. Adjuk meg:

a) A konfliktus-sorbarendezhető ütemezések számát.
b) A sorba rendezhető ütemezések számát, feltéve, hogy a tranzakcióknak ugyanolyan a hatásuk az adatbázis-állapotra, mint a 9.2.1. feladatban.

9.2.4. feladat: Tekintsük az alábbi ütemezéseket:

* a) $r_1(A)$; $r_2(A)$; $r_3(B)$; $w_1(A)$; $r_2(C)$; $r_2(B)$; $w_2(B)$; $w_1(C)$;
b) $r_1(A)$; $w_1(B)$; $r_2(B)$; $w_2(C)$; $r_3(C)$; $w_3(A)$;
c) $w_3(A)$; $r_1(A)$; $w_1(B)$; $r_2(B)$; $w_2(C)$; $r_3(C)$;
d) $r_1(A)$; $r_2(A)$; $w_1(B)$; $w_2(B)$; $r_1(B)$; $r_2(B)$; $w_2(C)$; $w_1(D)$;
e) $r_1(A)$; $r_2(A)$; $r_1(B)$; $r_2(B)$; $r_3(A)$; $r_4(B)$; $w_1(A)$; $w_2(B)$;

Válaszoljunk mindegyik esetében a következő kérdésekre:

i) Mi az ütemezés megelőzési gráfja?
ii) Konfliktus-sorbarendezhető-e az ütemezés? Ha igen, akkor melyek az összes ekvivalens soros ütemezések?
! iii) Vannak-e olyan soros ütemezések, amelyek ekvivalensek (függetlenül attól, hogy hogyan hatnak a tranzakciók az adatokon), de nem konfliktusekvivalensek?

!! **9.2.5. feladat:** Azt mondjuk, hogy a T tranzakció *megelőzi* az U tranzakciót egy S ütemezésben, ha a T összes művelete megelőzi az U összes műveletét az S-ben. Megjegyezzük, hogyha az S-ben csak a T és U tranzakciók vannak, akkor az, hogy T megelőzi az U-t, ugyanazt jelenti, hogy az S a (T, U) soros ütemezés. De, ha az S más tranzakciókat is tartalmaz a T-n és U-n kívül, akkor nem biztos, hogy az S sorba rendezhető, és valójában a többi tranzakció hatása miatt még az is előfordulhat, hogy nem konfliktus-sorbarendezhető. Adjunk példát az S olyan ütemezésére, amely:

i) S-ben T_1 megelőzi T_2-t, és
ii) S konfliktus-sorbarendezhető, de
iii) Minden S-sel konfliktusekvivalens soros ütemezésben T_2 megelőzi T_1-et!

! **9.2.6. feladat:** Magyarázzuk meg, hogyan lehet bármely $n > 1$ esetén találni olyan ütemezést, amelynek a megelőzési gráfja tartalmaz n hosszúságú kört, de kisebb kört nem!

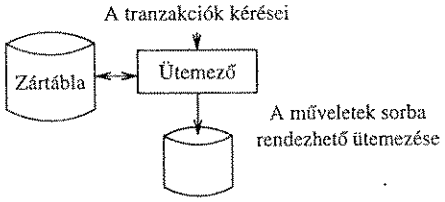
9.3. A sorbarendezhetőség biztosítása zárakkal

Képzeljünk el olyan tranzakcióhalmazt, amely megszorítások nélkül hajtja végre a műveleteit. Ezek a műveletek is egy ütemezést alkotnak, de nem valószínű, hogy ez az ütemezés sorba rendezhető lenne. Az ütemező feladata az, hogy megakadályozza az olyan műveleti sorrendeket, amelyek nem sorba rendezhető ütemezésekhez vezetnek. Ebben a részben az ütemező legáltalánosabb szerkezetét tekintjük, egy olyat, amelyben az adatbáziselemekre kiadott „zárak" akadályozzák meg a nem sorba rendezhető viselkedést. Intuíció alapján arról van szó, hogy a tranzakciók zárolják azokat az adatbáziselemeket, amelyekhez hozzáférnek, hogy megakadályozzák azt, hogy ugyanakkor más tranzakciók is hozzáférjenek ezekhez az elemekhez, mivel ekkor felmerülne a nem sorbarendezhetőség kockázata.

Ebben a fejezetben egy (nagyon is) leegyszerűsített zárolási sémával vezetjük be a zárolás fogalmát. Ebben a sémában csak egyféle zár van, amelyet a tranzakcióknak meg kell kapniuk az adatbáziselemre, ha bármilyen műveletet végre akarnak hajtani ezen az elemen. A 9.4. részben sokkal valósabb zárolási sémákat tanulmányozunk, különböző zármódokkal, beleértve az általános osztott/kizárólagos zárakat, amelyek az olvasási és írási jogoknak felelnek meg.

9.3.1. Zárak

A 9.11. ábrán egy ütemezőt láthatunk, amely zártábla segítségével végzi a feladatát. Emlékeztetünk arra, hogy az ütemező felelős azért, hogy fogadja a tranzakcióktól a kéréseket, és vagy megengedi a műveleteket az adatbázison, vagy addig késlelteti, amikor már biztonságosan végre tudja hajtani őket. A továbbiakban kifejtjük, hogyan irányítja ezt a döntést a zártábla felhasználásával.



9.11. ábra. *A döntéseket zártábla felhasználásával irányító ütemező*

Az lenne az ideális, ha az ütemező akkor, és csak akkor továbbítana egy kérést, ha annak végrehajtása nem vezethetne inkonzisztens adatbázis-állapotba, miután az összes aktív tranzakciót vagy véglegesen végrehajtottuk, vagy abortáltuk (vagyis leállítottuk a befejezése előtt, más szóval sikertelenül fejeztük be). Ezt a kérdést viszont túl nehéz lenne valós időben eldönteni. Így minden ütemező csak egy egyszerű tesztet hajt végre a sorbarendezhetőség biztosítására, azonban letilthat olyan műveleteket is, amelyek önmagunkban nem vezetnének inkonzisztenciához. A zárolási ütemező, mint a legtöbb ütemező, a konfliktus-sorbarendezhetőséget követeli meg, pedig mint ezt már tanultuk, ez erősebb követelmény, mint a sorbarendezhetőség.

Ha az ütemező zárakat használ, akkor a tranzakcióknak zárakat kell kérniük, és feloldaniuk az adatbáziselemek olvasásán és írásán felül. A zárak használatának két értelemben is helyesnek kell lennie, mind a tranzakciók szerkezetére, mind pedig az ütemezők szerkezetére alkalmazva.

- *Tranzakciók konzisztenciája* (consistency of transactions): A műveletek és a zárak az alábbi elvárások szerint kapcsolódnak egymáshoz:

  1. A tranzakció csak akkor olvashat vagy írhat egy elemet, ha már korábban zárolta az elemet, és még nem oldotta fel a zárat.
  2. Ha egy tranzakció zárol egy elemet, akkor később azt fel kell szabadítania.

- *Az ütemezések jogszerűsége* (legality of schedules): A zárak értelme feleljen meg a szándék szerinti elvárásnak: nem zárolhatja két tranzakció ugyanazt az elemet, csak úgy, ha az egyik előbb már feloldotta a zárat.

Kibővítjük a jelöléseinket a zárolás és a feloldás műveletekkel:

$l_i(X)$: T_i tranzakció az X adatbáziselemre *zárolást kér* (request a lock, l = „lock").
$u_i(X)$: T_i tranzakció az X adatbáziselem *zárolását feloldja* (release its lock, u = „unlock").

Így a tranzakciók konzisztenciafeltétele úgy is kimondható, hogy: „Amikor egy T_i tranzakcióban van egy $r_i(X)$ vagy egy $w_i(X)$ művelet, akkor van korábban egy $l_i(X)$ művelet, de közben nincs $u_i(X)$, és van később egy $u_i(X)$ művelet." Az ütemezések jogszerűsége azt mondja ki, hogy: „Ha egy ütemezésben van olyan $l_i(X)$ művelet, amelyet $l_j(X)$ követ, akkor e két művelet között lennie kell egy $u_i(X)$ műveletnek."

9.10. példa: Tekintsük a 9.1. példában szereplő T_1, T_2 tranzakciókat. Emlékeztetőül, a T_1 hozzáad az A és B adatbáziselemekhez 100-at, a T_2 pedig megduplázza az értéküket. Itt most úgy adjuk meg a tranzakciókat, hogy a zárolási és az aritmetikai számolási műveleteket is leírjuk segítségképpen, hogy emlékezzünk, mit tesznek a tranzakciók.[3]

T_1: $l_1(A)$; $r_1(A)$; `A := A+100;` $w_1(A)$; $u_1(A)$; $l_1(B)$; $r_1(B)$; `B := B+100;` $w_1(B)$; $u_1(B)$;
T_2: $l_2(A)$; $r_2(A)$; `A := A*2;` $w_2(A)$; $u_2(A)$; $l_2(B)$; $r_2(B)$; `B := B*2;` $w_2(B)$; $u_2(B)$;

Mindkét tranzakció konzisztens. Mindkettő felszabadítja az A-ra és B-re kiadott zárakat. Továbbá mindkettő csak olyan lépésekben dolgozik az A-n és a B-n, amikor előzőleg már zárolták az elemet, és még nem oldották fel a zár alól.

A 9.12. ábrán ennek a két tranzakciónak egy jogszerű ütemezése található. Helymegtakarítás miatt több műveletet írtunk egy sorban. Ez az ütemezés jogszerű, ugyanis a két tranzakció sohasem zárolja egyidejűleg az A-t vagy a B-t. Pontosabban, a T_2 nem végzi el az $l_2(A)$-t, csak miután a T_1 végrehajtotta az $u_1(A)$-t, és a T_1 nem végzi el az

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$;			
`A := A+100;`			
$w_1(A)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$;		
	`A := A*2;`		
	$w_2(A)$; $u_2(A)$;	250	
	$l_2(B)$; $r_2(B)$;		
	`B := B*2;`		
	$w_2(B)$; $u_2(B)$;		50
$l_1(B)$; $r_1(B)$;			
`B := B+100;`			
$w_1(B)$; $u_1(B)$;			150

9.12. ábra. *Konzisztens tranzakciók jogszerű ütemezése. Sajnos nem sorba rendezhető*

[3] Megjegyezzük, hogy a tranzakciók aktuális számításait rendszerint nem ábrázoljuk a jelenlegi jelölésünkben, ugyanis az ütemező sem tudja ezt figyelembe venni, amikor arról dönt, hogy engedélyezze vagy elutasítsa a tranzakciókéréseket.

$l_1(B)$-t, csak miután a T_2 végrehajtotta az $u_2(B)$-t. Láthatjuk a kiszámított értékek nyomon követésével, hogy bár ez az ütemezés jogszerű, mégsem sorba rendezhető. A 9.3.3. részben látni fogunk egy további feltételt, a „kétfázisú zárolás"-t, amivel biztosíthatjuk majd, hogy a jogszerű ütemezések konfliktus-sorbarendezhetők legyenek. □

9.3.2. A zárolási ütemező

A zároláson alapuló ütemező feladata, hogy akkor és csak akkor engedélyezze a kéréseket, ha a kérés jogszerű ütemezést eredményez. Ezt a döntést segíti a zártábla, amely minden adatbáziselemhez megadja azt a tranzakciót, ha van ilyen, amelyik pillanatnyilag éppen zárolja az adott elemet. Részletesen később a 9.5.2. részben beszélünk a zártábla szerkezetéről. Ha viszont csak egyféle zárolás van, mint ahogyan eddig feltételeztük, akkor úgy tekinthetjük a táblát, mint (X, T) párokból álló `Zárolások(elem, tranzakció)` relációt, ahol a T tranzakció zárolja az X adatbáziselemet. Az ütemezőnek csak le kell kérdeznie ezt a relációt, illetve egyszerű `INSERT` és `DELETE` utasításokkal kell módosítania.

9.11. példa: A 9.12. ábrán található ütemezés jogszerű, ahogyan ezt már láttuk, így a zárolási ütemező engedélyezhetné az összes kérést abban a sorrendben, ahogyan beérkeznek. Néha azonban előfordulhat, hogy nem lehet engedélyezni a kéréseket. Tekintsük a 9.10. példából a T_1 és T_2 tranzakciókat egy apró (de a 9.3.3. részben majd látni fogjuk, hogy lényeges) változtatással, mégpedig a T_1 is és a T_2 is előbb zárolja B-t, és csak azután oldja fel A zárolását.

T_1: $l_1(A)$; $r_1(A)$; `A := A+100`; $w_1(A)$; $l_1(B)$; $u_1(A)$; $r_1(B)$; `B := B+100`; $w_1(B)$; $u_1(B)$;
T_2: $l_2(A)$; $r_2(A)$; `A := A*2`; $w_2(A)$; $l_2(B)$; $u_2(A)$; $r_2(B)$; `B := B*2`; $w_2(B)$; $u_2(B)$;

A 9.13. ábrán látható, hogy amikor T_2 kéri B zárolását, az ütemezőnek el kell utasítania ezt a kérést, ugyanis T_1 még zárolja a B-t. Így T_2 áll, és a következő műveleteket a T_1 végzi. Végül a T_1 végrehajtja $u_1(B)$-t, amely felszabadítja a B-t. Most T_2 már zárolhatja a B-t, amelyet a következő lépésben végre is hajt. Megjegyezzük, hogy mivel T_2-nek várakoznia kellett, emiatt a B-t akkor szorozza meg 2-vel, miután a T_1 már hozzáadott 100-at B-hez, és ez konzisztens adatbázis-állapotot eredményez. □

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$;			
`A := A+100;`			
$w_1(A)$; $l_1(B)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$;		
	`A := A*2;`		
	$w_2(A)$;	250	
	$l_2(B)$; **Elutasítva**		
$r_1(B)$; `B := B+100;`			
$w_1(B)$; $u_1(B)$;			125
	$l_2(B)$; $u_2(A)$; $r_2(B)$;		
	`B := B*2;`		
	$w_2(B)$; $u_2(B)$;		250

9.13. ábra. *A zárolási ütemező késlelteti azt a kérést, amely jogtalan ütemezéshez vezetne*

9.3.3. A kétfázisú zárolás

Van egy meglepő feltétel, amellyel biztosítani tudjuk, hogy konzisztens tranzakciók jogszerű ütemezése konfliktus-sorbarendezhető legyen. Ezt a feltételt, amelyet a gyakorlatban elterjedt zárolási rendszerek leginkább követnek, *kétfázisú zárolásnak* (two-phase locking) vagy *2FZ-nek* (2PL) nevezzük. A 2FZ feltétel:

- Minden tranzakcióban minden zárolási művelet megelőzi az összes zárfeloldási műveletet.

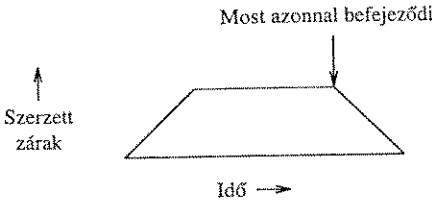
A „kétfázis", amelyre a 2FZ-vel hivatkozunk, abból adódik, hogy az első fázisban csak zárolásokat adunk ki, a második fázisban pedig csak megszüntetünk zárolásokat. A kétfázisú zárolás a konzisztenciához hasonlóan, a tranzakcióban a műveletek sorrendjére egy feltétel. Azokat a tranzakciókat, amelyek eleget tesznek a 2FZ feltételnek *kétfázisú zárolású tranzakcióknak* (two-phase-locked transaction) vagy 2FZ *tranzakcióknak* nevezzük.

9.12. példa: A 9.10. példában a tranzakciók nem engedelmeskednek a kétfázisú zárolási szabálynak. Például a T_1 előbb oldja fel az A zárolását, mint zárolná a B-t. A 9.11. példában található tranzakciók változata azonban már *eleget tesz* a 2FZ feltételnek. Megjegyezzük, hogy T_1 az A-t és B-t is az első öt műveleten belül zárolja, és a következő öt műveleten belül feloldja a zárakat; és a T_2 is hasonlóan viselkedik. Ha összehasonlítjuk a 9.12. és 9.13. ábrákat látjuk, hogy a kétfázisú zárolású tranzakciók hogyan működnek együtt az ütemezővel a konzisztencia biztosítására, ellenben a nem 2FZ tranzakciók esetén előfordulhat inkonzisztencia (és emiatt nem konfliktus-sorbarendezhető) viselkedés.

9.3.4. Miért működik a kétfázisú zárolás?

Igaz, bár közel sem nyilvánvaló, hogy a 2FZ példánkban észlelt előnyei általában is érvényesek. Intuíció alapján, mindegyik kétfázisú zárolású tranzakcióról azt gondolhatjuk, hogy rögtön végrehajtásra kerülnek, amint az első zárfeloldási kérés kiadásra kerül, ahogy ezt a 9.14. ábra javasolja. A 2FZ tranzakciók S ütemezésével konfliktusekvivalens soros ütemezés pont olyan, mint amelyikben a tranzakciók ugyanabban a sorrendben vannak, amilyenek az első zárfeloldásaik.[4]

[4] Bizonyos ütemezések esetén más konfliktusekvivalens soros ütemezések is lehetnek.



9.14. ábra. *Minden kétfázisú zárolású tranzakciónak van olyan pontja, amikor azt mondhatjuk, hogy azonnal befejeződik*

Megmutatjuk, hogyan lehet konzisztens, kétfázisú zárolású tranzakciók bármely S jogszerű ütemezését átalakítani konfliktusekvivalens soros ütemezéssé. A konverziót legjobban az S-ben részt vevő tranzakciók n száma szerinti indukcióval tudjuk leírni. A következőkben lényeges megjegyeznünk, hogy a konfliktusekvivalencia csak az olvasási és írási műveletekre vonatkozik. Amikor felcseréljük az olvasások és írások sorrendjét, akkor figyelmen kívül hagyjuk a zárolási és zárfeloldási műveleteket. Amikor megkaptuk az olvasási és írási műveletek sorrendjét, akkor úgy helyezzük el köréjük a zárolási és zárfeloldási műveleteket, ahogyan azt a különböző tranzakciók megkövetelik. Mivel minden tranzakció felszabadítja az összes zárolást a tranzakció befejezése előtt, tudjuk, hogy a soros ütemezés jogszerű lesz.

Alapeset: Ha $n = 1$, akkor semmit sem kell tennünk, az S már soros ütemezés.

Indukció: Tételezzük fel, hogy S n darab tranzakciót tartalmaz: $T_1, T_2, \ldots, T_n$, és legyen T_i az a tranzakció, amelyik a teljes S ütemezésben a legelső zárfeloldási műveletet végzi, mondjuk az $u_i(X)$-et. Azt állítjuk, hogy a T_i összes olvasási és írási műveletét az ütemezés legelejére tudjuk vinni anélkül, hogy konfliktusműveleteken kellene áthaladnunk.

Tekintsük a T_i valamelyik műveletét, mondjuk $w_i(Y)$-t. Megelőzheti-e ezt az S-ben valamely konfliktus művelet, például $w_j(Y)$? Ha így lenne, akkor az S ütemezésben $u_j(Y)$ és $l_i(Y)$ műveletek az alábbi módon helyezkednének el a művelet sorozatban

$$\ldots; w_j(Y); \ldots; u_j(Y); \ldots; l_i(Y); \ldots; w_i(Y); \ldots$$

Mivel T_i az első, amelyik zárat old fel, így az S-ben az $u_i(X)$ megelőzi az $u_j(Y)$-t, vagyis az S a következőképpen néz ki:

$$\ldots; w_j(Y); \ldots; u_i(X); \ldots; u_j(Y); \ldots; l_i(Y); \ldots; w_i(Y); \ldots$$

illetve az $u_i(X)$ előfordulhat még a $w_j(Y)$ előtt is. Mindegyik esetben az $u_i(X)$ az $l_i(Y)$ előtt van, ami azt jelenti, hogy a T_i *nem* lenne kétfázisú zárolású, amint azt feltételeztük. Ahogyan beláttuk, hogy nem létezhetnek konfliktuspárok az írásra, ugyanúgy be lehet látni bármely két lehetséges műveletre, az egyiket T_i-ből, a másikat pedig egy másik T_j-ből választva, hogy nem lehetnek konfliktuspárok.

Bebizonyítottuk, hogy valóban az S legelejére lehet vinni a T_i összes műveletét

konfliktusmentes olvasási és írási műveletpárok cseréjével. Ezután elhelyezhetjük a T_i zárolási és feloldási műveleteit. Vagyis az S-et a következő alakba írhatjuk át

(T_i műveletei) (a többi $n - 1$ tranzakció műveletei)

Az $n - 1$ tranzakcióból álló második része szintén konzisztens, 2FZ tranzakciókból álló jogszerű ütemezés, így az indukciós feltevést alkalmazhatjuk rá. Átalakítjuk a második részt konfliktusekvivalens soros ütemezéssé, ily módon a teljes S konfliktus-sorbarendezhetővé vált.

9.3.5. Feladatok

9.3.1. feladat: Az alábbiakban megadunk két tranzakciót a zárolási kéréseikkel és a tranzakciók szemantikájával. Megjegyezzük, hogy a 9.2.1. feladatban láttuk ezeknek a tranzakcióknak azt a szokatlan tulajdonságát, hogy úgy ütemezhetők, hogy ne legyenek konfliktus-sorbarendezhetők, de a szemantikájuk miatt mégis sorba rendezhetők.

A holtpont kockázata

Az egyik probléma, amelyet nem lehet a kétfázisú zárolással megoldani, a holtpontok bekövetkezésének a lehetősége, vagyis amikor az ütemező arra kényszeríti a tranzakciókat, hogy örökké várakozzanak egy olyan zárra, amelyet egy másik tranzakció tart zárolva. Például tekintsük a 9.11. példa 2FZ tranzakcióit, de most a T_2 dolgozza fel előbb a B-t:

T_1: $l_1(A)$; $r_1(A)$; `A := A+100`; $w_1(A)$; $l_1(B)$; $u_1(A)$; $r_1(B)$; `B := B+100`; $w_1(B)$; $u_1(B)$;
T_2: $l_2(B)$; $r_2(B)$; `B := B*2`; $w_2(B)$; $l_2(A)$; $u_2(B)$; $r_2(A)$; `A := A*2`; $w_2(A)$; $u_2(A)$;

A tranzakciós műveletek egy lehetséges végrehajtása:

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$;			
	$l_2(B)$; $r_2(B)$;		
`A := A+100;`			
	`B := B*2;`		
$w_1(A)$;		125	
	$w_2(B)$;		50
$l_1(B)$; **Elutasítva**	$l_2(A)$; **Elutasítva**		

Most egyik tranzakció sem folytatódhat, hanem örökké várakozniuk kell. A 10.3. részben megvizsgáljuk azokat a módszereket, amelyek megszüntetik ezt a helyzetet. Viszont vegyük észre, hogy nem tudjuk mind a két tranzakciót folytatni, ugyanis ha így lenne, akkor az adatbázis végső állapotában nem lehetne $A = B$.

T_1: $l_1(A)$; $r_1(A)$; A := A+2; $w_1(A)$; $u_1(A)$; $l_1(B)$; $r_1(B)$; B := B*3; $w_1(B)$; $u_1(B)$;
T_2: $l_2(B)$; $r_2(B)$; B := B*2; $w_2(B)$; $u_2(B)$; $l_2(A)$; $r_2(A)$; A := A+3; $w_2(A)$; $u_2(A)$;

Az alábbi kérdésekben az ütemezéseknek csak az írás és olvasás műveleteit tekintsük, a zárolási, feloldási és értékadási lépésektől tekintsünk el.

* a) Adjunk példát a zárolások miatt tiltott ütemezésre!

! b) Nyolc olvasás és írás művelet $\binom{8}{4} = 70$ sorrendjéből mennyi a jogszerű ütemezés (vagyis amelyeket zárakkal megengedünk)?

! c) A jogszerű ütemezések közül mennyi sorba rendezhető (a tranzakciók fent megadott szemantikája szerint)?

! d) A jogszerű és sorba rendezhető ütemezések közül mennyi konfliktus-sorbarendezhető?

!! e) Mivel a T_1 és a T_2 nem kétfázisú zárolású, azt várnánk, hogy bizonyos nem sorba rendezhető viselkedés fordulna elő. Vannak-e olyan jogszerű ütemezések, amelyek nem sorba rendezhetők? Ha igen, adjunk rá példát, ha nem, akkor magyarázzuk meg, hogy miért nem!

*! **9.3.2. feladat:** A 9.3.1. feladat tranzakcióit kétfázisú zárolásúvá írtuk át úgy, hogy az összes zárfeloldási műveletet a végére vittük:

T_1: $l_1(A)$; $r_1(A)$; A := A+2; $w_1(A)$; $l_1(B)$; $r_1(B)$; B := B*3; $w_1(B)$; $u_1(A)$; $u_1(B)$;
T_2: $l_2(B)$; $r_2(B)$; B := B*2; $w_2(B)$; $l_2(A)$; $r_2(A)$; A := A+3; $w_2(A)$; $u_2(B)$; $u_2(A)$;

Ezeknek a tranzakcióknak az összes írási és olvasási műveleteiből hány jogszerű ütemezést tudunk képezni?

9.3.3. feladat: A 9.2.4. feladat mindegyik ütemezése esetén tételezzük fel, hogy minden tranzakció közvetlenül azelőtt zárolja az adatbáziselemeket, mielőtt olvasná vagy írná az elemet, és minden tranzakció azonnal feloldja az elem zárolását, miután utoljára használta az elemet. Mondjuk meg, hogy mit tenne a zárolási ütemező ezeknek az ütemezéseknek mindegyikével, vagyis melyik kérést késleltetnénk, és mikor adnánk lehetőséget a tranzakció folytatására?

! **9.3.4. feladat:** Az alábbiakban megadott tranzakciók mindegyikénél tételezzük fel, hogy beszúrjuk a zárolás és feloldás műveletet minden egyes adatbáziselemhez, amihez hozzáférünk!

* a) $r_1(A)$; $w_1(B)$;
b) $r_2(A)$; $w_2(A)$; $w_2(B)$;

Adjuk meg, hogy a zárolási, feloldási, olvasási és írási műveleteknek hány sorrendje lehet az alábbi tranzakcióknál:

i) Konzisztens és kétfázisú zárolású.
ii) Konzisztens, de nem kétfázisú zárolású.
iii) Nem konzisztens, de kétfázisú zárolású.
iv) Sem nem konzisztens, sem nem kétfázisú zárolású.

9.4. Különböző zármódú zárolási rendszerek

A 9.3. rész zárolási sémája bemutatja a zárolás mögött álló legfőbb elveket, de túl egyszerű ahhoz, hogy a gyakorlatban is használható séma legyen. Az a legfőbb probléma, hogy a T tranzakciónak akkor is zárolnia kell egy X adatbáziselemet, ha csak olvasni akarja X-et, és nem akarja írni. Nem kerülhetjük el a zárolást ekkor sem, mert ha nem zárolnánk, akkor esetleg egy másik tranzakció azalatt írna az X-be új értéket, mialatt a T aktív, ami nem sorba rendezhető viselkedést okoz. Másrészről pedig miért is ne olvashatná több tranzakció egyidejűleg az X értékét mindaddig, amíg egyiknek sincs engedélyezve, hogy írjon az X-be.

Ez indokolja, hogy bevezessük a legelterjedtebb zárolási sémát, amikor két különböző zárat használunk, az egyiket az olvasáshoz (ezt „osztott zárnak" vagy „olvasási zárnak" nevezzük), és egyet az íráshoz (amelyet „kizárólagos zárnak" vagy „írási zárnak" hívunk). Ezután pedig megvizsgálunk egy fejlettebb sémát, amikor a tranzakcióknak később lehetőségük lesz osztott zárat „felminősíteni" kizárólagos zárrá. Megnézzük a „növelési zárakat" is olyan speciális írási műveletek kezelésére, amelyek növelik az adatbáziselemet. A lényeges különbség az, hogy a növelési műveletek felcserélhetők egymással, ellenben az általános írási műveletek nem. Ezek a példák elvezetnek a „kompatibilitási mátrix" segítségével megadott zárolási séma általános fogalmához, amely azt jelzi, hogy milyen zárakat engedélyezhetünk az olyan adatbáziselemekre, amelyek már zárolva vannak.

9.4.1. Osztott és kizárólagos zárak

Mivel ugyanannak az adatbáziselemnek két olvasási művelete nem eredményez konfliktust, így ahhoz nincs szükség zárolásra vagy más konkurenciavezérlési működésre, hogy az olvasási műveleteket egy bizonyos sorrendbe soroljuk. Mint a bevezetőben javasoltuk, továbbra is szükséges azt az elemet is zárolni, amelyet olvasunk, ugyanis ennek az elemnek az írását nem szabad közben megengednünk. Az íráshoz szükséges zár viszont „erősebb", mint az olvasáshoz szükséges zár, ugyanis ennek mind az olvasásokat, mind az írásokat meg kell akadályoznia.

Emiatt olyan zárolási ütemezőt tekintsünk, amely két különböző zárat alkalmaz: az *osztott zárakat* (shared locks) és a *kizárólagos zárakat* (exclusive locks). Intuíció alapján tetszőleges X adatbáziselemet vagy egyszer lehet zárolni kizárólagosan, vagy akárhányszor lehet zárolni osztottan, ha még nincs kizárólagosan zárolva. Amikor írni akarjuk az X-et, akkor az X-en kizárólagos zárral kell rendelkeznünk, de ha csak ol-

vasni szándékozzuk az X-et, akkor az X-en akár osztott, akár kizárólagos zár megfelel. Feltételezzük, hogy ha olvasni akarjuk az X-et, de írni nem, akkor előnyben részesítjük az osztott zárolást.

Az $sl_i(X)$ jelölést használjuk arra, hogy „a T_i tranzakció osztott zárat kér az X adatbáziselemre", és $xl_i(X)$ jelölést pedig arra, hogy „a T_i kizárólagos zárat kér az X-re". Továbbra is $u_i(X)$-szel jelöljük, hogy a T_i feloldja az X zárását, vagyis felszabadítja minden zár alól az X-et.

A három követelmény, a tranzakciók konzisztenciája, a tranzakciók 2FZ-je, és az ütemezések jogszerűsége, mindegyikének van megfelelője az osztott/kizárólagos zárolási rendszerben. Az alábbiakban összegezzük ezeket a követelményeket:

1. *Tranzakciók konzisztenciája:* Nem írhatunk kizárólagos zár fenntartása nélkül és nem olvashatunk valamilyen zár fenntartása nélkül. Pontosabban fogalmazva, bármely T_i tranzakcióban.

   a) Az $r_i(X)$ olvasási műveletet meg kell hogy előzze egy $sl_i(X)$ vagy $xl_i(X)$ úgy, hogy közben nincs $u_i(X)$.
   b) A $w_i(X)$ írási műveletet meg kell hogy előzze egy $xl_i(X)$ úgy, hogy közben nincs $u_i(X)$.

   Minden zárolást követnie kell ugyanannak az elemnek a zárolását feloldó műveletnek.
2. *Tranzakciók kétfázisú zárolása:* A zárolásoknak meg kell előzniük a zárak feloldását. Pontosabban fogalmazva, bármely T_i kétfázisú zárolású tranzakcióban egyetlen $sl_i(X)$ vagy $xl_i(X)$ műveletet sem előzhet meg egyetlen $u_i(Y)$ művelet sem semmilyen Y-ra.
3. *Az ütemezések jogszerűsége:* Egy elemet vagy egyetlen tranzakció zárol kizárólagosan, vagy több is zárolhatja osztottan, de a kettő egyszerre nem lehet. Pontosabban fogalmazva:

   a) Ha $xl_i(X)$ szerepel egy ütemezésben, akkor ezután nem következhet $xl_j(X)$ vagy $sl_j(X)$ valamely i-től különböző j-re anélkül, hogy közben ne szerepelne $u_i(X)$.
   b) Ha $sl_i(X)$ szerepel egy ütemezésben, akkor ezután nem következhet $xl_j(X)$ $j \neq i$-re anélkül, hogy közben ne szerepelne $u_i(X)$.

Megjegyezzük, hogy engedélyezett, hogy egy tranzakció ugyanazon elemre kérjen és tartson mind osztott, mind kizárólagos zárat, feltéve, ha ezzel nem kerül konfliktusba más tranzakciók zárolásaival. Ha a tranzakciók előre tudnák, milyen zárakra lesz szükségük, akkor biztosan csak a kizárólagos zárolást kérnék, de ha nem láthatók előre a zárolási igények, lehetséges, hogy egy tranzakció osztott és kizárólagos zárakat is kér különböző időpontokban.

9.13. példa: Nézzük az alábbi osztott és kizárólagos zárakat használó két tranzakciónak egy lehetséges ütemezését:

T_1: $sl_1(A)$; $r_1(A)$; $xl_1(B)$; $r_1(B)$; $w_1(B)$; $u_1(A)$; $u_1(B)$;
T_2: $sl_2(A)$; $r_2(A)$; $sl_2(B)$; $r_2(B)$; $u_2(A)$; $u_2(B)$;

A T_1 is és a T_2 is olvassa A-t és B-t, de csak a T_1 írja B-t. Egyik sem írja A-t.

A 9.15. ábrán T_1 és T_2 műveleteinek olyan ütemezése látható, amelyet a T_1 kezd az A osztott zárolásával. Ezután a T_2 következik, az A és B mindegyikét osztottan zárolja. Most a T_1-nek lenne szüksége a B kizárólagos zárolására, ugyanis olvassa is és írja is a B-t. Viszont nem kaphatja meg a kizárólagos zárat, hiszen a T_2-nek már osztott zárja van a B-n. Így az ütemező várakozni kényszeríti a T_1-et. Végül a T_2 feloldja a B zárját, és ekkor befejeződhet a T_1. □

T_1	T_2
$sl_1(A)$; $r_1(A)$;	
	$sl_2(A)$; $r_2(A)$;
	$sl_2(B)$; $r_2(B)$;
$xl_1(B)$ **Elutasítva**	
	$u_2(A)$; $u_2(B)$;
$xl_1(B)$; $r_1(B)$; $w_1(B)$;	
$u_1(A)$; $u_1(B)$;	

9.15. ábra. *Osztott és kizárólagos zárolást használó ütemezés*

Megjegyezzük, hogy a 9.15. ábrán látható ütemezés eredménye konfliktus-sorbarendezhető. A konfliktusekvivalens soros sorrend (T_2, T_1), hiába kezdődött el a T_1 előbb. Bár itt most nem bizonyítjuk, hogy konzisztens 2FZ tranzakciók jogszerű ütemezése konfliktus-sorbarendezhető, ugyanazok a meggondolások alkalmazhatók az osztott és kizárólagos zárakra is, mint amelyeket a 9.3.4. részben adtunk. A 9.15. ábrán a T_2 előbb old fel zárat, mint a T_1, így azt várjuk, hogy a T_2 megelőzi a T_1-et a soros sorrendben. Ezzel ekvivalensen megvizsgálva a 9.15. ábra olvasási és írási műveleteit, észrevehető, hogy az $r_1(A)$-t a T_2 összes műveletén át hátra tudjuk cserélgetni, amíg a $w_1(B)$-t nem tudjuk az $r_2(B)$ elé vinni, ami pedig szükséges lenne ahhoz, hogy ha a T_1 megelőzze T_2-t egy konfliktusekvivalens soros ütemezésben.

9.4.2. Kompatibilitási mátrixok

Ha több zármódot használunk, akkor az ütemezőnek valamilyen elvre van szüksége ahhoz, hogy mikor engedélyezzen egy zárolási kérést, ha már adva vannak más zárak is azon az adatbáziselemen. Míg az osztott/kizárólagos rendszerek egyszerűek, fogjuk látni, hogy a zárolási módoknak viszonylag összetettebb rendszerei is léteznek a gyakorlatban. A zárolást engedélyező elvek következő fogalmait előbb az egyszerű osztott/kizárólagos rendszerek környezetében vezetjük be.

A *kompatibilitási mátrix* minden egyes zármódhoz rendelkezik egy-egy sorral és egy-egy oszloppal. A sorok egy másik tranzakció által az X elemre már érvényes záraknak felelnek meg, az oszlopok pedig az X-re kért zármódoknak felelnek meg. A kompatibilitási mátrix használatának szabálya a zárolást engedélyező döntésekre az alábbi:

- *C* módú zárat akkor és csak akkor engedélyezhetünk, ha a táblázat minden olyan *R* sorára, amelyre más tranzakció már zárolta az *X*-et *R* módban, a *C* oszlopban „Igen" szerepel.

9.14. példa: A 9.16. ábrán osztott (*S*) és kizárólagos (*X*) zárak kompatibilitási mátrixa látható. Az *S* oszlop azt mondja meg, hogy akkor engedélyezhetünk osztott zárat egy elemre, ha arra az elemre jelenleg is csak osztott zárak vannak. Az *X* oszlop azt mondja meg, hogy csak akkor engedélyezhetünk kizárólagos zárat, ha jelenleg nincs más zár rajta. Megjegyezzük, hogy ezek a szabályok az ütemezések jogszerűségének a definícióját tükrözik erre a zárolási rendszerre. □

		Kért zár	
		S	*X*
Érvényes zár ebben a módban	*S*	Igen	Nem
	X	Nem	Nem

9.16. ábra. *Osztott (S) és kizárólagos (X) zárak kompatibilitási mátrixa*

9.4.3. Zárak felminősítése

Az a *T* tranzakció, amelyik osztott zárat helyez az *X*-re „barátságos" a többi tranzakcióhoz, ugyanis a többinek is lehetősége van az *X*-et a *T*-vel egy időben olvasni. Így kíváncsiak vagyunk arra, vajon még barátságosabb-e az a *T* tranzakció, amelyik beolvasni és új értékkel visszaírni akarja az *X*-et, előbb csak osztott zárat tesz az *X*-re, és később, amikor a *T* már készen áll az új érték beírásával, akkor *felminősíti a zárat kizárólagossá* (upgrade the lock to exclusive) (vagyis később kéri az *X* kizárólagos zárolását azon túl, hogy már osztott zárat tart fenn az *X*-en). Nincs akadálya, hogy a tranzakció ugyanarra az adatbáziselemre újabb különböző zármódú kéréseket adjon ki. Továbbra is fenntartjuk azt a megszokott jelölést, hogy $u_i(X)$ a T_i tranzakció által fennálló összes zárat feloldja az *X*-en, bár be lehetne vezetni zárolási módoktól függő feloldási műveleteket, ha lenne hasznuk.

9.15. példa: A következő példában a T_1 tranzakció a T_2-vel konkurensen tudja végrehajtani a számításait, amely nem lenne lehetséges, ha a T_1 a kezdetben kizárólagosan zárolta volna a *B*-t. A két tranzakció:

T_1: $sl_1(A)$; $r_1(A)$; $sl_1(B)$; $r_1(B)$; $xl_1(B)$; $w_1(B)$; $u_1(A)$; $u_1(B)$;
T_2: $sl_2(A)$; $r_2(A)$; $sl_2(B)$; $r_2(B)$; $u_2(A)$; $u_2(B)$;

Itt a T_1 beolvassa *A*-t és *B*-t, és végrehajtja a (valószínűleg hosszadalmas) számításokat velük, és a legvégén az eredményt beírja a *B* új értékének. Megjegyezzük, hogy a T_1 előbb osztottan zárolja a *B*-t, és később, miután az *A*-val és *B*-vel kapcsolatos számításait befejezte, kér egy kizárólagos zárat a *B*-re. A T_2 tranzakció csak beolvassa az *A*-t és *B*-t, és nem ír rájuk.

A 9.17. ábra a műveletek egy lehetséges ütemezését mutatja. T_2 egy osztott zárat kap a *B*-n, a T_1 előtt, de a negyedik sorban T_1 is képes osztottan zárolni a *B*-t. Így a T_1 rendelkezésére áll az *A* is és a *B* is, és az értékeik felhasználásával végre tudja hajtani a számításokat. Attól kezdve, hogy a T_1 megpróbálja a *B*-n a zárat felminősíteni kizárólagossá, az ütemező a kérést elutasítja, és arra kényszeríti a T_1-et, hogy várjon addig, amíg a T_2 felszabadítja a *B*-n a zárat. Ezután megkapja a T_1 a *B*-n a kizárólagos zárat, beírja *B*-t, és befejeződik a tranzakció.

T_1	T_2
$sl_1(A)$; $r_1(A)$;	
	$sl_2(A)$; $r_2(A)$;
	$sl_2(B)$; $r_2(B)$;
$sl_1(B)$; $r_1(B)$;	
$xl_1(B)$ **Elutasítva**	
	$u_2(A)$; $u_2(B)$;
$xl_1(B)$; $w_1(B)$;	
$u_1(A)$; $u_2(B)$;	

9.17. ábra. *A zárak felminősítésével több konkurens művelet lehet*

Megjegyezzük, hogy ha a T_1 a kezdéskor kért volna kizárólagos zárat a *B*-re, mielőtt beolvasta volna a *B*-t, akkor ezt a kérést az ütemező elutasította volna, ugyanis a T_2-nek már volt egy osztott zárja a *B*-n. A T_1 nem tudta volna elvégezni a számításait a *B* beolvasása nélkül, és így T_1-nek sokkal több dolga lett volna, miután a T_2 felszabadította a zárakat. Végeredményben a T_1 később fejezte volna be, mint most, amikor a felminősítő stratégiát alkalmazta, ha csak kizárólagos zárat használt volna a *B*-n. □

9.16. példa: Sajnos a felminősítés válogatás nélküli alkalmazása a holtpontok új és potenciálisan komoly forrását jelenti. Tételezzük fel, hogy a T_1 is és a T_2 is beolvassa az *A* adatbáziselemet, és egy új értéket ír vissza az *A*-ba. Ha mindkét tranzakció a felminősítéssel dolgozik, akkor előbb osztott zárat kapnak az *A*-ra, és azután minősítik ezt át kizárólagossá, így a 9.18. ábrán javasolt eseménysorozat következhet be, amikor a T_1 és a T_2 közel egyidejűleg kezdődik.

A T_1 és T_2 is kaphat osztott zárat az *A*-ra. Ezután mindkettő megpróbálja ezt felminősíteni kizárólagossá, de az ütemező mindkettőt várakozásra kényszeríti, hiszen a másik már osztottan zárolja az *A*-t. Emiatt egyik végrehajtása sem folytatódhat, vagy

T_1	T_2
$sl_1(A)$;	
	$sl_2(A)$;
$xl_1(A)$ **Elutasítva**	
	$xl_2(A)$ **Elutasítva**

9.18. ábra. *Két tranzakció általi felminősítés holtpontot okozhat*

mindkettőnek örökösen kell várakoznia, vagy addig, amíg a rendszer felfedezi, hogy holtpont alakult ki, abortálja a két tranzakció valamelyikét, és a másiknak engedélyezi az A-ra a kizárólagos zárat. □

9.4.4. Módosítási zárak

El tudjuk kerülni a 9.16. példában vázolt holtpont problémát egy harmadik zárolási mód, az úgynevezett *módosítási zárak* (update locks) használatával. Az $ul_i(X)$ módosítási zár a T_i tranzakciónak csak az X olvasására ad jogot, az X írására nem. Azonban csak a módosítási zárat lehet később felminősíteni. Az olvasási zárat nem lehet felminősíteni. Módosítási zárat akkor is engedélyezhetünk az X-en, amikor az X már osztott módon zárolva van, ha azonban az X-en már van egy módosítási zár, akkor ez megakadályozza, hogy bármilyen más újabb zárat, akár osztott, akár módosítási, akár kizárólagos zárat kapjon az X. Ennek az az oka, hogy ha nem utasítanánk el ezeket az újabb zárolásokat, akkor előfordulhat, hogy a módosításinak soha sem lenne lehetősége kizárólagossá való felminősítésre, ugyanis mindig valamilyen más zár lenne az X-en.

Ez a szabály nem szimmetrikus kompatibilitási mátrixot eredményez, ugyanis az U módosítási zár úgy néz ki, mintha osztott zár lenne, amikor kérjük, és úgy néz ki, mintha kizárólagos zár lenne, amikor már megvan. Emiatt az U és S zárak oszlopai megegyeznek, és az U és X sorai úgyszintén megegyeznek. A mátrixot a 9.19. ábrán láthatjuk.[5]

	S	X	U
S	Igen	Nem	Igen
X	Nem	Nem	Nem
U	Nem	Nem	Nem

9.19. ábra. *Osztott (S), kizárólagos (X) és módosítási (U) zárak kompatibilitási mátrixa*

9.17. példa: A módosítási zárak használata nem befolyásolja a 9.15. példát. A harmadik művelet az lenne, hogy a T_1 módosítási zárat tenne az B-re, nem pedig osztott zárat. A módosítási zárat viszont megkapná, ugyanis csak osztott zárak vannak a B-n, és a 9.17. ábrával megegyező műveletsorozat fordulna elő.

Módosítási zárakkal megszüntethető a 9.16. példában bemutatott probléma. Most mind a T_1, mind a T_2 előbb módosítási zárat kér az A-n, és csak később kizárólagos zárat. A T_1 és T_2 egy lehetséges leírása az alábbi:

T_1: $ul_1(A)$; $r_1(A)$; $xl_1(A)$; $w_1(A)$; $u_1(A)$;
T_2: $ul_2(A)$; $r_2(A)$; $xl_2(B)$; $w_2(A)$; $u_2(A)$;

A 9.18. ábrának megfelelő eseménysorozatot a 9.20. ábrán láthatjuk. Itt a T_2-t el-

[5] Megjegyezzük, hogy az ütemezések jogszerűségével kapcsolatban van egy további feltétel, amely nem tükröződik a mátrixban: egy olyan tranzakciónak, amely az X-en osztott zárat tart fenn, de módosítási zárat nem, nem adható az X-re kizárólagos zár, még akkor sem, ha általában nem tiltjuk a tranzakcióknak, hogy egy elemen több zárat tartsanak fenn.

utasítjuk, amelyik másodikként kérte az A módosítási zárolását. A T_1-nek megengedjük, hogy befejeződjön, és ezután folytatódhat a T_2. A zárolási rendszer hatékonyan megakadályozta a T_1 és T_2 konkurens végrehajtását, ebben a példában viszont lényeges mennyiségű konkurens végrehajtás vagy holtpontot vagy inkonzisztens adatbázis-állapotot eredményez. □

T_1	T_2
$ul_1(A)$; $r_1(A)$;	
	$ul_2(A)$ **Elutasítva**
$xl_1(A)$; $w_1(A)$; $u_1(A)$;	
	$ul_2(A)$; $r_2(A)$;
	$xl_2(A)$; $w_2(A)$; $u_2(A)$;

9.20. ábra. *Helyes végrehajtás a módosítási zárak használatával*

9.4.5. Növelési zárak

Egy másik érdekes zárolási mód, amely bizonyos helyzetekben hasznos lehet, a „növelési zár". Számos tranzakciónak csak az a hatása az adatbázison, hogy növeli vagy csökkenti a tárolt értékeket. Például:

1. Olyan tranzakció, amely pénzt utal át az egyik bankszámláról egy másikra.
2. Olyan tranzakció, amely repülőjegyeket árusít, és csökkenti az adott gépen a szabad ülőhelyek számát.

A növelési műveletek érdekes tulajdonsága, hogy tetszőleges sorrendben kiszámíthatók, ugyanis ha két tranzakció egy-egy konstanst ad hozzá ugyanahhoz az adatbáziselemhez, akkor nem számít, hogy melyiket hajtjuk végre előbb, ahogyan ezt a 9.21. ábrán látható adatbázis-állapot diagram javasolja. Másrészt a növelés nem cserélhető fel sem az olvasással, sem az írással. Ha azelőtt vagy azután olvassuk be az A-t, hogy valaki növelte, különböző értékeket kapunk, és ha azelőtt vagy azután növeljük az A-t, hogy más tranzakció új értéket írt be az A-ba, akkor is különböző értékei lesznek az A-nak az adatbázisban.

Vezessünk be a tranzakciókon mint egy lehetséges műveletet, a *növelési műveletet* (increment action), amelyet `NÖV(A,c)`-vel rövidítünk. Ez a művelet megnöveli c konstanssal az A adatbáziselemet, amelyről feltételezzük, hogy egyszerű szám. Megjegyezzük, hogy c negatív is lehet, ebben az esetben valójában csökkentjük az A értékét. Gyakorlatban alkalmazhatjuk a `NÖV`-öt a sor egy komponensére, annak ellenére, hogy maga a sor, és nem a komponense a zárolható elem.

Formálisan, a `NÖV(A,c)` a következő lépések atomi végrehajtására szolgál: `READ(A, t); t := t+c; WRITE(A,t);`. Nem ismertetjük azt a hardver és/vagy szoftverműködést, amely ezt a műveletet atomivá teszi, csak azt kell megjegyeznünk, hogy az atomiságnak ez az alakja alsóbb szintű, mint a tranzakcióknak a zárolások által támogatott atomisága.

9.21. ábra. *Két növelési művelet kiszámítása, mivel a végső adatbázis-állapot nem függ attól, hogy melyiket hajtottuk előbb végre*

Szükségünk van a növelési műveletnek megfelelő *növelési zárra* (increment lock), amelyet $il_i(X)$-szel fogunk jelölni, mely a T_i tranzakciónak az X-re vonatkozó növelési zárolásra kérése. A $növ_i(X)$ rövidítést arra a műveletre használjuk, amelyben a T_i tranzakció megnöveli az X adatbáziselemet valamely konstanssal. Annak nincs jelentősége, hogy pontosan mi is ez a konstans.

A növelési műveletek és zárak létezése szükségessé teszi, hogy több helyen módosítsuk a konzisztens tranzakciók, konfliktusok és jogszerű ütemezések definícióit. A változtatások az alábbiak:

a) Egy konzisztens tranzakció csak akkor végezheti el az X-en a növelési műveletet, ha egyidejűleg növelési zárat tart fenn az X-en. A növelési zár viszont nem teszi lehetővé sem az olvasási, sem az írási műveleteket.
b) Egy jogszerű ütemezésben bármennyi tranzakció bármikor fenntarthat az X-re növelési zárat. De ha egy tranzakció növelési zárat tart fenn az X-en, akkor egyidejűleg semelyik más tranzakció sem tarthat fenn sem osztott, sem kizárólagos zárat az X-en. Ezeket a követelményeket a kompatibilitási mátrix segítségével fejeztük ki, mely a 9.22. ábrán látható, ahol az I jelenti a növelési módú zárat (I az angol „increment" rövidítése).
c) A $növ_i(X)$ művelet konfliktusban áll az $r_j(X)$-szel, és a $w_j(X)$-szel is, $j \neq i$-re, de nem áll konfliktusban $növ_j(X)$-szel.

	S	X	I
S	Igen	Nem	Nem
X	Nem	Nem	Nem
I	Nem	Nem	Igen

9.22. ábra. *Osztott (S), kizárólagos (X) és növelési (I) zárak kompatibilitási mátrixa*

9.18. példa: Tekintsünk két tranzakciót, mindkettő beolvassa az A adatbáziselemet, és azután növeli a B-t. Lehet, hogy az A-t adják hozzá a B-hez, vagy egy olyan konstanssal növelik a B-t, amelynek a kiszámítása valamilyen más módon függ az A-tól.

T_1: $sl_1(A)$; $r_1(A)$; $il_1(B)$; $növ_1(B)$; $u_1(A)$; $u_1(B)$;
T_2: $sl_2(A)$; $r_2(A)$; $il_2(B)$; $növ_2(B)$; $u_2(A)$; $u_2(B)$;

Megjegyezzük, hogy a tranzakciók konzisztensek, ugyanis csak akkor végeznek növelést, amikor növelési zárral rendelkeznek, és csak akkor olvasnak, amikor osztott zárat tartanak fenn. A 9.23. ábra a T_1 és T_2-nek egy lehetséges ütemezését mutatja. A

T_1	T_2
$sl_1(A)$; $r_1(A)$;	
	$sl_2(A)$; $r_2(A)$;
	$il_2(B)$; $növ_2(B)$;
$il_1(B)$; $növ_1(B)$;	
	$u_2(A)$; $u_2(B)$;
$u_1(A)$; $u_1(B)$;	

9.23. ábra. *Növelési műveletekkel és zárakkal rendelkező tranzakciók ütemezése*

T_1 olvassa először az A-t, azután a T_2 beolvassa az A-t és növeli a B-t. Ezután viszont a T_1-nek is megengedjük, hogy növelési zárat kapjon a B-re, és folytatódjon.

Megjegyezzük, hogy az ütemezőnek a 9.23. ábrán egyik kérést sem kell késleltetnie. Például tételezzük fel, hogy T_1 növeli a B-t A-val, és T_2 növeli a B-t 2A-val. Bármelyik sorrendben végrehajthatjuk, ugyanis az A értéke nem változik, és a növelést is bármely sorrendben elvégezhetjük.

Másképpen kifejezve, megnézhetjük a nem zárolási műveletek sorozatát a 9.23. ábra ütemezésében:

S: $r_1(A)$; $r_2(A)$; $növ_2(B)$; $növ_1(B)$;

Az utolsó műveletet, a $növ_1(B)$-t előrébb tudjuk hozni a második helyre, ugyanis ez nincs konfliktusban ugyanannak az elemnek egy másik növelésével, és biztosan nincs konfliktusban egy másik elem olvasásával. A cseréknek ez a sorozata mutatja, hogy az S konfliktusekvivalens a következő soros ütemezéssel:

$r_1(A)$; $növ_1(B)$; $r_2(A)$; $növ_2(B)$;.

Hasonlóan tudjuk az első műveletet, az $r_1(A)$-t cserékkel a harmadik helyre hátrább vinni, amely azt a soros ütemezést adja, amelyben a T_2 megelőzi T_1-et. □

9.4.6. Feladatok

9.4.1. feladat: A T_1, T_2 és T_3 tranzakciók minden alábbi ütemezésére:

a) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(D)$;
b) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(A)$;
c) $r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(D)$; $w_1(C)$; $w_2(D)$; $w_3(E)$;
* d) $r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(D)$; $w_1(A)$; $w_2(B)$; $w_3(C)$;
e) $r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(A)$; $w_1(A)$; $w_2(B)$; $w_3(C)$;

Végezzük el a következőket:

i) Illesszük be az osztott és a kizárólagos zárakat, és illesszük be a zárak feloldási műveleteit! Helyezzünk osztott zárat közvetlenül minden olyan olvasási művelet elé,

amelyik után nem következik ugyanannak a tranzakciónak ugyanarra az elemre való írási művelete! Helyezzünk kizárólagos zárat minden más olvasási és írási művelet elé! Helyezzük el minden tranzakció végére a szükséges zárfeloldásokat!

ii) Adjuk meg mi történik, amikor minden ütemezést osztott és kizárólagos zárakat támogató ütemező futtat!

iii) Illesszük be az osztott és kizárólagos zárakat oly módon, amely lehetővé teszi a felminősítést. Helyezzünk osztott zárat minden olvasás elé, és kizárólagos zárat minden írás elé, és helyezzük el a szükséges zárfeloldásokat a tranzakciók végére.

iv) Adjuk meg mi történik, amikor az iii)-ból minden ütemezést osztott, kizárólagos zárakat és felminősítést támogató ütemező futtat.

v) Illesszük be az osztott, kizárólagos és módosítási zárakat, a feloldási műveletekkel együtt. Helyezzünk osztott zárat minden olyan olvasási művelet elé, amelyiket nem fogunk felminősíteni, helyezzünk módosítási zárat minden olyan olvasási művelet elé, amelyeket felminősítünk, és helyezzünk kizárólagos zárat minden írási művelet elé! Helyezzük el a zárfeloldásokat a tranzakciók végére, mint rendszerint!

vi) Adjuk meg mi történik, amikor az v)-ből minden ütemezést osztott, kizárólagos és módosítási zárakat támogató ütemező futtat!

! **9.4.2. feladat:** Tekintsük az alábbi két tranzakciót:

T_1: $r_1(A)$; $r_1(B)$; $növ_1(A)$; $növ_1(B)$;
T_2: $r_2(A)$; $r_2(B)$; $növ_2(A)$; $növ_2(B)$;

Válaszoljunk a következőkre:

* a) Ezeknek a tranzakcióknak mennyi átlapolása (ütemezése) sorba rendezhető?

b) Ha T_2-ben megfordítanánk a növelések sorrendjét [vagyis $növ_2(B)$ után következne $növ_2(A)$], akkor mennyi sorba rendezhető ütemezés lenne?

9.4.3. feladat: Az alábbi ütemezések mindegyikére, illesszük be a megfelelő zárolásokat (olvasási, írási vagy növelési) minden művelet elé, és a zárak feloldási műveleteit a tranzakciók végére. Ezután magyarázzuk el mi történik, amikor az ütemezést egy olyan ütemező futtatja, amelyik ezt a három zárolási típust támogatja!

a) $r_1(A)$; $r_2(B)$; $növ_1(B)$; $növ_2(C)$; $w_1(C)$; $w_2(D)$;
b) $r_1(A)$; $r_2(B)$; $növ_1(B)$; $növ_2(A)$; $w_1(C)$; $w_2(D)$;
c) $növ_1(A)$; $növ_2(B)$; $növ_1(B)$; $növ_2(C)$; $w_1(C)$; $w_2(D)$;

9.4.4. feladat: A 9.1.1. feladatban láttunk egy repülőjárat-helyfoglalással kapcsolatos feltételezett tranzakciót. Ha a tranzakciókezelőnek lehetősége lenne osztott, kizárólagos, módosítási és növelési zárakat alkalmaznia, akkor milyen zárat javasolnánk a tranzakció minden egyes lépéséhez?

9.4.5. feladat: Egy konstans tényezővel való szorzási műveletet modellezhetünk egy saját művelettel. Tételezzük fel, hogy `MC(X,c)` a `READ(X,t); t := c*t; WRITE(X,t);` lépéseknek atomi végrehajtását jelenti. Bevezethetünk egy olyan zárolási módot is, amely lehetővé teszi a konstans tényezővel való szorzást.

a) Adjuk meg az olvasási, írási és konstanssal való szorzási zárolások kompatibilitási mátrixát.

! b) Adjuk meg az olvasási, írási, növelési és konstanssal való szorzási zárolások kompatibilitási mátrixát.

! **9.4.6. feladat:** A feladat kedvéért tegyük fel, hogy az adatbáziselemek kétdimenziós vektorok. Négy műveletet tudunk ezekkel a vektorokkal végrehajtani, és mindegyikhez saját típusú zárolás tartozik.

i) Megváltoztatjuk az értékeket az x tengely mentén (X zár).
ii) Megváltoztatjuk az értékeket az y tengely mentén (Y zár).
iii) Megváltoztatjuk a vektor hajlásszögét (A zár, ahol A az angol „angle" rövidítése).
iv) Megváltoztatjuk a vektor nagyságát (M zár, ahol M az angol „magnitude"-ból származik).

Válaszoljunk az alábbi kérdésekre:

* a) Mely műveletpárok felcserélhetők? Például, ha elforgatjuk a vektort úgy, hogy a hajlásszöge 120° legyen, és azután megváltoztatjuk az x koordinátáját 10-re, ez ugyanaz-e, mintha először változtatnánk meg az x koordinátáját 10-re, és azután változtatnánk a hajlásszöget 120°-ra?

b) Az a) válasz alapján mi a négy zártípus kompatibilitási mátrixa?

!! c) Tegyük fel, hogy megváltoztatjuk a négy műveletet úgy, hogy ahelyett, hogy új értéket adnánk egy mértéknek, inkább növeljük a mértéket (pl. „adjunk hozzá 10-et az x koordinátához", vagy „forgassuk el a vektort 30°-kal az óramutató irányába"). Mi lenne ekkor a kompatibilitási mátrix?

! **9.4.7. feladat:** Az alábbi ütemezésből egy művelet hiányzik:

$r_1(A)$; $r_2(B)$; ???; $w_1(C)$; $w_2(A)$;

Az a feladatunk, hogy találjunk bizonyos művelettípusú műveleteket a ??? helyettesítésére, amivel az ütemezés nem lenne sorba rendezhető! Adjuk meg az összes nem sorba rendezhető helyettesítést az alábbi művelettípusok mindegyikére:

* a) olvasási műveletek;
b) írási műveletek;
c) módosítási műveletek;
d) növelési műveletek.

9.5. A zárolási ütemező felépítése

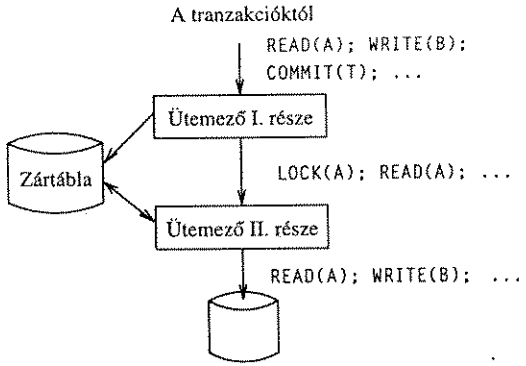
Eddig már számos zárolási sémát láttunk, most a leginkább arra van szükségünk, hogy megnézzük, hogyan működik egy olyan ütemező, amely ezek közül a sémák közül használja valamelyiket. Itt csak néhány megadott elven alapuló egyszerű ütemező felépítését tekintjük:

1. Maguk a tranzakciók nem kérnek zárakat, vagy figyelmen kívül hagyjuk, hogy ezt teszik. Az ütemező feladata, hogy zárolási műveleteket szúrjon be az adatokhoz hozzáférő olvasási, írási illetve egyéb műveletek sorába.
2. Nem a tranzakciók, hanem az ütemező oldja fel a zárakat, mégpedig akkor, amikor a tranzakciókezelő a tranzakció véglegesítésére vagy abortálására készül.

9.5.1. Zárolási műveleteket beszúró ütemező

A 9.24. ábra egy olyan két részből álló ütemezőt mutat be, amely olvasás, írás, végrehajtás, abortálás kéréseket fogad a tranzakcióktól. Az ütemező karbantartja a zártáblát, amelyet bár másodlagosan tárolt adatként ábrázoltunk, lehet, hogy részben vagy egészben a központi memóriában tárolunk. A zártábla által használt központi memória általában nem a lekérdezés-végrehajtás és a naplózás által használt pufferterület része. A zártábla az adatbázis-kezelő rendszernek csak egy komponense, és az operációs rendszer foglal le neki helyet ugyanúgy, mint az adatbázis-kezelő rendszer többi kódjának és belső adatainak.

A tranzakciók által kért műveletek általában az ütemezőn jutnak keresztül, és az adatbázison kerülnek végrehajtásra. Bizonyos körülmények esetén viszont *késleltett* a tranzakció, zárolásra vár, és a kérései (még) nem jutottak el az adatbázishoz. Az ütemező két része a következő műveleteket hajtja végre:



9.24. ábra. *Egy ütemező, amely beszúrja a zárolási kéréseket a tranzakciók kéréseinek sorába*

1. Az I. rész fogadja a tranzakciók által generált kérések sorát, és minden adatbázis-hozzáférési művelet elé, mint az olvasás, írás, növelés vagy a módosítás, beszúrja a megfelelő zárolási műveletet. Az adatbázis-hozzáférési műveleteket ezután átküldi a II. részhez. Az ütemező I. részének kell kiválasztania a megfelelő zárolási módot az ütemező által használt zármódok halmazából.
2. A II. rész fogadja az I. részen keresztül érkező zárolási és adatbázis-hozzáférési műveletek sorozatát, és mindegyiket pontosan végrehajtja. Ha egy zárolási vagy adatbázis-hozzáférési kérés érkezik a II. részhez, eldönti, hogy az igénylő a T tranzakciót késlelteti-e, mivel a zárat nem tudja engedélyezni. Ha így van, akkor magát a műveletet késlelteti, és hozzáadja azoknak a műveleteknek a listájához, amelyeket még végre kell hajtania a T tranzakciónak. Ha a T nem késleltetett (vagyis az összes előzőleg kért zár már korábban engedélyezve van), akkor

   a) Ha a művelet adatbázis-hozzáférés, akkor továbbítja az adatbázishoz, és végrehajtja.
   b) Ha zárolási művelet érkezik a II. részhez, megvizsgálja a zártáblát, hogy vajon a zár engedélyezhető-e.

      i) Ha igen, akkor úgy módosítja a zártáblát, hogy az éppen engedélyezett zárat is tartalmazza.
      ii) Ha nem, akkor egy bejegyzést kell elkészítenie a zártáblában, mely jelzi a zárolási kérést. Az ütemező II. része ezután késlelteti a T tranzakció további műveleteit mindaddig, amíg nem tudja engedélyezni a zárat.

3. Amikor a T tranzakciót véglegesítjük vagy abortáljuk, akkor a tranzakciókezelő értesíti az I. részt, hogy oldja fel az összes T által fenntartott zárat. Ha bármelyik tranzakció várakozik ezen zárak valamelyikére, akkor az I. rész értesíti a II. részt.
4. Amikor a II. rész értesül, hogy valamelyik X adatbáziselemen elérhetővé vált egy zár, akkor eldönti, hogy melyik a következő tranzakció vagy tranzakciók, amelyek megkapják most a zárat az X-re. A tranzakció(k), amely(ek) megkapták a zárat, a késleltetett műveleteik közül annyit végrehajthatnak, amennyit csak végre tudnak hajtani mindaddig, amíg vagy befejeződnek, vagy egy másik zárolási kéréshez érkeznek el, amelyet nem lehet engedélyezni.

9.19. példa: Ha csak egymódú zárak vannak, mint a 9.3. részben, akkor az ütemező I. részének a feladata egyszerű. Ha bármilyen műveletet lát az X adatbáziselemen, és még nem szúrt be zárolási kérést az X-re az adott tranzakcióhoz, akkor beszúrja a kérést. Amikor a tranzakciót véglegesítjük vagy abortáljuk, az I. rész törölheti ezt a tranzakciót, miután feloldotta a zárakat, így az I. részhez igényelt memória nem nő korlátlanul.

Amikor többmódú zárak vannak, az ütemezőnek szüksége lehet arra, hogy azonnal értesüljön, milyen későbbi műveletek fognak előfordulni ugyanazon az adatbáziselemen. Nézzük meg újból az osztott-kizárólagos-módosítási zárak esetét, a 9.15. példa tranzakcióit használva, amelyeket most a zárolások nélkül írunk fel:

T_1: $r_1(A)$; $r_1(B)$; $w_1(B)$;
T_2: $r_2(A)$; $r_2(B)$;

Az ütemező I. részéhez küldött üzenetnek nemcsak az olvasási és írási kéréseket kell tartalmaznia, hanem ugyanazon az elemen bekövetkező későbbi műveletekre vonatkozó jelzést is. Amikor az $r_1(B)$ érkezik be például, az ütemezőnek tudnia kell, hogy lesz-e később $w_1(B)$ művelet (vagy lehet-e ilyen művelet, ha a T_1 tranzakció kódjában elágazás szerepel). Több módon válhat az információ elérhetővé. Például, ha a tranzakció egy lekérdezés, akkor tudjuk, hogy semmit sem fog írni. Ha a tranzakció egy SQL-adatbázist módosítási utasítás, akkor a lekérdező processzor azonnal megadhatja azokat az adatbáziselemeket, melyeket olvashatunk és írhatunk is egyben. Ha a tranzakció beágyazott SQL-program, akkor a fordító hozzá tud férni az összes SQL-utasításhoz (és csakis ezekkel lehet írni az adatbázisba), és meghatározhatja, mely adatbáziselemek esélyesek az írásra.

A példánkban tételezzük fel, hogy a 9.17. ábrán javasolt sorrendben következnek be az események. Ekkor a T_1 először $r_1(A)$-t adja ki. Mivel nincs később kizárólagos zárrá való felminősítés erre a zárra, az ütemező beszúrja az $sl_1(A)$-t az $r_1(A)$ elé. Ezután a T_2 kérései – $r_2(A)$ és $r_2(B)$ – érkeznek az ütemezőhöz. Megint nincs később felminősítés, így az ütemező I. része a következő műveletsorozatot adja ki: $sl_2(A)$; $r_2(A)$; $sl_2(B)$; $r_2(B)$;.

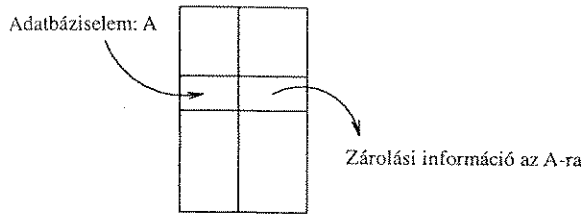
Ezután az $r_1(B)$ művelet érkezik be az ütemezőhöz azzal a figyelmeztetéssel, hogy ezt a zárat fel lehet minősíteni. Az ütemező I. része ekkor kibocsátja $ul_1(B)$; $r_1(B)$-t a II. résznek. Az utóbbi megnézi a zártáblát, és azt találja, hogy a T_1 engedélyezheti a módosítási zárat B-re, ugyanis csak osztott zárak vannak a B-n.

Amikor a $w_1(B)$ művelet beérkezik az ütemezőhöz, az I. rész kibocsátja az $xl_1(B)$; $w_1(B)$-t. A II. rész viszont nem teljesítheti az $xl_1(B)$ kérést, ugyanis a T_2-nek már van osztott zárja a B-n. A T_1-nek ezt és ez utáni minden műveletét késlelteti, egyben a II. rész tárolja a későbbi végrehajtáshoz. Végül a T_2 végrehajtja a véglegesítést, és az I. rész feloldja a zárakat az A-n és a B-n, amelyet a T_2 tartott fenn. Ugyanekkor felfedezi, hogy a T_1 várakozik a B zárolására. Értesíti a II. részt, és ez az $xl_1(B)$ zárolást most már végrehajthatónak találja. Beviszi ezt a zárat a zártáblába, és folytatja a T_1-től tárolt műveletek végrehajtását mindaddig, ameddig tudja. Az esetünkben a T_1 befejeződik. □

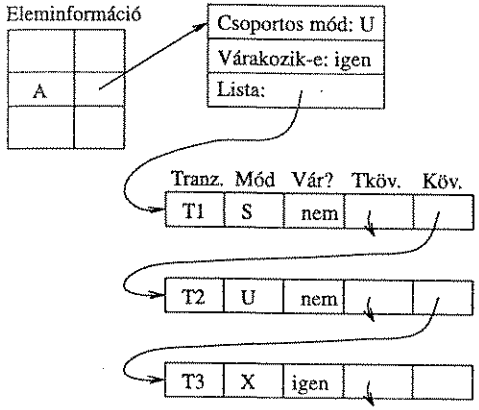
9.5.2. A zártábla

Absztrakt szinten a zártábla egy olyan reláció, amely összekapcsolja az adatbáziselemeket az elemre vonatkozó zárolási információval, mint ahogyan ezt a 9.25. ábra mutatja. A táblát például egy olyan tördelőtáblával lehet megvalósítani, amely az adatbáziselemek (címeit) használja tördelőkulcsként. Azok az elemek, amelyek nincsenek zárolva, nem fordulnak elő a táblában, így a méret csak a zárolt elemek számával arányos, nem pedig a teljes adatbázis méretével.

A 9.26. ábrán egy példát láthatunk, hogy milyen információk találhatók a zártábla-bejegyzésnél. Ez a példaszerkezet feltételezi, hogy az ütemező a 9.4.4. rész osztott-



9.25. ábra. *A zártábla az adatbáziselemekről a zárolási információkra történő leképezés*



9.26. ábra. *Zártábla-bejegyzések szerkezete*

kizárólagos-módosítási zársémáit alkalmazza. Az A-hoz, egy tipikus adatbáziselemhez, a bejegyzés a következő komponensekből álló sor:

1. A *csoportos mód* (group mode) a legszigorúbb feltételek összefoglalása, amivel egy tranzakció szembesül, amikor egy új zárolást kér az A-n. Ahelyett, hogy összehasonlítanánk a zárolási kérést a többi tranzakciónak ugyanazon az elemen fenntartott minden zárolásával, egyszerűsíthetjük az engedélyezési/elutasítási döntést azzal, hogy a kérést csak a csoportos móddal[6] hasonlítjuk össze. Az osztott-kizárólagos-módosítási (rövidítve: *SXU*) zárolási sémákhoz egyszerű a szabály: egy csoportos módban:

[6] A záróláskezelőnek viszont foglalkoznia kell azzal a lehetőséggel, hogy a kérést kiadó tranzakciónak már van egy másik módban zárja ugyanazon az elemen. Például az *SXU* zárolási rendszerre vonatkoztatva, a záróláskezelő elfogadhat egy *X* zár kérést, ha az igénylő tranzakció pont az, amely *U* zárat tart fenn ugyanazon az elemen. Azoknál a rendszereknél, amelyek nem támogatják, hogy egy tranzakció egy elemen több zárat is tartson, a csoportos mód mindig megadja mindazt, amit a záróláskezelőnek tudnia kell.

a) *S* azt jelenti, hogy csak osztott zárak vannak (*S* az angol „shared" rövidítése)
b) *U* azt jelenti, hogy egy módosítási zár van (*U* az angol „update" rövidítése), és lehet még egy vagy több osztott zár is.
c) *X* azt jelenti, hogy csak egy kizárólagos zár van (*X* az angol „eXclusive" szóból származik), és semmilyen más zár nincs.

A többi zárolási sémához is mindig találunk a csoportos mód összegzésének megfelelő rendszert. Ezeket a példákat feladatként javasoljuk elvégezni.

2. A *várakozási bit* (waiting bit) azt adja meg, hogy van-e legalább egy tranzakció, amely az *A* zárolására várakozik.

3. Az összes olyan tranzakciót leíró lista, amelyek vagy jelenleg zárolják az *A*-t, vagy az *A* zárolására várakoznak. Hasznos információk, amelyeket minden listabejegyzés tartalmazhat:

a) A zárolást fenntartó vagy a zárolásra váró tranzakció neve.
b) Ennek a zárnak a módja.
c) A tranzakció fenntartja-e a zárat vagy várakozik-e a zárra.

A 9.26. ábrán két kapcsolást mutatunk minden bejegyzéshez. Az egyik (`Köv`) magukhoz az adatbáziselemre vonatkozó bejegyzésekhez tartozó kapcsolás, a másik pedig (az ábrán `Tköv`) egy bizonyos tranzakció összes bejegyzéséhez kapcsolás. Az utóbbi kapcsolás akkor használható, amikor a tranzakciót véglegesítjük vagy abortáljuk, így könnyen megtalálhatjuk az összes zárat, amelyet fel kell oldanunk.

Zárolási kérések kezelése

Tételezzük fel, hogy a *T* tranzakció zárat kér az *A*-ra. Ha nincs az *A*-ra bejegyzés a zártáblában, akkor biztos, hogy zárak sincsenek az *A*-n, így létrehozhatjuk a bejegyzést, és engedélyezhetjük a kérést. Ha a zártáblában létezik bejegyzés az *A*-ra, akkor ezt felhasználjuk a zárolási kéréssel kapcsolatos döntésünkben. Megkeressük a csoportos módot, amely a 9.26. ábrán az *U*, vagyis „módosítási". Amikor már van módosítási zár egy elemen, akkor semmilyen más zárat nem engedélyezhetünk (kivéve azt az esetet, amikor maga a *T* tartja fenn az *U* zárat, és a többi zárak kompatibilisek *T* kérésével). Tehát a *T*-nek ezt a kérését elutasítjuk, és egy bejegyzést helyezünk el a listában, amely szerint *T* zárat kért (bármilyen módban kérte), és `Vár? = 'igen'`.

Ha a csoportos mód *X*, vagyis kizáró lenne, akkor ugyanez történne. Ha azonban a csoportos mód *S*, vagyis osztott lenne, akkor lehetne adni egy másik osztott vagy módosítási zárat. Ebben az esetben, a *T* bejegyzése a listában `Vár? = 'nem'`, és a csoportos módot az *U*-ra kellene cserélni, ha az új zár módosítási zár, egyébként pedig a csoportos mód az *S* maradna. Akár adtunk engedélyt a zárolásra, akár nem, az új lista bejegyzéshez megfelelő kapcsolat létesül, a `Tköv` és a `Köv` mezőkön keresztül. Megjegyezzük, hogy akár engedélyezzük a zárat, akár nem, a zártáblában a bejegyzés megadja az ütemezőnek azt, amit tudnia kell anélkül, hogy megvizsgálná a zárolások listáját.

Zárfeloldások kezelése

Most tételezzük fel, hogy a *T* tranzakció feloldja *A*-t. Ekkor *T* bejegyzését *A*-ra a listában töröljük. Ha a *T* által fenntartott zár nem egyezik meg a csoportos móddal (pl. *T* egy *S* zárat tart fenn, amíg a csoportos mód *U*), akkor nincs okunk, hogy megváltoztassuk a csoportos módot. Másrészt, ha a *T* által fenntartott zár van a csoportos módban, akkor meg kell vizsgálnunk a teljes listát, hogy megtaláljuk az új csoportos módot. A 9.26. ábrán található példában láttuk, hogy csak egyetlen *U* zár lehet egy elemen, így ha azt a zárat feloldjuk, az új csoportos mód csak az *S* lehetne (ha maradt még osztott zár), vagy semmi (ha nincs más zár jelenleg fenntartva).[7] Ha a csoportos mód *X*, akkor tudjuk, hogy nincsenek más zárolások, és ha a csoportos mód *S*, akkor el kell döntenünk, hogy van-e további osztott zár.

Ha a `Várakozik` értéke `'igen'`, akkor engedélyeznünk kell egy vagy több zárat a kért zárak listájáról. Több különböző megközelítés lehetséges, és mindegyiknek megvan a saját előnye:

1. *Első-beérkezett-első-kiszolgása* (first-come-first-served): Azt a zárolási kérést engedélyezzük, amelyik a legrégebb óta várakozik. Ez a stratégia azt biztosítja, hogy ne legyen kiéheztetés, vagyis a tranzakció ne várjon örökké egy zárra.
2. *Osztott záraknak elsőségadás* (priority to shared locks): Először az összes várakozó osztott zárat engedélyezzük. Ezután egy módosítási zárolást engedélyezünk, ha várakozik ilyen. A kizárólagos zárolást csak akkor engedélyezzük, ha semmilyen más igény nem várakozik. Ez a stratégia csak akkor engedi a kiéheztetést, ha a tranzakció *U* vagy *X* zárolásra vár.
3. *Felminősítésnek elsőségadás* (priority to upgrading): Ha van olyan *U* zárral rendelkező tranzakció, amely *X* zárrá való felminősítésre vár, akkor ezt engedélyezzük előbb. Máskülönben a fent említett stratégiák valamelyikét követjük.

9.5.3. Feladatok

9.5.1. feladat: Melyek a zártáblához a megfelelő csoportos módok, ha az alkalmazott zárolási módok az alábbiak:

a) osztott és kizárólagos zárak;
*! b) osztott, kizárólagos és növelési zárak;
!! c) a 9.4.6. feladatban szereplő zárolási módok.

9.5.2. feladat: A 9.2.4. feladat minden ütemezéséhez adjuk meg azokat a lépéseket, amelyeket ebben a fejezetben leírt zárolási ütemező végezne el.

---

[7] Valójában sohasem találunk „semmi" csoportos módot, ugyanis ha nincs sem zár, sem zárolási kérés egy elemen, akkor nincs bejegyzés sem a tárolási táblában erre az elemre.

9.6. Adatbáziselemekből álló hierarchiák kezelése

Most térjünk vissza a különféle zárolási sémák feltárásához, amelyet a 9.4. részben elkezdtünk. Különösen két olyan problémára összpontosítunk, amelyek akkor merülnek fel, amikor fastruktúra tartozik az adatainkhoz.

1. Az első fajta fastruktúra, amelyet figyelembe veszünk, a zárolható elemek hierarchiája. Ebben a részben megvizsgáljuk, hogyan engedélyezünk zárolást mind a nagy elemekre, pl. relációkra, mind a kisebb elemekre, mint pl. a reláció néhány sorát tartalmazó blokkokra vagy egyedi sorokra.
2. A másik lényeges hierarchiafajtát képezik a konkurenciavezérlési rendszerekben azok az adatok, amelyek önmagukban faszervezésűek. Jelentősebb példa a B-fa-indexek. A B-fák csomópontjait adatbáziselemeknek tekinthetjük, viszont ha így tekintjük, mint ahogyan azt a 9.7. részben látni fogjuk, az eddig tanult zárolási sémákat szegényesen használhatjuk, emiatt egy új megközelítésre van szükségünk.

9.6.1. Többszörös szemcsézettségű zárak

Emlékezzünk vissza, hogy az „adatbáziselem" kifejezést szándékosan definiálatlanul hagytuk, ugyanis a különböző rendszerek különböző méretű adatbáziselemeket zárolnak, mint pl. sorokat, lapokat vagy blokkokat, relációkat. Bizonyos alkalmazásoknál a kis adatbáziselemek előnyösek, mint amilyen a sorok, amíg másoknál a nagy elemek nyújtják a legtöbbet.

9.20. példa: Tekintsünk egy banki adatbázist. Ha a relációkat kezeljük adatbáziselemekként, akkor így csak egy zárat tudunk kiadni arra a teljes relációra, amely a számlák egyenlegét adja meg, ezért a rendszer nagyon kis konkurenciát engedélyezne. Mivel a legtöbb tranzakció a számla egyenlegét változtatja, vagy pozitívan vagy negatívan, a legtöbb tranzakciónak kizárólagosan kellene zárolnia a számlaegyenlegek relációt. Így csak egyetlen befizetést vagy kivételt tudnánk egyidejűleg elvégezni, nem számítana, hogy hány olyan processzor lenne, amely alkalmas lenne ezeknek a tranzakcióknak az elvégzésére. Jobb megközelítés, hogy egyedi oldalakat vagy adatblokkokat zárolunk. Így két olyan számla, amelynek a sorai különböző blokkban vannak, egyidejűleg módosítható. Ez biztosítja szinte a teljes konkurenciát, amely elérhető a rendszerben. A másik véglet az lenne, ha minden egyes sorra biztosítanánk zárolást, így bármilyen számlahalmazt egyszerre tudnánk módosítani, de a záraknak ennyire finom szemcsézettsége valószínűleg nem érné meg a sok fáradságot.

Ellentétes esetben, tekintsünk egy dokumentumokból álló adatbázist. Ezeket a dokumentumokat időnként szerkeszteni szokták, és a legtöbb tranzakció teljes dokumentumokhoz fér hozzá. Az adatbáziselem ésszerű megválasztása ekkor a teljes dokumentum. Mivel a legtöbb tranzakció *csak olvasási tranzakció* (vagyis nem végez írási műveletet), a zárolás csak azért szükséges, hogy elkerüljük a szerkesztés közben a dokumentumok olvasását. Ha kisebb szemcsézettségű elemeket zárolnánk, mint például paragrafusokat, mondatokat vagy szavakat, akkor ennek semmilyen előnyét sem látnánk, viszont sokkal költségesebb lenne. Az egyetlen tevékenység, amelyet a kisebb szemcsézettségű zárak támogatnának az, hogy a dokumentum egy részét tudnánk olvasni a dokumentum szerkesztése közben is. □

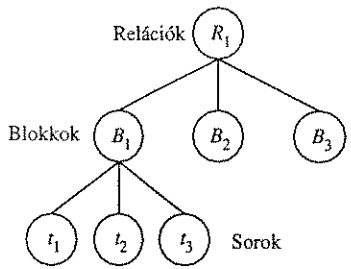
Bizonyos alkalmazások mind a nagy, mind a kis szemcsézettségű zárakat is tudják alkalmazni. Például a 9.20. példában tárgyalt banki adatbázisnál világos, hogy blokk- vagy sorszintű zárolás is szükséges, de néhány esetben a teljes számlareláció zárolása is szükséges lehet, annak érdekében, hogy ellenőrizzük a számlákat (pl. ellenőrizzük, hogy helyesek-e a számlaösszegek). De ha osztott zárat teszünk a számlarelációra annak érdekében, hogy kiszámoljunk a reláción valamilyen csoportfüggvényt, és egyidejűleg az egyéni számlák soraihoz kizárólagos zárat adunk, ez könnyen nem sorba rendezhető viselkedéshez vezethet, ugyanis a reláció valójában megváltozik, amíg egy feltehetően befagyasztott másolatát olvassuk a csoportfüggvényes lekérdezéshez.

9.6.2. A figyelmeztető zárak

A probléma megoldását, hogy hogyan kezeljük az újfajta zárolással kapcsolatos különféle szemcsézettségeken levő zárakat, „figyelmeztetés" nevű új zárat vezetünk be. Ezek a zárak akkor hasznosak, amikor adatbáziselemek beágyazott vagy hierarchikus struktúrákat mutatnak, mint azt a 9.27. ábrán láthatjuk. Itt az adatbáziselemek három szintjét figyelhetjük meg:

1. a relációk a legnagyobb zárolható elemek;
2. minden reláció egy vagy több blokkból vagy lapból épül fel, amelyekben a soraik vannak;
3. minden blokk egy vagy több sort tartalmaz.

Az adatbáziselemek hierarchiáján a zárak kezelésére szolgáló szabályok alkotják a *figyelmeztető protokollt* (warning protocol), amely tartalmazza mind a „közönséges" zárakat, mind a „figyelmeztető" zárakat. A zárolási sémát úgy adjuk meg, hogy a közönséges zárak *S* és *X* (osztott és kizárólagos). A figyelmeztető zárakat a közönséges



9.27. ábra. *Hierarchikusan szervezett adatbáziselemek*

zárak elé helyezett *I* előtaggal jelöljük (az angol „intention to" = szándékszik rövidítése). Például *IS* azt jelenti, hogy szándékunkban áll osztott zárat kapni egy részelemen. A figyelmeztető protokoll szabályai:

1. Ahhoz, hogy elhelyezzünk egy közönséges *S* vagy *X* zárat valamely elemen, a hierarchia gyökerénél kell kezdenünk.
2. Ha már annál az elemnél tartunk, amelyet zárolni akarunk, akkor nem kell tovább folytatnunk, hanem kérjük az *S* vagy *X* zárolást arra az elemre.
3. Ha az elem, amelyet zárolni szeretnénk, lejjebb van a hierarchiában, akkor elhelyezünk egy figyelmeztetést ezen a csomóponton. Vagyis ha osztott zárat szeretnénk kérni egy részelemen, akkor ebben a csomópontban egy *IS* zárat kérünk. Ha kizárólagos zárat akarunk egy részelemen kérni, akkor ebben a csomópontban egy *IX* zárat kérünk. Amikor a jelenlegi csomópontban levő zárat megkaptuk, akkor az ehhez a csomóponthoz tartozó utódcsomóponttal folytatjuk (azzal, amelyikhez tartozó részfa tartalmazza azt a csomópontot, amelyet zárolni kívánunk). Ezután megfelelően a 2. vagy 3. lépéssel folytatjuk mindaddig, amíg elérjük a keresett csomópontot.

Ahhoz, hogy eldöntsük, engedélyezhetjük-e ezek közül a zárak közül valamelyiket vagy sem, a 9.28. ábrán található kompatibilitási mátrixot használjuk. Ennek a mátrixnak az értelmezéséhez először nézzük meg az *IS* oszlopot. Ha *IS* zárat kérünk egy *N* csomópontban, az *N* egy leszármazottját szándékozzuk olvasni. Ez a szándék csak abban az esetben okozhat problémát, ha már egy másik tranzakció korábban jogosulttá vált arra, hogy az *N* által reprezentált teljes adatbáziselemről egy új példányt készítsen, emiatt „Nem" található az *X*-hez tartozó sorban. Megjegyezzük, hogy ha más tranzakció azt tervezi, hogy csak egy részelemét írja, ezt az *N*-en *IX* zárral jelölve meg, akkor lehetőségünk van arra, hogy engedélyezzük az *IS* zárat az *N*-en, és a konfliktust alsóbb szinten oldhatjuk meg, ha valóban az írási szándék és az olvasási szándék egy közös elemhez kapcsolódik.

	IS	*IX*	*S*	*X*
IS	Igen	Igen	Igen	Nem
IX	Igen	Igen	Nem	Nem
S	Igen	Nem	Igen	Nem
X	Nem	Nem	Nem	Nem

9.28. ábra. *Osztott (S), kizárólagos (X), és szándékjelölő (I előtaggal jelölt) zárak kompatibilitási mátrixa*

Most tekintsük az *IX*-hez tartozó oszlopot. Ha az *N* csomópont egy részelemét szándékozzuk írni, akkor meg kell akadályoznunk az *N* által képviselt teljes elem olvasását vagy írását. Ekkor „Nem"-et látunk az *S* és *X* zármódok bejegyzéseiben. Azonban az *IS* oszloppal kapcsolatban leírtaknak megfelelően, más tranzakció, amely egy részelemet olvas vagy ír, a potenciális konfliktusokat az adott szinten kezeli le, így az *IX* nincs konfliktusban egy másik *IX*-szel vagy egy *IS*-sel az *N*-en.

Ezután nézzük az *S*-hez tartozó oszlopot. Az *N* csomópontnak megfeleltetett elem olvasása nincs konfliktusban sem egy másik olvasási zárral az *N*-en, sem egy olvasási zárral az *N* egy részelemén, amelyet az *N*-en *IS* reprezentál. Emiatt „Igen"-t találunk az *S* és az *IS* soraiban is. Azonban egy *X* vagy egy *IX* azt jelenti, hogy más tranzakció írni fogja legalábbis egy részét az *N* által reprezentált elemnek. Ezért nem tudjuk engedélyezni a teljes olvasását az *N*-nek, amelyet a „Nem" bejegyzés fejez ki az *S* oszlopban.

Végül a *X* oszlopban csak „Nem" bejegyzések vannak. Nem tudjuk megengedni az *N* csomópont egyik részének írását sem, ha más tranzakciónak már joga van olvasni vagy írni az *N*-et, vagy arra, hogy megszerezze ezt a jogot az *N* egy részelemére.

9.21. példa: Tekintsük a következő relációt

```
Film(filmCím, év, hossz, stúdióNév)
```

Tételezzük fel, hogy a teljes relációra és az egyedi sorokra követelünk zárolást. Ekkor a T_1 tranzakció, amely az alábbi kérdést tartalmazza:

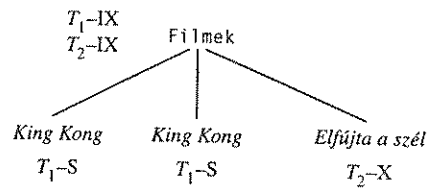
```
SELECT *
FROM Film
WHERE filmCím = 'King Kong';
```

azzal kezdődik, hogy *IS* módon zárolja a teljes relációt. Ezután veszi az egyedi sorokat (két film szerepel a *King Kong* filmcímmel), és *S* módú zárolást ad ki ezekre.

Most tételezzük fel, hogy mialatt az első lekérdezést végezzük, elkezdődik a T_2 tranzakció, amely a sorok év komponensét változtatja meg:

```
UPDATE Film
SET év = 1939
WHERE filmCím = 'Elfújta a szél';
```

Ekkor a T_2-nek szüksége van a reláció *IX* módú zárolására, ugyanis azt tervezi, hogy új értéket ír be az egyik sorba. Ez kompatibilis a T_1-nek a relációra vonatkozó *IS* zárolásával, így a zárat engedélyezzük. Amikor a T_2 elérkezik az *Elfújta a szél*hez tartozó sorhoz, ezen a soron nem talál zárat, így megkapja az *X* módú zárat, és átírja a sort. Ha a T_2 a *King Kong* filmek valamelyikéhez próbált volna új értéket beírni, akkor várnia kellett volna, amíg a T_1 felszabadítja az *S* zárakat, ugyanis az *S* és *X* nem kompatibilisek. A 9.29. ábrán láthatjuk a zárak kollekcióját. □



9.29. ábra. *Engedélyezett zárak a* `Film` *soraihoz hozzáférő két tranzakcióhoz*

Csoportos mód a szándékzárolásokhoz

A 9.28. ábrán szereplő kompatibilitási mátrix olyan helyzetet mutat be, amelyet eddig még nem láttunk a zármódok erejét illetően. A korábbi zárolási sémákban, valahányszor lehetőségünk volt arra, hogy egy adatbáziselemet egyszerre kétféleképpen, *M* és *N* módban is zároljunk, ezek közül az egyik *dominánsabb* volt a másiknál, mégpedig abban az értelemben, hogy amikor az egyik mód sorában és oszlopában minden olyan pozícióban „Nem" áll, amelyben a másik mód sorában vagy oszlopában a „Nem" áll. Például a 9.19. ábrán látjuk, hogy az *U* dominánsabb az *S*-nél, és az *X* dominánsabb az *S*-nél is és az *U*-nál is. Egy előnye annak, hogy tudjuk, mindig van egy domináns zár egy elemen az, hogy több zárolás hatását össze tudjuk foglalni egy „csoportos mód"-dal, amint azt a 9.5.2. részben tárgyaltuk.

Amint a 9.28. ábrán látjuk, az *S* és *IX* módok közül egyik sem dominánsabb a másiknál. Továbbá egy elemet *S* és *IX* módok mindkettőjében zárolhatunk egyidejűleg, feltéve, hogy ugyanaz a tranzakció kérte a zárolást (vigyázzunk, hogy a „nem" bejegyzések a kompatibilitási mátrixban csak azokra a zárakra alkalmazhatók, amelyeket *más* tranzakciók tartanak fenn). Egy tranzakció mindkét zárolást kérheti, ha egy teljes elemet akar beolvasni, és azután a részelemek csak kis részhalmazát akarja írni. Ha egy tranzakciónak *S* és *IX* zárolásai is vannak egy elemen, akkor ez korlátozza a többi tranzakciót olyan mértékben, ahogy bármelyik zár teszi. Vagyis elképzelhetünk egy másik *SIX* zárolási módot, amelynek a sorai és oszlopai a „Nem"-et tartalmazzák az *IS* bejegyzés kivételével mindenhol. Az *SIX* zárolási mód csoportmódként szolgál, ha van olyan tranzakció, amelynek van *S*, illetve *IX* módú, de nincs *X* módú zárolása.

Elképzelhetjük ugyanezt a helyzetet a 9.22. ábrán levő mátrixnál a növelési zárolásokra. Vagyis egy tranzakció az *S* és az *I* módokban is fenntarthatna zárakat. Ez a helyzet viszont ekvivalens az *X* módú zárolással, így ekkor az *X*-et csoportos módként használhatnánk.

9.6.3. Fantomok és a beszúrások helyes kezelése

Amikor a tranzakciók egy zárolható elem új részelemeit hozzák létre, néhány kedvező lehetőség rosszra fordulhat. Az a probléma, hogy csak létező egyedeket tudunk zárolni. Nem könnyű olyan adatbáziselemeket zárolni, amelyek nem léteznek, de később beszúrhatók. A következő példával világítjuk meg ezt az esetet.

9.22. példa: Tegyük fel, hogy ugyanaz a `Film` relációnk van, mint a 9.21. példában, és az első tranzakció, amelyet végrehajtunk a T_3, amely az alábbi lekérdezés:

```
SELECT SUM(hossz)
FROM Film
WHERE stúdióNév = 'Disney';
```


T_3-nak be kell olvasnia az összes Disney-filmről a sorokat, így azzal kezdődhet, hogy a relációt *IS* zárolja, és a Disney-filmekhez tartozó minden sort *S* zárolja[8].

Most egy T_4 tranzakció is megjelenik, és beszúr egy új Disney-filmet. Úgy tűnik, hogy a T_4-nek nincs szüksége zárolásokra, de a T_3 eredményét helytelenné változtatja. Ez a tény önmagában nem konkurenciaprobléma, ugyanis a (T_3, T_4) soros sorrend azzal ekvivalens, ami valójában történt. Lehetne még más *X* elem is, amelyet a T_3 és a T_4 is úgy ír, hogy a T_4 írja előbb, és így az összetettebb tranzakcióknak *lehetne* nem sorba rendezhető viselkedése.

Pontosabban kifejezve, tegyük fel, hogy D_1 és D_2 korábban létező Disney-filmek, és D_3 a T_4 által beszúrt új Disney-film. Legyen *L* a T_3 által kiszámolt Disney-filmek hosszainak az összege, és legyen az a konzisztenciamegszorítás az adatbázison, hogy az *L*-nek egyenlőnek kell lennie azon a Disney-filmek hosszának az összegével, amelyek léteztek, amikor az *L*-t utoljára kiszámoltuk. Ekkor a figyelmeztetési protokoll alatt jogszerű az alábbi eseménysorozat:

$r_3(D_1)$; $r_3(D_2)$; $w_4(D_3)$; $w_4(X)$; $w_3(L)$; $w_3(X)$;

Itt $w_4(D_3)$-et használtuk arra, hogy a T_4 tranzakció létrehozza a D_3-at. A fenti ütemezés nem sorba rendezhető. Ténylegesen az *L* értéke nem a D_1, D_2 és D_3 hosszának az összege, amelyek a jelenleg létező Disney-filmek. Továbbá az a tény, hogy *X* értékét a T_3 írta, és nem a T_4, kizárja azt a lehetőséget, hogy a T_3 a T_4 előtt következzen a feltételezett ekvivalens soros elrendezésben. □

A 9.22. példában az a probléma, hogy az új Disney-filmnek van egy *fantom* (phantom) sora, amelyet zárolni kellett volna, de nem tettük meg, ugyanis még nem létezett akkor, amikor a zárolásokat elvégeztük. Mégis van egy egyszerű út, hogy elkerüljük a fantomokat. A sorok beszúrását és törlését az egész relációra vonatkozó írásként kell tekintenünk. Így a T_4 tranzakciónak a 9.22. példában meg kell kapnia az *X* zárat a `Film` reláción. Minthogy a T_3 már *IS* módban zárolta ezt a relációt, és az a mód nem kompatibilis az *X* móddal, a T_4-nek várnia kell, amíg a T_3 befejeződik.

9.6.4. Feladatok

9.6.1. feladat: Tekintsünk a változatosság kedvéért egy objektumorientált adatbázist. A *C* osztály objektumait két blokkban tároljuk, a B_1-ben és a B_2-ben. A B_1 tartalmazza az O_1 és O_2 objektumokat, míg a B_2 tartalmazza az O_3, O_4 és O_5 objektumokat. Az osztálykiterjedések, a blokkok és az objektumok zárolható adatbáziselemekből álló hierarchiát alkotnak. Adjuk meg a zárolási kérések sorozatát és a figyelmeztető protokoll alapú ütemező feladatát az alábbi kérési sorozatokhoz. Feltehetjük, hogy minden kérés éppen azelőtt fordul elő, mint amikor szükségünk van rá, és minden zárfeloldás a tranzakció befejeztével történik.

[8] Ha viszont sok Disney-film lenne, akkor hatékonyabb lehetne csak egy *S* zárat kérni a teljes relációra.

* a) $r_1(O_1)$; $w_2(O_2)$; $r_2(O_3)$; $w_1(O_4)$;
b) $r_1(O_5)$; $w_2(O_5)$; $r_2(O_3)$; $w_1(O_4)$;
c) $r_1(O_1)$; $r_1(O_3)$; $r_2(O_1)$; $w_2(O_4)$; $w_2(O_5)$;
d) $r_1(O_1)$; $r_2(O_2)$; $r_3(O_1)$; $w_1(O_3)$; $w_2(O_4)$;$w_3(O_5)$; $w_1(O_2)$;

9.6.2. feladat: Változtassuk meg a 9.22. példában található eseménysorozatot úgy, hogy a $w_4(D_3)$ művelet a teljes `Film` relációnak a T_4 általi írása legyen. Ezután mutassunk be egy figyelmeztető protokoll alapú ütemező működést ezen a kérési sorozaton!

!! **9.6.3. feladat:** Mutassuk be, hogyan adjuk hozzá a növelési zárakat a figyelmeztető protokoll alapú ütemezőhöz!

9.7. Faprotokoll

Ebben a fejezetben az elemekből álló fákkal kapcsolatosan egy másik problémát tekintünk. A 9.6. részben a beágyazott szerkezetű adatbáziselemekkel létrehozott fákkal foglalkoztunk, amelyben a gyerekek a szülők részei voltak. Most maguknak az elemeknek a kapcsolati sémájából álló fastruktúrákkal foglalkozunk. Az adatbáziselemek diszjunkt adatdarabok, azonban csak egyféleképpen, a szülőkön keresztül lehet elérni egy csomópontot. A B-fák az ilyen típusú adatoknak fontos példái. Tudjuk, hogy csak egy bizonyos útvonalon jutunk el egy elemhez, és ez lényeges szabadságot ad nekünk abban, hogy a már látott kétfázisú zárolási megközelítéstől eltérő módon kezeljük a zárakat.

9.7.1. Fa alapú zárolások idítékai

Tekintsünk egy B-fa-indexet olyan rendszerben, amely az egyedi csomópontokat (vagyis blokkokat) zárolható adatbáziselemekként kezeli. A csomópont a zárolás szemcsézettségének a megfelelő szintje, ugyanis nem előnyös, ha kisebb darabokat kezelünk elemekként. Ha pedig a teljes B-fát kezeljük adatbáziselemként, akkor ez megakadályozza az index olyan konkurens használatát, mint amilyen elérhető a 9.7. rész tárgyát alkotó működési mechanizmus által.

Ha a zármódoknak egy szabványos halmazát használjuk, mint az osztott, kizárólagos és módosítási zárak, valamint használjuk a kétfázisú zárolást, akkor a B-fa konkurens használata szinte lehetetlen. Ennek az az oka, hogy az indexet használó minden tranzakciónak a B-fa gyökércsomópontját kell először zárolnia. Ha a tranzakció 2FZ, akkor nem lehet addig feloldani a gyökéren a zárolást, amíg meg nem szerezte az összes zárat, amelyre szüksége van, mind a B-fa-csomópontokon mind pedig más adatbáziselemeken.[9] Továbbá, mivel elvben bármely tranzakció, amely beszúrásokat vagy törléseket végez, a B-fa gyökerének az átírásával fejeződhet be, ily módon a tranzakciónak legalább egy módosítási zárolásra szüksége van a gyökércsomóponton, vagy kizárólagos zárra van szüksége, ha a módosítási mód nem elérhető. Így csak egyetlen nem csak olvasási tranzakció férhet hozzá bármikor a B-fához.

[9] Ezenkívül jó oka van annak, amiért a tranzakciók minden zárat addig tartanak, amíg készen nem állnak a véglegesítésre. Lásd a 10.1. részt.

Mégis az esetek többségében majdnem közvetlenül levezethetjük, hogy egy B-fa csomópontját nem kell átírni, még akkor sem, ha a tranzakció beszúr vagy töröl egy sort. Például, ha a tranzakció beszúr egy sort, de a gyökérnek az a gyereke, amelyhez hozzáférünk, nincs teljesen tele, akkor tudjuk, hogy a beszúrás nem kerül fel a gyökérig. Hasonlóan, ha a tranzakció egyetlen sort töröl, és a gyökérnek abban a gyerekében, amelyhez hozzáfértünk, a minimum számnál több kulcs és mutató van, akkor biztosak lehetünk, hogy a gyökér nem változik meg.

Így, amikor a tranzakció a gyökérnek egyik gyereke felé irányul, és észleli azt a (teljesen szokványos) helyzetet, ami kizárja a gyökér átírását, azonnal szeretnénk feloldani a gyökéren a zárat. Ugyanezt a megfigyelést alkalmazhatjuk a B-fa bármely belső csomópontjának a zárolására is, bár a konkurens B-fánál a legtöbb lehetőség abból származik, hogy a gyökéren a zárat korán oldjuk fel. Sajnos, a gyökéren levő zárolás korai feloldása ellentmond a 2FZ-nek, így nem lehetünk biztosak abban, hogy a B-fához hozzáférő több tranzakciónak az ütemezése sorba rendezhető lesz. A megoldás egy speciális protokoll a B-fákhoz hasonló fastruktúrájú adatokhoz hozzáférő tranzakciók részére. A protokoll ellentmond a 2FZ-nek, de azt a tényt használja, hogy az elemekhez való hozzáférés lefelé halad a fán, a sorbarendezhetőség biztosítása érdekében.

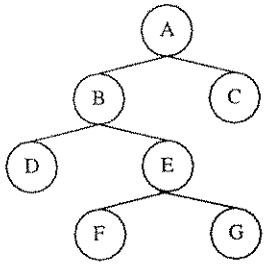
9.7.2. Faszerkezetű adatok hozzáférési szabályai

Az alábbi megszorítások a zárakon a *faprotokollt* (tree protocol) adják. Tételezzük fel, hogy csak egyféle zár van, amelyet az $l_i(X)$ alakú zárolási kérésekkel ábrázolunk, de ezt az ötletet bármely zárolási módokból álló halmazra általánosíthatjuk. Tételezzük fel, hogy a tranzakciók konzisztensek, az ütemezéseknek jogszerűnek kell lenniük (vagyis az ütemező csak akkor adja meg az elvárt megszorításokat a zárak engedélyezésével, amikor nincs konfliktusban azokkal a zárakkal, amelyek már a csomóponton vannak), és ugyanakkor nincs kétfázisú zárolási követelmény a tranzakciókon.

1. Egy tranzakciónak az első zárja a fa bármely csomópontján lehet.[10]
2. Rákövetkező zárakat csak akkor lehet szerezni, ha a tranzakciónak jelenleg van zárja a szülő csomóponton.
3. A csomópontok zárját bármikor feloldhatjuk.
4. Egy tranzakció nem zárolhatja újból azt a csomópontot, amelyen feloldotta a zárat, még akkor sem, ha még tartja a csomópont szülőjén a zárat.

9.23. példa: A 9.30. ábra a csomópontok hierarchiáját, míg a 9.31. ábra ezeken az adatokon három tranzakció műveleteit mutatja. T_1 az A gyökéren kezdődik, és lefelé folytatódik B, C és D felé. T_2 a B-n kezdődik, és az E felé próbál haladni, de először

[10] A 9.7.1. rész B-fa példájában az első zárnak mindig a gyökéren kell lennie.



9.30. ábra. *Zárolható elemekből álló fa*

T_1	T_2	T_3
$l_1(A)$; $r_1(A)$;		
$l_1(B)$; $r_1(B)$;		
$l_1(C)$; $r_1(C)$;		
$w_1(A)$; $u_1(A)$;		
$l_1(D)$; $r_1(D)$;		
$w_1(B)$; $u_1(B)$;		
	$l_2(B)$; $r_2(B)$;	
		$l_3(E)$; $r_3(E)$;
$w_1(D)$; $u_1(D)$;		
$w_1(C)$; $u_1(C)$;		
	$l_2(E)$; **Elutasítva**	
		$l_3(F)$; $r_3(F)$;
		$w_3(F)$; $u_3(F)$;
		$l_3(G)$; $r_3(G)$;
		$w_3(E)$; $u_3(E)$;
	$l_2(E)$; $r_2(E)$;	
		$w_3(G)$; $u_3(G)$;
	$w_2(B)$; $u_2(B)$;	
	$w_2(E)$; $u_2(E)$;	

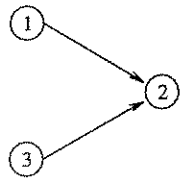
9.31. ábra. *A faprotokollt követő három tranzakció*

elutasítjuk, ugyanis már a T_3-nak van zárja az E-n. A T_3 tranzakció az E-n kezdődik, és folytatja az F-fel és G-vel. Megjegyezzük, hogy a T_1 nem 2FZ tranzakció, ugyanis az A-n előbb töröljük a zárat, mielőtt megszerezzük a zárat a D-n. Hasonlóan a T_3 sem 2FZ tranzakció, de a T_2 véletlenül éppen 2FZ. □

9.7.3. Miért működik a faprotokoll?

A faprotokoll az ütemezésben részt vevő tranzakciókon egy soros sorrendet kényszerít ki. A következőképpen definiálhatjuk a megelőzési sorrendet. Azt mondjuk, hogy $T_i <_S T_j$ az S ütemezésben, ha a T_i és T_j tranzakciók egyrészt közösen zárolnak egy csomópontot, másrészt a T_i zárolja a csomópontot először.

9.24. példa: A 9.31. ábra S ütemezésében a T_1 és T_2 közösen zárolják a B-t, és a T_1 zárolja először. Így $T_1 <_S T_2$. Azt találjuk még, hogy T_2 és T_3 közösen zárolják az E-t, és a T_3 zárolja először, így $T_3 <_S T_2$. A T_1 és T_3 között viszont nincs megelőzés, hiszen nincs olyan csomópont, amelyet közösen zárolnak. Így ezekből a megelőzési relációkból levezetett megelőzési gráf a 9.32. ábrán látható. □



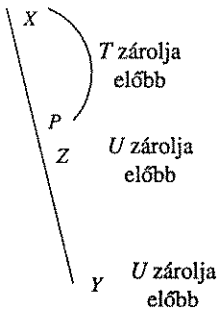
9.32. ábra. *A 9.31. ábra ütemezéséből származó megelőzési gráf*

Ha a fent definiált megelőzési relációkból rajzolt megelőzési gráf nem tartalmaz kört, akkor azt állítjuk, hogy a tranzakciók bármely topologikus sorrendje egy ekvivalens soros ütemezés. Például a 9.31. ábrához vagy a (T_1, T_3, T_2) vagy a (T_3, T_1, T_2) az ekvivalens soros ütemezés. Ennek az az oka, hogy az ilyen soros ütemezésben minden csomóponthoz ugyanabban a sorrendben nyúlnak a tranzakciók, mint az eredeti ütemezésben.

Ahhoz, hogy megértsük a fent leírt megelőzési gráfnak miért kell körmentesnek lennie, először vegyük észre a következőt:

- Ha két tranzakció közösen zárol néhány elemet, akkor ugyanabban a sorrendben zárolják mindegyiket.

Tekintsünk valamilyen T és U tranzakciókat, amelyek két vagy több elemet közösen zárolnak. Először, megjegyezzük, hogy mindegyik tranzakció faformájú halmazát zárolja az elemeknek, és a két fa metszete maga is fa. Emiatt van egy legmagasabb X



9.33. ábra. *Két tranzakció által közösen zárolt elemek útja*

elem, amelyet a T is és az U is zárol. Tételezzük fel, hogy T zárolja az X-et először, de van egy másik Y elem, amelyet az U előbb zárol, mint a T. Ekkor az elemekből álló fában van út az X-től az Y-ba, és a T-nek is és az U-nak is zárolnia kell minden elemet az út mentén, ugyanis egyik sem zárolhat úgy egy csomópontot, hogy ne lenne már ennek a szülőjén zárja.

Tekintsük az első elemet az út mentén, mondjuk legyen Z, amelyet az U zárol először, mint azt a 9.33. ábrán látjuk. Ekkor T előbb zárolja Z-nek a P szülőjét, mint az U. Ekkor viszont a T még mindig tartja a zárolást P-n, amikor zárolja Z-t, így U még nem zárolta P-t, amikor a Z-t zárolja. Az nem lehet, hogy Z lenne az első elem, amelyet az U a T-vel közösen zárol, mivel mindkettő zárolta az ősét, X-et (amely lehet a P is, csak a Z nem). Így az U addig nem zárolhatja a Z-t, amíg meg nem szerezte a P-n a zárat, amely azután van, hogy a T zárolta a Z-t. Arra következtetünk, hogy a T megelőzi az U-t minden csomópontban, amelyet közösen zárolnak.

Most tekintsük a $T_1, T_2, \ldots, T_n$ tranzakciók tetszőleges halmazát, amely eleget tesz a faprotokollnak, és az S ütemezésnek megfelelően zárolja a fa valamely csomópontjait. Először azok a tranzakciók, amelyek zárolják a gyökeret, ezt valamilyen sorrendben végzik, és olyan szabály alapján, amelyet éppen megfigyeltünk:

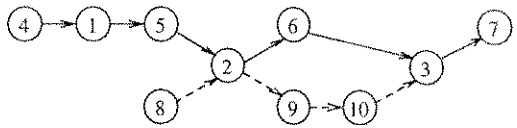
- Ha a T_i előbb zárolja a gyökeret, mint a T_j, akkor a T_i minden T_j-vel közös $T_i <_S T_j$ csomópontot előbb zárol, mint a T_j. Vagyis $T_i <_S T_j$, de nem $T_j <_S T_i$.

A fa csomópontjainak a száma szerinti teljes indukcióval megmutathatjuk, hogy a teljes tranzakcióhalmazhoz van az S-sel ekvivalens soros sorrend.

Alapeset: Ha csak egyetlen csomópont van, a gyökér, akkor ahogyan már megfigyeltük, a megfelelő sorrend az, ahogy a tranzakciók a gyökeret zárolják.

Indukció: Ha egynél több csomópont van a fában, tekintsük a gyökér mindegyik részfájához az olyan tranzakciókból álló halmazt, amelyek egy vagy több csomópontot zárolnak abban a részfában. Megjegyezzük, hogy a gyökeret záró tranzakciók egy vagy több részfához tartozhatnak, de egy olyan tranzakció, amely nem zárolja a gyökeret, az csak egyetlen részfához tartozik. Például a 9.31. ábrán található tranzakciók közül csak a T_1 zárolja a gyökeret, és az mindkét részfához tartozik, a B gyökerű fához is és a C gyökerű fához is. A T_2 és a T_3 viszont csak a B gyökerű fához tartoznak.

Az indukciós feltevés szerint, van soros sorrend az összes olyan tranzakcióhoz, amelyek ugyanabban a tetszőleges részfában zárolnak csomópontokat. Csupán egybe kell olvasztanunk a különböző részfákhoz tartozó soros sorrendeket. Mivel a tranzakcióknak ezekben a listáiban csak azok a tranzakciók közösek, amelyek a gyökeret záró tranzakciók, és megállapítottuk, hogy ezek a tranzakciók minden közös csomópontot ugyanabban a sorrendben zárolnak, ahogy a gyökeret zárolják, nem fordulhat elő két, a gyökeret záró tranzakció különböző sorrendben két részlistán. Pontosabban, ha T_i és T_j előfordul a gyökér valamely C gyermekéhez tartozó listán, akkor ezek a C-t ugyanabban a sorrendben zárolják, ahogyan a gyökeret zárolják, és emiatt a listán is ebben a sorrendben fordulnak elő. Így felépíthetjük a soros sorrendet a teljes tranzakcióhalmazhoz azokból a tranzakciókból kiindulva, amelyek a gyökeret zárolják, a megfelelő sorrendjükben, és beleolvasztjuk azokat a tranzakciókat, amelyek nem zárolják a gyökeret, a részfáik soros sorrendjével konzisztens tetszőleges sorrendben.



9.34. ábra. *A részfákhoz tartozó soros sorrendek egyesítése az összes tranzakcióhoz tartozó soros sorrenddé*

9.25. példa: Tegyük fel, hogy van 10 darab tranzakció $T_1, T_2, \ldots, T_{10}$, és ezekből T_1, T_2 és T_3 ugyanabban a sorrendben zárolja a gyökeret. Tegyük fel azt is, hogy a gyökérnek van két gyereke, az elsőt a T_1-től a T_7-ig zárolják a tranzakciók, és a másodikat T_2, T_3, T_8, T_9 és T_{10} zárolja. Tegyük fel, hogy az első részfához a soros sorrend $(T_4, T_1, T_5, T_2, T_6, T_3, T_7)$. Megjegyezzük, hogy ennek a sorrendnek tartalmaznia kell T_1, T_2 és T_3-at ebben a sorrendben. Legyen továbbá a második részfához a soros sorrend $(T_8, T_2, T_9, T_{10}, T_3)$. Mint az előző esetben, a T_2 és T_3 tranzakciók, amelyek a gyökeret zárolják, abban a sorrendben fordulnak elő, ahogyan a gyökeret zárolták.

Ezeknek a tranzakcióknak a soros sorrendjére felállított megszorításokat a 9.34. ábrán mutatjuk be. A folyamatos vonalak a gyökér első gyerekének a rendezése szerinti megszorításokat jelölik, és a szaggatott vonalak pedig a második gyereknél levő rendezést jelölik. Ennek a gráfnak több topologikus sorrendjéből a $(T_4, T_8, T_1, T_5, T_2, T_9, T_6, T_{10}, T_3, T_7)$ az egyik. □

9.7.4. Feladatok

9.7.1. feladat: Tegyük fel, hogy végrehajtjuk az alábbi műveleteket a 4.23. ábrán szereplő B-fán. Ha a faprotokollt alkalmazzuk, mikor tudjuk feloldani az írási zárakat minden egyes megvizsgált csomóponton?

* a) Beszúrjuk a 10-et.
b) Beszúrjuk a 20-at.
c) Töröljük az 5-öt.
d) Töröljük a 23-at.

! **9.7.2. feladat:** Tekintsük az alábbi tranzakciókat, amelyek a 9.30. ábrán található fán működnek.

T_1: $r_1(A)$; $r_1(B)$; $r_1(E)$;
T_2: $r_2(A)$; $r_2(C)$; $r_2(B)$;
T_3: $r_3(B)$; $r_3(E)$; $r_3(F)$;

Válaszoljunk a következőkre:

* a) Hányféleképpen lehet T_1-et és T_2-t átlapolni, ha a faprotokollt követjük?
b) Hányféleképpen lehet T_1-et és T_3-t átlapolni, ha a faprotokollt követjük?
!! c) Hányféleképpen lehet mind a hármat átlapolni, ha a faprotokollt követjük?

! **9.7.3. feladat:** Tegyük fel, hogy van nyolc tranzakció T_1, T_2, ..., T_8, amelyekből a páratlan számú tranzakciók, T_1, T_3, T_5 és T_7, a fa gyökerét zárolják ebben a sorrendben. Három gyereke van a gyökérnek, az elsőt T_1, T_2, T_3 és T_4 zárolja ebben a sorrendben. A második gyereket T_3, T_6 és T_5 zárolja ebben a sorrendben, és a harmadik gyereket T_8 és T_7 zárolja ebben a sorrendben. A tranzakcióknak hány olyan soros sorrendje van, amely konzisztens ezekkel az állításokkal?

!! **9.7.4. feladat:** Tegyük fel, hogy az olvasáshoz, illetve íráshoz megfelelő osztott, illetve kizárólagos zárakkal rendelkező faprotokollt használjuk. A 2. szabályt, amely megköveteli, hogy zárolva legyen a szülő ahhoz, hogy a csomópontot zároljuk, meg kell változtatnunk, hogy megakadályozzuk a nem sorba rendezhető viselkedést. Mi az osztott és kizárólagos záraknak megfelelő helyes 2. szabály? *Útmutató:* ugyanolyan típusú zárolása szükséges-e a szülőknek, mint amilyen a gyereken van?

9.8. Konkurenciavezérlés időbélyegzőkkel

A következőkben a zárolástól különböző két másik módszert nézünk meg, amelyeket néhány rendszerben használnak a tranzakciók sorbarendezhetőségének biztosítására:

1. *Időbélyegzés* (timestamping). Minden tranzakcióhoz hozzárendelünk egy „időbélyegzőt", minden adatbáziselem utolsó olvasását és írását végző tranzakció időbélyegzőjét rögzítjük, és összehasonlítjuk ezeket az értékeket, hogy biztosítsuk, hogy a tranzakciók időbélyegzőinek megfelelő soros ütemezés ekvivalens legyen a tranzakciók aktuális ütemezésével. Ez a megközelítés lesz a jelenlegi rész témája.
2. *Érvényesítés* (validation). Megvizsgáljuk a tranzakciók időbélyegzőit és az adatbáziselemeket, amikor a tranzakció véglegesítésre kerül. Ezt az eljárást a tranzakciók „érvényesítésének" nevezzük. Az a soros ütemezés, amely az érvényesítési idejük alapján rendezi a tranzakciókat, ekvivalens kell hogy legyen az aktuális ütemezéssel. Az érvényesítési megközelítést a 9.9. részben tárgyaljuk.

Mindkét megközelítés *optimista* abban az értelemben, hogy feltételezik: nem fordul elő nem sorba rendezhető viselkedés, és csak akkor tisztázza a helyzetet, amikor a megszegés nyilvánvaló. Ezzel ellentétben, minden zárolási módszer azt feltételezi, hogy a dolgok rosszra fordulnak, hacsak a tranzakciókat azonnal meg nem akadályozzák a nem sorba rendezhető viselkedésbe kerülésben. Az optimista megközelítések abban különböznek a zárolásoktól, hogy az egyetlen ellenszerük, amikor valami rossz-

ra fordul, hogy azt a tranzakciót, amely nem sorba rendezhető viselkedésbe próbált kerülni, abortáljuk (leállítjuk), és aztán újraindítjuk. A zárolási ütemezők ezzel ellentétben késleltetik a tranzakciókat, de nem abortálják őket.[11] Általában az optimista ütemezők akkor jobbak a zárolásinál, amikor sok tranzakció csak olvasási, ugyanis az ilyen tranzakciók önmagukban soha nem okozhatnak nem sorba rendezhető viselkedést.

9.8.1. Időbélyegzők

Annak érdekében, hogy az időbélyegzést konkurenciavezérlési módszerként használjuk, az ütemezőnek minden egyes T tranzakcióhoz hozzá kell rendelnie egy egyedi számot, a TS(T) *időbélyegzőt* (ahol TS az angol *timestamp* rövidítése). Az időbélyegzőket növekvő sorrendben kell kiadni abban az időpontban, amikor a tranzakció az elindításáról először értesíti az ütemezőt. Két megközelítés az időbélyegzők generálásához:

a) Az egyik lehetőség, hogy az időbélyegzőket a rendszeróra felhasználásával hozzuk létre, feltéve, hogy az ütemező nem működik annyira gyorsan, hogy két tranzakcióhoz ugyanazt az órapercegést rendelné időbélyegzőként.
b) A másik megközelítés szerint az ütemező karbantart egy számlálót. Minden alkalommal, amikor egy tranzakció elindul, a számláló növekszik 1-gyel, és ez az új érték lesz a tranzakció időbélyegzője. Ebben a megközelítésben az időbélyegzőknek semmi közük sincs az „idő"-höz, azonban azzal a bármely időbélyegző-generáló rendszer esetén szükséges fontos tulajdonsággal rendelkeznek, miszerint egy később elindított tranzakció nagyobb időbélyegzőt kap, mint egy korábban elindított tranzakció.

Bármelyik módszert is használjuk az időbélyegzők generálására, az ütemezőnek karban kell tartania a jelenleg aktív tranzakciók és időbélyegzőik tábláját.

Ahhoz, hogy időbélyegzőket használjunk konkurenciavezérlési módszerként, minden egyes X adatbáziselemhez hozzá kell kapcsolnunk két időbélyegzőt és egy további bitet:

1. RT(X), az X *olvasási ideje* (ahol RT az angol *read time* rövidítése), amely a legmagasabb időbélyegző, ami egy olyan tranzakcióhoz tartozik, amely már olvasta az X-et.
2. WT(X), az X *írási ideje* (ahol WT az angol *write time* rövidítése), amely a legmagasabb időbélyegző, ami egy olyan tranzakcióhoz tartozik, amely már írta X-et.
3. C(X), az X *véglegesítési bitje* (C az angol *commit bit* szóból származik), amely akkor és csak akkor igaz, ha a legújabb tranzakció, amely az X-et írta, már véglegesítve van. Ennek a bitnek az a célja, hogy elkerüljük azt a helyzetet, amelyben egy T tranzakció egy másik U tranzakció által írt adatokat olvas be, és utána az U-t

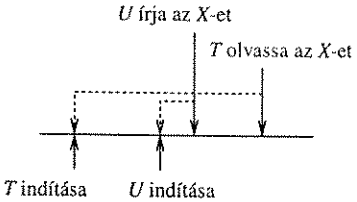
[11] Ez nem azt jelenti, hogy az a rendszer, amely zárolási ütemezőt használ, soha nem abortáltatja a tranzakciót. Például a 10.3. részben tárgyaljuk a holtpontok feloldását szolgáló tranzakcióabortáltatást. Egy zárolási ütemező viszont soha nem használ tranzakcióabortálást egyszerűen mint egy választ a zárolási kéréshez, amelyet nem lehet engedélyezni.

abortáljuk. Ez a probléma, amikor a T nem véglegesített adatok „piszkos olvasását" hajtja végre, bizonyosan az adatbázis-állapot inkonzisztensé válását is okozhatja. Így bármely ütemezőhöz szükség van olyan mechanizmusra, amely megakadályozza a piszkos olvasást.[12]

9.8.2. Fizikailag nem megvalósítható viselkedések

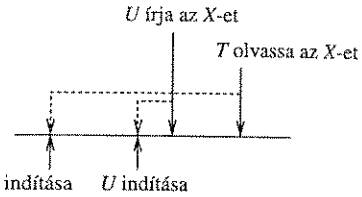
Azért, hogy megértsük az időbélyegzőn alapuló ütemező felépítését és szabályait, emlékeztetnünk kell arra, hogy az ütemező feltételezi, hogy a tranzakciók időbélyegző szerinti sorrendje egyúttal olyan soros sorrend, amely a végrehajtás sorrendjét is jelenti. Így az ütemező feladata azon túl, hogy hozzárendeli az időbélyegzőket a tranzakciókhoz, és módosítja az RT-t, WT-t és C-t az adatbáziselemek számára, még az is, hogy ellenőrzi, amikor egy olvasás vagy írás fordul elő, hogy az úgy történt *volna-e* a valós időben is, ha minden tranzakciót azonnal, az időbélyegző általi időpillanatban hajtottuk volna végre. Ha nem, akkor azt mondjuk, hogy a viselkedés *fizikailag nem megvalósítható*. Kétféle probléma merülhet fel:

1. *Túl késői olvasás:* A T tranzakció megpróbálja olvasni az X adatbáziselemet, de az X írási ideje azt jelzi, hogy az X jelenlegi értékét azután írtuk, miután a T-t már elméletileg végrehajtottuk. Vagyis $TS(T) < WT(X)$. A 9.35. ábra mutatja ezt a problémát. A vízszintes tengely jelenti azt a valós időt, amikor az események előfordulnak. A pontozott vonalak kapcsolják össze az aktuális eseményt azzal az időponttal, amikor a tranzakciók időbélyegzője szerint elméletileg végre kellett volna hajtani az eseményt. Így látjuk, hogy az U tranzakciót a T tranzakció után indítottuk el, mégis az X értékét előbb írta, mielőtt a T beolvasta volna az X-et. T-nek nem az U által írt értéket kellene olvasnia, ugyanis elméletileg az U-t a T után hajtjuk végre. A T-nek viszont nincs más választása, ugyanis az X-nek az U által írt értéke az egyetlen, amelyet a T most be tud olvasni. A megoldás, hogy a T-t abortáljuk, amikor ez a probléma felmerül.
2. *Túl késői írás:* A T tranzakció megpróbálja írni az X adatbáziselemet, de az X olvasási ideje azt jelzi, hogy van egy másik tranzakció is, amelynek a T által beírt értéket kellene olvasnia, ám ehelyett más értéket olvas. Vagyis $WT(X) < TS(T) < RT(X)$. A 9.36. ábra mutatja ezt a problémát, amely egy U tranzakciót mutat, amelyet a T után indítottunk el, mégis előbb olvassa az X-et, mint a T-nek lehetősége lett volna írnia az X-et. Amikor a T megpróbálja írni az X-et, úgy találjuk, hogy $RT(X) > TS(T)$, ami azt jelenti, hogy az U tranzakció már beolvasta az X-et, amelyet elméletileg a T-nél később kellett volna elvégeznie. Valamint úgy találjuk, hogy $WT(X) < TS(T)$, ami azt jelenti, hogy semelyik más tranzakció sem írta az X-et, amellyel felülírta volna a T általi értéket, így érvénytelenítette volna a T hatását, és olyan érték került volna az X-be, amelyet az U beolvashat.



9.35. ábra. *A T tranzakció túl késői olvasást próbál végezni*

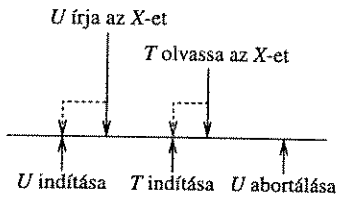
[12] Bár a piaci rendszerek általában a felhasználóra bízzák, hogy megengedhetők-e a piszkos olvasások.



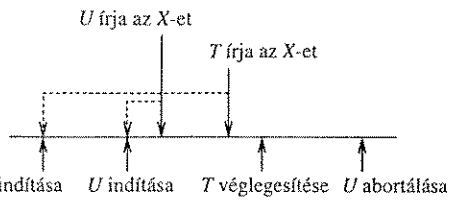
9.36. ábra. *A T tranzakció túl késői írást próbál végezni*

9.8.3. Piszkos adatok problémái

Van egy problémákból álló osztály, amelynek kezelésére bevezették a véglegesítési bitet. A problémák egyike a „piszkos olvasás", amelyet a 9.37. ábra szemléltet. Itt a T tranzakció olvassa az X-et, és ezen X-et utoljára az U írta. Az U időbélyegzője kisebb, mint a T-é, és a valóságban a T általi olvasás az U általi írás után történik, így az esemény úgy tűnik, hogy fizikailag megvalósítható. Mégis lehetséges, hogy miután a T beolvasta az U által írt X-beli értéket, az U tranzakciót abortáljuk. Esetleg az U talált hibás feltételt a saját adataiban, mint pl. 0-val való osztást, vagy mint ahogyan később látni fogjuk a 9.8.4. részben, az ütemező kényszeríti ki az U abortálását, ugyanis az valamilyen fizikailag nem megvalósítható viselkedést eredményező dolgot próbált végezni. Így, bár nincs fizikailag nem megvalósítható abban, hogy a T olvassa az X-et, mégis jobb a T olvasását akkorra elhalasztani, amikor az U véglegesítését vagy abortálását már elvégeztük. Meg tudjuk mondani, hogy az U még nincs véglegesítve, ugyanis a $C(X)$ véglegesítési bit hamis lesz.



9.37. ábra. *A T tranzakció piszkos adat olvasást tud végezni, ha akkor olvassa az X-et, amikor az ábrán látható*



9.38. ábra. *Egy írást elhagyunk, ugyanis van egy későbbi időbélyegzővel ellátott írás, azonban az író tranzakció később abortál*

Egy másik lehetséges problémát a 9.38. ábra szemléltet. Itt az U, a T-nél későbbi időbélyegzőjű tranzakció írja először az X-et. Amikor a T írni próbál, a megfelelő művelet semmit sem végez. Nyilvánvalóan nincs más V tranzakció, amelynek az X-ből a T általi értékét kellene beolvasnia, és ehelyett az U általi értéket olvasná, ugyanis ha a V megpróbálná olvasni az X-et, abortálnia kellene a túl késői olvasás miatt. A későbbi X olvasásoknál az U általi értéket kell olvasni, vagy az X még későbbi, de nem a T általi értékét. Ezt az ötletet, miszerint azokat az írásokat kihagyhatjuk, amelyeknél már elvégeztünk egy későbbi írási idejű írást, *Thomas-féle írási szabálynak* nevezzük.

A Thomas-féle írási szabállyal azonban van egy lényegi probléma. Ha az U-t később abortáljuk, amint az a 9.38. ábrán látható, akkor az U által írt X értéket ki kell törölnünk, továbbá az előző értéket és írási időt vissza kell állítanunk. Minthogy a T-t véglegesítettük, úgy látszik, hogy a T által írt X értéket kell a későbbi olvasáshoz használnunk. Mi viszont már kihagytuk a T általi írást, és már túl késő, hogy helyrehozhassuk ezt a hibát.

Sokféle módon lehet kezelni a most vázolt problémát, azonban egy viszonylag egyszerű elvet mutatunk be, amely az időbélyegzőn alapuló ütemezőre épül.

- Amikor a T tranzakció írja az X adatbáziselemet, az írás „kísérleti", és vissza lehet állítani, ha a T-t abortáljuk. A $C(X)$ véglegesítési bitet hamisra állítjuk, egyúttal az ütemező másolatot készít az X régi értékéről, és az előző $WT(X)$-ről.

9.8.4. Az időbélyegzőn alapuló ütemezések szabályai

Összegezhetjük azokat a szabályokat, amelyeket az időbélyegzőket használó ütemezőnek követnie kell ahhoz, hogy biztosan ne fordulhasson elő semmiféle fizikailag nem megvalósítható viselkedés. Az ütemezőnek egy T tranzakciótól érkező olvasási vagy írási kérésre adott válaszában az alábbi választásai lehetnek:

a) Engedélyezi a kérést.
b) Abortálja a T-t (ha a T megsérti a fizikai valóságot), és egy új időbélyegzővel újraindítja a T-t (azt az abortálást, amelyet újraindítás követ gyakran *visszagörgetésnek* vagy *rollbacknek* nevezzük).
c) Késlelteti a T-t, és később dönti el, hogy abortálja a T-t, vagy engedélyezi a kérést (ha a kérés olvasás és az olvasás piszkos is lehet, mint a 9.8.3. részben).

A szabályok a következők:

1. Tegyük fel, hogy az ütemezőhöz érkező kérés $r_T(X)$.

   a) Ha $TS(T) \geq WT(X)$, az olvasás fizikailag megvalósítható.

      i) Ha $C(X)$ igaz, engedélyezzük a kérést. Ha $TS(T) > RT(X)$, akkor $RT(X) := TS(T)$, egyébként nem változtatjuk meg $RT(X)$-t.
      ii) Ha $C(X)$ hamis, késleltessük a T-t addig, amíg $C(X)$ igazzá válik, vagy addig, amíg az X-et író tranzakció abortál.

   b) Ha $TS(T) < WT(X)$, az olvasás fizikailag nem megvalósítható. Visszagörgetjük a T-t, vagyis abortáljuk T-t, és újraindítjuk egy új, nagyobb időbélyegzővel.

2. Tegyük fel, hogy az ütemezőhöz érkező kérés $w_T(X)$.

   a) Ha $TS(T) \geq RT(X)$ és $TS(T) \geq WT(X)$, az írás fizikailag megvalósítható, és az alábbiakat kell végrehajtani:

      i) Írjuk be az új X értéket.
      ii) Állítsuk be $WT(X) := TS(T)$.
      iii) Állítsuk be $C(X) :=$ `hamis`.

   b) Ha $TS(T) \geq RT(X)$, de $TS(T) < WT(X)$, akkor az írás fizikailag megvalósítható, de az X-nek már egy későbbi értéke van. Ha $C(X)$ igaz, az X előző írását végző tranzakció véglegesítve van, és egyszerűen figyelmen kívül hagyjuk a T írását, megengedjük, hogy a T folytatódjon, és ne változtassa meg az adatbázist. Ha viszont a $C(X)$ hamis, akkor késleltetnünk kell a T-t, mégpedig az 1a)ii) pontban leírtak szerint.
   c) Ha $TS(T) < RT(X)$, az írás fizikailag nem megvalósítható, és a T-t vissza kell görgetnünk.

3. Tegyük fel, hogy az ütemezőhöz érkező kérés a T véglegesítése. Meg kell találnunk (az ütemező karbantartási listája alapján) az összes olyan X adatbáziselemet, amelybe a T írt, és állítsuk be $C(X)$-et `igaz`-ra. Ha vannak az X véglegesítésére várakozó tranzakciók (az ütemező egy másik karbantartási listáján találjuk meg), ezeknek a tranzakcióknak megengedjük, hogy folytatódjanak.
4. Tegyük fel, hogy az ütemezőhöz érkező kérés a T abortálása, vagy a T visszagörgetésére való döntés, mint az 1.b) vagy 2.c) esetekben. Ekkor bármely olyan tranzakcióra, amely egy X elem T általi írására várakozott, meg kell ismételnünk ezt az olvasási vagy írási kísérletet, és meglátjuk, hogy a művelet most jogszerű-e, miután az abortált tranzakció írásait visszavontuk.

9.26. példa: A 9.39. ábrán három tranzakció T_1, T_2 és T_3 ütemezése látható, amelyek három A, B és C adatbáziselemhez férnek hozzá. Az események előfordulásának valós

T_1	T_2	T_3	A	B	C
200	150	175	RT = 0 WT = 0	RT = 0 WT = 0	RT = 0 WT = 0
$r_1(B)$;				RT = 200	
	$r_2(A)$;		RT = 150		
		$r_3(C)$;			RT = 175
$w_1(B)$;				WT = 200	
$w_1(A)$;			WT = 200		
	$w_2(C)$;				
	Abortál;				
		$w_3(A)$;			

9.39. ábra. *Három tranzakció időbélyegzőn alapuló ütemező alatti végrehajtása*

ideje a szokás szerint a lapon lefelé nő. Most azonban a tranzakciók időbélyegzői és az elemek olvasási és írási ideje is jelölve vannak. Tegyük fel, hogy kezdetben minden adatbáziselemhez az olvasási és az írási idő is 0. A tranzakciók abban a pillanatban kapnak időbélyegzőt, amikor értesítik az ütemezőt az elindításukról. Megjegyezzük, hogy bár a T_1 hajtja végre az első adathozzáférést, mégsem neki van a legkisebb időbélyegzője. Tegyük fel, hogy T_2 az első, amelyik az indításáról értesíti az ütemezőt, és a T_3 volt a következő, és T_1-et indítottuk el utoljára.

Az első műveletben a T_1 beolvassa a B-t. Mivel a B írási ideje kisebb, mint a T_1 időbélyegzője, ez az olvasás fizikailag megvalósítható, és engedélyezzük a végrehajtást. B olvasási idejét 200-ra állítjuk, a T_1 időbélyegzőjére. A második és a harmadik olvasási művelet hasonlóan jogszerű, és mindegyik adatbáziselem olvasási idejének értékét az őt olvasó tranzakció időbélyegzőjére állítjuk.

A negyedik lépésben T_1 írja a B-t. Mivel a B olvasási ideje nem nagyobb, mint a T_1 időbélyegzője, az írás fizikailag megvalósítható. Mivel a B írási ideje nem nagyobb, mint a T_1 időbélyegzője, ténylegesen végre kell hajtanunk az írást. Amikor ezt elvégezzük, a B írási idejét 200-ra növeljük, amely az őt felülíró T_1 tranzakció időbélyegzője.

Ezután T_2 megpróbálja írni a C-t. C-t viszont már beolvasta a T_3 tranzakció, amelyet elméletileg a 175-ös időpontban hajtottunk végre, míg a T_2-nek az értéket a 150-es időpontban kellett volna beírnia. Így a T_2 olyan dologgal próbálkozik, amely fizikailag nem megvalósítható viselkedést eredményezne, és a T_2-t vissza kell görgetnünk.

Az utolsó lépés, hogy a T_3 írja az A-t. Mivel az A írási ideje 150, kevesebb, mint a T_3 időbélyegzője, ami 175, az írás jogszerű. Viszont az A-nak már egy későbbi értéke van tárolva ebben az adatbáziselemben, mégpedig a T_1 által beírt érték, elméletileg a 200-as időpontban. Így a T_3-at nem görgetjük vissza, de be sem írjuk az értékét. □

9.8.5. Többváltozatú időbélyegzők

Az időbélyegzés egyik fontos változata karbantartja az adatbáziselemek régi változatait is, az adatbázisban magában tárolt jelenlegi változaton kívül. A cél az, hogy megengedjünk olyan $r_T(X)$ olvasásokat, amelyek egyébként a T tranzakció abortálását

okoznák (ugyanis az X jelenlegi változatát egy T-nél későbbi írta felül) úgy, hogy az X-nek a T időbélyegzőjű tranzakcióhoz megfelelő régebbi változatának a beolvasásával folytatjuk T-t. A módszer különösen hasznos, ha az adatbáziselemek lemezblokkok vagy lapok, ugyanis ekkor csak annyit kell a pufferkezelőnek tennie, hogy bizonyos blokkok a memóriában legyenek, amelyek néhány jelenleg aktív tranzakció számára hasznosak lehetnek.

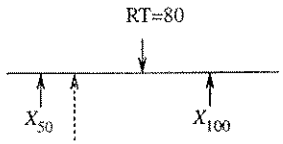
9.27. példa: Tekintsük a 9.40. ábrán szereplő, az A adatbáziselemhez hozzáférő tranzakciók halmazát. Ezek a tranzakciók egy közönséges időbélyegzőn alapuló ütemező alatt működnek, és amikor a T_3 megpróbálja az A-t olvasni, azt találja, hogy a WT(A) nagyobb, mint a saját időbélyegzője, és abortálni kell. Viszont megvan az A-nak a T_1 által írt, és a T_2 által felülírt régi értéke, amely alkalmas lenne a T_3-nak, hogy olvassa. Ebben a változatban az A-nak 150-es volt az írási ideje, ami kevesebb, mint a T_3 175-ös időbélyegzője. Ha az A-nak ez a régi értéke hozzáférhető lenne, a T_3 engedélyt kaphatna az olvasásra, még ha ez az A-nak nem is a „jelenlegi" értéke. □

T_1	T_2	T_3	T_4	A
150	200	175	225	RT = 0 WT = 0
$r_1(A)$;				RT = 150
$w_1(A)$;				WT = 150
	$r_2(A)$;			RT = 200
	$w_2(A)$;			WT = 200
		$r_3(A)$;		
		Abortál;		
			$r_4(A)$;	RT = 225

9.40. ábra. *A T_3-at abortálnunk kell, ugyanis nem tud hozzáférni az A régi értékéhez*

A többváltozatú időbélyegzés ütemező az alábbiakban különbözik az 9.8.4. részben leírt ütemezőtől:

1. Amikor egy új $w_T(X)$ írás fordul elő, ha ez jogszerű, akkor az X adatbáziselemnek egy új változatát hozzuk létre. Az írási ideje TS(T), és X_t-vel fogunk rá hivatkozni, ahol t = TS(T).
2. Amikor egy $r_T(X)$ olvasás fordul elő, az ütemező megkeresi az X-nek azt az X_t változatát, amelyre $t \leq$ TS(T), de nincs más $X_{t'}$ változata, amelyre $t < t' \leq$ TS(T) lenne. Vagyis az X-nek azt a változatát, amelyet a T elméleti végrehajtása előtt közvetlenül írtak, olvassa be a T.
3. Az írási időket egy elem *változataihoz* kapcsoljuk, és soha nem változtatjuk meg.
4. Az olvasási időket is a változatokhoz kapcsoljuk. Arra használjuk őket, hogy visszautasítsunk bizonyos írásokat, mégpedig azokat, amelyek ideje kisebb, mint az előző verzió olvasási ideje. A 9.41. ábrán mutatjuk be ezt a problémát, ahol az X változatai az X_{50} és az X_{100}, a korábbi a 80-as időpontban olvasásra került, és megjelent a 60-as időbélyegzőjű T tranzakció általi új írás. Ez az írás a T abortálá-



9.41. ábra. *Egy tranzakció az X egyik változatát próbálja írni, amely az eseményt fizikailag megvalósíthatatlanná tenné*

sát kell hogy okozza, ugyanis az X-beli értékét a 80-as időbélyegzőjű tranzakciónak kellett volna olvasnia, ha T végrehajtását engedtük volna.

5. Amikor egy X_t változat t írási ideje olyan, hogy nincs a t-nél kisebb időbélyegzőjű aktív tranzakció, akkor törölhetjük az X-nek az X_t-t *megelőző* változatait.

9.28. példa: Tekintsük újból a 9.40. ábrán szereplő műveleteket, amikor többváltozatú időbélyegzést használunk. Először, az A-nak három változata létezik: A_0, amelyik a tranzakciók elindítása előtt létezik, A_{150}, amelyet a T_1 írt, és A_{200}, amelyet a T_2 írt. A 9.42. ábra mutatja azt az eseménysorozatot, amikor a változatokat létrehozzuk, és amikor ezeket beolvassuk. Megjegyezzük, hogy T_3-at nem kell abortálni, ugyanis be tudja olvasni az A-nak egy korábbi változatát. □

T_1	T_2	T_3	T_4	A_0	A_{150}	A_{200}
150	200	175	225			
$r_1(A)$;				Olvasás		
$w_1(A)$;					Létrehozás	
	$r_2(A)$;				Olvasás	
	$w_2(A)$;					Létrehozás
		$r_3(A)$;			Olvasás	
			$r_4(A)$;			Olvasás

9.42. ábra. *Többváltozatú konkurenciavezérlést alkalmazó tranzakciók végrehajtása*

9.8.6. Az időbélyegzők és zárolások

Általában az időbélyegzés azokban a helyzetekben kiváló, amikor a tranzakciók többsége csak olvasási, vagy ritka az az eset, hogy konkurens tranzakciók ugyanazt az elemet próbálják meg olvasni és írni. Az erősen konfliktusos helyzetekben jobb a zárolásokat használni. Ehhez az ökölszabályhoz az érvek az alábbiak:

- A zárolások gyakran késleltetik a tranzakciókat azzal, hogy a zárakra várnak, és még holtpontok is kialakulhatnak, amikor néhány tranzakció hosszú ideje várakozik, és ezután az egyiket vissza kell görgetni.

- Ha viszont a konkurens tranzakciók gyakran olvasnak és írnak közös elemeket, akkor a visszagörgetés lesz gyakori, ami még több késedelmet okoz, mint egy zárolási rendszer.

Több piaci rendszer érdekes kompromisszumot alkalmaz. Az ütemező felosztja a tranzakciókat csak olvasási tranzakciókra és olvasási/írási tranzakciókra. Az olvasási/írási tranzakciókat kétfázisú zárolást használva hajtjuk végre úgy, hogy a zárolt elemek hozzáférését megakadályozzuk a többi és a csak olvasási tranzakciók esetén is.

A csak olvasási tranzakciókat a többváltozatú időbélyegzéssel hajtjuk végre. Amikor az olvasási/írási tranzakciók létrehozzák egy adatbáziselem új változatait, ezeket a változatokat úgy kezeljük, ahogyan a 9.8.5. részben leírtuk. Csak olvasási tranzakciónak megengedjük, hogy egy adatbáziselem bármelyik változatát olvassa, amelyik megfelel az időbélyegzőjének. Csak olvasási tranzakciót emiatt soha nem kell abortálnunk, és csak nagyon ritkán kell késleltetnünk.

9.8.7. Feladatok

9.8.1. feladat: Az alábbiakban több eseménysorozatot találunk, beleértve az *indítási* eseményeket is, ahol az st_i (az angol *start* rövidítéssel) azt jelenti, hogy a T_i tranzakciót elindítottuk. Ezek a sorozatok valós időt jelentenek, és az időbélyegzőn alapuló ütemező a tranzakciókhoz az időbélyegzőket az indítási sorrendjük szerint adja. Mondjuk meg, hogy mi történik, amikor ezeket végrehajtjuk.

* a) st_1; st_2; $r_1(A)$; $r_2(B)$; $w_2(A)$; $w_1(B)$;
b) st_1; $r_1(A)$; st_2; $w_2(B)$; $r_2(A)$; $w_1(B)$;
c) st_1; st_2; st_3; $r_1(A)$; $r_2(B)$; $w_1(C)$; $r_3(B)$; $r_3(C)$; $w_2(B)$; $w_3(A)$;
d) st_1; st_3; st_2; $r_1(A)$; $r_2(B)$; $w_1(C)$; $r_3(B)$; $r_3(C)$; $w_2(B)$; $w_3(A)$;

9.8.2. feladat: Mondjuk meg, hogy mi történik az alábbi eseménysorozatok folyamán, ha többváltozatú, időbélyegzőn alapuló ütemezőt használunk. Mi történik helyette, ha az ütemező nem támogat többszörös többváltozatokat?

* a) st_1; st_2; st_3; st_4; $w_1(A)$; $w_2(A)$; $w_3(A)$; $r_2(A)$; $r_4(A)$;
b) st_1; st_2; st_3; st_4; $w_1(A)$; $w_3(A)$; $r_4(A)$; $r_2(A)$;
c) st_1; st_2; st_3; st_4; $w_1(A)$; $w_4(A)$; $r_3(A)$; $w_2(A)$;

!! **9.8.3. feladat:** Észrevettük a zároláson alapuló ütemezők tanulmányozásában, hogy számos okból alakulhatnak ki holtpontok a zárakat elnyert tranzakciók esetén. A $C(X)$ véglegesítési bitet használó időbélyegzőn alapuló ütemező esetén kialakulhat-e holtpont?

9.9. Konkurenciavezérlés érvényesítéssel

Az *érvényesítés* (validation) az optimista konkurenciavezérlés másik típusa, amelyben a tranzakcióknak megengedjük, hogy zárolások nélkül hozzáférjenek az adatokhoz, és a megfelelő időben ellenőrizzük a tranzakció sorba rendezhető viselkedését. Az érvényesítés alapvetően abban különbözik az időbélyegzőtől, hogy itt az ütemező egy nyilvántartást vezet arról, mit tesznek az aktív tranzakciók ahelyett, hogy az összes adatbáziselemhez feljegyezné az olvasási és írási időt. Mielőtt a tranzakció írni kezdene értékeket az adatbáziselemekbe, egy „érvényesítési fázison" megy keresztül, ahol a beolvasott és írandó elemek halmazait összehasonlítjuk más aktív tranzakciók írásai halmazaival. Ha a fizikailag nem megvalósítható viselkedés kockázata lépne fel, akkor a tranzakciót visszagörgetjük.

9.9.1. Érvényesítésen alapuló ütemező felépítése

Amikor az érvényesítést használjuk konkurenciavezérlési működésként, az ütemezőnek meg kell adnunk minden T tranzakcióhoz a T által olvasott adatbáziselemek halmazát, és a T által írt elemek halmazát. Ezek a halmazok az $\mathtt{RS}(T)$ *olvasási halmaz* (ahol *RS* az angol *read set* rövidítése), és a $\mathtt{WS}(T)$ *írási halmaz* (ahol *WS* az angol *write set* rövidítése). A tranzakciókat három fázisban hajtjuk végre:

1. *Olvasás*. Az első fázisban a tranzakciók beolvassák az adatbázisból az összes elemet az olvasási halmazba. A tranzakció ki is számítja a lokális címhelyen az összes eredményt, amelyet be fog írni.
2. *Érvényesítés*. A második fázisban, az ütemező érvényesíti a tranzakciót oly módon, hogy összehasonlítja az olvasási és írási halmazait a többi tranzakcióéval. Az érvényesítési eljárást a 9.9.2. részben fogjuk leírni. Ha az érvényesítés hibát jelez, akkor a tranzakciót visszagörgetjük, egyébként pedig folytatódik a harmadik fázissal.
3. *Írás*. A harmadik fázisban a tranzakció az írási halmazában levő elemek értékeit beírja az adatbázisba.

Intuitív alapon minden sikeresen érvényesített tranzakcióról azt gondolhatjuk, hogy az érvényesítés pillanatában került végrehajtásra. Így az érvényesítésen alapuló ütemező a tranzakciók feltételezett soros sorrendjével dolgozik. Az alapja annak a döntésnek, hogy érvényesítsen-e vagy sem egy tranzakciót az, hogy a tranzakciók viselkedése konzisztens legyen ezzel a soros sorrenddel.

Ahhoz a döntéshez, hogy az ütemező érvényesítheti-e a tranzakciót, fenntart három halmazt:

1. *KEZD*, a már elindított, de még nem teljesen érvényesített tranzakciók halmaza. Ebben a halmazban minden T tranzakcióhoz az ütemező karbantartja a $\mathtt{KEZD}(T)$-t a T indításának időpontjában.

2. *ÉRV*, a már érvényesített, de a 3. fázisban az írásokat még nem befejezett tranzakciók halmaza. Ebben a halmazban minden T tranzakcióhoz az ütemező karbantartja a $\mathtt{KEZD}(T)$-t és az $\mathtt{ÉRV}(T)$-t a T érvényesítésekor. Megjegyezzük, hogy $\mathtt{ÉRV}(T)$ az az idő, amikor a T végrehajtását gondoljuk a végrehajtás feltételezett soros sorrendjében.
3. *BEF*, a 3. fázist befejezett tranzakciók halmaza. Ezekhez a T tranzakciókhoz az ütemező rögzíti a $\mathtt{KEZD}(T)$-t, az $\mathtt{ÉRV}(T)$-t és a $\mathtt{BEF}(T)$-t a T befejezésekor. Elméletben ez a halmaz nő, de amint látni fogjuk, nem kell megjegyeznünk a T tranzakciót, ha $\mathtt{BEF}(T) < \mathtt{KEZD}(U)$ bármely U aktív tranzakcióra (vagyis bármely U-ra a *KEZD*-ben vagy az *ÉRV*-ben). Az ütemező így időnként tisztogathatja a *BEF* halmazt, hogy megakadályozza a méretének a korlátlan növekedését.

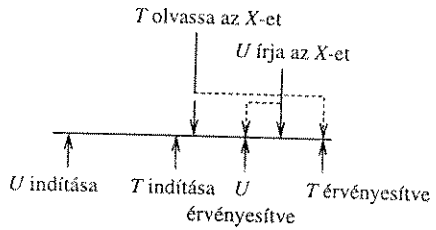
9.9.2. Az érvényesítési szabályok

Ha az ütemező elvégzi a karbantartását, akkor a 9.9.1. részben leírt információ elég ahhoz, hogy észlelje a tranzakciók feltételezett soros sorrendjének (a tranzakciók érvényesítési sorrendjének) bármely lehetséges megsértését. A szabályok megértése szempontjából először vizsgáljuk meg, hogy mi lehet hibás, amikor megpróbáljuk a T tranzakciót érvényesíteni.

1. Tegyük fel, hogy van olyan U tranzakció, hogy:

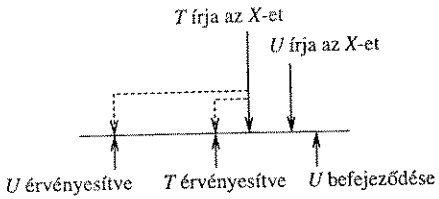
   a) U az *ÉRV*-ben vagy a *BEF*-ben van, vagyis az U-t már érvényesítettük.
   b) $\mathtt{BEF}(U) > \mathtt{KEZD}(T)$, vagyis az U nem fejeződött be a T indítása előtt.[13]
   c) $\mathtt{RS}(T) \cap \mathtt{WS}(U)$ nem üres, legyen X a metszetben levő adatbáziselem.

   Ekkor lehetséges, hogy az U azután írja az X-et, miután a T olvassa az X-et. Tulajdonképpen lehet, hogy az U még nem írta az X-et. Az az eset, amikor az U nem időben írta az X-et, a 9.43. ábrán látható. Az ábrához magyarázatként megjegyez-



9.43. ábra. *A T-t nem érvényesíthetjük, ha egy korábbi tranzakció most ír valamit, amelyet a T-nek olvasnia kellett volna*

[13] Megjegyezzük, hogy ha U az *ÉRV*-ben van, akkor az U még nem fejeződött be a T érvényesítésekor. Ebben az esetben a $\mathtt{BEF}(U)$ technikailag nem definiált. Viszont tudjuk, hogy a $\mathtt{KEZD}(T)$-nél nagyobbnak kell lennie ebben az esetben.



9.44. ábra. *A T tranzakció nem érvényesíthető akkor, ha egy korábbi tranzakció előtt tudna írni*

zük, hogy a pontozott vonalak kapcsolják össze a valós idejű eseményeket azzal az idővel, amikor elő kellett volna fordulniuk, ha a tranzakciókat az érvényesítés pillanatában hajtottuk volna végre. Mivel nem tudjuk, hogy a T-nek be kell-e olvasnia az U-tól származó értékét vagy sem, vissza kell görgetnünk T-t, hogy elkerüljük annak a kockázatát, hogy a T és az U műveletei nem lesznek konzisztensek a feltételezett soros sorrenddel.

2. Tegyük fel, hogy van olyan U tranzakció, amelyre:

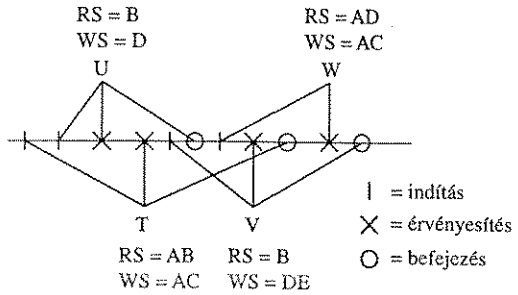
   a) U az *ÉRV*-ben van; vagyis az U-t már sikeresen érvényesítettük.
   b) $\mathtt{BEF}(U) > \mathtt{ÉRV}(T)$; vagyis az U-t nem fejeztük be mielőtt a T az érvényesítési fázisába lépett volna.
   c) $\mathtt{WS}(T) \cap \mathtt{WS}(U) \neq \emptyset$ legyen X mind a két írási halmazban.

   Ekkor a lehetséges problémát a 9.44. ábra mutatja. Mind a T-nek, mind az U-nak írnia kell az X értékét, és ha megengedjük a T érvényesítését, lehetséges, hogy az U előtt fogja írni az X-et. Mivel nem lehetünk biztosak a dolgunkban, visszagörgetjük a T-t, hogy biztosan ne szegjük meg azt a feltételezett soros sorrendet, amelyben T-t követi az U.

A fent leírt két problémával kerülhetünk csak olyan helyzetbe, amikor a T által végzett írás fizikailag nem megvalósítható. A 9.43. ábrán, ha az U a T elindítása előtt fejeződött volna be, akkor a T biztosan olyan X értéket olvasna, amelyet vagy U vagy valamely későbbi tranzakció írt. A 9.44. ábrán, ha az U a T érvényesítése előtt fejeződik be, akkor biztos, hogy az U a T előtt írta az X-et. Így a T tranzakció érvényesítésére vonatkozó észrevételeinket az alábbi szabállyal foglalhatjuk össze:

- Összehasonlítjuk az $\mathtt{RS}(T)$-t a $\mathtt{WS}(U)$-val, és ellenőrizzük, hogy $\mathtt{RS}(T) \cap \mathtt{WS}(U) = \emptyset$, minden olyan U-ra, amely még nem fejeződött be a T elindítása előtt, vagyis, ha $\mathtt{BEF}(U) > \mathtt{KEZD}(T)$, és az U egy korábban érvényesített tranzakció.
- Összehasonlítjuk a $\mathtt{WS}(T)$-t a $\mathtt{WS}(U)$-val, és ellenőrizzük, hogy $\mathtt{WS}(T) \cap \mathtt{WS}(U) = \emptyset$, minden olyan U-ra, amely még nem fejeződött be a T-t érvényesítése előtt, vagyis, ha $\mathtt{BEF}(U) > \mathtt{ÉRV}(T)$, és az U egy korábban érvényesített tranzakció.

9.29. példa: A 9.45. ábra egy idővonalat ábrázol, amely mentén T, U, V és W négy tranzakció végrehajtási és érvényesítési kísérletei láthatók. Az ábrán jelöltük mindegyik tranzakció olvasási és írási halmazait. A T-t indítjuk el elsőnek, de az U-t érvényesítjük elsőnek.



9.45. ábra. *Négy tranzakció és az érvényesítéseik*

1. Az U érvényesítése: amikor az U-t érvényesítjük, nincs más érvényesített tranzakció, és így nem kell semmit sem ellenőriznünk. Az U-t sikeresen érvényesítjük, és értéket írunk a D adatbáziselembe.
2. A T érvényesítése: amikor a T-t érvényesítjük, az U már érvényesítve van, de még nincs befejezve. Így ellenőriznünk kell, hogy a T-nek sem az olvasási, sem az írási halmazában nincs semmi közös a $\mathtt{WS}(U) = \{D\}$-vel. Mivel $\mathtt{RS}(T) = \{A, B\}$, és $\mathtt{WS}(T) = \{A, C\}$, mindkét ellenőrzés sikeres, ily módon a T-t érvényesítjük.
3. A V érvényesítése: amikor a V-t érvényesítjük, az U már érvényesítve van, és befejeződött, a T már érvényesítve van, de még nem fejeződött be. Továbbá a V-t az U befejeződése előtt indítjuk el. Így össze kell hasonlítanunk mind az $\mathtt{RS}(V)$-t, mind a $\mathtt{WS}(V)$-t a $\mathtt{WS}(T)$-vel, azonban csak az $\mathtt{RS}(V)$-t kell összehasonlítanunk a $\mathtt{WS}(U)$-val. Azt találjuk, hogy:

   - $\mathtt{RS}(V) \cap \mathtt{WS}(T) = \{B\} \cap \{A, C\} = \emptyset$.
   - $\mathtt{WS}(V) \cap \mathtt{WS}(T) = \{D, E\} \cap \{A, C\} = \emptyset$.
   - $\mathtt{RS}(V) \cap \mathtt{WS}(U) = \{B\} \cap \{D\} = \emptyset$.

   Így a V-t sikeresen érvényesítjük.

4. A W érvényesítése: Amikor a W-t érvényesítjük, azt tapasztaljuk, hogy az U a W elindítása előtt befejeződött, és így nem kell elvégeznünk a W és U összehasonlítását. A T a W érvényesítése előtt fejeződött be, de nem fejeződött be a W elindítása előtt, ezért csak az $\mathtt{RS}(W)$-t kell összehasonlítanunk a $\mathtt{WS}(T)$-vel. A V már érvényesítve van, de még nem fejeződött be, így össze kell hasonlítanunk mind az $\mathtt{RS}(W)$-t, mind a $\mathtt{WS}(W)$-t a $\mathtt{WS}(T)$-vel. Ezek az ellenőrzések:

   - $\mathtt{RS}(W) \cap \mathtt{WS}(T) = \{A, D\} \cap \{A, C\} = \{A\}$.
   - $\mathtt{RS}(W) \cap \mathtt{WS}(V) = \{A, D\} \cap \{D, E\} = \{D\}$.
   - $\mathtt{WS}(W) \cap \mathtt{WS}(V) = \{A, C\} \cap \{D, E\} = \emptyset$.

   Mivel a metszetek nem mind üresek, a W-t nem érvényesítjük, hanem a W-t visszagörgetjük, és így nem ír értéket sem az A-ba, sem a C-be.

□

Csak egy pillanat

Úgy gondoljuk, hogy az érvényesítés egy pillanat alatt vagy észrevétlenül rövid idő alatt játszódik le. Például úgy képzeljük, hogy el tudjuk dönteni, hogy egy U tranzakció már érvényesített-e akkor, amikor a T tranzakció érvényesítése elindult. Előfordulhat-e, hogy az U érvényesítése a T-t érvényesítése alatt fejeződik be?

Ha egyprocesszoros rendszeren futtatunk, és csak egy ütemező végzi a feldolgozást, akkor valóban azt gondolhatjuk az érvényesítésről és az ütemező többi tevékenységéről, hogy egy pillanat alatt bekövetkeznek. Azért, mert ha az ütemező a T-t érvényesíti, akkor nem érvényesítheti ezalatt az U-t is, így a T érvényesítése alatt az U érvényesítési állapota sem változhat meg.

Ha többprocesszoros rendszer alatt futtatunk, és több ütemező végzi a feldolgozást, akkor lehet, hogy az egyik a T-t érvényesíti, mialatt a másik az U-t érvényesíti. Ha így van, akkor a többprocesszoros rendszer olyan szinkronizációs működésére kell támaszkodnunk, amely biztosítja, hogy az érvényesítést atomi tevékenységként végezzük el.

9.9.3. Három konkurenciavezérlés működésének összehasonlítása

A sorbarendezhetőséghez három megközelítést néztünk meg – a zárolásokat, az időbélyegzőket és az érvényesítést – mindegyiknek megvannak az előnyei. Először hasonlítsuk őket össze a tár felhasználása szempontjából:

- *Zárak:* A zártábla által lefoglalt tár a zárolt adatbáziselemek számával arányos.
- *Időbélyegzők:* Egy naiv megvalósításban minden adatbáziselemhez akár hozzáférünk jelenleg, akár nem, az olvasási és írási időkhöz szükségünk van tárra. Egy körültekintőbb megvalósítás azonban úgy kezeli az összes olyan időbélyegzőt, amely a legkorábbi aktív tranzakciók előtti, hogy „mínusz végtelen" értékűnek tekinti, és nem rögzíti ezeket. Ebben az esetben, a zártáblával analóg táblában tudjuk tárolni az olvasási és írási időket, amelyben csak a legújabban elért adatbáziselemek szerepelnek.
- *Érvényesítés:* Tárat használunk az időbélyegzőkhöz és minden jelenleg aktív tranzakció olvasási/írási halmazaihoz, hozzávéve még egy pár olyan tranzakciót, amelyek azután fejeződnek be, miután valamelyik jelenleg aktív tranzakció elkezdődött.

Így mindegyik megközelítésben az összes aktív tranzakcióra felhasznált tár a tranzakciók által hozzáfért adatbáziselemek számának az összegével megközelítőleg arányos. Az időbélyegzés és az érvényesítés kicsit több helyet használhat fel, ugyanis nyomon kell követnünk a korábban véglegesített tranzakciók bizonyos hozzáféréseit, amelyeket a zártábla nem rögzített volna. Az érvényesítéssel kapcsolatban egy lényeges probléma, hogy a tranzakcióhoz tartozó írási halmazt az írás elvégzése előtt kell már ismernünk (de a tranzakció helyi számításai befejeződése után).

Összehasonlíthatjuk a módszereket abból a szempontból is, hogy késleltetés nélkül befejeződnek-e a tranzakciók. A három módszer hatékonysága attól függ, hogy vajon a tranzakciók közötti *egymásra hatás* erős vagy gyenge (milyen valószínűséggel akar egy tranzakció hozzáférni egy olyan elemhez, amelyhez egy konkurens tranzakció már hozzáfért).

- A zárolás késlelteti a tranzakciókat, azonban elkerüli a visszagörgetéseket, még ha erős is az egymásra hatás. Az időbélyegzők és az érvényesítés nem késlelteti a tranzakciókat, azonban visszagörgetést okozhat, amely a késleltetésnek egy problémásabb formája, azonfelül erőforrásokat is pazarol.
- Ha gyenge az egymásra hatás, akkor sem az időbélyegzés, sem az érvényesítés nem okoz sok visszagörgetést, és előnyösebb lehet a zárolásnál, ugyanis ezeknek általában alacsonyabbak a költségei, mint a zárolási ütemezőnek.
- Amikor szükséges a visszagörgetés, az időbélyegzők hamarabb feltárják a problémákat, mint az érvényesítés, amelyek mindig hagyják, hogy a tranzakció elvégezze az összes belső munkáját, mielőtt megnézné, hogy vissza kell-e görgetni a tranzakciót.

9.9.4. Feladatok

9.9.1. feladat: A következő eseménysorozatokban, az $R_i(X)$ azt jelöli, hogy „T_i tranzakciót elindítjuk, melynek az olvasási halmaza az X adatbáziselemek listája". V_i azt jelenti, hogy „T_i megpróbálja az érvényesítést", és $W_i(X)$ azt jelenti, hogy „T_i befejeződik, és az írási halmaza az X". Mondjuk meg, mi történik, amikor minden sorozatot érvényesítésen alapuló ütemező hajt végre.

* a) $R_1(A, B)$; $R_2(B, C)$; V_1; $R_3(C, D)$; V_3; $W_1(A)$; V_2; $W_2(A)$; $W_3(B)$;
b) $R_1(A, B)$; $R_2(B, C)$; V_1; $R_3(C, D)$; V_3; $W_1(A)$; V_2; $W_2(A)$; $W_3(D)$;
c) $R_1(A, B)$; $R_2(B, C)$; V_1; $R_3(C, D)$; V_3; $W_1(C)$; V_2; $W_2(A)$; $W_3(D)$;
d) $R_1(A, B)$; $R_2(B, C)$; $R_3(C)$; V_1; V_2; V_3; $W_1(A)$; $W_2(B)$; $W_3(C)$;
e) $R_1(A, B)$; $R_2(B, C)$; $R_3(C)$; V_1; V_2; V_3; $W_1(C)$; $W_2(B)$; $W_3(A)$;
f) $R_1(A, B)$; $R_2(B, C)$; $R_3(C)$; V_1; V_2; V_3; $W_1(A)$; $W_2(C)$; $W_3(B)$;

9.10. Összefoglalás

- *Konzisztens adatbázis-állapotok:* A tervezők által megadott megszorításokat kielégítő adatbázis-állapotokat, legyenek azok implikáltak vagy deklaráltak, konzisztensnek nevezzük. Fontos, hogy a műveletek megőrizzék az adatbázis konzisztenciáját, vagyis az adatbázist konzisztens állapotból konzisztens állapotba vigyék.
- *Konkurens tranzakciók konzisztenciája:* Normálisan több tranzakció egyidejűleg fér hozzá az adatbázishoz. Az egymástól elszigetelten futó tranzakciókról feltételezzük, hogy megőrzik az adatbázis konzisztenciáját. Az ütemező feladata annak

biztosítása, hogy a konkurensen működő tranzakciók is megőrizzék az adatbázis konzisztenciáját.

- *Ütemezések:* A tranzakciók műveletekből állnak, többségében az adatbázis olvasásából és írásából. Egy vagy több tranzakció ilyen műveleteinek sorozatát ütemezésnek nevezzük.
- *Soros ütemezések:* Ha a tranzakciókat sorban egymás után, egyenként hajtjuk végre, akkor az ütemezést sorosnak mondjuk.
- *Sorba rendezhető ütemezések:* Az olyan ütemezést, amelynek az adatbázisra való hatása ekvivalens valamely soros ütemezéssel, sorba rendezhetőnek hívunk. Több tranzakciótól származó műveletek átlapolása előfordulhat egy sorba rendezhető ütemezésben, míg maga az ütemezés nem soros, ugyanakkor nagyon elővigyázatosaknak kell lennünk, milyen műveletsorozatokat engedélyezünk, különben az átlapolás az adatbázist nem konzisztens állapotba fogja átalakítani.
- *Konfliktus-sorbarendezhetőség:* Egyszerű teszt, amely a sorbarendezhetőségre elégséges feltétel, mely szerint az ütemezést konfliktus nélküli szomszédos műveletek cseréinek sorozatával sorossá alakíthatjuk át. Az ilyen ütemezést konfliktus-sorbarendezhetőnek nevezzük. Konfliktus fordulhat elő, ha megpróbáljuk ugyanannak a tranzakciónak két műveletét felcserélni, vagy két, ugyanahhoz az adatbáziselemhez hozzáférő műveletet cserélünk meg, amelyek közül legalább az egyik művelet írás.
- *Megelőzési gráf:* Könnyű teszt a konfliktus-sorbarendezhetőséghez, hogy elkészítjük az ütemezés megelőzési gráfját. A csomópontok a tranzakcióknak felelnek meg, a T-ből vezet él az U-ba, $T \rightarrow U$, ha az ütemezésben a T valamelyik művelete konfliktusban áll az U-nak egy későbbi műveletével. Egy ütemezés akkor és csak akkor konfliktus-sorbarendezhető, ha a megelőzési gráf körmentes.
- *Zárolás:* A legáltalánosabb megközelítés a sorba rendezhető ütemezések biztosításához, hogy zároljuk az adatbáziselemeket mielőtt hozzáférnénk, és feloldjuk a zárakat, miután befejeztük az elemhez való hozzáférést. A zárak megakadályozzák a többi tranzakció hozzáférését az adott elemhez.
- *Kétfázisú zárolás:* Önmagában a zárolás nem biztosítja a sorbarendezhetőséget. A kétfázisú zárolásban minden tranzakció előbb olyan fázisba lép, amikor csak zárakat igényel, és ezután lép olyan fázisba, amikor csak feloldja a zárakat. A kétfázisú zárolás biztosítja a sorbarendezhetőséget.
- *Zárolási módok:* Ahhoz, hogy elkerüljük a tranzakciók felesleges zárolásait, a rendszerek általában több zárolási módot is alkalmaznak, minden módhoz külön szabályok tartoznak, amelyekkel megadjuk, hogy mikor engedélyezhetjük a zárat. A legáltalánosabb rendszer csak olvasáshoz osztott zárakat, az írást is tartalmazó hozzáférésekhez pedig kizárólagos zárakat alkalmaz.
- *Kompatibilitási mátrixok:* A kompatibilitási mátrix hasznos összefoglalása annak, hogy mikor jogos egy bizonyos zármódú zárat engedélyeznünk, amikor ugyanabban vagy más módban, ugyanazon az elemen más zárak is adottak lehetnek.
- *Módosítási zárak:* Az ütemező lehetőséget nyújt arra, hogy egy tranzakció, amely előbb olvassa, és azután írja az elemet, előbb módosítási zárat helyezzen el, és később minősítse át ezt a zárat kizárólagossá. Módosítási zárat akkor is engedélyezhetünk, amikor már vannak osztott zárak az adott elemen, de ha egyszer kiadtunk egy módosítási zárat erre az elemre, ez megakadályozza, hogy más zárakat engedélyezzünk.

- *Növelési zárak:* Abban a speciális esetben, amikor egy tranzakció csak hozzáad vagy levon egy konstanst egy elemből, a növelési zár megfelelő. Ugyanannak az elemnek a növelési zárjai nem mondanak ellent egymásnak, viszont az osztott és kizárólagos zárakkal konfliktust alkotnak.
- *Szemcsézettség hierarchiájú elemek zárolása:* Amikor nagy és kis elemeket – relációkat, lemezblokkokat és esetleg sorokat – kell zárolnunk, a zárak figyelmeztető rendszere biztosítja a sorbarendezhetőséget. A tranzakciók szándékot kifejező zárakat helyeznek el a nagy elemekre, hogy figyelmeztessék a többi tranzakciót, hogy ennek egy vagy több részeléméhez való hozzáférést szándékoznak elvégezni.
- *Faelrendezésű elemek zárolása:* Ha az adatbáziselemekhez csak úgy tudunk hozzáférni, hogy egy fán haladunk lefelé, mint egy B-fa-indexen, akkor egy nem kétfázisú zárolási stratégiával tudjuk biztosítani a sorbarendezhetőséget. A szabályok szerint zárolnunk kell a szülőt, amikor a gyerekre igényelünk zárat, de később a szülőn való zár feloldható, és további zárakat is kiadhatunk.
- *Optimista konkurenciavezérlés:* A zárolás helyett az ütemező felteheti, hogy a tranzakciók sorba rendezhetők lesznek, és csak akkor abortálja a tranzakciót, ha valamilyen lehetséges nem sorba rendezhető viselkedést tapasztal. Ez a megközelítés, amelyet optimistának nevezünk, két részre oszlik, az időbélyegzőn alapuló és az érvényesítésen alapuló ütemezésre.
- *Időbélyegzőn alapuló ütemezők:* Az ütemezőnek ez a típusa időbélyegzőt rendel a tranzakciókhoz, amint elkezdődnek. Az adatbáziselemekhez hozzárendelt olvasási és írási idők az adott műveleteket legújabban végrehajtó tranzakcióknak az időbélyegzői. Ha egy lehetetlen helyzetet észlelne, mint például amikor egy tranzakció egy későbbi tranzakció által felülírt értéket olvasna be, a megsértő tranzakciót visszagörgeti, vagyis abortálja, és utána újraindítja.
- *Érvényesítésen alapuló ütemezők:* Ezek az ütemezők azután érvényesítik a tranzakciókat, miután beolvastak mindent, amire szükségük van, de még mielőtt írtak volna. Azok a tranzakciók, amelyek valamely más tranzakció írásának feldolgozása alatt álló elemet olvastak be, vagy fognak írni, kétes eredményhez vezethetnek, így ezeket a tranzakciókat nem érvényesítjük. Azokat a tranzakciókat, amelyeket nem érvényesítünk, visszagörgetjük.
- *Többváltozatú időbélyegzők:* A gyakorlatban elterjedt technika, hogy a csak olvasási tranzakciókat időbélyegzőkkel ütemezzük, de többszörös változatokkal. Vagyis egy elem írásakor nem írjuk felül az elem korábbi értékeit mindaddig, ameddig azok a tranzakciók be nem fejeződnek, amelyeknek esetleg szüksége lenne ezekre a korábbi értékre. Az írási tranzakciókat a hagyományos zárakkal ütemezzük.

9.11. Irodalomjegyzék

A [6] könyv az ütemezésről, valamint a zárolásról nyújt lényeges forrásanyagot. A [3] ugyanennek egy másik fontos forrása. A konkurenciavezérlés legújabb eredményei a [12]-ben és a [11]-ben találhatók.

Valószínűleg a legjelentősebb cikk a tranzakciófeldolgozásban a [4] a kétfázisú zárolásról. A figyelmeztető protokoll a szemcsézettség hierarchiákra az [5]-ből származik. A fák nem kétfázisú zárolása a [10]-ből ered. A kompatibilitási mátrixot a zárolási módok tanulmányozására a [7]-ben vezették be.

Az időbélyegzők mint konkurenciavezérlési módszerek a [2]-ben és az [1]-ben fordulnak elő. Az érvényesítésen alapuló ütemezés a [8]-ból származik. A többszörös változatok használatát a [9]-ben tanulmányozták.

1. P. A. Bernstein, Goodman, N., „Timestamp-based algorithms for concurrency controll in distributed database systems", *Proc. Intl. Conf. on Very Large Databases* (1980), pp. 285–300.
2. P. A. Bernstein, Goodman, N., Rothnie, J. B. Jr., Papadimitriou, C. H., „Analysis of serializability in SDD-1: a system of distributed databases (the fully redundant case)", *IEEE Trans. on Software Engineering* **SE-4**:3 (1978), pp. 154–168.
3. P. A. Bernstein, Hadzilacos, V., Goodman, N., *Concurrency Control and Recovery in Database Systems,* Addison-Wesley, Reading MA, 1987.
4. Eswaran, K. P., Gray, J. N., Lorie, R. A., Traiger, I. L., „The notions of consistency and predicate locks in a database system", *Comm. ACM* **19**:11 (1976), pp. 624–633.
5. Gray, J. N, Putzolo, F., Traiger, I. L, „Granularity of locks and degrees of consistency in a shared data base", in Nijssen, G. M. (ed.), *Modeling in Data Base Management Systems,* North Holland, Amsterdam, 1976.
6. Gray, J. N, Reuter, A., *Transaction Processing: Concepts and Techniques,* Morgan-Kaufmann, San Francisco, 1993.
7. Korth, H. F., „Locking primitives in a database system", *J. ACM* **30**:1, (1983), pp. 55–79.
8. Kung, H.-T., Robinson, J. T., „Optimistic concurrency control", *ACM Trans. on Database Systems* **6**:2 (1981), pp. 312–326.
9. Papadimitriou, C. H., Kanellakis, P. C., „On concurrency control by multiple versions", *ACM Trans. on Database Systems* **9**:1 (1984), pp. 89–99.
10. Silberschatz, A., Kedem, Z., „Consistency in hierarchical database systems", *J. ACM* **27**:1 (1980), pp. 72–80.
11. Thompsian, A., „Concurrency control: methods, performance, and analysis", *Computing Surveys* **30**:1 (1998), pp. 170–231.
12. Thuraisingham, B., Ko, H.-P., „Concurrency control in trusted database management systems: a survey", *SIGMOD Record* **22**:4 (1993), pp. 52–60.

10. fejezet

Bővebben a tranzakciókezelésről

Ebben a fejezetben a tranzakciókezelés olyan kérdéseiről lesz szó, amelyekkel a 8. és a 9. fejezetben nem foglalkoztunk. Megnézzük, hogyan egyeztethető össze az előző két fejezet nézőpontja: milyen kölcsönhatásban áll egymással a helyreállítás, a tranzakciók abortáltathatóságának és a sorbarendezhetőség fenntartásának a szükségessége? Ezután megtárgyaljuk a tranzakciók közötti holtpontkezelés lehetőségeit. Holtpont tipikusan abban az esetben alakul ki, amikor több tranzakciónak kell várnia egy olyan erőforrásra (például egy zárra), amely az adott pillanatban egy másik tranzakció birtokában van.

A fejezet során bevezetést nyerünk az osztott adatbázisok világába is. Közelebbről megvizsgáljuk az esetleg többszörözött példányok segítségével megosztott adatok zárolási problémáját. Azt a kérdést is áttekintjük, hogy mi alapján lehet dönteni egy olyan tranzakció abortálásáról, illetve véglegesítéséről, amely egyszerre több helyszínen is végez műveletet.

Végül tárgyalásra kerülnek a „hosszú tranzakciókból" eredő problémák. Léteznek olyan alkalmazások, például CAD[1] vagy „munkafolyamat" rendszerek, amelyekben az emberi és a számítógépes eljárások akár több napon keresztül is kölcsönhatásban vannak egymással. Hasonlóan a rövid tranzakciós rendszerekhez (banki műveletek, repülőjegy-foglalás) itt is szükség van az adatbázis-állapot konzisztenciájának megőrzésére. A 9. fejezetben bevezetett konkurenciavezérlő módszerek azonban nem működnek ésszerűen, amikor a zárak napokra vannak kiosztva, vagy az érvényesítési döntéseket több nappal ezelőtt végbement események kapcsán kell meghoznunk.

10.1. Tranzakciók, melyek nem véglegesített adatokat olvasnak

A 8. fejezetben néhány naplózási eljárással és ezeknek a rendszerhiba utáni helyreállításban betöltött szerepével ismerkedtünk meg. Az adatbázison végrehajtott számításokat olyan folyamatoknak tekintettük, amely során az értékek a nem felejtő lemez, a

[1] *Computer Aided Design,* magyarul Számítógéppel Támogatott Tervezés. *A fordító megjegyzése.*

felejtő központi memória és a tranzakció memóriaterülete között mozognak. A különféle naplózási módszerek segítségével egy esetleges rendszerhiba esetén biztosan helyreállíthatók voltak a véglegesített tranzakciók műveletei és ezek hatása az adatbázislemezen tárolt példányán. A naplózási rendszerek azonban nem tesznek kísérletet a sorbarendezhetőség támogatására; az adatbázis állapotát még akkor is visszaállítják az eredeti formájára, ha az nem sorba rendezhető műveletek eredményeképpen jött létre. Ami azt illeti, a forgalomban lévő adatbázisrendszerek sem mindig ragaszkodnak a sorbarendezhetőséghez, és néhányuk csak a felhasználó kifejezett kérésére tesz erre irányuló törekvéseket.

A 9. fejezetben viszont egyedül a sorbarendezhetőségről volt szó. A fejezet alapelvei szerint tervezett ütemező olyan dolgokat művelhet, amelyeket a naplóvezérlő nem tűrhet el. Például a sorbarendezhetőség definíciójában semmi sem gátolja meg a tranzakciót abban, hogy a véglegesítés előtt a megfelelő zár birtokában A-nak új értéket adjon az adatbázisban, és ezzel megsértse a naplózási politikát. Még rosszabb esetben a tranzakció az adatbázisba írás után abortál, ami könnyen eredményezhet inkonzisztens állapotot még rendszerhiba nélkül is, annak ellenére, hogy az ütemező elméletileg támogatja a sorbarendezhetőséget.

10.1.1. A piszkos adat probléma

Már volt arról szó, hogy az adatot „piszkosnak" nevezzük, ha egy olyan tranzakció írta, amely még nem fejeződött be. Piszkos adat feltűnhet a pufferben, a lemezen vagy mindkét helyen; ezek közül bármelyik problémát okozhat.

10.1. példa: Vegyük újra a 9.13. ábra sorba rendezhető ütemezését, de tegyük fel, hogy T_1 a B olvasása után valamilyen oknál fogva abortál. Ekkor a 10.1. ábrán látható események követik egymást. T_1 abortáltatása után az ütemező feloldja a B-re T_1-nek kiosztott zárat. Ez a lépés elkerülhetetlen, mert különben egyik tranzakció sem zárolhatná B-t.

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$;			
`A := A+100;`			
$w_1(A)$; $l_1(B)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$;		
	`A := A*2;`		
	$w_2(A)$;	250	
	$l_2(B)$ **Elutasítva**		
$r_1(B)$;			
Abort; $u_1(B)$;			
	$l_2(B)$; $u_2(A)$; $r_2(B)$;		
	`B := B*2;`		
	$w_2(B)$; $u_2(B)$;		50

10.1. ábra. *T_1 piszkos adatot ír, majd abortál*

T_2 azonban most olyan adatot olvasott, amely egy inkonzisztens adatbázisállapothoz tartozik. Azaz T_2 a T_1 által megváltoztatott A értéket és a T_1 működése előtti B-t olvasta. Ebben az esetben nem számít, hogy a T_1 által írt $A = 125$ érték kikerült-e a lemezre vagy sem, T_2 úgyis a pufferből kapja az értékeket. Az inkonzisztens állapot olvasása következtében T_2 inkonzisztens állapotban hagyja az adatbázist (a lemezen), ahol $A \neq B$.

A 10.1. ábra ütemezésében az okozza a problémát, hogy a T_1 által írt A piszkos, akár a pufferben van, akár a lemezen. Az a tény, hogy T_2 olvasta és a saját számításaihoz felhasználta A-t, megkérdőjelezi a tranzakció helyes működését. A 10.1.2. részben látni fogjuk, hogy ha ilyen eset előfordulhat, akkor T_2-t és T_1-et is abortáltatni kell, és vissza kell görgetni (rollback). Ha még azt is megengedjük, hogy a T_1 által írt A érték a lemezen is megjelenjen, az elég sokba kerül, hiszen a változtatás meg nem történtté tevéséhez a naplót kell felhasználnunk. Ezért olyan szabályokat kell kidolgoznunk, amelyek segítségével megelőzhető, hogy a piszkos adat kikerüljön a lemezre. □

10.2. példa: Most nézzük a 10.2. ábrán látható eseménysorozatot, amely a 9.8. részben megismert időbélyegző alapú ütemező felügyelete alatt játszódik le. Tegyük fel azonban, hogy ez az ütemező nem használja a 9.8.1. részben bevezetett véglegesítés bitet. Abban a részben láthattuk, hogy ennek a bitnek a segítségével megelőzhető, hogy egy tranzakció olyan adatot olvasson, amelyet egy még folyamatban lévő másik tranzakció írt. Így, amikor a második lépésben T_1 B-t olvassa, nincs véglegesítés bit ellenőrzés, ami késleltethetné T_1-et. T_1 továbbléphet, akár a lemezre is írhat, és véglegessé válhat. B olvasása utáni műveleteit nem tüntettük fel az ábrán.

Végül T_2 fizikailag nem megvalósítható módon próbálja C-t olvasni, ezért abortál. T_2-nek a B-n végrehajtott változtatásait érvénytelenítjük, B értékét és írási idejét visszaállítjuk arra, amit T_2 felülírt. Mindezek ellenére T_1 hozzáférhetett B érvénytelen értékéhez, és ezt bármire felhasználhatja, például új értékeket számolhat A-nak, B-nek és/vagy C-nek, és ezeket kiírhatja a lemezre. Vagyis mivel T_1 B piszkos értékét olvasta, inkonzisztens állapotba hozta az adatbázist. Vegyük észre, hogyha használtuk volna a véglegesítés bitet, akkor a 2. lépésben B olvasását késleltethettük volna egészen addig, amíg T_2 nem abortál, és az általa írt B értéket vissza nem állítjuk az előző (feltehetően véglegesített) értékére. □

T_1	T_2	T_3	A	B	C
200	150	175	RT = 0	RT = 0	RT = 0
			WT = 0	WT = 0	WT = 0
	$w_2(B)$;			WT = 150	
$r_1(B)$;					
	$r_2(A)$;		RT=150		
		$r_3(C)$;			RT = 175
	$w_2(C)$;				
	Abort;			WT = 0	
		$w_3(A)$;	WT = 175		

10.2. ábra. *T_1 a T_2-ből olvas piszkos adatot, ezért T_2-vel együtt őt is abortáltatni kell*

Az elkülönítés szintjei SQL-ben

Az SQL2-szabvány nem teszi fel, hogy a tranzakciók sorba rendezhető módon futnak, ennek megfelelően a forgalomban lévő rendszerek a felhasználóra bízzák a kívánt konkurenciavezérlési szint beállítását. SQL2-ben a legmagasabb „elkülönítési szint" a *sorba rendezhető* (serializable), amely pontosan azt jelenti, amit a szó takar: ezen a szinten a tranzakciónak úgy kell lefutnia, mintha egy pillanat alatt történt volna, és mintha minden más tranzakció teljes egészében vagy előtte, vagy utána ment volna végbe.

Az „olvasásbiztos" elkülönítési szint nem követeli meg a sorbarendezhetőséget, viszont megtiltja a piszkos adat olvasását. Lehetséges azonban, hogy ha ezen a szinten egy T tranzakció kétszer olvassa A-t, akkor két különböző értéket kap, amelyeket két különböző véglegesített tranzakció írt. Ehhez kapcsolódik a „megismételhető olvasás" szintje, amely egy kicsit erősebb feltételt szab: nemcsak hogy ha T olvassa A-t, akkor A nem lehet piszkos, de ha A egy reláció, akkor A minden további olvasásánál csak bővebb részhalmazt kaphatunk; vagyis amíg T aktív, A-nak semmilyen része sem tűnhet el.

A negyedik, „nem olvasásbiztos" szinten semmilyen megszorítás sincs arra vonatkozólag, hogy a tranzakció milyen adatokhoz férhet hozzá. Csak ez a szint engedi meg a piszkos adat olvasását.

10.1.2. Továbbgyűrűző visszagörgetés

A fenti példákból látszik, hogy ha a tranzakciók hozzáférhetnek piszkos adatokhoz, akkor néha végre kell hajtanunk egy *továbbgyűrűző visszagörgetést* (cascading rollback). Ez azt jelenti, hogy amikor T abortál, meg kell határoznunk, hogy milyen tranzakciók olvasták a T által írt adatokat, ezeket abortáltatnunk kell, és rekurzívan az összes olyan tranzakciót is, amelyek az ezek által írt adatokat olvasták. Tehát meg kell találnunk minden olyan U tranzakciót, amely a T által írt piszkos adato(ka)t olvasta, abortáltatnunk kell U-t, majd meg kell találnunk minden olyan V tranzakciót, amely az U által írt piszkos adatot olvasta, abortáltatnunk kell V-t és így tovább. Hogy az abortáltatott tranzakció hatását érvénytelenítsük az adatbázison, használhatjuk a naplót, ha megtalálhatók benne a változók előző értékei (semmisségi vagy semmisségi/helyrehozó naplózás). Ha a piszkos adat hatása még nem érte el a lemezt, akkor az adatok helyreállításához az adatbázis lemezen található változata is megfelelő alapot nyújt. Ezekről a módszerekről részletesebben a következő részben lesz szó.

Mint azt már megjegyeztük, az időbélyegző alapú ütemező a véglegesítés bit használatával megakadályozza az olyan tranzakció továbblépését, amely piszkos adatot olvasott, azaz ebben az esetben nem lesz szükség visszagörgetésre. Az érvényesítés alapú ütemező is elkerüli a továbbgyűrűző visszagörgetést, hiszen az adatbázisba írás (sőt már a pufferbe írás is) csak azután történik meg, miután az ütemező megállapította, hogy a tranzakció végleges lesz.

10.1.3. A visszagörgetés kezelése

Most nézzük meg, hogyan kezelhető a továbbgyűrűző visszagörgetés problémája a zárolás alapú ütemezők esetén. Egyszerűen biztosítható, hogy ne legyen szükség visszagörgetésre:

- *Szigorú zárolás:* A tranzakció egészen addig nem engedheti el az írás zárjait (illetve az olyan típusú zárjait, például növelési zár, amelyek birtokában megváltoztathatja az adat értékét), amíg nem lett végleges vagy nem abortált, és az ennek megfelelő naplóbejegyzés nem került ki a lemezre.

Vegyük észre, hogy a feltétel, amely szerint a tranzakció befejezésére vonatkozó feljegyzés lemezre írása megelőzi a zárakról való lemondást, biztosítja a következőt: ha rendszerhiba következik be, miután a T tranzakció eleresztette a zárjait, és egy másik tranzakció olvasta a T által írt adatokat, akkor ezek az adatok nem lehetnek piszkosak amiatt, hogy a helyreállító eljárás abortáltatja T-t. A tranzakciók olyan ütemezését, amely megfelel a szigorú zárolás szabályának, *helyreállíthatónak* (recoverable) nevezzük.

Világos, hogy egy helyreállítható ütemezésben a tranzakciók nem olvashatnak piszkos adatot, hiszen egy még folyamatban lévő tranzakció által pufferbe írt adat egészen addig zár alatt marad, amíg a tranzakció be nem fejeződik. A pufferben lévő adatok helyreállítása azonban még mindig problémát jelent abban az esetben, ha a tranzakció abortál, hiszen a végrehajtott változtatások hatását érvényteleníteni kell. Hogy ez mennyire bonyolult, az attól is függ, hogy az adatbáziselemek blokk méretűek vagy annál kisebbek. Most mindkét esetet megvizsgáljuk.

Blokkok visszagörgetése

Ha a zárolható adatbáziselemek blokkok, akkor létezik a visszagörgetésnek egy egyszerű módja, amelyhez nem kell a naplót felhasználnunk. Tegyük fel, hogy T tranzakció rendelkezik az A blokk kizárólagos zárjával. A új értékét beírta a pufferbe, majd abortálnia kellett. Mivel A zárolva volt, mióta T felülírta, más tranzakció nem férhetett hozzá az értékéhez. Könnyű visszaállítani A régi értékét, ha a következő szabály szerint járunk el:

- A még folyamatban lévő tranzakciók által írt blokkok a központi memóriában vannak csatolva, ami azt jelenti, hogy ezeket a puffereket nem írhatjuk ki a lemezre.

Ilyenkor T abortálása után a „visszagörgetés" annyiból áll, hogy utasítjuk a pufferkezelőt, hogy ne vegye figyelembe A értékét. Vagyis az A által elfoglalt puffer nem íródik ki sehova, hanem az elérhető „szabad" pufferek közé kerül. Biztosak lehetünk benne, hogy A értéke a lemezen a legutóbbi végleges tranzakció eredménye, amely pontosan az, amit szeretnénk.

Ha a 9.8.5. és 9.8.6. részben megismert többverziós időbélyegzős rendszert használjuk, akkor is könnyű dolgunk van a visszagörgetéssel. Ebben az esetben is fel kell tennünk, hogy a még folyamatban lévő tranzakciók által írt blokkok a memóriában csatolva vannak. Ekkor az A értékeit tartalmazó listából egyszerűen töröljük azt, amely T hatására jött létre. Vegyük észre, hogy mivel T értékváltoztató tranzakció volt, ezért az általa írt A érték egészen addig el volt zárva a többi tranzakció elől, amíg T nem abortált (feltéve, hogy a 9.8.6. rész időbélyegző/zár sémáját használtuk).

Kis adatbáziselemek visszagörgetése

Amikor a zárolható adatbáziselemek nem egész blokkok, csak blokkrészletek (például sorok vagy objektumok), akkor az abortált tranzakciók által megváltoztatott pufferek helyreállítása egyszerű módszerrel nem oldható meg. Az a probléma ugyanis, hogy egy pufferben több adatelem is megtalálható, és ezeket két vagy több tranzakció is írhatja, vagyis ha ezek közül az egyik abortál, a többi által véghezvitt változtatásokat még meg kell őriznünk. Egy abortált tranzakció által írt kis A adatbáziselem régi értékének visszaállítására több lehetőségünk is van.

1. A eredeti értékét a pufferben a lemezen található adatbázis alapján állítjuk vissza.
2. Semmisségi vagy semmisségi/helyrehozó naplózás esetén A előző értékét a napló alapján állítjuk vissza.
3. Készíthetünk minden tranzakcióhoz a központi memóriában egy külön naplót arra az időre, amíg a tranzakció aktív. Ebbe a tranzakciók által végrehajtott változtatásokat jegyezzük fel, tehát A régi értéke is megtalálható a megfelelő „naplóban".

Egyik módszer sem ideális. Az elsőben biztosan a lemezen tárolt információhoz kell fordulnunk. A második esetben, ha a napló megfelelő része még mindig a pufferben van, akkor lehet, hogy nem kell a lemezhez fordulnunk. Rosszabb esetben azonban alaposan át kell vizsgálnunk a lemezen tárolt naplót, hogy az A-ra vonatkozó bejegyzést megtaláljuk. Az utolsó módszerrel nem kell a lemezről olvasnunk, viszont a központi memória „naplók" sok helyet foglalhatnak a memóriában.

10.1.4. Csoportos véglegesítés

Bizonyos körülmények között akkor is elkerülhető a piszkos adat olvasása, ha nem írjuk ki azonnal a lemezre a naplóban található commit rekordokat. Ha a naplóbejegyzéseket abban a sorrendben írjuk ki a lemezre, ahogy a naplóban szerepelnek, akkor amint a napló pufferben lévő részébe kerül egy commit rekord, feloldhatjuk a megfelelő zárakat.

10.3. példa: Tegyük fel, hogy T_1 tranzakció az X írása után véglegessé válik, kiírja a commit rekordját a naplóba, de ez a pufferben marad. Bár T_1 nem végleges abban az

Mikor végleges igazán egy tranzakció?

A csoportos véglegesítés kifinomultságáról eszünkbe juthat, hogy egy lezárult tranzakció több különböző állapotban lehet a műveletei befejezése és a valódi „végleges" állapota között, ahol ez utóbbi azt jelenti, hogy a tranzakció hatása semmilyen körülmények között, még a rendszer összeomlása esetén sem vész el. Ahogy a 8. fejezetben megjegyeztük, lehetséges, hogy egy tranzakció véget ér, és még a `COMMIT` feljegyzés is bekerül a központi memória egy pufferébe, de ha ez nem éri el a lemezt, rendszerhiba esetén mégis elvész a tranzakció hatása. Sőt a 8.5. részben azt láttuk, hogy ha a `COMMIT` rekord már el is érte a lemezt, de mentés még nem készült róla, az adathordozó hibája esetén a tranzakció ugyanúgy semmissé válhat, és elveszthetí a hatását.

Ha a hiba lehetőségét kizárjuk, ezek az állapotok mind ekvivalensek abban az értelemben, hogy minden tranzakció a lezárulását követően biztosan eléri a tartósság szintjét, azaz még az adathordozó hibája esetén sem vész el a hatása. Ha azonban a hibákat és ezek helyreállítását is számításba kell vennünk, fontos felismernünk a fenti állapotok közti különbséget, amelyekre egyébként mindre „véglegesként" hivatkozhatnánk.

értelemben, hogy a commit rekordja túlélne egy rendszerhibát, mégis feloldjuk a zárjait. Ezután T_2 olvassa X-et és „véglegessé válik", de a commit rekord, amely T_1 commit rekordját követi a naplóban, szintén a pufferben marad. Mivel a naplóbejegyzések az eredeti sorrendben kerülnek ki a lemezre, a helyreállítás-kezelő csak akkor foghatja fel T_2-t végleges tranzakciónak (a lemezen talált commit rekord következtében), ha T_1-et is véglegesként kezeli. Így a helyreállítás-kezelő szempontjából három eset különböztethető meg:

1. Sem T_1, sem T_2 commit rekordja nem kerül ki a lemezre. Ekkor a helyreállítás-kezelő mindkettőt abortáltatja, és az a tény, hogy T_2 a le nem zárult T_1 által írt X-et olvasta, lényegtelen.
2. T_1 végleges, T_2 nem. Két okból kifolyólag sincs probléma: T_2 tranzakció X-et egy véglegesített tranzakció után olvasta, vagyis X nem volt piszkos, másrészt T_2-t egyébként is abortáltatjuk az adatbázisra való hatás nélkül.
3. Mindkettő végleges. Ekkor T_2 nem piszkos adatot olvasott.

Ezzel szemben tegyük fel, hogy a T_2 commit rekordját tartalmazó puffer kikerült a lemezre (például, mert a pufferkezelő ezt a puffert valami másra akarta használni), T_1 commit rekordja viszont nem. Ha ezen a ponton a rendszer összeomlik, a helyreállítás-kezelő számára úgy fog tűnni, mintha T_2 véglegesített lett volna, T_1 pedig nem. T_2 hatása végleges marad az adatbázison, annak ellenére, hogy ez az X piszkos adat olvasásán alapult. □

Konkurencia és helyreállíthatóság

Talán feltűnt, hogy a 8. fejezetben megismert három naplózási módszer nem mindig felel meg a helyreállítható ütemezés követelményeinek. Az olyan rendszereket, amelyek mind a konkurencia, mind a helyreállíthatóság elvárásainak megfelelnek, gyakran nevezzük ACID- (atomosság, konzisztencia, elkülönítés, tartósság) rendszereknek (lásd az 1.2.4. rész idevágó részeit). Itt a megfelelő alkalom tehát, hogy megjegyezzük: a naplózás és a zárolás (vagy akármilyen más konkurenciavezérlő módszer) egymás ellen ható mechanizmusok.

Például lehet egy olyan rendszerünk, amely naplózza a tranzakciók műveleteit, hogy biztosítsa az atomosságukat, még akkor is, ha semmi szükségünk a sorbarendezhetőségre vagy a piszkos adatok olvasásának az elkerülésére. Ami ezt illeti, a forgalomban lévő rendszerek gyakran a felhasználóra bízzák, hogy igényli-e a tranzakciók sorbarendezhetőségét, illetve a piszkos olvasás megakadályozását; ahogy ezt a 10.1.1. részben az SQL2 „elkülönítési szintjeit" tárgyaló dobozban is említettük. Megfordítva, a megszokott ABKR-ekben lehetőség van a naplózás kikapcsolására, de ezzel együtt az összes vagy néhány tranzakció futása sorba rendezhető. Ennek az a következménye, hogy egy esetleges rendszerhiba esetén a sorbarendezhetőség és az adatbázis konzisztenciája nincs biztosítva, de a felhasználónak esetleg más módszerei is lehetnek a konzisztencia visszaállítására.

A 10.3. példa alapján levonhatjuk azt a következtetést, hogy a zárakat még azelőtt feloldhatjuk, mielőtt a tranzakció commit rekordja a lemezen megjelenne. Ez az eljárás, amit gyakran *csoportos véglegesítésnek* (group commit) nevezünk, a következőkben foglalható össze:

- Ne oldjuk fel a zárakat, amíg a tranzakció be nem fejeződik, és amíg a commit naplóbejegyzés legalább a pufferben fel nem tűnik.
- A naplóblokkokat abban a sorrendben írjuk ki a lemezre, amelyben keletkeztek.

A csoportos véglegesítés elve éppúgy, mint a 10.1.3. részben tárgyalt „helyreállítható ütemezés" elve, biztosítja, hogy piszkos adat olvasására soha ne kerüljön sor.

10.1.5. Logikai naplózás

A 10.1.3. részben láthattuk, hogy a „piszkos olvasásokat" könnyebb megelőzni, illetve ha az olvasás már megtörtént, akkor könnyebb kijavítani abban az esetben, ha a zárolható egységek blokkok, illetve lapok. Azonban még ekkor is legalább két probléma adódik:

1. Minden naplózási módszer számon tartja az adatbáziselemek régi és/vagy új értékét. Ha egy blokkon belül csak kis változtatás történik, például egy sor egy mezőjét átírjuk, vagy beszúrunk/törlünk egy sort, akkor a naplóba nagy mennyiségű redundáns adat kerül be.
2. Az a követelmény, hogy az ütemezés helyreállítható legyen, azaz a zárak feloldása csak a tranzakció véglegesítése után következzen be, a konkurenciát komolyan gátolhatja. Például emlékezzünk vissza a 9.7.1. részre, ahol éppen a korai zárfeloldás előnyeit tárgyaltuk B-fa-index használata esetén. Ha megköveteljük, hogy a zárakat egészen a tranzakció véglegesítéséig fenn kell tartani, akkor ez az előny nem használható ki, és gyakorlatilag egyszerre csak egy értékváltoztató tranzakciónak engedjük meg, hogy a B-fához hozzáférjen.

Mindkét probléma indokolja a *logikai naplózás* használatát, ahol csak a blokkok változtatásai vannak feltüntetve. A változtatás természetétől függően a bonyolultságnak különböző fokai lehetnek.

1. Az adatbáziselemeknek csak néhány bájtján változtatunk, például egy rögzített hosszúságú mezőt módosítunk. Ebben a helyzetben egy kézenfekvő módszer, ha csak a megváltoztatott bájtokat és azok pozícióját jegyezzük fel. A 10.4. példában bemutatjuk ezt az esetet, és a módosítást leíró naplórekord megfelelő formáját.
2. Az adatbáziselemen végrehajtott változtatás egyszerűen leírható és könnyen visszaállítható, de az adatbáziselem legtöbb/összes bájtját érinti. A 10.5. példában egy gyakori szituáció kerül elő, ahol egy változó hosszú mezőn változtatunk, és a mező rekordjait és még más rekordokat is el kell csúsztatnunk a blokkon belül. A blokk új és régi értéke nagyon különböző lehet, ha nincs a tudatunkban, és nem jelöljük a változás egyszerű okát.
3. Az adatbáziselemen sok bájtot érintő változtatást hajtunk végre, és a további módosítások megakadályozhatják a meg nem történtté tevését. Ez valódi „logikai" naplózás, mert a semmisségi/helyrehozó folyamatot még csak nem is az adatbáziselemeken, hanem az általuk képviselt magasabb szintű „logikai" struktúrán keresztül látjuk. A 10.6. példában B-fák (a lemezblokkoknak mint adatbáziselemeknek megfelelő logikai struktúra) segítségével szemléltetjük a logikai naplózásnak ezt a komplex formáját.

10.4. példa: Tegyük fel, hogy az adatbáziselemek olyan blokkok, amelyek mindegyike tartalmaz néhány sort valamilyen relációból. Egy attribútum módosítását egy ehhez hasonló naplóbejegyzéssel fejezhetjük ki: „t sor a attribútumát v_1-ről v_2-re változtattuk". Ha a blokk egy üres részére beszúrunk egy új sort, azt a következőképpen fejezhetjük ki: „a p eltolási érték pozíciótól kezdve beszúrtunk egy t sort az $(a_1, a_2, \ldots, a_k)$ értékkel". Ha a módosított attribútum vagy a beszúrt sor méretben nem összemérhető a blokk nagyságával, akkor ezek a feljegyzések sokkal kisebb helyet foglalnak el, mint maga a blokk. Ezenfelül a semmissé tevő és helyrehozó műveletek végrehajtását is támogatják.

Vegyük észre, hogy a naplóba feljegyzett mindkét művelet idempotens: ha egy blokkon többször is végrehajtjuk őket, ugyanazt kapjuk, mintha csak egyszer alkal-

maztuk volna a műveletet. Hasonlóan az inverz műveleteik ($t[a]$ visszaállítása v_2-ről v_1-re, illetve t sor törlése) is idempotensek. Vagyis az ilyen típusú bejegyzések a helyreállítás során pontosan ugyanúgy használhatók, mint a 8. fejezetben használt módosítást leíró rekordok. □

10.5. példa: Ismét tegyük fel, hogy az adatbáziselemek sorokat tartalmazó blokkok, de most a sorok valamilyen változó hosszúságú mezőt is tartalmaznak. Ha a 10.4. példában említett módosításhoz hasonlóra kerül sor, akkor lehet, hogy a blokk nagy részét el kell csúsztatnunk ahhoz, hogy a hosszabb mezőnek helyet biztosítsunk, vagy hogy helyet takarítsunk meg, ha a mező megrövidül. Rendkívüli esetben túlcsordulásblokkot kell létrehoznunk az eredeti blokk egy részének tárolására (lásd 3.5. részt), illetve a túlcsordulásblokkot el kell távolítanunk, ha egy rövidebb mező miatt lehetővé válik két blokk egyesítése.

Ha a blokkot a túlcsordulásblokkjaival együtt egy adatbáziselem részeként tekintjük, akkor kézenfekvő a módosított mező régi és/vagy új értékeinek a használata a változtatás meg nem történtté tevéséhez illetve újra elvégzéséhez. A blokk-plusz-túlcsordulásblokk(ok)ra azonban úgy kell gondolnunk, mintha az adott sorokat „logikai" szinten tartalmaznák. Talán nem is leszünk képesek a blokkok bájtjait az eredeti állapotukba visszaállítani egy semmissé tevés vagy helyrehozás után, hiszen olyan módosítások miatt, amelyek más mezők hosszúságát változtatták, a blokkok újraszervezésére is sor kerülhetett. Ha viszont az adatbáziselemekre úgy gondolunk, mint blokkok gyűjteményére, amelyek együtt jelentenek bizonyos sorokat, akkor a helyrehozás, illetve semmissé tevés csakugyan visszaállítja az elem logikai „állapotát". □

Nem mindig lehet azonban a blokkokat a túlcsordulásblokkok mechanizmusán keresztül kiterjeszthető egységként kezelni, mint ahogy ezt a 10.5. példában láttuk. Ezért lehet, hogy a műveletek semmissé tevésére, illetve helyrehozására csak a blokkoknál magasabb szinten van lehetőség. A következő példa a B-fa-indexek fontos esetét tárgyalja, ahol a blokkok kezelése nem teszi lehetővé a túlcsordulásblokkok alkalmazását, és a semmissé tevés, illetve helyrehozás műveletekre úgy kell gondolnunk, mintha a blokkok helyett a B-fa logikai szintjén történnének meg.

10.6. példa: Nézzük meg, hogyan lehetne a B-fán történt változtatásokat a logikai naplózás módszerével nyilvántartani. Az egész csúcspont (blokk) régi és/vagy új értéke helyett csak egy rövid bejegyzést írunk a naplóba, amely a következő módosítások valamelyikét írja le:

1. Kulcs-mutató pár beszúrása vagy törlése a leszármazott (gyerek) csúcspontba vagy csúcspontból.
2. A mutatóhoz rendelt kulcs megváltoztatása.
3. Csúcspontok szétválasztása vagy egyesítése.

A fenti módosítások mindegyike leírható egy rövid naplóbejegyzéssel. Még a szétválasztás műveletéhez is csak azt kell megmondanunk, hogy melyik csúcs osztódik és

melyek az új csúcsok. Hasonlóan az egyesítés műveleténél is csak az érintett csúcspontokat kell megadnunk, hiszen az egyesítés módját a használt B-fa-kezelő algoritmus határozza meg.

A logikai módosító rekordok használatával lehetőségünk nyílik arra, hogy a zárakat korábban feloldjuk, mint ahogy erre egy helyreállítható ütemezés szerint sor kerülhetne. Ennek az az oka, hogy a piszkos B-fa-blokkok olvasása nem jelent problémát a tranzakció részéről, feltéve, hogy az olvasás egyetlen célja a tranzakció számára szükséges adatok helymeghatározása.

Tegyük fel például, hogy T tranzakció az N levelet olvassa, de az U tranzakció, amely N-t utoljára írta, abortál, ezért az N-en történt változtatásokat (például az U által beszúrt sor miatt az N-be beszúrtunk egy új kulcs-mutató párt) érvényteleníteni kell. Ha T is beszúrt egy kulcs-mutató párt N-be, akkor N-et már nem lehet az U működése előtti állapotába visszavinni. U-nak az N-re gyakorolt hatását azonban érvényteleníteni tudjuk: ebben a példában az U által beszúrt kulcs-mutató párt töröl-nénk. Az így kapott N persze nem lesz ugyanaz, mint ami U előtt volt, hiszen tartalmazni fogja a T által beszúrt kulcs-mutató párt. Az adatbázis azonban konzisztens maradt, mivel a B-fa továbbra is csak a végleges tranzakciók hatását tükrözi. A B-fát tehát visszaállítottuk logikai szinten, de fizikai szinten nem. □

10.1.6. Feladatok

* **10.1.1. feladat:** Hogyan lehetne az

$r_1(A)$; $r_1(B)$; $w_1(A)$; $w_1(B)$;

műveletsorozatba a 9.3. részben megismert egyszerű zárolásokat beilleszteni úgy, hogy a T_1 tranzakció alkalmazza:

a) A kétfázisú zárolás és a szigorú zárolás módszerét?
b) A kétfázisú zárolás módszerét, de ne alkalmazzon szigorú zárolást?

Adjuk meg az összes lehetséges megoldást!

10.1.2. feladat: Tegyük fel, hogy a következő ütemezések mindegyike a T tranzakció abortálásával zárul. Mely tranzakciókat kell visszagörgetnünk?

* a) $r_1(A)$; $r_2(B)$; $w_1(B)$; $w_2(C)$; $r_3(B)$; $r_3(C)$; $w_3(D)$;
b) $r_1(A)$; $w_1(B)$; $r_2(B)$; $w_2(C)$; $r_3(C)$; $w_3(D)$;
c) $r_2(A)$; $r_3(A)$; $r_1(A)$; $w_1(B)$; $r_2(B)$; $r_3(B)$; $w_2(C)$; $r_3(C)$;
d) $r_2(A)$; $r_3(A)$; $r_1(A)$; $w_1(B)$; $r_3(B)$; $w_2(C)$; $r_3(C)$;

10.1.3. feladat: Vegyük a 10.1.2. feladat ütemezéseit, de most tegyük fel, hogy mindhárom tranzakció véglegessé válik, és az utolsó műveletük után rögtön be is írják a

naplóba a commit rekordjukat. A rendszer összeomlása miatt azonban a napló vége már nem kerül ki a lemezre, így az utolsó néhány rekord elvész. Válaszoljunk a következő kérdésekre attól függően, hogy a napló mekkora része élte túl a rendszerhibát!

i) Mely tranzakciók nem tekinthetők véglegesnek?
ii) Létrejön-e piszkos olvasás a helyreállítás közben? Ha igen, mely tranzakciókat kell visszagörgetni?
iii) Milyen további piszkos olvasások jöhettek volna létre, ha a bejegyzések nem a napló végéről, hanem valahonnan a közepéről vesztek volna el?

10.2. Nézet-sorbarendezhetőség

A 9.1.4. részben azt mondtuk, hogy az ütemezők tervezésekor az a valódi célunk, hogy csak sorba rendezhető ütemezések jöhessenek létre. Azt is láttuk, hogy egy ütemezés sorbarendezhetősége a tranzakciók adatokon végzett műveleteitől is függhet. A 9.2. részben pedig azt is megtudtuk, hogy az ütemezők rendszerint a „konfliktus-sorbarendezhetőséget" juttatják érvényre, amely a tranzakciók adatokon végrehajtott módosításaira való tekintet nélkül garantálja a sorbarendezhetőséget.

Azonban a konfliktus-sorbarendezhetőségnél gyengébb feltételek is biztosíthatják a sorbarendezhetőséget. Ebben a részben egy ilyen feltétellel, a „nézet-sorbarendezhetőséggel" foglalkozunk. A nézet-sorbarendezhetőség veszi az összes olyan esetet T és U tranzakció kapcsán, amikor T írja az U által olvasott adatbáziselemet. A kétféle sorbarendezhetőség közötti lényeges különbség akkor jön elő, amikor T tranzakció írja A-t, de azt az értéket nem olvassa más tranzakció (mert később egy másik tranzakció felülírja A-t). Ilyenkor a $w_T(A)$ művelet az ütemezés egyéb olyan pozícióira is kerülhet, ahol éppúgy sohasem olvassák, viszont a konfliktus-sorbarendezhetőség definíciója mellett nem lennének megengedhetők. Ebben a részben pontosan definiáljuk a nézet-sorbarendezhetőség fogalmát, majd megadunk egy eljárást, amely segítségével ellenőrizhető, hogy egy adott ütemezés sorba rendezhető-e ilyen értelemben.

10.2.1. Nézetekvivalencia

Tegyük fel, hogy ugyanazokhoz a tranzakciókhoz van két ütemezés: S_1 és S_2. Az ütemezések legelejére, illetve legvégére vegyünk fel további két hipotetikus tranzakciót, T_0-t, illetve T_f-t. Úgy képzeljük, hogy a többi tranzakció által olvasott adatbáziselemek kezdeti értékét T_0 írta, az általuk írt értékeket pedig T_f olvassa az ütemezés végén. Ekkor minden $r_i(A)$ olvasáshoz megadható az a $w_j(A)$ írásművelet, amely megelőzi $r_i(A)$-t, és a legközelebb esik hozzá.[2] Ilyenkor a T_j tranzakciót az $r_i(A)$ olvasás *forrásának* nevezzük. Vegyük észre, hogy T_j lehet a hipotetikus T_0 kezdeti tranzakció, illetve T_i lehet a T_f tranzakció is.

[2] Bár az eddigiekben nem akadályoztuk meg a tranzakciókat abban, hogy ugyanazt az elemet kétszer is írják, erre általában nincs szükségük. A továbbiakban hasznos feltennünk, hogy a tranzakciók az egyes elemeket csak egyszer írják.

Ha az olvasásműveletek forrása mindkét ütemezésben ugyanaz, akkor azt mondjuk, hogy az S_1 és az S_2 ütemezések nézetekvivalensek. Kétségtelen, hogy a nézetekvivalens ütemezések valóban ekvivalensek: akármilyen adatbázis-állapoton hajtjuk őket végre, azonos lesz a hatásuk. Ha az S ütemezés nézetekvivalens egy soros ütemezéssel, akkor S-t *nézet-sorbarendezhető* ütemezésnek nevezzük.

10.7. példa: Vegyük a következő S ütemezést:

T_1:			$r_1(A)$		$w_1(B)$				
T_2:	$r_2(B)$	$w_2(A)$				$w_2(B)$			
T_3:				$r_3(A)$			$w_3(B)$		

A tranzakciók műveleteit most függőleges irányban tagoltuk, hogy jobban lehessen követni, melyik tranzakció mit csinál; az ütemezés a megszokott módon balról jobbra olvasandó.

S-ben T_1 és T_2 is írja B-t, de ezek az értékek elvesznek (felülíródnak), csak a T_3 által írt B érték éri meg az ütemezés végét, ahol a hipotetikus T_f tranzakció „kiolvassa". S nem konfliktus-sorbarendezhető. Hogy lássuk, miért, először vegyük észre, hogy T_2 azelőtt írja A-t, mielőtt azt T_1 olvasná, azaz egy feltételes konfliktus ekvivalens soros ütemezésben T_2-nek meg kell előznie T_1-et. Az a körülmény viszont, hogy a $w_1(B)$ művelet megelőzi $w_2(B)$-t, azt vonja maga után, hogy a feltételes konfliktus ekvivalens soros ütemezésben T_1-nek kell megelőznie T_2-t. Pedig sem $w_1(B)$-nek, sem $w_2(B)$-nek nincs hosszú távú hatása az adatbázisra. A nézet-sorbarendezhetőség az ilyen típusú, lényegtelen írásműveleteket hagyja figyelmen kívül, amikor megállapítja, hogy melyek azok a tényleges megszorítások, amelyekhez igazodnia kell egy ekvivalens soros ütemezésnek.

Formálisabban, tekintsük S-ben az egyes olvasásműveletek forrásait:

1. $r_2(B)$ forrása T_0, mivel ezelőtt nem írjuk B-t S-ben.
2. $r_1(A)$ forrása T_2, mivel az olvasás előtt T_2 írta legutóbb A-t.
3. Hasonlóan, $r_3(A)$ forrása T_2.
4. A hipotetikus $r_{T_f}(A)$ művelet forrása T_2.
5. A hipotetikus $r_{T_f}(B)$ művelet forrása T_3, amely legutoljára írta B-t.

Természetes, hogy T_0, illetve T_f a valódi tranzakciók előtt, illetve után tűnik fel, akármilyen ütemezést is veszünk. Ha a valódi tranzakciók (T_2, T_1, T_3) sorrendjét vesszük, akkor minden olvasás forrása ugyanaz, mint S-ben. Azaz T_2 olvassa B-t és biztosan T_0 az utolsó „író" tranzakció. T_1 olvassa A-t, de T_2 már írta A-t, vagyis $r_1(A)$ forrása T_2, ahogy S-ben. T_3 is olvassa A-t, de mivel az ezt megelőző T_2 írta A-t, $r_3(A)$ forrása T_2, mint S-ben. Végül a hipotetikus T_f olvassa A-t és B-t, de A, illetve B utolsó írói a (T_2, T_1, T_3) ütemezésben rendre T_2, illetve T_3, szintén, mint S-ben. Megállapíthatjuk, hogy S egy nézet-sorbarendezhető ütemezés, és hogy (T_2, T_1, T_3) egy vele nézetekvivalens ütemezés. □

10.2.2. Poligráfok és nézet-sorbarendezhetőségi teszt

A 9.2.2. részben a konfliktus-sorbarendezhetőség tesztelésére a megelőzési gráfot használtuk. Ez a gráf általánosítható úgy, hogy a nézet-sorbarendezhetőség definíciójában megkövetelt összes megelőzési megszorítást tükrözze. Az ütemezéshez tartozó *poligráfot* definíció szerint a következő alkotóelemekből kell összeállítani:

1. Minden tranzakcióhoz vegyünk egy-egy csomópontot, és még további kettőt a hipotetikus T_0-nak és T_f-nek.
2. Ha $r_i(X)$ művelet forrása T_j, vegyük fel a T_j-ből T_i-be mutató irányított élt.
3. Tegyük fel, hogy az $r_i(X)$ olvasás forrása T_j, de T_k is írja X-t. Nem engedhetjük meg, hogy T_k tranzakció T_j és T_i közé essen, tehát vagy T_j előtt vagy T_i után kell állnia. Ezt a feltételt a T_k-ból T_j-be és a T_i-ből T_k-ba vezető élpár felvételével (szaggatott vonallal rajzolva) adhatjuk meg. Az élpár egyik fele „valódi", de nem foglalkozunk vele, hogy melyik. Amikor majd körmentessé próbáljuk a gráfot tenni, az élpárnak azt a felét tartjuk meg, amelyik ebben jobban segít. Vannak azonban fontos speciális esetek, amikor az élpár csak egyetlen élből áll:

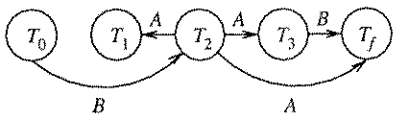
   a) Ha $T_j = T_0$, akkor T_k-nak nincs lehetősége arra, hogy T_j előtt szerepeljen. Ilyenkor az élpár helyett a $T_i \rightarrow T_k$ élt használjuk.
   b) Ha $T_i = T_f$, akkor T_k nem követheti T_i-t. Ilyenkor az élpár helyett a $T_k \rightarrow T_j$ élt használjuk.

10.8. példa: Vegyük a 10.7. példa ütemezését. A 10.3. ábrán látható, hogyan kezdtük el az S-hez tartozó poligráfot felépíteni. A csúcsokon kívül csak a 2. szabály alapján berajzolható élek vannak meg; ezekre azt az adatelemet is ráírtuk, ami miatt bekerültek a gráfba. A-t tehát T_2 továbbadja T_1-nek, T_3-nak és T_f-nek, míg B T_0-tól T_2-höz, illetve T_3-tól T_f-hez kerül.

Most meg kell gondolnunk, hogy milyen tranzakciók léphetnek közbe azáltal, hogy ugyanazt az adatelemet írják, mint ami összeköttetésekben szerepel. Ezeket a lehetséges beavatkozásokat zárjuk ki a 3. szabály élpárjaival, amelyek ebben a példában mind egyetlen élt jelentenek majd valamelyik speciális eset miatt.

Vegyük a $T_2 \rightarrow T_1$ irányított élt. Az A adatelemet csak T_0 és T_2 írja, és ezek közül egyik sem kerülhet T_2 és T_1 közé, mivel T_0 nem változtathatja meg a pozícióját, T_2 pedig már ott szerepel az él egyik végén. Vagyis nem kell új élt berajzolnunk. Hasonló érvelés alapján a $T_2 \rightarrow T_3$ és a $T_2 \rightarrow T_f$ esetén sem kell újabb éleket rajzolnunk ahhoz, hogy az A-t író tranzakciókat kívül tartsuk ezektől az élektől.

Most nézzük a B-hez tartozó éleket. Vegyük észre, hogy T_0, T_1, T_2 és T_3 is írja B-t.

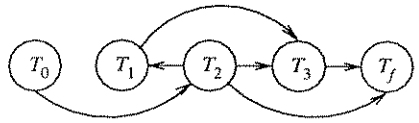


10.3. ábra. *Poligráf kezdemény a 10.8. példához*

Vegyük először a $T_0 \rightarrow T_2$-t. Ahogy az előbb láttuk, T_0-val és T_2-vel most nem kell foglalkoznunk, hiszen az él végpontjai nem okozhatnak problémát; a B adatelemet író tranzakciók közül tehát marad T_1 és T_3. Mivel T_1 nem szerepelhet T_0 és T_2 között, elméletileg be kellene rajzolnunk a $(T_1 \rightarrow T_0, T_2 \rightarrow T_1)$ élpárt. T_0-t azonban semmi sem előzheti meg, vagyis a $T_1 \rightarrow T_0$ lehetőség valójában nem lehetőség. Ebben a speciális esetben elég csak a $T_2 \rightarrow T_1$ irányított él felvétele. De ez az él már megvan A miatt, vagyis gyakorlatilag semmit sem változtatunk a poligráfon annak érdekében, hogy T_1 ne kerülhessen T_0 és T_2 közé.

T_3 sem kerülhet T_0 és T_2 közé. Az előző esethez hasonlóan itt is az élpár helyett csak a $T_2 \rightarrow T_3$ élt kellene felvennünk a gráfba, de ez ugyancsak szerepel már A miatt. Vagyis most sem változtatunk semmin.

Most nézzük a $T_3 \rightarrow T_f$ irányított élt. Mivel T_3-on kívül B-t még T_0, T_1 és T_2 írja, ezeket mind kívül kell tartanunk ezen az élen. T_0 nem eshet T_3 és T_f közé, de T_1 vagy T_2 igen. Mivel semelyikük sem tehető T_f mögé, biztosítanunk kell, hogy T_1 és T_2 mind T_3 előtt jelenjenek meg. A $T_2 \rightarrow T_3$ él már szerepel a gráfban, de a $T_1 \rightarrow T_3$-at hozzá kell adnunk. Ez az egyetlen változtatás, amit végre kell hajtanunk a poligráfon. Az S-hez tartozó végleges poligráf a 10.4. ábrán látható. □



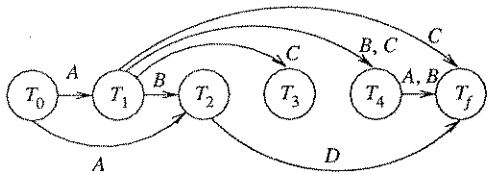
10.4. ábra. *Kész poligráf a 10.8. példához*

10.9. példa: A 10.8. példában minden élpárról kiderült, hogy valamelyik speciális eset miatt tulajdonképpen csak egyetlen élből áll. A 10.5. ábrán egy olyan ütemezést láthatunk, amely poligráfjában egy valódi élpár is szerepel.

T_1	T_2	T_3	T_4
	$r_2(A)$;		
$r_1(A)$; $w_1(C)$;			
		$r_3(C)$;	
$w_1(B)$;			
			$r_4(B)$;
		$w_3(A)$;	
			$r_4(C)$;
	$w_2(D)$; $r_2(B)$;		
			$w_4(A)$; $w_4(B)$;

10.5. ábra. *Az ütemezéshez tartozó poligráf valódi élpárt tartalmaz*

A 10.6. ábrán látható poligráf csak a forrás-olvasó kapcsolatokat mutatja. Mint a 10.3. ábrán, itt is ráírtuk az élekre, hogy melyik adatelem miatt kellett berajzolnunk. Ezek után végig kell mennünk az összes lehetséges eseten, amikor élpárt kéne felven-



10.6. ábra. *Poligráfkezdemény a 10.9. példához*

nünk. Ahogy a 10.8. példában láttuk, többféleképpen is egyszerűsíthető ez az eljárás. A $T_j \to T_i$ él vizsgálata esetén T_k szerepében (amelyek nem lehetnek középen) csak a következő tranzakciókat kell megvizsgálnunk:

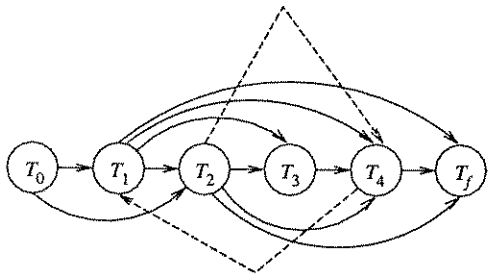
- A $T_j \to T_i$ élhez tartozó adatelemek írói.
- De nem T_0 vagy T_f, amelyek sohasem vehetik fel T_k szerepét, és
- Nem T_i vagy T_j, a kérdéses él végpontjai.

Ezek fényében nézzük végig az A adatbáziselemhez tartozó éleket. Az A-t író tranzakciók: T_0, T_3, T_4. Ezek közül T_0-val egyáltalán nem kell foglalkoznunk. T_3 nem kerülhet T_4 és T_f közé, ezért felvesszük a $T_3 \to T_4$ élt a gráfba; az élpár másik fele: a $T_f \to T_3$ nem jöhet számításba. Hasonlóan T_3 nem kerülhet T_0 és T_1, illetve a T_0 és T_2 közé sem, így a gráf a $T_0 \to T_3$, illetve $T_2 \to T_3$ élekkel bővül tovább.

Most nézzük meg, milyen megszorítások vonatkoznak T_4-re A írása miatt. T_4 a $T_4 \to T_f$ él egyik végpontja, azaz ezzel az éllel nem kell foglalkoznunk. T_4 nem kerülhet T_0 és T_1, illetve T_0 és T_2 közé, tehát fel kell vennünk a gráfba a $T_1 \to T_4$, illetve a $T_2 \to T_4$ éleket.

Következő lépésként vegyük a B-hez tartozó éleket. A B-t író tranzakciók: T_0, T_1, T_4. T_0-t most is figyelmen kívül hagyhatjuk. B miatt a következő éleket kell sorba vennünk: $T_1 \to T_2$, $T_1 \to T_4$, $T_4 \to T_f$. T_1-nek nincsen jelentősége az első két éllel kapcsolatban, viszont a harmadik miatt fel kell vennünk a $T_1 \to T_4$ élt.

T_4-nek csak a $T_1 \to T_2$ élre lehet hatása. Mivel ennek egyik végpontja sem T_0 vagy T_f, egy valódi élpárral bővül a gráf: ($T_4 \to T_1$, $T_2 \to T_4$)-gyel. Az új élekkel kibővített poligráf a 10.7. ábrán látható.



10.7. ábra. *Kész poligráf a 10.9. példához*

A C adatelemet T_0 és T_1 írja. Az előbbiekhez hasonlóan T_0 most sem jelenthet problémát. T_1 viszont minden C-hez tartozó élnek része, tehát vele sem kell foglalkoznunk. A helyzet gyakorlatilag D-vel kapcsolatban is ugyanez, vagyis megállapíthatjuk, hogy nincs szükség további élek felvételére. A végleges poligráf tehát megegyezik azzal, amelyet a 10.7. ábrán láthatunk. □

10.2.3. A nézet-sorbarendezhetőség tesztelése

Mivel minden élpárból csak az egyik élt kell kiválasztanunk, az S ütemezéshez pontosan akkor adható meg vele ekvivalens soros ütemezés, ha az S-hez tartozó poligráf az élpárok egy-egy tagjának elhagyásával körmentessé tehető. Először tegyük fel, hogy van egy ilyen körmentes gráfunk. Ekkor a gráf egy topologikus sorrendje a tranzakciók egy olyan rendezését adja, amelyben a forrás és az olvasó tranzakció közé nem ékelődhet be író tranzakció, továbbá minden író tranzakció megelőzi a hozzá tartozó olvasó tranzakciókat. Így a soros ütemezésben az olvasó-forrás kapcsolatok pontosan ugyanazok, mint S-ben; a két ütemezés nézetekvivalens, tehát S nézet-sorbarendezhető.

Megfordítva, ha S nézet-sorbarendezhető, akkor létezik hozzá egy S' nézetekvivalens soros ütemezés. Az S-hez tartozó poligráfban minden ($T_k \to T_j$, $T_i \to T_k$) élpár esetén az S' ütemezésben T_k vagy megelőzi T_j-t vagy T_i után következik. Ellenkező esetben T_k írásművelete megszakítaná a T_j és T_i közötti kapcsolatot, ami azt jelentené, hogy S és S' nem nézetekvivalens. Hasonlóan, a poligráf minden irányított éle is megjelenik az S'-beli tranzakciók sorrendjében. Ebből az következik, hogy a poligráf minden élpárjából ki tudjuk választani az egyik élt úgy, hogy az így kapott gráf irányított éleinek megfelel az S' soros ütemezése. Tehát a gráf körmentes.

10.10. példa: Vegyük a 10.4. ábrán látható poligráfot. Ez már ebben az állapotában is körmentes gráf. Ehhez egyetlen topologikus sorrend adható, így a 10.8. példában bemutatott ütemezés nézetekvivalens soros ütemezése (T_2, T_1, T_3).

Most nézzük a 10.7. ábra poligráfját. Két esetet kell megvizsgálnunk aszerint, hogy az élpárból melyik él marad meg. Ha a $T_4 \to T_1$ élt választjuk, akkor létrejön egy irányított kör a gráfban. A $T_2 \to T_4$ választása esetén azonban a gráf körmentes lesz. A gráfhoz megadható egyetlen topologikus sorrend (T_1, T_2, T_3, T_4), amely egyben nézetekvivalens soros ütemezésként is szolgál, és igazolja, hogy az eredeti ütemezés nézet-sorbarendezhető. □

10.2.4. Feladatok

10.2.1. feladat: Adjuk meg a következő ütemezésekhez tartozó poligráfokat és az összes nézetekvivalens soros ütemezést:

* a) $r_1(A)$; $r_2(A)$; $r_3(A)$; $w_1(B)$; $w_2(B)$; $w_3(B)$;
b) $r_1(A)$; $r_2(A)$; $r_3(A)$; $r_4(A)$; $w_1(B)$; $w_2(B)$; $w_3(B)$; $w_4(B)$;

c) $r_1(A)$; $r_3(D)$; $w_1(B)$; $r_2(B)$; $w_3(B)$; $r_4(B)$; $w_2(C)$; $r_5(C)$; $w_4(E)$; $r_5(E)$; $w_5(B)$;
d) $w_1(A)$; $r_2(A)$; $w_3(A)$; $r_4(A)$; $w_5(A)$; $r_6(A)$;

! **10.2.2. feladat:** Határozzuk meg, hogy az alábbi soros ütemezésekkel az előző feladat ütemezései közül hány i) konfliktusekvivalens, illetve ii) nézetekvivalens:

* a) $r_1(A)$; $w_1(B)$; $r_2(A)$; $w_2(B)$; $r_3(A)$; $w_3(B)$; azaz mindhárom tranzakció először olvassa A-t, majd írja B-t.
b) $r_1(A)$; $w_1(B)$; $w_1(C)$; $r_2(A)$; $w_2(B)$; $w_2(C)$; azaz mindhárom tranzakció először olvassa A-t, majd írja B-t és C-t.

10.3. Holtpontkezelés

Már többször megfigyelhettük, hogy az egyszerre végrehajtott tranzakciók versenyezhetnek egymással az erőforrásokért, és ezáltal *holtpontra* (deadlock) juthatnak, ami azt jelenti, hogy minden tranzakció egy másik tranzakció által birtokolt erőforrásra vár, és egyik sem tud továbblépni.

- A 9.3.4. részben láttuk, hogy a kétfázisú zárolást használó tranzakciók szokásos műveletei még mindig vezethetnek holtponthoz, ha mindegyik tranzakció valami olyasmit zárolt, amelyre egy másiknak is szüksége van.
- A 9.4.3. részben pedig azt láttuk, hogy az a lehetőség, hogy az osztott zár kizárólagos zárrá minősíthető fel, szintén okozhat holtpontot, ha minden tranzakció rendelkezik ugyanarra az elemre egy osztott zárral, és ezt egyszerre akarják felminősíteni.

A holtpontkezelés problémája két fő irányból közelíthető meg. Vagy valahogy rájövünk, hogy néhány tranzakció holtpontra jutott, és ebből a helyzetből keresünk kiutat, vagy már eleve úgy kezeljük a tranzakciókat, hogy soha ne juthassanak holtpontra.

10.3.1. Holtpontérzékelés időkorláttal

Ha a holtpont már bekövetkezett, akkor általában nem lehet a helyzeten úgy javítani, hogy minden tranzakció továbbléphessen. Azaz legalább egy tranzakciót vissza kell görgetni – abortáltatni kell, majd újraindítani.

A holtpontok érzékelésére és feloldására a legegyszerűbb megoldást az *időtúllépés* (timeout) módszere adja. Időkorlátokat vezetünk be, amely arra vonatkozik, hogy az egyes tranzakciók mennyi ideig lehetnek aktívak, és ha ezt a határt túllépik, akkor visszagörgetjük őket. Például egy egyszerű rendszerben, ahol a tipikus tranzakciók ezredmásodpercek alatt lefutnak, az egyperces időkorlátnak tényleg csak a holtpontra jutott tranzakcióra lenne hatása. De ha van néhány összetettebb tranzakció is, akkor az időtúllépés bekövetkezéséhez hosszabb időt választhatunk.

Vegyük észre, hogyha a holtpontra jutott tranzakció túllépi az időkorlátját, akkor a többi erőforrással együtt az eddig birtokolt zárjairól is lemond. Így tehát van esély arra, hogy a holtponton álló többi tranzakció még azelőtt be tudja fejezni a tevékenységét, mielőtt kifutna az időből. De mivel a holtpontra jutott tranzakciók valószínűleg körülbelül ugyanabban az időpontban indultak (különben az egyik befejeződött volna, még mielőtt a másik elkezdődött volna), az is lehetséges, hogy a rendszer hamis időtúllépéseket érzékel, azaz úgy görgeti vissza a tranzakciókat, hogy azok már túljutottak a közös holtponton.

10.3.2. A várakozási gráf

Azok a helyzetek, amikor a holtpont azért alakul ki, mert az egyik tranzakció a másik birtokában lévő zárakra vár, jól kezelhetők a *várakozási gráffal* (waits-for graph). Ebben a gráfban azt tartjuk nyilván, hogy melyik tranzakció melyikre vár. A várakozási gráf segítségével érzékelhetővé válnak a már kialakult holtpontok, de meg is előzhető a kialakulásuk. Mi az utóbbival próbálkozunk, ami azzal jár, hogy a várakozási gráfot egész idő alatt nyilván kell tartanunk, és az olyan műveleteket, amelyek következtében a gráfban kör alakulna ki, nem szabad megengednünk.

Idézzük fel a 9.5.2. rész alapján, hogy a zártáblában minden X adatbáziselemhez létezik egy lista, amelyben azon tranzakciók mellett, amelyek arra várnak, hogy zárolhassák X-t, azok is fel vannak sorolva, amelyek rendelkeznek X zárjával. A várakozási gráf csúcsai a listában található tranzakcióknak felelnek meg. A gráfban irányított él fut T-ből U-ba, ha létezik olyan A adatbáziselem, hogy:

1. U zárolja A-t.
2. T arra vár, hogy zárolhassa A-t, és
3. T csak akkor kapja meg a számára megfelelő módban A zárját, ha először U lemond róla.[3]

Ha nincsen (irányított) kör a gráfban, akkor végül minden tranzakció be tudja fejezni a működését. Lesz legalább egy olyan tranzakció, amelyik nem vár semelyik másikra, így ez biztosan befejeződhet. Ekkor viszont megint lesz legalább egy tranzakció, amelyik nem várakozik, ezért továbbléphet és így tovább.

Ha azonban a gráf nem körmentes, akkor a körben részt vevő tranzakciók nem léphetnek tovább, azaz holtpontra jutottak. A holtpont-megelőzési stratégia tehát abból áll, hogy minden olyan tranzakciót visszagörgetünk, amelynek valami olyan igénye van, ami kört idézne elő a várakozási gráfban.

[3] Egyszerű esetekben, mint amikor osztott és kizárólagos zárakat használunk, minden egyes várakozó tranzakciónak meg kell várnia, amíg *minden* zárral rendelkező tranzakció feloldja a saját zárját. Vannak azonban olyan zármód rendszerek is, ahol egy tranzakció már akkor is megkaphatja a zárat, ha még csak néhány zárbirtokos mondott le róla. Lásd 10.3.6. feladat.

10.11. példa: Tegyük fel, hogy a következő négy tranzakcióval rendelkezünk:

T_1: $l_1(A)$; $r_1(A)$; $l_1(B)$; $w_1(B)$; $u_1(A)$; $u_1(B)$;
T_2: $l_2(C)$; $r_2(C)$; $l_2(A)$; $w_2(A)$; $u_2(C)$; $u_2(A)$;
T_3: $l_3(B)$; $r_3(B)$; $l_3(C)$; $w_3(C)$; $u_3(B)$; $u_3(C)$;
T_4: $l_4(D)$; $r_4(D)$; $l_4(A)$; $w_4(A)$; $u_4(D)$; $u_4(A)$;

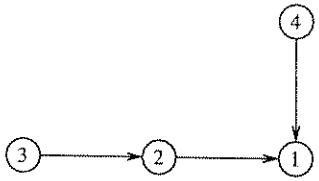
Mindegyik az egyik elemet olvassa, a másikat írja. Egy egyszerű zárrendszert használunk egyféle zárral, bár ugyanezt a jelenséget figyelhetnénk meg, ha egy osztott/kizárólagos rendszerben a szokásos módon osztanánk a zárakat: osztott zárat az olvasáshoz, kizárólagos zárat az íráshoz.

A 10.8. ábrán egy lehetséges ütemezés kezdeti szakasza látható. Az első négy lépésben mindegyik tranzakció zárolja azt az elemet, amelyet olvasni szeretne. Az 5) lépésben T_2 megpróbálja zárolni A-t, de nem tudja, mert a zár már T_1 birtokában van. T_2 tehát várakozik T_1-re, ezért a várakozási gráfba berajzolunk egy élt a T_2-nek megfelelő csúcsból a T_1-nek megfelelő csúcs felé.

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$			
2)		$l_2(C)$; $r_2(C)$;		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(D)$; $r_4(D)$;
5)		$l_2(A)$; **Elutasítva**		
6)			$l_3(C)$; **Elutasítva**	
7)				$l_4(A)$; **Elutasítva**
8)	$l_1(B)$; **Elutasítva**			

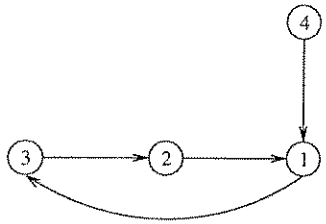
10.8. ábra. *Egy ütemezés első néhány lépése, amelyben holtpont alakul ki*

Hasonlóan, a 6) lépésben T_3 nem tudja zárolni C-t T_2 miatt, a 7) lépésben pedig T_4 vall kudarcot A zárolásával T_1 miatt. Az ebben az állapotban egyelőre körmentes várakozási gráf a 10.9. ábrán látható.



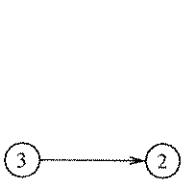
10.9. ábra. *A várakozási gráf az ütemezés 7) lépése után*

A 8) lépésben T_1-nek várnia kell B zárolásával T_3 miatt. Ha megengednénk T_1-nek, hogy várjon erre a zárra, akkor T_1, T_2 és T_3 mentén kör jönne létre a várakozási gráfban, ahogy ezt a 10.10. ábra is mutatja. Mivel a körben mindegyik tranzakció arra vár, hogy a másik befejeződjön, egyik sem tud továbblépni, vagyis ennek a három tranzakciónak a részvételével holtpont alakul ki. Véletlen egybeesés, hogy T_4 sem fejeződhet be annak ellenére, hogy nincs benne a körben. Az ő előrejutása azonban T_1 továbblépésén múlik.



10.10. ábra. *A várakozási gráf az ütemezés 8) lépése után kört tartalmaz*

Mivel a kört okozó tranzakciókat visszagörgetjük, így teszünk T_1-gyel is. A várakozási gráf a 10.11. ábrának megfelelően alakul. T_1 feloldja A zárolását, amelyet vagy T_2 vagy T_4 vesz át. Tegyük fel, hogy a zár T_2 birtokába kerül. T_2 befejeződik, ezáltal feloldódik a zár A-n és C-n. Most T_3, amely C-t akarja zárolni, és T_4 is, amely A-t, lezárulhat. Valamivel később T_1-et újraindítjuk, de nem kaphatja meg sem A, sem B zárját, amíg T_2, T_3 és T_4 be nem fejeződött. □



10.11. ábra. *A várakozási gráf T_1 visszagörgetése után*

10.3.3. Holtpontmegelőzés az elemek sorbarendezésével

Most ismerkedjünk meg néhány más módszerrel is, amellyel megelőzhető a holtpont kialakulása. Az elsőben sorrendbe kell raknunk az adatbáziselemeket valamilyen tetszőleges, de előre rögzített rendezés szerint. Ha például az adatbáziselemek blokkok, akkor rendezhetjük őket lexikografikusan a fizikai címük szerint. Emlékezzünk vissza, a 3.3.1. részben volt arról szó, hogy egy blokk fizikai címe általában egy olyan bájtsorozat, amely a blokknak a tárrendszeren belüli helyét írja le.

Ha minden tranzakciótól megkívánjuk, hogy az adatbáziselemeket a megadott sorrendnek megfelelően próbálja meg zárolni (ami egyébként a legtöbb alkalmazásban nem reális feltétel), akkor a (foglalt) zárakra való várakozások miatt nem alakulhat ki holtpont. Ugyanis tegyük fel, hogy T_2 várakozik A_1 zárolásával T_1 miatt, T_3 várakozik A_2 zárolásával T_2 miatt, és így tovább, T_n várakozik A_{n-1} zárolásával T_{n-1} miatt, T_1 pedig A_n zárolásával várakozik T_n miatt. Mivel T_2 zárolta A_2-t, de várnia kell A_1-re,

$A_2 < A_1$ relációnak kell teljesülnie az adott rendezésen. Hasonlóan, $A_i < A_{i-1}$, ahol $i = 3, 4, \ldots, n$. De mivel T_1 zárolta A_1-et és várakozik A_n-re, $A_1 < A_n$ is fennáll. Ekkor a következő teljesül: $A_1 < A_n < A_{n-1} < \ldots < A_2 < A_1$, ami lehetetlen, hiszen ebből $A_1 < A_1$ következne.

10.12. példa: Tegyük fel, hogy az adatbáziselemek alfabetikusan vannak rendezve. Ekkor, ha a 10.11. példa négy tranzakcióját vesszük, akkor T_2-t és T_4-t át kell írnunk, hogy a megfelelő sorrendben zárolják az elemeket. Így a négy tranzakció a következő:

T_1: $l_1(A)$; $r_1(A)$; $l_1(B)$; $w_1(B)$; $u_1(A)$; $u_1(B)$;
T_2: $l_2(A)$; $l_2(C)$; $r_2(C)$; $w_2(A)$; $u_2(C)$; $u_2(A)$;
T_3: $l_3(B)$; $r_3(B)$; $l_3(C)$; $w_3(C)$; $u_3(B)$; $u_3(C)$;
T_4: $l_4(A)$; $l_4(D)$; $r_4(D)$; $w_4(A)$; $u_4(D)$; $u_4(A)$;

A 10.12. ábrán látható, hogy mi történik, ha ugyanazt az ütemezést használjuk, mint ami a 10.8. ábrán adott. Az első lépésben T_1 zárolja A-t. A következő lépésben T_2 próbálja végrehajtani az első műveletét, de A zárolásával T_1 miatt várnia kell. Ezek után T_3 zárolja B-t, majd T_4 is megpróbálja zárolni A-t, de várakoznia kell.

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$			
2)		$l_2(A)$; **Elutasítva**		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(A)$; **Elutasítva**
5)			$l_3(C)$; $w_3(C)$;	
6)			$u_3(B)$; $u_3(C)$;	
7)	$l_1(B)$; $w_1(B)$;			
8)	$u_1(A)$; $u_1(B)$;			
9)		$l_2(A)$; $l_2(C)$;		
10)		$r_2(C)$; $w_2(A)$;		
11)		$u_2(A)$; $u_2(C)$;		
12)				$l_4(A)$; $l_4(D)$;
13)				$r_4(D)$; $w_4(A)$;
14)				$u_4(A)$; $u_4(D)$;

10.12. ábra. *Az elemeket a tranzakciók alfabetikus sorrendben zárolják, ezzel megelőzhető a holtpont kialakulása*

Mivel T_2 elakadt, a 10.8. ábra ütemezése szerint T_3 következik. Sikeresen zárolja C-t, majd a 6) lépésben lezárul. Mivel ezzel egy időben elengedi B-t és C-t, T_1 is befejeződhet, ami meg is történik a 8) lépésben. Ezen a ponton A felszabadul a zárolás alól, és feltehető, hogy FIFO alapon T_2 birtokába kerül a zár. Most T_2 mindkét szükséges adatelemét zárolja, majd a 11) lépésben lezárul. Végül T_4 is megkapja az óhajtott zárakat és befejeződik. □

10.3.4. Holtpontérzékelés időbélyegzővel

A 10.3.2. részben láttuk, hogy a várakozási gráf segítségével észlelni tudjuk a holtpontok kialakulását. Ez a gráf azonban nagy lehet, és minden alkalommal kört keresni benne, valahányszor egy tranzakciónak várnia kell egy zárra, nagyon időigényessé válhat. A várakozási gráf mellett egy másik lehetőség az időbélyegzők bevezetése. Minden egyes tranzakcióhoz hozzárendelünk egy-egy időbélyegzőt, amely:

- Csak a holtpont érzékelésére használatos. Ez nem ugyanaz, mint az időbélyegző, amelyet a 9.8. részben a konkurenciavezérléshez használtunk, még akkor sem, ha éppen az időbélyegző alapú konkurenciavezérlés van érvényben.
- Például, ha egy tranzakciót vissza kell görgetni, akkor egy új, későbbi konkurencia időbélyegzővel kezdi újra a működését, de a holtpontészleléshez használt időbélyegzője sohasem változik.

Az időbélyegzőt akkor használjuk, amikor egy T tranzakciónak az U tranzakció birtokában lévő zárra kell várakoznia. Attól függően, hogy T vagy U az *idősebb* (korábbi időbélyegzővel rendelkező), két különböző dolog történhet. Két különféle eljárásmód használható a tranzakciók kezelésére és a holtpontok érzékelésére.

1. *Megvár-meghal séma:*

   a) Ha T idősebb U-nál (azaz T időbélyegzője korábbi, mint U időbélyegzője), akkor megengedjük, hogy T az U által birtokolt zár(ak)ra várakozzon.
   b) Ha U idősebb T-nél, akkor T „meghal": visszagörgetjük.

2. *Megsebez-megvár séma:*

   a) Ha T idősebb U-nál, akkor „megsebzi" U-t. A „seb" általában végzetes: U-t vissza kell görgetni, és le kell mondania a szükséges zárakról T javára. Egyetlen eset képez kivételt: ha, mire a „sebzésnek" hatása lenne, U befejeződik, és elengedi a zárjait. Ilyenkor U életben marad, és nem kell visszagörgetni.
   b) Ha U idősebb T-nél, akkor T az U birtokában lévő zár(ak)ra várakozik.

10.13. példa: Vegyük a 10.12. példa tranzakcióit, és nézzük meg, hogyan működik a megvár-meghal séma. Feltesszük, hogy T_1, T_2, T_3, T_4 az időbélyegzők sorrendje, azaz T_1 a legidősebb tranzakció. Azt is feltesszük, hogy amikor egy tranzakciót visszagörgetünk, az nem indul újra olyan hamar, hogy még azelőtt aktívvá válna, mielőtt a többi tranzakció lezárulna.

Az események egy lehetséges sorrendje a megvár-meghal sémát használva a 10.13. ábrán látható. A zárját először T_1 kapja meg. Amikor T_2 akarja zárolni A-t, meghal, mert T_1 idősebb nála. A 3) lépésben T_3 zárolja B-t, de a 4) lépésben T_4 akarja zárolni A-t, és meghal, mert a zár birtokosa, T_1, idősebb T_4-nél. A következő lépésben T_3 megkapja C zárját, majd befejeződik. Amikor T_1 folytatja a működését, elérhető lesz számára B zárja, majd a 8) lépésben szintén befejeződik.

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$			
2)		$l_2(A)$; **Meghal**		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(A)$; **Meghal**
5)			$l_3(C)$; $w_3(C)$;	
6)			$u_3(B)$; $u_3(C)$;	
7)	$l_1(B)$; $w_1(B)$;			
8)	$u_1(A)$; $u_1(B)$;			
9)				$l_4(A)$; $l_4(D)$;
10)		$l_2(A)$; **Várakozik**		
11)				$r_4(D)$; $w_4(A)$;
12)				$u_4(A)$; $u_4(D)$;
13)		$l_2(A)$; $l_2(C)$;		
14)		$r_2(C)$; $w_2(A)$;		
15)		$u_2(A)$; $u_2(C)$;		

10.13. ábra. *A tranzakciók műveletei a megvár-meghal sémában érzékelik a holtpontot*

Most újrakezdődik a két visszagörgetett tranzakció: T_2 és T_4. A holtpontkezeléshez használt időbélyegzőjük nem változik; T_2 továbbra is idősebb, mint T_4. Feltesszük azonban, hogy T_4 kezdődik előbb, a 9) lépésben, és amikor a 10) lépésben az idősebb T_2 zárolni akarja A-t, akkor erre várnia kell, de nem abortál. T_4 lezárul a 12) lépésben, majd az utolsó három lépésben T_2 is befejezi a működését. □

10.14. példa: Most a 10.14. ábra segítségével kövessük nyomon, hogy ugyanezek a tranzakciók hogyan viselkednek, ha a megsebez-megvár sémát használjuk. Mint a 10.13. ábrán látható, T_1 itt is A zárolásával indul. Amikor a 2) lépésben T_2 akarja zárolni A-t, várakoznia kell, hiszen T_1 idősebb T_2-nél. Miután a 3) lépésben T_3 zárolja B-t, T_4-nek is várnia kell A zárjára.

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$			
2)		$l_2(A)$; **Várakozik**		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(A)$; **Várakozik**
5)	$l_1(B)$; $w_1(B)$;		**Megsebezve**	
6)	$u_1(A)$; $u_1(B)$;			
7)		$l_2(A)$; $l_2(C)$;		
8)		$r_2(C)$; $w_2(A)$;		
9)		$u_2(A)$; $u_2(C)$;		
10)				$l_4(A)$; $l_4(D)$;
11)				$r_4(D)$; $w_4(A)$;
12)				$u_4(A)$; $u_4(D)$;
13)			$l_3(B)$; $r_3(B)$;	
14)			$l_3(C)$; $w_3(C)$;	
15)			$u_3(B)$; $u_3(C)$;	

10.14. ábra. *A tranzakciók műveletei a megsebez-megvár sémában érzékelik a holtpontot*

Miért működik jól az időbélyegző alapú holtpontérzékelés?

Azt állítjuk, hogy sem a megvár-meghal, sem a megsebez-megvár sémával nem alakulhat ki kör a várakozási gráfban, így holtpont sem jöhet létre. Tegyük fel az ellenkezőjét, vagyis hogy létezik egy $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$ kör. A legidősebb tranzakció legyen mondjuk T_2.

A megvár-meghal sémában mindig csak újabb tranzakciókra lehet várni. Így nem lehetséges, hogy T_1 várjon T_2-re, mivel T_2 biztosan idősebb T_1-nél. A megsebez-megvár sémában csak idősebb tranzakciókra lehet várni. Tehát semmiképp sem lehetséges, hogy T_2 az újabb T_3-ra várakozzon. Ezekből következik, hogy a kör nem létezhet, ezért holtpont sincsen.

Ez után tegyük fel, hogy az 5) lépésben T_1 folytatja a működését, zár alá akarja helyezni B-t. B zárja már T_3 birtokában van, de mivel T_1 az idősebb, „megsebzi" T_3-t. T_3 még nem zárult le, a seb halálos: feloldjuk T_3 zárjait, őt pedig visszagörgetjük, így T_1 befejeződhet.

Amikor T_1 feloldja A zárját, tegyük fel, hogy ez T_2-höz kerül, amely ekkor továbbléphet. T_2 után a zárat T_4 kapja meg, amely ezután befejeződik. Végül T_3 újraindul és közbeavatkozás nélkül lezárul. □

10.3.5. A holtpontkezelő módszerek összehasonlítása

A megvár-meghal és a megsebez-megvár sémában is az idősebb tranzakciók maguknál újabb tranzakciókat végeznek ki. Mivel a tranzakciók mindig az eredeti időbélyegzőjükkel indulnak újra, egyszer mindegyikre teljesülni fog, hogy éppen ő a legidősebb a rendszerben, és mint ilyen, biztosan lezárul. Erre a biztosítékra, hogy végül mindegyik tranzakció befejeződik, úgy hivatkozunk, hogy *nincs kiéheztetés*. Vegyük észre, hogy a fejezetben leírt többi séma nem feltétlenül előzi meg a kiéheztetést; ha nem teszünk további intézkedéseket, a tranzakciókkal újra és újra megtörténhet, hogy az indításuk után holtpontra jutnak, és ezért visszagörgetjük őket. Ezzel kapcsolatban nézzük meg a 10.3.7. feladatot.

A megvár-meghal és a megsebez-megvár sémák működése között azonban van egy finom különbség. A megsebez-megvár sémában valahányszor egy idősebb tranzakciónak egy újabb tranzakció birtokában lévő zárra van szüksége, az újabb tranzakció meghal. Ha feltesszük, hogy a tranzakciók az indítás után nem sokkal már birtokba veszik a szükséges zárakat, akkor ritkán fog az előfordulni, hogy egy újabb tranzakció megszerzi a zárakat az idősebb elől. Ebben a sémában tehát a tranzakciók visszagörgetésére várhatóan csak ritkán kerül sor.

Másrészt, amikor a megvár-meghal séma görget vissza egy tranzakciót, akkor az

még mindig a „zárgyűjtő" stádiumában van, amely feltehetően a tranzakció legkorábbi szakaszát jelenti. Így, habár a megvár-meghal séma talán több tranzakciót görget vissza, mint a megsebez-megvár séma, ezek a tranzakciók csak rövid ideig működtek az abortálás előtt. Ezzel szemben a megsebez-megvár sémában visszagörgetett tranzakció valószínűleg már birtokba vette az összes szükséges zárat, és tekintélyes mennyiségű processzoridőt vett igénybe az eddigi működése. A feldolgozandó tranzakcióktól függően tehát hol az egyik, hol a másik módszer eredményez több fölösleges munkát.

Most vizsgáljuk meg a két séma előnyeit és hátrányait a várakozási gráf kézenfekvő konstrukciójával és használatával szemben is. A következő szempontok a lényegesek:

- A megsebez-megvár és a megvár-meghal sémát is könnyebb megvalósítani, mint egy olyan rendszert, amely folyamatosan nyilvántartja vagy időszakosan létrehozza a várakozási gráfot. A várakozási gráf felépítésével járó hátrány még rendkívülibb, amikor osztott adatbázisokkal van dolgunk, és a gráfot a különböző munkaállomásokról[4] összegyűjtött zártáblák alapján kell elkészíteni. Erről bővebben a 10.6. részben lesz szó.
- A várakozási gráf használatával minimálisra csökkenthető a holtpont miatt abortáltatott tranzakciók száma. A tranzakciót mindig csak akkor abortáltatjuk, ha tényleg holtpontra jutott. Másrészt azonban, a megsebez-megvár és a megvár-meghal sémával is előfordulhat néha, hogy olyankor görget vissza egy tranzakciót, amikor nem jött volna létre holtpont, és a tranzakció életben hagyásával sem alakulhatott volna ki.

10.3.6. Feladatok

10.3.1. feladat: Tegyük fel, hogy az alábbi műveletsorozatokban minden egyes olvasás-, illetve írásműveletet közvetlenül megelőzi az osztott, illetve kizárólagos zár igénylése. Tegyük fel továbbá, hogy a zárak feloldása rögtön a tranzakció utolsó művelete után történik meg. Adjuk meg azokat a műveleteket, amelyeknek a végrehajtását az ütemező megtagadja, és mondjuk meg, hogy létrejön-e holtpont vagy sem! Adjuk meg továbbá, hogy hogyan alakul a műveletek végrehajtása során a várakozási gráf! Ha létrejön egy holtpont, abortáltassuk az egyik tranzakciót, és mutassuk meg, hogyan folytatódik a műveletsorozat!

* a) $r_1(A)$; $r_2(B)$; $w_1(C)$; $r_3(D)$; $r_4(E)$; $w_3(B)$; $w_2(C)$; $w_4(A)$; $w_1(D)$;
b) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(D)$;
c) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(A)$;
d) $r_1(A)$; $r_2(B)$; $w_1(C)$; $w_2(D)$; $r_3(C)$; $w_1(B)$; $w_4(D)$; $w_2(A)$;

10.3.2. feladat: Hogyan játszódnak le a 10.3.1. feladat műveletsorozatai a megsebez-megvár holtpont-megelőzési rendszerben? Tegyük fel, hogy a holtpont-időbélyegzők sorrendje megegyezik a tranzakciók indexével, azaz T_1, T_2, T_3, T_4. Tegyük fel azt is, hogy a tranzakciók esetleges újraindítása a visszagörgetés sorrendjében történik.

[4] Angolul *site*. A hálózatban részt vevő számítógépeket, „helyszíneket" nevezzük így. *A fordító megjegyzése.*

10.3.3. feladat: Hogyan játszódnak le a 10.3.1. feladat műveletsorozatai a megvár-meghal holtpont-megelőzési rendszerben? Használjuk ugyanazokat a feltevéseket, mint a 10.3.2. feladatban!

! **10.3.4. feladat:** Igaz-e, hogy minden $n > 1$ egészhez létezik olyan várakozási gráf, amelyben a legrövidebb kör hossza n? Mi a helyzet $n = 1$, vagyis hurokél esetén?

!! **10.3.5. feladat:** A holtpontok elkerülésére a következő módszer is megadható: minden tranzakció a futása elején bejelenti, hogy mely zárakra lesz szüksége, ezeket vagy mind megkapja, vagy egyiket sem kapja meg, és várakoznia kell. Elkerülhető-e ezzel a módszerrel a zárolások miatt kialakuló holtpont? Ha igen, állításunkat indokoljuk; ha nem, adjunk ellenpéldát!

! **10.3.6. feladat:** Vegyük a 9.6. részben megismert szándékjelölő zárolási rendszert. Írjuk le, hogyan lehetne ehhez a zárrendszerhez várakozási gráfot készíteni! Vegyük például azt a lehetőséget, amikor az A adatbáziselemet különböző tranzakciók S, illetve IX módban zárolják. Milyen éleket rajzolunk a gráfba, ha A zárolásával valamelyik tranzakciónak várnia kell?

*! **10.3.7. feladat:** A 10.3.5. részben rámutattunk arra, hogy a megsebez-megvár, illetve megvár-meghal sémától eltérő holtpontérzékelő módszerek nem feltétlenül előzik meg a tranzakciók kiéheztetését, amikor is a tranzakciót ismételten visszagörgetjük, így az sohasem éri el a végpontját. Adjunk példát arra, hogy hogyan vezethet a tranzakciók kiéheztetéséhez annak a módszernek a használata, amikor minden olyan tranzakciót visszagörgetünk, amely kört hozna létre a várakozási gráfban! Megelőzhető-e a kiéheztetés, ha a tranzakciók csak egy rögzített sorrendben zárolhatják az elemeket? Mit mondhatunk az időtúllépésről, mint holtpontfeloldó mechanizmusról?

10.4. Osztott adatbázisok

Ebben a fejezetben az osztott adatbázisrendszerek alapelemeit tekintjük át. Egy osztott rendszerben sok, viszonylag autonóm processzorral rendelkezünk, amelyek mind részt vehetnek az adatbázis-műveletekben. Az osztott adatbázisok alkalmazásában számos lehetőség rejlik:

1. Mivel egyszerre több gép is dolgozhat ugyanazon a problémán, jobbak a lehetőségek a párhuzamosításra és arra, hogy a lekérdezésekre gyorsan kapjuk meg a választ.
2. Mivel az adatok többszörözés útján számos helyszínen is jelen lehetnek, a rendszernek valószínűleg nem kell leállnia a feldolgozással csak azért, mert az egyik munkaállomás vagy alkotóelem az adott pillanatban nem érhető el.

Másrészt azonban az osztott feldolgozás minden tekintetben növeli az adatbázisrendszer összetettségét, ezért újra kell gondolnunk az ABKR legalapvetőbb elemeinek a tervezését is. Sok osztott környezetben a kommunikációs költség túlnőhet a feldolgozás költségén, így kritikus témává válik az elküldött üzenetek száma. Ebben a fejezetben az osztott adatbázisok fő kérdései kerülnek bevezetésre, az utána következő részekben pedig az osztott adatbázisok két fontos problémájának, az osztott véglegesítésnek és az osztott zárolásnak megoldására összpontosítunk.

10.4.1. Osztott adatok

Az adatok szétosztásának egyik fontos oka lehet, hogy az adatbázist felhasználó szervezet maga is több helyszínre van szétosztva, és minden munkaállomáson megvannak az elsősorban odatartozó adatok. Lássunk néhány példát:

1. Egy banknak lehet sok fiókja. Minden fiók (vagy egy adott város fiókcsoportja) rendelkezik a nála (vagy a városban) vezetett számlák adatbázisával. Az ügyfelek akármelyik fiókban elintézhetik a pénzügyeiket, de általában a „saját" fiókjukba járnak, ahol a számla adatait is tárolják. A banknak lehetnek a központi fiókban tárolt adatai is, például az alkalmazottakról vagy a banki politikáról, amely például az aktuális kamatlábat is jelentheti. Természetesen az egyes fiókokban tárolt adatrekordokról készült biztonsági másolatot (backup) is tárolják valahol, de valószínűleg nem az adott és nem is a központi fiókban.
2. Egy üzlethálózat sok önálló áruházból állhat. Minden üzlet (vagy egy város üzletcsoportja) rendelkezik az adott üzletben történt vásárlásokra vonatkozó adatbázissal és egy raktáradatbázissal. Lehet, hogy van egy központi részleg, ahol az alkalmazottakról, az üzletláncszintű leltárról vagy a hitelkártyával fizető vásárlókról tárolnak adatokat, a szállítókról pedig olyan információkat tartanak nyilván, mint például a nem teljesített megrendelések, vagy hogy mennyivel tartozik még az üzletlánc. Ezenfelül létezhet még egy „adattárház" is, amelyben megtalálható az összes üzletben nyilvántartott vásárlási adatok másolata. Ezt az elemzők arra használják, hogy ad hoc lekérdezések segítségével analizálják, illetve előre jelezzék a várható keresletet (lásd 11.3. részt).
3. On-line könyvekkel és egyéb dokumentumokkal rendelkező egyetemek közös erővel létrehozhatnak egy digitális könyvtárat. Akármelyik munkaállomáson indítunk egy keresést, az az elérhető dokumentumok *uniójának* a katalógusát fogja megvizsgálni, és ha valamelyik munkaállomáson elérhető a keresett dokumentum, a felhasználó megkapja az elektronikus másolatát.

Néhány esetben az a reláció, amire logikailag úgy gondolunk, mintha egyetlen reláció lenne, valójában sok különböző helyen megtalálható részre van szétosztva. Úgy képzelhetjük például, hogy az üzlethálózat egyetlen vásárlási relációval rendelkezik:

```
Eladások(árucikk, dátum, ár, vevő)
```


A kommunikációs költségben szerepet játszó tényezők

Mivel manapság egyre olcsóbban egyre nagyobb sávszélesség érhető el, eltűnődhetünk azon, vajon számításba kell-e vennünk a kommunikációs költséget az osztott adatbázisrendszer tervezése közben. A legnagyobb elektronikusan kezelt objektumok között most az adatok bizonyos fajtái is megtalálhatók, tehát még egy nagyon olcsó kommunikációs megoldás mellett sem hanyagolható el egy terabájt méretű adatdarab továbbításának a költsége. A legtöbb esetben azonban a kommunikációs költség nem csupán a bitek átküldéséből áll – figyelembe kell vennünk a különböző protokollrétegeket is, amelyek előkészítik, majd a vevő oldalon helyreállítják a továbbításra szánt adatokat, és kezelik az egész kommunikációt. Ezen protokollok mindegyike tekintélyes mennyiségű számítást igényel. A számítások elvégzése is egyre kevesebbe kerül, a kommunikáció miatt végrehajtott számítások mennyisége azonban valószínűleg még mindig jelentős marad ahhoz képest, amennyire a legfontosabb adatbázis-műveletek elvégzéséhez a hagyományos, egyprocesszoros rendszerekben szükség van.

Ez a reláció azonban fizikailag nem létezik, csak mint a hálózat üzleteiben tárolt, azonos sémával rendelkező számos reláció uniója. Ezeket a helyi relációkat *töredékeknek* (fragments) hívjuk, a logikai reláció fizikai részekre bontását pedig az `Eladások` reláció *vízszintes irányú (horizontális) dekompozíciójának* nevezzük. A felbontásra azért hivatkozunk „vízszintesként", mert a „nagy" `Eladások` reláció részekre bontását úgy is elképzelhetjük, mintha vízszintes vonalakkal választanánk szét az egyes üzletekhez tartozó sorokat egymástól.

Más helyzetekben úgy tűnik, mintha az osztott adatbázis a relációt „függőlegesen" bontotta volna fel, mégpedig úgy, hogy ami logikailag egy reláció lenne, azt két vagy több, más-más munkaállomáson megtalálható relációra osztja szét, amelyek attribútumai az eredeti attribútumhalmaz részhalmazát képezik. Például, ha meg akarjuk keresni azokat a vásárlásokat a bostoni áruházban, ahol a vevő több mint 90 napos hátralékban van a hitelkártyaszámla kiegyenlítésével, akkor hasznos lenne egy olyan reláció (vagy nézettábla), amely az `Eladások` táblázatból az `árucikk`, `dátum` és `vevő` információkkal együtt a vevő utolsó hitelkártyaszámla befizetésének az időpontját is tartalmazná. Az itt leírt esetben azonban ez a reláció függőlegesen fel van bontva, ezért össze kellene kapcsolnunk a központban tárolt hitelkártyás vásárló relációt a bostoni áruházban nyilvántartott `Eladások` töredékkel.

10.4.2. Osztott tranzakciók

Az adatok megosztásának az az egyik következménye, hogy a tranzakciók különböző helyeken végrehajtandó folyamatokat is magukban foglalnak. Így az eddigi tranzakciómodellünket meg kell változtatni. A tranzakciót nem tekinthetjük többé egy darab kódnak, amit az egyetlen ütemezővel és az egyetlen naplókezelővel kommunikáló

egyetlen processzor egyetlen számítógépen hajt végre. A tranzakció most egymással kapcsolatot tartó, különböző munkaállomásokon megtalálható *tranzakció-alkotórészekből* áll, amelyek mindegyike az adott munkaállomáson működő lokális ütemezővel és naplókezelővel kommunikál. Két fontos kérdést kell újra átgondolnunk:

1. Hogyan kezeljük a véglegesítés/abortálás döntést osztott tranzakciók esetén? Mi történik, ha a tranzakció egyik alkotórésze abortáltatni akarja az egész tranzakciót, de a többi alkotórésznek nincs problémája, és inkább véglegessé szeretne válni? A 10.5. részben megtárgyaljuk a „kétfázisú véglegesítés" technikáját, melynek segítségével helyes döntés hozható, és amely gyakran akkor is figyelembe veszi a még elérhető munkaállomásokat, ha néhány másik már átmenetileg működésképtelenné vált.
2. Hogyan biztosítható az olyan tranzakciók sorbarendezhetősége, amelyek különböző helyeken végrehajtott alkotórészeket foglalnak magukban? A 10.6. részben különös figyelmet szentelünk a zárolás problémájának, és megnézzük, hogyan használhatók a lokális zártáblák az adatbáziselemek globális zárolásához, és így hogyan támogatják egy megosztott környezetben a tranzakciók sorbarendezhetőségét.

10.4.3. Adattöbbszörözés

Az osztott rendszereknek egy fontos előnye az, hogy az adatokat *többszörözni* tudjuk, azaz az adatelemekről másolatokat tarthatunk több helyen is. Ez egyrészt hasznos abban az esetben, ha az egyik munkaállomás meghibásodik, mert ilyenkor egy másik helyen is megtalálhatók azok az adatok, amelyekhez most a meghibásodás miatt nem lehet hozzáférni. Másrészről pedig növelhetjük a válaszadás sebességét azzal, hogy a szükséges adatokból másolatokat tárolunk azon a munkaállomáson, ahol a lekérdezést elindítják.

Például:

1. A bank minden fiókjában tárolhat egy-egy másolatot az aktuális kamatlábakról, így az erre vonatkozó lekérdezéseket nem kell elküldeni a központi fiókba.
2. Az áruház minden egyes üzletében tárolhat egy-egy másolatot a szállítókról nyilvántartott információkról, így ha lokálisan szükség van valamilyen adatra (például az üzletvezető tudni szeretné egy szállító telefonszámát, mert ellenőrizni akarja a szállítmányt), akkor ehhez nem kell a központi részlegbe üzeneteket küldeni.
3. A digitális könyvtár ideiglenesen eltárolhatja (cache-elheti) egy népszerű dokumentum másolatát abban az iskolában, ahol a diákoknak ez kötelező olvasmány.

A többszörözött adatokkal kapcsolatban azonban számos problémával is szembe kell néznünk.

a) Hogyan oldható meg, hogy a másodpéldányok mind azonosak legyenek? A többszörözött adat módosítása lényegében egy olyan osztott tranzakcióvá válik, amely az összes másolatot módosítja.

b) Hogyan döntsük el, hogy hol, mennyi másodpéldányt tároljunk? Minél több másolattal rendelkezünk, annál nagyobb erőfeszítést vesz igénybe az adat módosítása, viszont annál könnyebb lesz a lekérdezések végrehajtása. Egy ritkán módosított relációból például a maximális hatásfok érdekében mindenhol tárolhatunk másolatot, viszont egy gyakran módosított relációból csak egy vagy kettő másodpéldányunk lehetne.
c) Mi történik hálózati kommunikációs hiba esetén, amikor ugyanannak az adatnak különböző másodpéldányai egymástól független változásokon mehetnek keresztül, és amikor a hálózat újra helyreáll, az adat másolatait valahogy össze kell egyeztetni?

10.4.4. Osztott lekérdezésoptimalizálás

A megosztott adatok jelenléte hatást gyakorol a fizikai lekérdezési terv létrehozásakor választható lehetőségekre és a terv összetettségére is (lásd 7.7. részt). Többek között a következő kérdéseket kell eldöntenünk, amikor fizikai tervet választunk:

1. Ha a szükséges R relációnak több másolata is létezik, melyikből vegyük R értékét?
2. Ha egy kétrelációs műveletet, például összekapcsolást alkalmazunk R-re és S-re, akkor számos lehetőség közül kell kiválasztanunk egyet. Néhány ezekből:

   a) Lemásolhatjuk S-t az R-t tároló munkaállomásra, és a számítást ott végezzük el.
   b) Lemásolhatjuk R-t az S-t tároló munkaállomásra, és a számítást ott végezzük el.
   c) Lemásolhatjuk R-t és S-t is egy harmadik helyre, és a számítást ott végezzük el.

   Hogy melyik a legjobb választás, az többek között olyan tényezőktől is függ, hogy melyik helyen van elérhető számítási kapacitás, és hogy a művelet eredményét kombináljuk-e egy harmadik munkaállomáson található adattal. Például ha $(R \bowtie S) \bowtie T$ eredményét kell kiszámítanunk, azt a lehetőséget is választhatjuk, hogy R-t és S-t is továbbítjuk a T-t tároló munkaállomásra, és mindkét összekapcsolást ott végezzük el.

Ha az R reláció az R_1, R_2, ..., R_n töredékek képében van több helyre szétosztva, akkor a logikai lekérdezési terv választásakor R helyett mindenhol az

$$R_1 \cup R_2 \cup \ldots \cup R_n$$

uniót kell írnunk. A lekérdezéstől függően aztán jelentősen egyszerűsíthetjük a kifejezést. Például, ha mindegyik R_i a 10.4.1. részben tárgyalt `Eladások` reláció egy-egy töredéke, és minden töredék egy-egy üzlethez van hozzárendelve, akkor a bostoni áruház eladásaival kapcsolatos lekérdezésekben a bostoni töredéken kívül minden más töredéket elhagyhatunk az unióból.

10.4.5. Feladatok

*!! **10.4.1. feladat:** Ebben a feladatban lehetőségünk lesz néhány olyan probléma felvetésére, amelyek az adattöbbszörözési stratégia kiválasztásakor jönnek elő. Tegyük fel, hogy az R relációhoz n munkaállomás férhet hozzá. Az i-edik munkaállomás másodpercenként q_i lekérdezést és u_i módosítást hajt végre R-en, $i = 1, 2, \ldots, n$. A lekérdezés végrehajtásának a költsége az R relációval rendelkező munkaállomáson c, de ha az adott munkaállomás nem tárol másodpéldányt R-ről, ezért a lekérdezést továbbítani kell egy másikra, a költség $10c$. A módosítás végrehajtása egy „helyi" R-en d, de minden távoli példányon $10d$. Ezen paraméterek függvényében, nagy n esetén, hogyan döntenénk el, hogy mely munkaállomások rendelkezzenek R másodpéldányával és melyek ne?

10.5. Osztott véglegesítés

Ebben a fejezetben arról lesz szó, hogy hogyan biztosítható a különböző helyeken végrehajtott alkotórészekből álló osztott tranzakció atomossága. A következő fejezet az osztott tranzakciók egy másik fontos tulajdonságát, a sorbarendezhetőséget tárgyalja. A következő példán keresztül bemutatjuk, hogy milyen problémák merülhetnek fel.

10.15. példa: Vegyük a 10.4. részben említett üzlethálózatot. Tegyük fel, hogy a hálózat igazgatója minden egyes áruházról tudni akarja, hogy mennyi fogkeféje van raktáron, majd ezek alapján kiegyenlíti a készleteket, azaz néhány áruháznak olyan utasítást fog adni, hogy valamelyik másik áruházba helyezze át a fogkefekészletének egy bizonyos részét. Az egész műveletet egyetlen globális T tranzakció végzi el, amelynek több alkotórésze is van: az i-edik üzletben T_i, az igazgató irodájában pedig T_0. T a következő tevékenységeket hajtja végre:

1. Létrehozza T_0 alkotórészt az igazgató munkaállomásán.
2. T_0 minden áruházba üzenetet küld, utasítva őket, hogy hozzák létre a T_i alkotórészeket.
3. Minden T_i végrehajt egy lekérdezést az i-edik üzletben, amelyből megtudja a raktáron lévő fogkefék számát, majd továbbítja ezt az értéket T_0-nak.
4. T_0 a visszajelzett értékek alapján, valamilyen, itt nem tárgyalt algoritmus segítségével meghatározza, hogy a fogkefeállományt hogyan kell átszervezni. Ezután T_0 olyan üzeneteket küld a megfelelő üzleteknek (ebben az esetben a 7-esnek és a 10-esnek), mint „a 10-es számú áruház szállítson át 500 fogkefét a 7-es számú áruházba".
5. Az üzletek a kapott utasítások alapján módosítják a leltárukat, és végrehajtják a szükséges szállításokat. □

10.5.1. Az osztott atomosság támogatása

A 10.15. példában számos dolog elromolhat, és ezek közül sok eredményezheti azt, hogy sérül T atomossága, azaz néhány műveletét végrehajtja a rendszer, de a többit nem. A naplózás és a helyreállítás mechanizmusa, amelyről feltesszük, hogy minden munkaállomáson jelen van, biztosítja az egyes T_i-k atomosságát, de nem biztosítja, hogy maga T is atomosan fusson le.

10.16. példa: Tegyük fel, hogy a fogkeféket újraosztó algoritmus hibás és ennek következtében a 10-es áruháznak több fogkefét kellene átszállítania, mint amennyi a raktárán van. Ezért T_{10} abortálni fog, és a 10-es üzletből nem szállítanak át egyetlen fogkefét sem, és az üzlet leltárát sem módosítják. T_7 viszont nem talál semmi problémát, és miután a várt fogkefeszállítmánynak megfelelően módosította a 7-es üzlet leltárát, véglegessé válik. Most tehát T nemcsak hogy nem volt atomos (mivel T_{10} soha nem fut le), de ráadásul még inkonzisztens állapotban is hagyta az adatbázist: a leltárban szereplő fogkefék száma nem egyenlő a raktárban ténylegesen megtalálható fogkefék számával. □

A problémák egy másik forrását jelenti annak a lehetősége, hogy egy munkaállomás meghibásodik, vagy leszakad a hálózatról az osztott tranzakció futása közben.

10.17. példa: Tegyük fel, hogy T_{10} még válaszol T_0 első kérdésére, vagyis megküldi a raktáron lévő fogkefék számát, de a 10-es üzlet számítógépe ezután működésképtelenné válik, így T_{10} soha nem kapja meg T_0 utasításait. Véglegessé válhat-e valaha is az osztott T tranzakció? Mit kellene T_{10}-nek tennie, miután a számítógép megjavul? □

10.5.2. Kétfázisú véglegesítés

Annak érdekében, hogy a 10.5.1. részben jelzett problémák elkerülhetők legyenek, az osztott ABKR-ek egy összetett protokollt használnak annak az eldöntésére, hogy egy osztott tranzakció véglegessé váljon-e vagy ne. Ez a *kétfázisú véglegesítés protokoll*[5], és ebben a fejezetben az eljárás alapötletével ismerkedünk meg. Hogy globális döntést hozunk a tranzakció véglegesítéséről, az azt jelenti, hogy vagy minden alkotórésze véglegessé válik, vagy egyetlenegy sem. Ahogy szokásos, most is feltesszük, hogy az egyes munkaállomásokon a lokális alkotórészek vagy véglegessé válnak, vagy nincs hatásuk egyáltalán az adott adatbázisra, vagyis hogy a tranzakció alkotórészei atomosak. Így annak a szabálynak a követésével, hogy az osztott tranzakció vagy minden alkotórésze véglegessé válik, vagy egyik sem, maga az osztott tranzakció is atomossá tehető.

Most következzék néhány kiugróan lényeges dolog a kétfázisú véglegesítés protokollal kapcsolatban:

---

[5] Ne keverjük össze a kétfázisú véglegesítést a kétfázisú zárolással. Ez két, egymástól független elgondolás, egymástól eltérő problémák megoldására.

- Föltesszük, hogy minden munkaállomás naplózza a saját eseményeit, és hogy nincs globális napló.
- Azt is feltesszük, hogy az egyik munkaállomás speciális szerepet játszik annak eldöntésében, hogy az osztott tranzakció véglegessé válhat-e vagy sem. Ezt a munkaállomást *koordinátornak* nevezzük. Koordinátor lehet például az a munkaállomás, ahonnan a tranzakció ered; ez a 10.5.1. rész példáiban a T_0 munkaállomása.
- A kétfázisú véglegesítés protokoll során a koordinátor és a többi munkaállomás bizonyos üzeneteket váltanak egymással. Minden üzenetet küldő munkaállomás naplózza az általa küldött üzeneteket, ezzel segítve az esetleg szükséges helyreállítás műveletét.

Ezeket észben tartva most már jellemezhető a két fázis a munkaállomások közti üzenetcserék leírásával.

Első fázis

A kétfázisú véglegesítés első fázisában a T osztott tranzakció koordinátora eldönti, hogy mikor kísérli meg T véglegesítését. A kísérlet feltehetően akkor történik meg, amikor a koordinátornál futó alkotórész már készen áll a véglegesítésre, de elvben a lépéseket akkor is végre kell hajtani, ha ez az alkotórész abortálni szándékozik (persze nyilvánvaló egyszerűsítések mellett, ahogy ezt majd látni fogjuk). A koordinátor a T tranzakció alkotórészeihez tartozó összes munkaállomást megszavaztatja arról, hogy a véglegesítés vagy az abortálás mellett van-e.

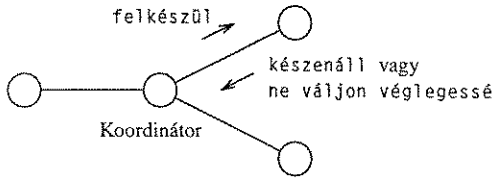
1. A koordinátor a saját naplójába felveszi a <T `Felkészül`> bejegyzést.
2. A koordinátor minden munkaállomásra (elméletileg a sajátjára is) elküldi a T `felkészül` üzenetet.
3. Minden munkaállomás, amely megkapta a T `felkészül` üzenetet, eldönti, hogy a nála található T alkotórészt véglegesíteni vagy abortáltatni akarja. A döntés késleltethető, ha az adott alkotórész még folyamatban van, de a munkaállomásnak végül vissza kell jeleznie.
4. Ha a munkaállomás véglegesíteni akarja az alkotórészt, akkor ennek az *előzetesen véglegesített* állapotba kell lépnie. Az alkotórészt ebben az állapotban a munkaállomás már csak akkor abortáltathatja, ha erre a koordinátortól utasítást kap. Hogy T alkotórésze az előzetesen véglegesített állapotba jusson, a következőket kell megtenni:

   a) Az összes olyan lépést végre kell hajtani, amely annak a biztosításához szükséges, hogy T lokális alkotórészének ne kelljen majd abortálnia, a munkaállomáson bekövetkező rendszerhibát követő helyreállítás során sem. Így nem csak a lokális T műveleteit kell elvégezni, hanem a megfelelő naplózási műveleteket is, hogy egy esetleges helyreállítás során T hatását ne semmisítsük meg, legfeljebb újra futtassuk a tranzakciót. A tényleges műveletek a naplózási módszertől függenek, de a lokális T működéséhez tartozó naplóbejegyzéseket mindeneképpen ki kell írni a lemezre.
   b) Fel kell venni a <T `Készenáll`> bejegyzést a lokális naplóba, és a naplót ki kell írni a lemezre.
   c) El kell küldeni a koordinátornak a T `készenáll` üzenetet.

T alkotórészét azonban még nem véglegesíti ebben a lépésben a munkaállomás; ezzel várnia kell a második fázisig.

1. Ha a munkaállomás inkább abortáltatni akarja az alkotórészt, akkor naplózza a <T `Nem Válik Véglegessé`> bejegyzést és elküldi a T `ne váljon véglegessé` üzenetet a koordinátornak. Biztonságos már ebben a lépésben abortáltatni az alkotórészt, hiszen T is biztosan abortál, még akkor is, ha csak egyetlenegy alkotórész szavazott a véglegesítés ellen.

Az első fázisban váltott üzeneteket a 10.15. ábrán foglaltuk össze.



10.15. ábra. *A kétfázisú véglegesítés első fázisában váltott üzenetek*

Második fázis

A második fázis akkor kezdődik, amikor a koordinátor mindegyik munkaállomástól megkapta a `készenáll` vagy a `ne váljon véglegessé` üzenetet. Lehetséges azonban, hogy valamelyik munkaállomás nem válaszol, talán mert meghibásodott, vagy leszakadt a hálózatról. Ebben az esetben a koordinátor egy megfelelő időkorlát túllépése után úgy fogja venni, mintha ez a munkaállomás a `ne váljon véglegessé` választ küldte volna.

1. Ha a koordinátor a T `készenáll` üzenetet kapta T minden alkotórészétől, akkor a T véglegesítése mellett fog dönteni. A koordinátor

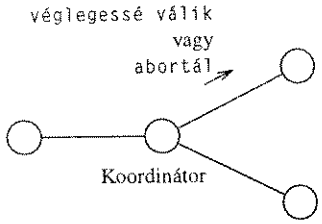
   a) Felveszi a saját naplójába a <`Commit` T> bejegyzést, és
   b) Elküldi a T `véglegessé válik` üzenetet minden T-hez tartozó munkaállomásra.

2. Ha a koordinátor egy vagy több munkaállomástól a `ne váljon véglegessé` üzenetet kapta, akkor

a) Felveszi a saját naplójába az <Abort *T*> bejegyzést, és
b) Elküldi a *T* abortál üzenetet minden *T*-hez tartozó munkaállomásra.

3. Ha egy munkaállomás *T* véglegessé válik üzenetet kap, akkor véglegesíti *T* alkotórészét, és felveszi a naplóba a <Commit *T*> bejegyzést.
4. Ha egy munkaállomás *T* abortál üzenetet kap, akkor abortáltatja *T* alkotórészét, és felveszi a naplóba az <Abort *T*> bejegyzést.

A második fázis üzeneteit a 10.16. ábrán foglaltuk össze.



10.16. ábra. *A kétfázisú véglegesítés második fázisában váltott üzenetek*

10.5.3. Az osztott tranzakciók helyreállítása

A kétfázisú véglegesítés folyamata alatt bármikor meghibásodhat egy munkaállomás. Biztosítanunk kell, hogy ami a helyreállításkor történik, az megfelel annak a döntésnek, amelyet a *T* osztott tranzakcióról hoztunk. Attól függően, hogy az adott helyen mi a *T*-vel kapcsolatos utolsó naplóbejegyzés, a következő eseteket kell végiggondolnunk:

1. Ha a *T*-re vonatkozó utolsó naplóbejegyzés <Commit *T*>, akkor a koordinátor biztosan véglegesítette *T*-t. A naplózási módszertől függően szükség lehet arra, hogy a helyreállítás során *T* alkotórészét újra végrehajtsák.
2. Ha az utolsó feljegyzés <Abort *T*>, akkor az előző esethez hasonlóan itt is ismerjük a globális döntést: *T* abortált. Ha a naplózási módszer igényli, a *T* alkotórész hatását meg kell semmisíteni.
3. Ha az utolsó bejegyzés <*T* Nem válik véglegessé>, akkor a globális döntés biztosan *T* abortálása volt. Ha szükséges, *T* hatását a lokális adatbázison megsemmisítjük.
4. Nehéz a dolgunk, ha *T*-vel kapcsolatban az utolsó feljegyzés <*T* Készenáll>. Ilyenkor a magát helyreállítani próbáló munkaállomás nem tudja, hogy mi lehetett a globális döntés, ezért ezt egy másik munkaállomástól kell megtudnia. Kommunikálhat a koordinátorral, ha az éppen nem működésképtelen, a koordinátor meghibásodása esetén azonban megkérdezhet egy másik munkaállomást is, amely majd a naplója alapján megadja a szükséges információt. Legrosszabb esetben, ha semelyik másik munkaállomással sem tud kapcsolatba lépni, *T* lokális alkotórészét

függőben kell tartania addig, amíg meg nem tudja állapítani, hogy mi volt a *T*-re vonatkozó véglegesítés/abortálás döntés.[6]
5. Az is előfordulhat, hogy a lokális napló egyáltalán nem tartalmaz a kétfázisú véglegesítés protokollal kapcsolatban *T*-re vonatkozó feljegyzést. Ebben az esetben az önhelyreállítást végző munkaállomás egyoldalúan a *T* alkotórész abortáltatása mellett dönthet, ami minden naplózási módszernek megfelel. Lehetséges, hogy a koordinátor egyébként is abortáltatni akarta *T*-t a meghibásodott munkaállomás időkorlát-túllépése miatt. Ha ez csak rövid ideig volt működésképtelen, akkor a többi helyen *T* még mindig aktív lehet, de a lokális *T* alkotórész abortáltatása soha nem okoz problémát, ha a munkaállomás a később kezdődő első fázisban a *T* ne váljon véglegessé választ adja.

A fenti elemzésben feltettük, hogy a meghibásodott munkaállomás nem a koordinátor volt. Ha a koordinátor válik működésképtelenné a kétfázisú véglegesítés folyamata alatt, akkor új problémák merülnek fel.

Először is, a túlélő résztvevőknek vagy meg kell várniuk, amíg a koordinátort helyreállítják, vagy új koordinátort kell választani. Mivel nem lehet előre tudni, hogy a koordinátor mikor lesz újra üzemképes, erős az indíttatás az új vezető választására, legalábbis abban az esetben, amikor a koordinátor még egy rövid várakozási idő eltelte után sem állt helyre.

A *vezetőválasztás* az osztott rendszerek egy méltán összetett problémája – részletesebb vizsgálata túlmutat e könyv célkitűzésein. Egy egyszerű módszer azonban a legtöbb esetben működni fog. Például feltehető, hogy minden résztvevő munkaállomás rendelkezik egy egyedi azonosítóval; az IP-cím sok helyzetben megfelel erre a célra. Minden résztvevő az összes többi résztvevőnek küld egy üzenetet a saját azonosítójával, jelezvén, hogy őt meg lehet választani vezetőnek. Megfelelő idő eltelte után minden résztvevő a legalacsonyabb azonosítószámmal rendelkező munkaállomást ismeri el új koordinátorként azok közül, amelyekről hallott, és erről minden más munkaállomásnak értesítést küld. Ha minden résztvevő azonos üzenetet kap, akkor egyértelmű és mindenki számára ismeretes az új koordinátor személye. Ha az üzenetek nem egyeznek meg egymással, vagy egy résztvevő nem válaszolt, erről szintén mindenki tud majd, és a választást újrakezdhetik.

Most az új vezető szavazás formájában információt gyűjt a munkaállomásoktól a *T* osztott tranzakcióról. Minden munkaállomás elküldi a naplójában található utolsó bejegyzést *T*-vel kapcsolatban, ha van ilyen. A lehetséges esetek a következők:

1. Valamelyik munkaállomás naplójában szerepel a <Commit *T*> bejegyzés. Az eredeti koordinátor bizonyára a *T* véglegessé válik üzenetet akarta mindenhova elküldeni, tehát *T* véglegesítése egy biztonságos megoldás.
2. Hasonlóan, ha valamelyik munkaállomás naplójában az <Abort *T*> bejegyzés sze-

[6] Ez az állapot, *T* blokkolódása az adott csomópontban, akkor is bekövetkezik, ha minden elérhető csomópontban a *T*-vel kapcsolatos utolsó feljegyzés <*T* Készenáll>. *A szerkesztő megjegyzése.*

repel, akkor az eredeti koordinátor biztos, hogy abortáltatni akarta T-t, tehát biztonságos lépés, ha az új koordinátor is ezt teszi.

3. Most tegyük fel, hogy egyetlen munkaállomás naplójában sem szerepel a <Commit T> vagy az <Abort T> bejegyzés, és legalább egy munkaállomás naplójában a <T Készenáll> feljegyzés sem található meg. Mivel az elküldött üzeneteket még a továbbítás előtt naplózzák, világos, hogy a régi koordinátor ettől a munkaállomástól nem kapott T készenáll üzenetet, ezért nem dönthetett T véglegesítése mellett. Az új koordinátor tehát nyugodt lélekkel abortáltathatja T-t.
4. Nehéz a dolgunk abban az esetben, ha egyik helyen sem található <Commit T> vagy <Abort T>, viszont minden túlélő munkaállomás naplójában szerepel a <T Készenáll> bejegyzés. Ilyenkor nem lehetünk biztosak abban, hogy a régi koordinátornak nem volt valamilyen oka T abortáltatására; a saját munkaállomásán történt események, vagy egy másik, jelenleg hibás munkaállomás T ne váljon véglegessé üzenete például megfelelő alapot nyújthatott ehhez a döntéshez. De az is lehet, hogy véglegesíteni akarta T-t, és a saját alkotórésze már véglegessé is vált. Vagyis az új koordinátor nem tudja eldönteni, hogy mit csináljon T-vel, ezért meg kell várnia, amíg az eredeti koordinátor helyreáll. A valódi rendszerekben az adatbázis-adminisztrátornak megvan a lehetősége arra, hogy közbelépjen és manuálisan kényszerítse a várakozó tranzakciókat a továbblépésre. Lehetséges, hogy ezzel sérül a tranzakció atomossága, de a blokkolt tranzakciót végrehajtó személy ennek tudatában megfelelően ellensúlyozza majd ezt a hatást.

10.5.4. Feladatok

! **10.5.1. feladat:** Vegyünk egy olyan T tranzakciót, amelyet egy otthoni számítógépről indítanak, és a segítségével a B bank egy számlájáról 10 000 dollárt akarnak átutalni egy C bankbeli számlára.

* a) Melyek lesznek az osztott T tranzakció alkotórészei? Mi a feladata a B-ben, illetve C-ben található alkotórésznek?

b) Milyen hibát okozhat az, ha a B bankban vezetett számlán nincs 10 000 dollár?

c) Milyen problémát okoz, ha az egyik vagy mindkét bank számítógépe meghibásodik, vagy ha a hálózati kapcsolat megszűnik?

d) Ha a c)-ben felsoroltak közül valamelyik bekövetkezik, hogyan folytatódna helyesen a tranzakció, amikor a számítógépek, illetve a hálózat rendbe jön?

10.5.2. feladat: Ebben a feladatban valahogy jelölnünk kell a kétfázisú véglegesítés folyamata során váltott üzeneteket. Jelentse (i, j, M) a következőt: i munkaállomás az M üzenetet küldi a j munkaállomásnak, ahol M az alábbi értékeket veheti fel: P (jelentése felkészül), R (készenáll), D (ne váljon véglegessé), C (commit), A (abort). Nézzünk meg egy egyszerű esetet, ahol a 0-s sorszámú munkaállomás a koordinátor, amelynek a tranzakcióban máskülönben nincs része, az 1-es és a 2-es munkaállomásokon pedig fut egy-egy alkotórész. Ekkor a tranzakció sikeres véglegesítése során például a következő üzenetsorozat jöhet létre:

$(0, 1, P)$, $(0, 2, P)$, $(2, 0, R)$, $(1, 0, R)$, $(0, 2, C)$, $(0, 1, C)$

* a) Adjunk példát egy olyan üzenetsorozatra, amely akkor jön létre, ha az 1-es munkaállomás a véglegesítés, a 2-es pedig az abortálás mellett van!

*! b) Hány, a fentihez hasonló lehetséges üzenetsorozat jöhet létre, ha a tranzakció sikeresen véglegessé válik?

! c) Hány különböző üzenetsorozat jöhet létre, ha az 1-es munkaállomás a véglegesítés, a 2-es viszont az abortálás mellett van, és feltehető, hogy nem fordul elő semmiféle hibajelenség?

! d) Hány különböző üzenetsorozat jöhet létre, ha az 1-es munkaállomás a véglegesítés mellett szavaz, viszont a 2-es valamilyen meghibásodásnál fogva nem válaszol az üzenetekre?

!! **10.5.3. feladat:** Az előző feladat jelöléseit használva tegyük fel, hogy a koordinátor mellett n másik munkaállomásunk van, és ezek adják a tranzakció alkotórészeit. Adjuk meg n függvényében, hogy hány különböző üzenetsorozat jöhet létre abban az esetben, ha a tranzakció sikeresen véglegessé válik!

10.6. Osztott zárolás

Ebben a fejezetben azzal fogunk foglalkozni, hogy hogyan lehet a zárolás alapú ütemezőket kiterjeszteni egy olyan környezetben, ahol a tranzakciók meg vannak osztva, és különböző helyeken futó alkotórészekből állnak. Feltesszük, hogy a zártáblákat az egyes munkaállomások külön-külön kezelik, és hogy az alkotórész csak azokat az adatbáziselemeket zárolhatja, amelyek azon a helyen megtalálhatók.

Amikor többszörözött adatokkal van dolgunk, úgy kell rendeznünk, hogy egy X adatbáziselem minden másodpéldánya egyformán tükrözze az X-en végrehajtott változtatásokat. Ezért különbséget kell tennünk az X *logikai* adatbáziselem és az X egy vagy több másolatának a zárolása között. Ebben a részben megismerkedünk egy költségmodellel, amelyet az osztott zárolási algoritmusokhoz fejlesztettek ki, és amely akkor is alkalmazható, ha nem többszörözött adatokat használunk. Ennek tárgyalása előtt azonban bemutatjuk a központi zárolás technikáját, amely egy kézenfekvő (és néha éppen megfelelő) megoldás az osztott zárolás problémájára.

10.6.1. Központosított zárolási rendszerek

Talán a legegyszerűbb módszer az, ha a munkaállomások közül kijelölünk egyet, hogy az legyen a *zárállomás* (lock site), azaz hogy az tartsa nyilván a logikai elemek zártábláit, függetlenül attól, hogy rendelkezik-e az elemek másodpéldányával vagy sem. Ha egy tranzakció zárolni akar egy X logikai elemet, akkor ezt az igényét a zárállomásnak nyújtja be, az pedig az adott helyzettől függően vagy engedélyezi, vagy

megtagadja a zárolást. Mivel X globális zárolása megegyezik X lokális zárolásával a zárállomáson, biztosak lehetünk benne, hogy a globális zárak helyesen működnek, feltéve, hogy a lokális zárak adminisztrációja a hagyományos módon történik. A költség általában három üzenet zárolásonként (igénylés, engedélyezés, feloldás), ha a tranzakció nem a zárállomáson fut.

Hogy csak egyetlen zárállomást használunk, ez néhány esetben megfelelő lehet, de ha sok munkaállomáson egyszerre sok tranzakció fut, a zárállomásnál hamar kialakulhat a torlódás. Ráadásul, ha az állomás összeomlik, ez teljesen lebénítja a rendszert, hiszen ilyenkor egyetlen tranzakció sem juthat hozzá semelyik zárhoz, függetlenül attól, hogy éppen hol fut. Ezen problémák miatt számos más módszer is született az osztott zárolás megoldására; ezekre a költségbecslés tárgyalása után kerül sor.

10.6.2. Költségmodell az osztott zárolási algoritmusokhoz

Tegyük fel, hogy minden adatelem pontosan egy helyen fordul elő (vagyis hogy nincs adattöbbszörözés), és hogy az egyes munkaállomásokon működő zárkezelő tartja nyilván az ott elérhető adatelemek zárjait és az azokra benyújtott igényeket. Lehetnek osztott tranzakciók a rendszerben, és mindegyikük egy vagy több különböző helyen futó alkotórészből áll.

A zárkezeléshez ugyan többféle költség is kapcsolódik, de sokuk rögzített, független a hálózaton keresztül történt zárigénylés módjától. Az egyetlen költségtényező, amelyet befolyásolni tudunk, a zárak kiosztásakor és feloldásakor váltott üzenetek száma. Ezért a különböző zárolási sémáknál mindig ezt fogjuk figyelni azon feltevés mellett, hogy az igényelt zárakat mindig ki is osztják. Persze lehet, hogy a zárkezelő nem engedélyezi a zárolást, és ezzel további üzenetek küldésére kerül sor a zárolás megtagadásával, illetve a későbbi engedélyezéssel kapcsolatban. Mivel azonban nem látjuk előre, hogy ez milyen arányban fog előfordulni, és ezt az értéket egyébként sem tudjuk befolyásolni, az összehasonlításban ezeket a pluszüzeneteket nem fogjuk figyelembe venni.

10.18. példa: Ahogy ezt már a 10.6.1. részben említettük, a központi zárolás módszerével a zárkérelmekhez jellemzően három üzenet szükséges: egy az igénylés bejelentéséhez, egy a központi zárállomástól a zár kiadásához, és a harmadik a zár feloldásához. Kivételt a következő esetek jelentenek:

1. Nincs szükség az üzenetekre, ha maga a központi zárállomás igényli a zárolást.
2. További üzenetek küldésére van szükség, ha az első kérelmet nem lehet teljesíteni.

Feltesszük azonban, hogy mindkét eset viszonylag ritkán fordul elő, azaz a legtöbb zárigénylés a központi zárállomástól különböző helyről fut be, és hogy a kérelmek többsége kielégíthető. A zárankénti három üzenet tehát egy jó becslés a központi zár módszer költségére. □

Most vizsgáljunk meg a központi zárolásnál egy rugalmasabb helyzetet, ahol minden X adatbáziselemhez a saját munkaállomásán tartjuk nyilván a hozzá tartozó zárat.

Úgy tűnhet, hogy mivel az X-et zárolni kívánó tranzakció az X munkaállomásán is rendelkezik egy alkotórésszel, a munkaállomások közötti üzenetcserére nincs is szükség: az alkotórész egyszerűen az adott munkaállomás zárkezelőjével tárgyal X-t illetően. Ha azonban az osztott tranzakciónak több elemet is zárolnia kell, mondjuk X-t, Y-t és Z-t is, akkor nem fejezheti be addig a számításait, amíg meg nem szerzi mindhárom elem zárját. Ha X, Y és Z különböző munkaállomásokon található, akkor az ott futó alkotórészeknek legalább szinkronizáló üzeneteket kell cserélniük, hogy a tranzakció nehogy „előbbre járjon saját magánál".

Az összes lehetséges variáció áttekintése helyett csak egy egyszerű modellt veszünk arra, hogy a tranzakciók hogyan gyűjtik be a szükséges zárakat. Feltesszük, hogy minden tranzakciónak az egyik alkotórésze, a *zárkoordinátor* felelős az egyes alkotórészek által igényelt zárak összegyűjtéséért. A zárkoordinátor a saját munkaállomásán üzenetcsere nélkül zárolja az elemeket, de egy másik helyen található X zárolásához három üzenet szükséges:

1. Az igény benyújtása X munkaállomására.
2. A válasz üzenet, amellyel a zárolást engedélyezik (korábban feltettük, hogy az igényeket azonnal kielégítik; ha mégsem, akkor ebben a pontban az üzenet a zárolás elutasítására vonatkozik, amelyet majd később követ az engedélyező üzenet).
3. A zárról való lemondás továbbítása X munkaállomására.

Mivel az osztott zárolási protokollokat csak összehasonlítani akarjuk és nem kívánjuk megadni az üzenetek átlagos számát, ez a leegyszerűsítés megfelel a céljainknak.

Ha azt a munkaállomást választjuk zárkoordinátornak, ahonnan a legtöbb zárat kell beszereznie a tranzakciónak, akkor minimalizáljuk a szükséges üzenetek számát. Ez a többi munkaállomáson található adatbáziselemek számának háromszorosával egyenlő.

10.6.3. Többszörözött elemek zárolása

Óvatosan kell az X adatelem zárolását értelmeznünk, amikor X-ből különböző helyeken másodpéldányok is megtalálhatók.

10.19. példa: Tegyük fel, hogy az X adatbáziselem két példányban létezik, ezek X_1 és X_2. Tegyük fel továbbá, hogy a T tranzakció X_1 munkaállomásán osztott zár alá helyezte X_1-et, U viszont kizárólagos zárat birtokol X_2 munkaállomásán X_2-n. Ilyenkor U csak X_2-t módosíthatja, X_1-et nem, és ez azt eredményezi, hogy X két példánya egymástól eltérő lesz. Sőt mivel T és U más elemeket is zár alá helyezhet, és az, hogy milyen sorrendben olvassák, illetve írják X-et, független a másodpéldányokon birtokolt záraktól, T-nek és U-nak arra is lehetősége van, hogy nem sorba rendezhető módon viselkedjenek. □

A 10.19. példában bemutatott probléma lényege, hogy többszörözött adatok esetén különbséget kell tennünk az X logikai elem osztott, illetve kizárólagos zárolása, és az

X egy másodpéldányának az adott munkaállomáson történő helyi zárolása között. Vagyis annak érdekében, hogy biztosítani lehessen a sorbarendezhetőséget, a tranzakcióknak globálisan kell zárolniuk a logikai elemeket. Viszont a logikai elemek fizikailag nem léteznek – csak a másodpéldányaik –, és globális zártábla sincs. Így a tranzakció csak egyetlen módon juthat hozzá X globális zárjához: ha megszerzi X egy vagy több másodpéldányán az adott munkaállomáson a lokális zárakat. Most olyan módszerekkel fogunk foglalkozni, amelyek segítségével a lokális zárak globális zárakká alakíthatók, és rendelkeznek a szükséges tulajdonságokkal:

- Semelyik két tranzakció sem birtokolhat globális kizárólagos zárat egy X logikai elemen ugyanabban az időben.
- Ha egy tranzakció birtokában van az X logikai elem globális kizárólagos zárja, akkor semelyik másik tranzakció nem birtokolhat globális osztott zárat X-en.
- Bármennyi tranzakció rendelkezhet X globális osztott zárjával, amíg nincs olyan tranzakció, amely birtokolná a globális kizárólagos zárat.

10.6.4. Az elsődleges példány zárolása

A központi zárolás módszere továbbfejleszthető úgy, hogy megosztjuk a zárállomás feladatát, de továbbra is ragaszkodunk ahhoz az elképzeléshez, hogy minden logikai elemhez tartozik egy olyan egyedi munkaállomás, amely az elem globális zárjáért felelős. Ezt az osztott zárolási módszert az *elsődleges példány* (primary copy) módszerének nevezzük. Ezzel a változtatással sikerül a központosított módszer egyszerűségét megőrizni, ugyanakkor elkerülhető annak a lehetősége, hogy a központi zárállomás túlterheltté váljon.

Az elsődleges példány zárolási módszerben minden X logikai elem másodpéldányai közül kijelölünk egyet „elsődleges példánynak". Ha egy tranzakció zárolni szeretné az X logikai elemet, akkor arra a munkaállomásra kell elküldenie az igényét, ahol X elsődleges példánya található. Ez a munkaállomás tartja nyilván X-et a helyi zártáblázatban, és az adott helyzettől függően vagy engedélyezi, vagy megtagadja a zár kiadását. Az előző módszerhez hasonlóan a globális (logikai) zárak helyes működése itt is az elsődleges példányhoz tartozó zárak megfelelő helyi nyilvántartásán múlik.

Ahogy ezt a központi zárállomás esetében láttuk, ennél a módszernél is a legtöbb zárigénylés három üzenettel jár, kivéve, ha a tranzakció az elsődleges példány munkaállomásán fut. Ha azonban az elsődleges példányokat okosan választjuk, várhatóan ez sokkal gyakrabban fog előfordulni, mint az ellenkezője.

10.20. példa: Az üzletláncos példánkban a vásárlásokra vonatkozó adatok elsődleges példányait úgy kellene kiválasztanunk, hogy azok annak az üzletnek az adatbázisában legyenek, ahol maga a vásárlás történt. Az adat többi példánya, amely például a központi részlegben vagy az elemzők által használt adattárházban van, nem elsődleges. A jellemző tranzakciók valószínűleg az áruházakban futnak le, és csak az adott áruházra vonatkozó vásárlási adatokat módosítják. Amikor egy ilyen típusú tranzakció zárolja

Osztott holtpontok

Miközben a tranzakció többszörözött adatot próbál globális zár alá helyezni, számos alkalma van arra, hogy holtpontra jusson. Annak is számos módja van, hogy globális várakozási gráfot készítsünk, és így észlelni tudjuk a holtpontok kialakulását. Osztott környezetben azonban gyakran az a legegyszerűbb és leghatékonyabb megoldás, ha az időkorlátos módszert alkalmazzuk. Bármely tranzakcióról, amely a megfelelő időmennyiség eltelte után még mindig nem zárult le, feltesszük, hogy holtpontra jutott, és ezért visszagörgetjük.

az elemeket, akkor nincs szükség üzenetek küldésére. A zárolással kapcsolatos üzenetekre csak akkor lenne szükség, ha a tranzakció egy másik áruház adatait vizsgálná vagy módosítaná. □

10.6.5. A lokális zártól a globálisig

Egy másik megoldás, ha több, összegyűjtött lokális zárból képzünk globális zárat. Ezekben a sémákban az X adatbáziselem egyik példánya sem „elsődleges" – az elem másolatai szimmetrikusak, és bármelyikükre igényelhető lokális osztott vagy kizárólagos zár. Egy jól működő globális zároló séma nyitja, hogy a tranzakcióknak be kell gyűjteniük bizonyos számú lokális zárat X példányain ahhoz, hogy az X-en lévő globális zárat birtokolhassák.

Tegyük fel, hogy az A adatbáziselem n példányban létezik. Választunk két értéket:

1. s – A példányai közül s számút kell osztott módban zárolni ahhoz, hogy egy tranzakció globális osztott módban zárolhassa A-t.
2. x – A példányai közül x számút kell kizárólagos módban zárolni ahhoz, hogy egy tranzakció globális kizárólagos módban zárolhassa A-t.

Ha $2x > n$ és $s + x > n$, akkor megvannak a kellő tulajdonságok: csak egy globális kizárólagos zár létezhet A-n, és nem létezhet egyszerre globális osztott és globális kizárólagos zár rajta. Ezeket a következőképpen magyarázhatjuk: ha két tranzakció birtokában is lenne egy-egy globális kizárólagos zár A-n, akkor mivel $2x > n$, A-nak legalább egy másodpéldányán mindkét tranzakció lokális kizárólagos zárat birtokolna (mert több lokális kizárólagos zár van kiosztva, mint ahány másodpéldánya van A-nak). Ekkor azonban a lokális zárolási módszer nem működne helyesen. Hasonlóan, ha egy tranzakció globális osztott módban, egy másik pedig globális kizárólagos módban zárolja A-t, akkor, mivel $s + x > n$, valamelyik másodpéldányon az egyikük lokális osztott, a másikuk pedig lokális kizárólagos zárat birtokolna egyszerre.

Általában, a globális osztott, illetve kizárólagos zár megszerzéséhez szükséges üzenetek száma rendre $3s$, illetve $3x$. Ez a szám igen nagynak tűnik, összehasonlítva a

központi módszerekkel, ahol záranként átlagosan három vagy annál kevesebb üzenet szükséges. Vannak azonban ezt ellensúlyzó tulajdonságai a rendszernek speciális (s, x) választása esetén, ahogy ezt a következő két példában látjuk:

- *Egy-olvasás-zár; Minden-írás-zár.* Itt $s = 1$, $x = n$. A globális kizárólagos zár megszerzése nagyon drága, de globális osztott zár esetén legfeljebb három üzenet is elég. Ráadásul ennek a sémának van egy előnye az elsődleges példány módszerével szemben: míg az utóbbival elkerülhető az üzenetek küldése, ha az elsődleges példányt olvassuk, addig az egy-olvasás-zár sémával ugyanez megtehető, valahányszor a tranzakció egy olyan helyen fut, ahol megtalálható az olvasni kívánt adatbáziselem *akármelyik példánya*. Így ez a séma jobban megfelelhet, amikor a tranzakciók többsége csak olvas, de az X-et olvasó tranzakciók más-más helyeken futnak. Példának hozható fel az osztott digitális könyvtár, amely egy dokumentum másodpéldányait azokon a helyeken tárolja el, ahol azokat gyakran olvassák.
- *Többségi zárolás.* Itt $s = x = \lceil (n+1)/2 \rceil$. Úgy tűnik, ebben a rendszerben a tranzakció helyétől függetlenül mindenképpen sok üzenetre lesz szükség. Van azonban számos más tényező, amely figyelembe vételével már elfogadhatóvá válik a séma. Ezek közé tartozik, hogy sok hálózati rendszer támogatja a *csomagszórást*[7], amely segítségével lehetővé válik, hogy a tranzakció egyetlen általános igény elküldésével próbálja begyűjteni az X elem lokális zárjait, hiszen ez az üzenet minden munkaállomáshoz eljut. Hasonlóan, a zárak feloldásához is elég lehet egyetlen üzenet. A módszer javára írandó még az is, hogy s és x értékének fenti megválasztása az üzenetek nagy számának ellenére olyan előnnyel jár, amely más értékek esetén nem érhető el: lehetőség van a rendszer részleges működésére, még akkor is, ha a hálózat több részre szakadt. Amíg a hálózatnak van olyan része, amely az X másodpéldányait tároló munkaállomások többségét tartalmazza, addig lehetősége van a tranzakciónak arra, hogy zárolni próbálja X-et. Ha más, a hálózatról leszakadt munkaállomások aktívak is maradnak, még osztott zárat sem kaphatnak X-en, így nincs veszélye annak, hogy a hálózat különböző (egymástól elszakadt) részein futó tranzakciók nem sorba rendezhető módon fognak viselkedni.

10.6.6. Feladatok

! **10.6.1. feladat:** Megmutattuk, hogyan lehet lokális osztott, illetve kizárólagos zárból a globális változatot létrehozni. Hogyan hoznánk létre a megfelelő típusú lokális zárból a következőket:

* a) Globális osztott, kizárólagos és növelési zárakat.
b) Globális osztott, kizárólagos és módosítási zárakat.
!! c) Globális osztott, kizárólagos és mindenféle típusú szándékjelölő zárakat.

[7] Angolul *broadcast*. Olyan üzenettovábbítás, amely a hálózat minden elemének szól. *A fordító megjegyzése.*

10.6.2. feladat: Tegyük fel, hogy van öt munkaállomás, amelyek mindegyike rendelkezik az X adatbáziselem egy-egy másodpéldányával. Ezek közül P a legfontosabb X-re nézve, és az elsődleges példány osztott zárolási rendszerben ezt használjuk X elsődleges munkaállomásaként. Az X elérésére vonatkozó statisztikák a következők:

i) A hozzáférések 50%-a P-ből ered és ezek során X-et csak olvassák.
ii) A többi négy munkaállomás mindegyike egyenként 10%-ban igényel hozzáférést, és ezek csak olvasó-hozzáférések.
iii) A hozzáférések maradék 10%-a kizárólagos, és az öt munkaállomás közül akármelyik igényelheti egyenlő valószínűséggel (azaz mindegyik 2%-ban igényli).

Az alábbi zárolási módszerek mindegyikéhez adjuk meg az egyes zárak megszerzéséhez szükséges üzenetek átlagos számát! Feltehető, hogy minden kérelmet jóváhagynak, azaz hogy elutasító üzenetekre nem lesz szükség.

* a) Egy-olvasás-zár; minden-írás-zár.
b) Többségi zárolás.
c) Elsődleges példány zárolás, az elsődleges példány P-nél található.

10.7. Hosszú tranzakciók

Az adatbázis-alkalmazásoknak van egy olyan csoportja, ahol az adatok nyilvántartásához megfelel az adatbázisrendszer, de a sok, rövid tranzakciós modell, amelyre az adatbázis konkurenciavezérlő mechanizmusai alapoznak, alkalmatlan. Ebben a fejezetben ilyen alkalmazásokkal és a velük kapcsolatban felmerülő problémákkal foglalkozunk, majd bemutatjuk a „kiegyenlítő tranzakciókon" alapuló megoldást. Ezek a tranzakciók érvénytelenítik az olyan véglegesített tranzakciók hatását, amelyeknek nem kellett volna véglegessé válniuk.

10.7.1. A hosszú tranzakciók problémái

Durván fogalmazva a *hosszú tranzakció* egy olyan tranzakció, amely túl sokáig tart ahhoz, hogy megengedhető legyen számára az, hogy olyan elemeket tartson zár alatt, amelyekre más tranzakcióknak is szükségük van. A „túl hosszú" a környezettől függően jelenthet másodperceket, perceket vagy órákat; mi feltesszük, hogy egy „hosszú" tranzakció legalább percekig, talán órákig is eltart. A hosszú tranzakciókkal kapcsolatos alkalmazások három nagy osztálya a következő:

1. *Hagyományos ABKR-alkalmazások.* A közönséges adatbázis-alkalmazások főleg rövid tranzakciókat futtatnak, sok esetben azonban szükség van alkalmanként hosszú tranzakciókra is. Például egy tranzakció végigvizsgálhatja egy bank összes

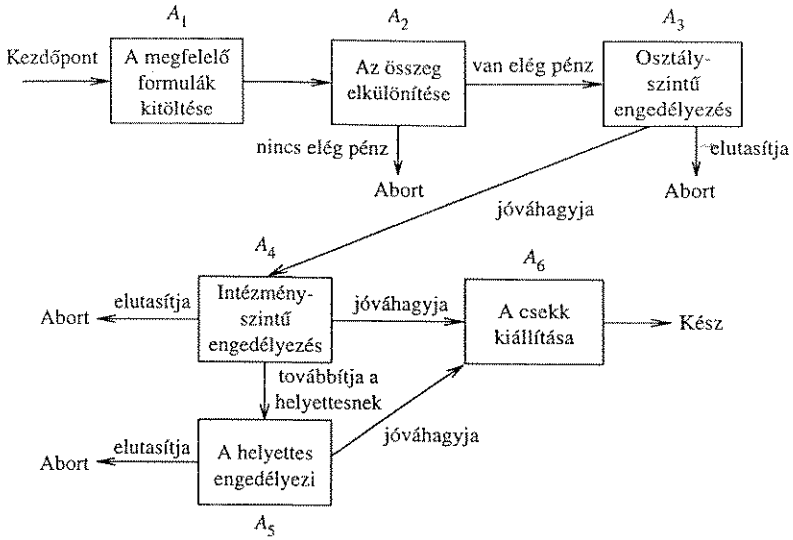
számláját, hogy megállapítsa, hogy a végösszeg helyes-e, egy másik alkalmazásban pedig alkalmanként szükség lehet egy-egy index újraszervezésére, hogy a teljesítmény továbbra is maximális legyen.

2. *Tervezőrendszerek.* Függetlenül attól, hogy a tervezni kívánt eszköz mechanikus (például gépkocsi) vagy elektronikus (például mikroprocesszor vagy programrendszer), a tervezőrendszerek egy dologban mindig megegyeznek: a terv alkotóelemeire van szétbontva (például fájlokra egy felhasználói program esetén), és a különböző részeken különböző tervezők dolgoznak egyszerre. Nem akarjuk viszont, hogy két tervező egyszerre ugyanazon a fájlon dolgozzon, hiszen ezzel csak felülírnák egymás munkáját, ezért egy *ki- és bejelentkező* rendszer segítségével lehetővé tesszük, hogy a tervező „kijelentsen" egy fájlt, majd a változtatások végeztével, talán órákkal vagy napokkal később, bejelentse. Egy tervező azonban valamilyen oknál fogva talán akkor is bele akar nézni egy fájlba, amikor azon éppen egy másik tervező dolgozik. Ha a fájl kijelentése egyenértékű lenne egy kizárólagos zárolással, akkor néhány indokolt és ésszerű műveletet talán csak napokkal később lehetne végrehajtani.
3. *Munkafolyamat-rendszerek.* Ezek a rendszerek olyan folyamatokat foglalnak magukban, amelyek közül néhányat egy szoftver hajt végre egyedül, néhánynak emberi beavatkozás is része, és néhány talán csak ember által végrehajtott műveletekből áll. Rövidesen láthatunk egy példát, amely egy számla kifizetésével járó hivatalos papírmunkát mutat be. Az ilyen alkalmazások végrehajtása napokig is eltarthat, és ez alatt az idő alatt lehet, hogy néhány adatbáziselemen változtatni kell. Ha a rendszer engedélyezné a tranzakcióhoz szükséges adatok kizárólagos zárolását, akkor a többi tranzakció napokig nem férhetne hozzájuk.

10.21. példa: Vegyük azt az esetet, amikor egy alkalmazott megpróbálja elszámoltatni az utazási költségeit. Azt szeretné, hogy a cége az A123-as számláról visszafizesse a kiadásait – a költségtérítés folyamata a 10.17. ábrán látható. Az eljárás az A_1 művelettel kezdődik, amikor az utazó titkárnője egy on-line kérdőív kitöltésével leírást ad az utazásról, megadja a terhelendő számla számát és a visszatérítendő összeget. Feltesszük, hogy ebben a példában a számlaszám A123 és az összeg 1000 dollár.

Az utazó számláit fizikailag átküldik az osztályos titkárságra, a kitöltött formula pedig a hálózaton keresztül eljut az A_2 automatikus eljáráshoz. A_2 ellenőrzi, hogy a megadott számlán (A123) van-e elég pénz a költségek (1000 dollár) fedezésére, és ha van, akkor a számlán elkülönít egy ennek megfelelő összeget; vagyis próbaképpen levon a számlán található pénzmennyiségből 1000 dollárt, de erről még nem állít ki csekket. Ha nincs elég pénz a számlán, akkor a tranzakció abortál, és feltehetően majd újraindul, vagy amikor már rendelkezésre áll a kívánt pénzösszeg, vagy ha egy másik számláról kérik a kifizetést.[8]

[8] Természetesen az utazó (aki úgysem a stanfordi egyetemen dolgozik) *soha* sem írná a költségeit – helytelenül – egy másik kormányszámla terhére, hanem megfelelő pénzügyi forrásokat keresne. Ezt azért kell hangsúlyoznunk, mert a kormány számvevői, akiknek fogalmuk sincs arról, hogy egy egyetemnek hogyan kéne működnie, még mindig itt nyüzsögnek Stanford körül.



10.17. ábra. *Munkafolyamat-ábra az utazási költségtérítésekhez*

Az A_3 műveletet, amelyre lehet, hogy csak napokkal később kerül sor, az osztály ügyintézője hajtja végre: megvizsgálja a benyújtott számlákat és az on-line formulát. Ha mindent rendben talál, a kérvényt jóváhagyja, és a számlákkal együtt továbbküldi egy magasabb adminisztrációs szintre, az intézményes ügyintézőhöz. Ellenkező esetben a tranzakció abortál, az utazónak pedig feltehetően módosított formában újra be kell adnia a jelentkezését.

Az A_4 művelet során, amelyre lehet, hogy csak napokkal később kerül sor, az intézményes ügyintéző vagy jóváhagyja, vagy elutasítja a kérelmet, vagy továbbadja a helyettesének, aki az A_5 műveletben dönt. Ha a kérvényt elutasítják, akkor a tranzakció abortál, és a jelentkezést újra be kell adni. Jóváhagyás esetén az A_6 művelet során kiállítják a csekket, és véglegesítik az 1000 dollár levonását a számláról.

Tegyük fel, hogy ez a munkafolyamat csak a hagyományos zárak használatával valósítható meg. Ekkor például, mivel a sikeresen lezárult tranzakció megváltoztathatja az A123-as számla egyenlegét, az A_2 műveletnél kizárólagos zár alá kell helyezni a számlát, és azt addig nem szabad feloldani, amíg a tranzakció nem abortál, vagy az A_6 művelet le nem zárul. A zárat talán napokig fenn kell tartani, amíg a kérvény elbírálásával megbízott emberek végre foglalkozni tudnak az üggyel. Ilyenkor viszont az A123-as számláról semmilyen más kifizetés sem történhet, még próbaképpen sem. Másrészről viszont, ha a számlához való hozzáférést egyáltalán nem szabályozzuk, akkor különböző tranzakciók egyszerre terhelhetik a számlát, illetve foglalhatnak le a számlán különféle összegeket, ez pedig a hitelkeret túllépéséhez vezethet. A két szélsőséges módszer helyett tehát egy kompromisszumos megoldást kell találni. □

10.7.2. Regék

A *rege* (saga) olyan műveletek összessége, amelyek együtt egy hosszú „tranzakciót" alkotnak (ilyen műveleteket láttunk a 10.21. példában). A rege tehát a következő részekből áll:

1. Valamilyen műveletek összessége.
2. Egy gráf, amelyben a csúcspontok vagy műveletek, vagy a speciális *Abort*, illetve *Kész* csúcsok, az élek pedig a csúcsok között futnak. A kétféle speciális csúcsból, amelyeket *végpontoknak* (terminal node) nevezünk, nem indulnak ki élek.
3. Azon csúcspont megjelölése, ahonnan az egész folyamat indul. Ez a csúcs a *kezdőpont* (start node).

A kezdőpontból akármelyik végpontba vezető út a műveletek egy lehetséges sorozatát adja. Az *Abort* csúcspontba vezető utak olyan műveletsorozatot jelentenek, amely után az egész tranzakciót vissza kell görgetni, a műveleteknek pedig változatlanul kell hagyniuk az adatbázist. A *Kész* csúcsba vezető utak sikeres műveletsorozatokat jelentenek, és az ezen műveletek által az adatbázisrendszeren véghezvitt változtatásoknak meg kell maradniuk az adatbázisban.

10.22. példa: A 10.17. ábrán látható gráfban az *Abort* csúcsba vezető utak a következők: A_1A_2, $A_1A_2A_3$, $A_1A_2A_3A_4$ és $A_1A_2A_3A_4A_5$. A *Kész* csúcsba vezetnek $A_1A_2A_3A_4A_6$ és $A_1A_2A_3A_4A_5A_6$. Vegyük észre, hogy most a gráf nem tartalmaz kört, vagyis a végpontokba vezető utak száma véges. Általános esetben azonban létrejöhetnek körök a gráfban, és ilyenkor az utak száma végtelen is lehet.[9] □

A regék konkurenciavezérlésére kétféle lehetőség kínálkozik:

1. Minden egyes műveletet tekinthetünk egy-egy önálló (rövid) tranzakciónak, amely a végrehajtáskor egy olyan hagyományos konkurenciavezérlő mechanizmust használ, mint a zárolás. Az A_2 művelet például megvalósítható úgy is, hogy arra a (rövid) időre, amíg az A123-as számla egyenlegét csökkenti az utazási elismervényen jelölt összeggel, zárolja a számlát, majd a zárat feloldja. A zárolás segítségével megelőzhető, hogy egyszerre két tranzakció próbáljon a számla egyenlegének új értéket adni, amivel az első változtatás hatása elveszne, és „csodával határos módon pénz jelenne meg a számlán".
2. Az egész tranzakciót, amelyet a végpontokba vezető utak bármelyike jelenthet, a „kiegyenlítő tranzakciók" mechanizmusán keresztül kezeljük, amelyek a rege csúcspontjaiban található tranzakciók inverzei. Feladatuk az, hogy a véglegessé vált műveletek hatását olyan módon görgessék vissza, amely nem függ attól, hogy mi történt az adatbázissal a véglegesített művelet és a kiegyenlítő tranzakció végrehajtása közötti időben. A kiegyenlítő tranzakciókról a következő részben lesz szó.

[9] A bürokrácia útvesztői... *A fordító megjegyzése.*

Mikor „ugyanaz" két adatbázis-állapot?

A kiegyenlítő tranzakciók tárgyalása közben óvatosnak kell lennünk azzal kapcsolatban, hogy mit jelent az adatbázist „ugyanabba" az állapotba visszaállítani, mint amelyben előtte volt. Már kaptunk egy kis ízelítőt ebből a problémából, amikor a 10.6. példában a B-fák logikai naplózásáról volt szó. Láthattuk, hogy amikor „visszaállítottunk" egy műveletet, a B-fa állapota nem feltétlenül volt azonos a művelet előtti állapottal, viszont ekvivalens volt vele a B-fa elérési műveleteit illetően. Általánosabban fogalmazva, megtörténhet, hogy egy műveletet és a hozzá tartozó kiegyenlítő tranzakciót végrehajtva, az adatbázis nem áll visszabitről bitre ugyanabba az állapotba, amelyben előtte volt, de a különbségeket nem érzékelheti egyetlen, az adatbázis által támogatott felhasználói program sem.

10.7.3. Kiegyenlítő tranzakciók

Egy regében minden A művelethez tartozik egy *kiegyenlítő tranzakció* (compensating transaction), amit A^{-1}-gyel jelölünk. Ha végrehajtjuk A-t, majd később A^{-1}-et, akkor eredményül ugyanazt az adatbázis-állapotot kapjuk, mintha sem A-t, sem A^{-1}-et nem hajtottuk volna végre. Formálisabban fogalmazva:

- Ha D egy tetszőleges adatbázis-állapot, $B_1B_2...B_n$ pedig műveletek és kiegyenlítő tranzakciók sorozata (akár a kérdéses regéből, akár egy másikból vagy akármilyen más tranzakcióból, amely szabályosan fut az adatbázison), akkor ugyanazt az adatbázis-állapotot kapjuk D-ből a $B_1B_2...B_n$ sorozat futtatása után, mint $AB_1B_2...B_nA^{-1}$ futtatása után.

Ha a rege végrehajtása az *Abort* csúcspontba vezet, akkor a visszagörgetés úgy történik, hogy a végrehajtott műveletek fordított sorrendjében lefuttatjuk az egyes műveletekhez tartozó kiegyenlítő tranzakciókat. A kiegyenlítő tranzakciók fent említett tulajdonsága miatt a rege hatása érvénytelenné válik, és az adatbázis-állapota ugyanaz lesz, mintha a regét soha nem is hajtottuk volna végre. Hogy miért biztos az, hogy a visszagörgetett rege hatása érvénytelenné válik, arra részletesebb magyarázatot a 10.7.4. részben találunk.

10.23. példa: Vegyük a 10.17. ábra műveleteit, és nézzük meg, hogy mik lehetnek a megfelelő kiegyenlítő tranzakciók. Elsőként A_1 létrehoz egy on-line dokumentumot. Ha ezt az adatbázisban tároljuk, akkor A_1^{-1}-nek ezt el kell onnan távolítania. Vegyük észre, hogy ez a kiegyenlítés engedelmeskedik a kiegyenlítő tranzakciók alaptulajdonságának: ha létrehozzuk a dokumentumot, és összeállítunk egy α műveletsorozatot (amely akár a dokumentum törlését is tartalmazhatja, ha akarjuk), akkor $A_1\alpha A_1^{-1}$ hatása megegyezik α hatásával.

A_2 implementálásával óvatosan kell bánnunk. A pénzt úgy „foglaljuk le", hogy a megfelelő mennyiséget levonjuk a számlaegyenlegből. Ez a pénz egészen addig törölve marad, amíg az A_2^{-1} kiegyenlítő tranzakció vissza nem helyezi a számlára. Azt állítjuk, hogy A_2^{-1} kiegyenlítő tranzakció helyesen fog működni, ha a számlavezetés általános szabályait követjük. Hogy jobban ráérezzünk a probléma lényegére, hasznos, ha megnézünk egy hasonló tranzakciót, ahol a logikusnak tűnő kiegyenlítés nem működik. A 10.24. példában látunk majd egy ilyen esetet.

Az A_3, A_4 és A_5 műveletek mindegyikében egy jóváhagyással egészül ki az on-line nyomtatvány. A kiegyenlítő tranzakciók tehát egyszerűen eltávolíthatják ezeket a jóváhagyásokat.[10]

Végül A_6-hoz, a csekket kiállító művelethez nem tudunk egyértelműen kiegyenlítő tranzakciót megadni. A gyakorlatban erre nincs is szükség, hiszen ha A_6-ot egyszer már végrehajtottuk, akkor a regét már nem görgethetjük vissza. A_6-nak viszont a szó szoros értelmében úgy sincs hatása az adatbázisra, mivel a csekket fedező összeget a számláról már levonta A_2. Ha az „adatbázist" esetleg tágabb határok között kéne értelmeznünk, ahol a csekk beváltásához hasonló eseményeknek hatásuk van az adatbázisra, akkor A_6^{-1}-et úgy kéne megterveznünk, hogy először próbálja meg a csekket visszavonni, ha ez nem megy, akkor levélben követelje vissza a pénzt attól, aki a csekket beváltotta, végül, ha ez a kísérlet is kudarcba fulladt, nyilatkozzon a behajthatatlan követelés miatti veszteségről, és állítsa vissza az egyenleg eredeti értékét. □

Most fogjunk hozzá a 10.23. példában említett eset felvázolásához, ahol is a számlán végrehajtott változtatást nem lehet kiegyenlíteni a változtatás inverzével. A problémát az okozza, hogy a számlák egyenlege általában nem lehet negatív.

10.24. példa: Tegyük fel, hogy B egy olyan tranzakció, amely 1000 dollárt rak egy olyan számlára, amelyen eredetileg 2000 dollár van, B^{-1} pedig a kiegyenlítő tranzakció, amely ugyanezt az összeget leszedi a számláról. Ésszerű továbbá azt is feltennünk, hogy az a tranzakció, amely egy számláról a rendelkezésre álló összegnél nagyobbat akar eltávolítani, nem mindig futhat le sikeresen. Legyen C egy olyan tranzakció, amely 2500 dollárt emel le ugyanarról a számláról. Ekkor BCB^{-1} nem ekvivalens C-vel. Ennek az az oka, hogy C magában nem fut le sikeresen, így a számlán ott marad 2000 dollár, viszont ha először B-t, majd C-t is végrehajtjuk, a számlán 500 dollár marad, ami után B^{-1} nem zárulhat le sikeresen.

Azt a következtetést kell levonnunk, hogy egy számlák közti tetszőleges átutalásokból álló regét és azt a szokást, hogy a számlák nem mehetnek negatívba, nem lehet egyszerűen kiegyenlítő tranzakciókkal támogatni. Valamilyen változtatást kell bevezetnünk a rendszerbe, például megengedhetjük a negatív egyenleget is. □

[10] A 10.17. ábrán bemutatott regében ezeket a műveleteket csak akkor kell kiegyenlíteni, amikor a nyomtatványt különben is megsemmisítenék. A kiegyenlítő tranzakciók definíciója azonban megköveteli, hogy a tranzakciók egymástól elkülönítve is működjenek, arra való tekintet nélkül, hogy valamelyik másik tranzakció esetleg lényegtelenné teszi az általuk eszközölt változtatásokat.

10.7.4. Miért működnek jól a kiegyenlítő tranzakciók?

Azt mondjuk, hogy két műveletsorozat *ekvivalens* ($\equiv$), ha bármely D adatbázis-állapotot azonos állapotba visznek. A kiegyenlítő tranzakciókról tett alapvető feltevésünk ekkor a következő alakban írható fel:

- Ha A egy művelet, α pedig legális műveletek és kiegyenlítő tranzakciók sorozata, akkor $A\alpha A^{-1} \equiv \alpha$.

Most azt kell megmutatnunk, hogy ha egy $A_1A_2\ldots A_n$ rege végrehajtása után a fordított sorrendben vett kiegyenlítő tranzakciók is lefutnak, akkor akármilyen műveletek is estek a rege műveletei illetve a tranzakciók közé, ennek olyan a hatása, mintha sem a rege műveletei, sem a kiegyenlítő tranzakcióik nem futottak volna le egyáltalán. Ezt az állítást n-re vonatkozó indukcióval bizonyítjuk.

Alap: Ha $n = 1$, akkor az A_1 és az ezt kiegyenlítő A_1^{-1} tranzakció közti műveletsorozat: $A_1\alpha A_1^{-1}$. A kiegyenlítő tranzakciókról való alapfeltevés szerint $A_1\alpha A_1^{-1} \equiv \alpha$; azaz a regének nincs hatása az adatbázis-állapotra.

Indukció: Tegyük fel, hogy az állítás teljesül minden legfeljebb $n-1$ műveletből álló útra. Vegyük most egy n hosszú út műveleteit, amelyeket a fordított sorrendben vett kiegyenlítő tranzakciók követnek, és a műveletek és a tranzakciók közé másféle műveletek is eshetnek. A sorozatot ilyen alakban írhatjuk fel:

$$A_1\alpha_1A_2\alpha_2\ldots\alpha_{n-1}A_n\beta A_n^{-1}\gamma_{n-1}\ldots\gamma_2A_2^{-1}\gamma_1A_1^{-1} \qquad (10.1.)$$

ahol minden görög betű nulla vagy több művelet sorozatát jelöli. A kiegyenlítő tranzakciók definíciója miatt $A_n\beta A_n^{-1} \equiv \beta$. Vagyis (10.1.) ekvivalens a következővel:

$$A_1\alpha_1A_2\alpha_2\ldots A_{n-1}\alpha_{n-1}\beta\gamma_{n-1}A_{n-1}^{-1}\gamma_{n-1}\ldots\gamma_2A_2^{-1}\gamma_1A_1^{-1} \qquad (10.2.)$$

Az indukciós feltevés miatt pedig ez ekvivalens

$$\alpha_1\alpha_2\ldots\alpha_{n-1}\beta\gamma_{n-1}\ldots\gamma_2\gamma_1$$

kifejezéssel, hiszen (10.2.)-ben csak $n-1$ művelet szerepel. Tehát ha a rege után a kiegyenlítését is lefuttatjuk, ez olyan állapotban hagyja az adatbázist, mintha a regét nem is hajtottuk volna végre.

10.7.5. Feladatok

*! **10.7.1. feladat:** Szoftverek „eltávolítását" (uninstalling), mint folyamatot felfoghatjuk az adott szoftver telepítésének (installing) a kiegyenlítő tranzakciójaként is. A telepítés és eltávolítás egy egyszerű modelljében feltehető, hogy egy művelet egy vagy több fájl *feltöltéséből* (loading) áll a forrásról (például CD-ROM) a számítógép merevle-

mezére. Egy *f* fájl feltöltésekor átmásoljuk *f*-et a CD-ROM-ról, felülírva az *f*-fel azonos nevű és útnevű fájlt, ha volt ilyen. Hogy meg tudjuk különböztetni az ilyen értelemben azonos fájlokat, feltehetjük, hogy minden fájl rendelkezik egy időbélyegzővel.

a) Mi a kiegyenlítő tranzakciója *f* fájl feltöltésének? Gondoljuk meg mindkét esetet: amikor létezett ugyanazon a helyen egy ugyanolyan nevű *f'* fájl, illetve amikor nem.
b) Magyarázzuk meg, hogy az a) feladat megoldásaként adott tranzakció miért lesz valóban kiegyenlítő tranzakció! *Segítség:* alaposan gondoljuk át azt az esetet, amikor *f*-fel felülírtuk *f'*-t, majd egy későbbi művelet egy másik, azonos útnevű fájllal felülírta *f*-et.

! **10.7.2. feladat:** Adjunk meg a repülőjegy-foglalás folyamatához egy regét! Vegyük számításba azt a lehetőséget, hogy az ügyfél csak érdeklődik egy jegy iránt, de nem foglalja le. A jegy lefoglalása után az ügyfél lemondhatja a foglalást, de az is lehetséges, hogy nem fizeti ki a jegyet időre, vagy kifizeti, de végül mégsem repül. Minden művelethez adjuk meg a megfelelő kiegyenlítő tranzakciót is!

10.8. Összefoglalás

- *Piszkos adat:* A központi memória puffereiben vagy a lemezen található adatot, amelyet egy, még folyamatban lévő tranzakció írt, „piszkos" adatnak nevezünk.
- *Továbbgyűrűző visszagörgetés:* Olyan naplózás és konkurenciavezérlés együttese esetén, amely megengedi, hogy egy tranzakció piszkos adatot olvasson, szükség lehet az olyan tranzakciók visszagörgetésére, amelyek egy később abortált tranzakció által írt (piszkos) adatokat olvastak.
- *Szigorú zárolás:* A szigorú zárolás elve azt követeli meg a tranzakcióktól, hogy a zárakat (az osztott zárak kivételével) ne csak a tranzakció lezárulásáig tartsák fenn, hanem még azt követően egészen addig, amíg a commit vagy abort naplóbejegyzés ki nem kerül a lemezre. A szigorú zárolás biztosítja, hogy egyetlen tranzakció sem olvas piszkos adatot, még visszamenőlegesen sem egy rendszerhiba és annak helyreállítása során sem.
- *Csoportos véglegesítés:* Gyengíthetünk a szigorú zárolásnak a naplóbejegyzésre vonatkozó feltételén, ha biztosítjuk, hogy a bejegyzések abban a sorrendben kerülnek ki a lemezre, amelyben eredetileg a pufferbe írtuk őket. Ekkor továbbra is garantált, hogy a tranzakciók nem olvasnak piszkos adatot, még egy esetleges hiba és helyreállítása során sem.
- *Az adatbázis állapotának helyreállítása abort után:* Ha egy tranzakció abortál, miután a pufferbe írt, az adatelemek régi értékét a napló vagy az adatbázis lemezen található példánya alapján állíthatjuk vissza. Ha az új értékek már elérték a lemezt, a napló még akkor is használható a régi értékek visszaállítására.
- *Logikai naplózás:* Olyan nagy adatbáziselemek esetén, mint a lemezblokkok, sok helyet takaríthatunk meg, ha a régi és az új értékeket a naplóba növekményesen jegyezzük be, azaz csak a változtatásokat tüntetjük fel. Néhány esetben a változtatások logikai feljegyzése – azaz a blokkok tartalmának absztrakt leírása – lehetővé teszi, hogy egy tranzakció abortálása után az adatbázis állapotát logikailag visszaállítsuk, még akkor is, ha szigorúan ugyanabba az állapotba való visszaállítás lehetetlen.
- *Nézet-sorbarendezhetőség:* Olyan ütemezésekben, ahol a tranzakciók talán olyan értékeket is írnak, amelyeket a későbbiekben egyetlen más tranzakció sem olvas, csak egy következő felülír, a konfliktus-sorbarendezhetőség túl erős feltételnek bizonyul. Egy gyengébb feltétel, amit nézet-sorbarendezhetőségnek nevezünk, csak azt követeli meg, hogy az ekvivalens soros ütemezésben minden tranzakció ugyanabból a forrásból olvassa az adatelemek értékeit, mint az eredeti ütemezésben.
- *Poligráf:* A nézet-sorbarendezhetőség ellenőrzése magában foglalja egy poligráf felépítését, amelyben az élek az értékek az írótól az olvasó irányába haladását jelölik, az élpárok pedig azt a követelményt tükrözik, amely szerint bizonyos írásműveletek nem eshetnek az adott írás- és olvasásművelet közé. Az ütemezés pontosan akkor nézet-sorbarendezhető, ha minden élpárnak elhagyhatjuk az egyik felét úgy, hogy eredményül egy körmentes gráfot kapjunk.
- *Holtpont:* Ez bármikor kialakulhat, amikor a tranzakcióknak olyan erőforrásokra, például zárakra, kell várakozniuk, amelyek az adott pillanatban egy másik tranzakció birtokában vannak. Megfelelő előkészületek nélkül fennáll a veszélye egy várakozási kör kialakulásának, amikor is az ebben részt vevő tranzakciók egyike sem képes a továbblépésre.
- *Várakozási gráf:* Rajzoljunk egy csúcsot minden várakozó tranzakcióhoz, és kössük össze azokkal a tranzakciókkal, amelyekre várakozik. A holtpont kialakulása pontosan azt jelenti, hogy létrejött egy vagy több kör a várakozási gráfban. A holtpontok kialakulása elkerülhető, ha a várakozási gráf nyilvántartásával minden olyan tranzakciót abortáltatunk, amely várakozásával kör jönne létre a gráfban.
- *Holtpontmegelőzés az erőforrások sorbarendezésével:* Ha a tranzakcióktól megkívánjuk, hogy az erőforrásokat valamilyen lexikografikus sorrendben használják fel, akkor ezzel megelőzhető a holtpontok kialakulása.
- *Időbélyegző alapú holtpontmegelőzés:* Más sémák időbélyegzőket vezetnek be, és ezek alapján döntik el, hogy az erőforrást igénylő tranzakciót abortáltassák-e vagy várakoztassák. A megvár-meghal sémában az erőforrást birtokló tranzakciónál idősebb tranzakció várakozik, az újabbakat pedig visszagörgetjük, de az időbélyegzőjük nem változik. A megsebez-megvár sémában az újabb tranzakció várakozik, az idősebb viszont kényszeríti az erőforrást birtokló tranzakciót, hogy mondjon le az erőforrásairól, és később kezdje el újra a működését.
- *Osztott adatok:* Egy osztott adatbázisban az adatok részekre bonthatók vízszintesen (a reláció sorai különböző munkaállomásokra vannak szétszórva), vagy függőlegesen (a reláció sémája több sémára van bontva [dekompozíció], és az ezekhez tartozó relációk különböző munkaállomásokon találhatók). Lehetőség van az adatok többszörözésére is, vagyis arra, hogy a relációnak egymással feltehetőleg azonos másodpéldányai legyenek különböző helyeken.
- *Osztott tranzakciók:* Az osztott adatbázisban egy logikai tranzakció több alkotórészből állhat, amelyek mindegyike különböző munkaállomáson fut. A konziszten-

cia megőrzésének érdekében az összes alkotórésznek egyet kell értenie abban, hogy a logikai tranzakció abortáljon vagy véglegessé váljon.

- *Kétfázisú véglegesítés:* Ez a módszer támogatja a tranzakció-alkotórészek döntéshozatalát a logikai tranzakció lezárását illetően, gyakran még rendszerhiba esetén is. Az első fázisban a koordinátor vezetésével minden alkotórésznek szavaznia kell arról, hogy az abortálás vagy a véglegesítés mellett van-e. A második fázisban a koordinátor pontosan akkor utasítja az alkotórészeket a véglegesítésre, ha mindegyikük azt akarta.
- *Osztott zárak:* Ha a tranzakcióknak olyan adatbáziselemeket kell zárolniuk, amelyek egyszerre több helyen is megtalálhatók, akkor megfelelő módszert kell alkalmaznunk a zárak összehangolására. A központi zárolási módszerben egy munkaállomás tartja nyilván az összes elem zárját. Az elsődleges példány módszerben az egyes elemek „szülőállomása" foglalkozik a zárak nyilvántartásával.
- *Többszörözött adat zárolása:* Amikor egy adatbáziselem többszörözve van, és több helyen is megtalálhatók a példányai, akkor az elem globális zárjához csak egy vagy több másodpéldány zárján keresztül lehet hozzájutni. A többségi zárolási módszerben a globális zár megszerzéséhez a másodpéldányok többségén kell osztott vagy kizárólagos zárat birtokolni. Az is megoldás lehet azonban, ha a globális osztott zár birtoklását már egy példány osztott zár alá helyezése után is lehetővé tesszük, globális kizárólagos zár esetén viszont megköveteljük az összes példány kizárólagos zárolását.
- *Regék:* Amikor a tranzakciók hosszabb időt igénybe vevő lépéseket is tartalmaznak, amelyek órákig vagy akár napokig is eltarthatnak, a hagyományos zárolási mechanizmusok túlságosan is korlátozhatják a konkurenciát. A rege műveletek hálózatából áll, amelyek mindegyike egy vagy több másik művelethez, az egész rege befejezéséhez vagy a rege abortálásához vezet.
- *Kiegyenlítő tranzakció:* Hogy egy regének értelme legyen, egy kiegyenlítő műveletnek kell tartoznia minden művelethez, amely érvényteleníti az első művelet hatását az adatbázis-állapoton, viszont érintetlenül hagy minden olyan más műveletet, amely egy másik, befejeződött vagy éppen működésben lévő regéhez tartozik. A rege abortáltatása esetén a kiegyenlítő műveletek egy megfelelő sorozatát hajtjuk végre.

10.9. Irodalomjegyzék

Az itt tárgyalt témákhoz néhány hasznos, általános forrásmű [2], [1] és [9]. A nézetsorbarendezhetőség és a poligráfteszt [10]-ből való. A holtpontmegelőzésről ad áttekintést [7]; a várakozási gráf is itt szerepel. A megvár-meghal és a megsebez-megvár sémák [11]-ből jönnek.

A kétfázisú véglegesítés protokollt [8]-ban ajánlották. Egy hatékonyabb (a könyvünkben nem tárgyalt) sémáról, amelyet háromfázisú véglegesítés protokollnak nevezünk, [12]-ben olvashatunk. A helyreállítást a vezetőválasztás szempontjából [4]-ben vizsgálják.

Az osztott zárolási módszereket [3]-ban (központi zárolás módszere), [13]-ban (elsődleges példány módszere) és [14]-ben (globális zárak a másodpéldányokon tartott zárakból) javasolták.

A hosszú tranzakciókat [6]-ban vezették be. A regéket [5] írja le.

1. N. S. Barghouti and G. E. Kaiser, „Concurrency control in advanced database applications," *Computing Surveys* **23**:3 (Sept., 1991), pp. 269–318.
2. S. Ceri and G. Pelagatti, *Distributed Databases: Principles and Systems*, McGraw-Hill, New York, 1984.
3. H. Garcia-Molina, „Performance comparison of update algorithms for distributed databases," TR Nos. 143 and 146, Computer Systems Laboratory, Stanford Univ., 1979.
4. H. Garcia-Molina, „Elections in a distributed computer system," *IEEE Trans. on Computers* **C-31**:1 (1982), pp. 48–59.
5. H. Garcia-Molina and K. Salem, „Sagas," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1987), pp. 249–259.
6. J. N. Gray, „The transaction concept: virtues and limitations," *Proc. Intl. Conf. on Very Large Databases* (1981), pp. 144–154.
7. R. C. Holt, „Some deadlock properties of computer systems," *Computing Surveys* **4**:3 (1972), pp. 179–196.
8. B. Lampson and H. Sturgis, „Crash recovery in a distributed data storage system," Technical report, Xerox Palo Alto Research Center, 1976.
9. M. T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems*, Prentice-Hall, Englewood Cliffs NJ, 1999.
10. C. H. Papadimitriou, „The serializability of concurrent updates," *J. ACM* **26**:4 (1979), pp. 631–653.
11. D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II, „System-level concurrency control for distributed database systems," *ACM Trans. on Database Systems* **3**:2 (1978), pp. 178–198.
12. D. Skeen, „Nonblocking commit protocols," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1981), pp. 133–142.
13. M. Stonebraker, „Retrospection on a database system," *ACM Trans. on Database Systems* **5**:2 (1980), pp. 225–240.
14. R. H. Thomas, „A majority consensus approach to concurrency control," *ACM Trans. on Database Systems* **4**:2 (1979), pp. 180–219.

11. fejezet

Információk egyesítése

Bár a modern adatbázisrendszerek sokféle irányba fejlődnek, az *információegyesítés* általános iránya az új alkalmazások népes családját tudhatja magáénak. Az ilyen alkalmazások két vagy több adatbázisban *(információforrásban)* tárolt adatokból hoznak létre egy nagy (virtuális) adatbázist, amelyben a forrásokban fellelhető összes információ megtalálható. Így a több helyen „szétszórtan" tárolt adatokat egységesen lehet lekérdezni. Az adatforrások lehetnek hagyományos vagy más típusú adatbázisok, például világhálós weboldalak gyűjteményei.

Ebben a fejezetben az információegyesítés fontos szempontjait vezetjük be. Először vázlatosan áttekintjük az információegyesítés legfontosabb módszereit: a szövetséget, a tárház létrehozását és a közvetítést. Ezután néhány alkalmazásban megtalálható integrált adatszervezési módszeren keresztül megismerkedünk egy különleges adatbáziselemmel, az úgynevezett „adatkockával". Az információegyesítéshez kapcsolódó speciális alkalmazásokat is áttekintjük. Így kerül sorra az „OLAP" (on-line analitikus feldolgozás)[1], és az „adatbányászat" témaköre.

11.1. Az információegyesítés módjai

Sokféle módszer létezik, amelynek segítségével adatbázisok vagy más (osztott) információforrások együttműködhetnek. Ebben a részben a három leggyakrabban alkalmazott eljárást mutatjuk be:

1. *Adatbázis-szövetség.* Az adatforrások egymástól függetlenek, de mindegyik kérhet a másiktól információt.
2. *Tárház létrehozása.* A különböző adatforrásokban megtalálható adatok másolatait egyetlen adatbázisban, az *adattárházban* tároljuk. Lehetséges, hogy az adatok még a tárházbavétel előtt valamilyen feldolgozó eljáráson mennek keresztül, például szűrhetjük vagy összesíthetjük (aggregáljuk) őket, vagy összekapcsolhatunk néhány relációt. Az adattárházat valamilyen időközönként, talán éjszakánként, frissítjük. Mivel az adatokat az adatforrásokból másoljuk, szükség lehet bizonyos átalakításokra, hogy az összes adat megfeleljen az adattárház sémájának.
3. *Közvetítés.* A közvetítő egy szoftverkomponens, amely egy olyan *virtuális adatbázist* támogat, amelyet a felhasználó úgy kérdezhet le, mintha az *valódi* lenne (azaz mintha fizikailag létezne, mint egy adattárház). Maga a közvetítő nem tárol semmiféle adatot – más módszert alkalmaz. A felhasználó lekérdezését lefordítja az adatforrások számára, egy vagy több lekérdezés képében, majd az adatforrások válaszait egységbe fogva adja meg a választ a felhasználónak.

[1] Az OLAP mozaikszó az angol *On-Line Analytic Processing* kifejezésnek felel meg. *A fordító megjegyzése.*

Mindegyik módszerrel részletesebben is megismerkedünk a fejezet során. Valamennyi megközelítésben kulcskérdés, hogy hogyan alakítjuk át az információforrásból nyert adatot. Az efféle információalakítók, a *borítékolók* (wrappers) vagy *adatkinyerők* (extractors) felépítését tárgyaljuk a 11.2. részben.[2] A következőkben bemutatunk néhány olyan problémát, amelyeket a borítékolóknak kell megoldaniuk.

11.1.1. Az egyesítés problémái

Akármilyen információegyesítő felépítést is választunk, kényes problémákba ütközünk, amikor a különböző adatforrások nyers adatainak próbálunk jelentést tulajdonítani. Az olyan adatforrások összességét, amelyek ugyanolyan, mégis sok apró részletben eltérő adatokkal dolgoznak, *heterogén* adatforrásnak nevezzük. Egy hosszabb példa segítségével bemutatjuk, miről van szó.

11.1. példa: A Süni Gépkocsigyártó Rt.-nek 1000 forgalmazója van, mindegyik adatbázist tart fenn a maga készletének a nyilvántartására. A cég létre akar hozni egy olyan egyesített adatbázist, amely az összes forgalmazójánál (azaz az 1000 adatforrásban) tárolt információt tartalmazza.[3]

Ha az egyik kereskedőnek éppen nincs raktáron valamilyen modellje, az egyesített adatbázis segítségével könnyen meg tudja állapítani, hogy melyik másiknál találhatja meg a hiányzó modellt. A cég elemzői pedig az előrelátható keresletre tehetnek megfigyeléseket, így a termelést a várhatóan népszerű modellekre lehet összpontosítani. Az 1000 forgalmazó azonban nem ugyanazt az adatbázissémát használja. Például az egyikük tárolhatja az autók adatait egyetlen relációban:

[2] Általában borítékolóról beszélünk közvetítő esetén, és adatkinyerőről tárház esetén, de nem mindig következetes a szóhasználat. *A fordító megjegyzése.*

[3] A legtöbb valódi gépkocsigyártó cég ma hasonló eszközökkel rendelkezik, bár ezeknek a fejlesztése eltérhet a példában leírtaktól. Lehet például, hogy a központi adatbázis jön létre először, majd később a forgalmazók, akik ebből letölthetik a saját adatbázisukba a megfelelő részeket. Az itt említett eset mégis arra szolgál például, amivel manapság sokféle szférában próbálkoznak.

```
Kocsik(sorszám, modell, szín, autoSeb, cdJátszó,...)
```

ahol minden lehetséges extra felszereléshez (automata sebességváltó, CD-lejátszó...) egy logikai érték van rendelve. Egy másik forgalmazó viszont nyilvántarthatja az extra felszereléseket egy külön relációban, például így:

```
Autók(sorsz, modell, szín)
Extrák(sorsz, extra)
```

Vegyük észre, hogy nem csak a két séma eltérő, hanem a nyilvánvalóan „azonos" nevek is megváltoztak: `Kocsikból` `Autók` lett, `sorszámból` `sorsz`.

Rosszabb esetben az azonos jelentésű adatok a különböző adatbázisokban másképpen vannak ábrázolva.

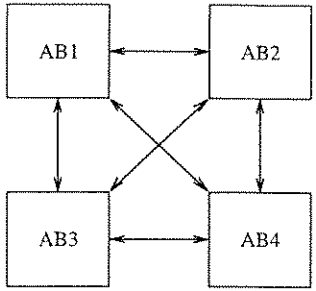
1. *Adattípus-különbségek.* A sorszámok lehetnek változó hosszúságú karakterláncok az egyik helyen vagy rögzített hosszúak a másikon. Maga a rögzített hossz is lehet más és más az egyes adatbázisokban, sőt néhány adatforrás használhat egész számokat a karakterlánc helyett.
2. *Értékkülönbségek.* Ugyanazt a fogalmat ábrázolhatják különböző konstansokkal a különböző adatforrásokban. A fekete színt például ábrázolhatják egy egész számmal, mint kóddal az egyik helyen, a `FEKETE` karakterlánccal egy másikon, és a `FE` kóddal egy harmadikon. Még rosszabb a helyzet, ha a `FE` a „fehéret" jelenti egy negyedik helyen.
3. *Szemantikus különbségek.* Sok fontos kifejezést másképpen értelmezhetnek az egyes adatforrások. Az egyik forgalmazó nyilvántarthat tehergépkocsikat is a `Kocsik` relációban, míg a másik csak személygépkocsikat tárol ugyanebben a relációban. Az egyik forgalmazó talán különbséget tesz furgon és kisteherautó között, a másik pedig nem.
4. *Hiányzó értékek.* Lehet, hogy az egyik adatforrás egyáltalán nem tárol egy olyan típusú információt, amit az összes többi (vagy a legtöbb) forrás nyilvántart. Például az egyik forgalmazó a színeket egyáltalán nem tartja nyilván. Hogy ezeket a helyzeteket kezelni tudjuk, néha használhatunk `NULL` értéket vagy valamilyen alapértelmezett értéket. A fejlődés iránya azonban a „felstrukturált" adatmodell használata felé mutat, amellyel olyan egyesített adatokat ábrázolhatunk, amelyek nem egészen egyeztethetők össze. Ezzel a témával kapcsolatban található néhány hasznos olvasmány a fejezet irodalomjegyzékében.

Az adatforrások közti minden efféle összeférhetetlenséget meg kell szüntetni valamilyen átalakítás, „fordítás" segítségével, még mielőtt az egyesített adatbázis létrejönne. □

11.1.2. Adatbázis-szövetség

Több adatbázis összefogásának talán az a legegyszerűbb módja, ha 1-1 kapcsolatot építünk azon adatbázispárok között, amelyek kommunikálni akarnak egymással. Ezeken a kapcsolatokon keresztül D_1 adatbázisrendszer lekérdezheti D_2-t, D_2 által érthető formában. A probléma ezzel a felépítéssel abból adódik, hogy ha az n adatbázis mindegyike szeretne kommunikálni a másik $(n-1)$-gyel, akkor $n(n-1)$ „rendszerközi" lekérdezéseket támogató kódot is kell írnunk. A 11.1. ábrán látható szövetségben a négy adatbázis mindegyikének három fordító komponensre van szüksége, hogy a másik hármat el tudja érni.

Mindezek ellenére bizonyos körülmények között lehet, hogy az adatbázis-szövetséget a legkönnyebb létrehozni, különösen, ha az adatbázisok egymás közti kommunikációja természeténél fogva korlátozott. Egy példán keresztül bemutatjuk, hogyan működhet a fordító komponens.



11.1. ábra. *A négy adatbázisból álló szövetségnek összesen 12 fordító komponensre van szüksége*

11.2. példa: Tegyük fel, hogy a Süni típusú gépkocsi forgalmazói meg akarják osztani egymás közt a leltárukat, de minden forgalmazónak csak arra van szüksége, hogy egy megfelelő lekérdezés segítségével megállapíthassa, hogy a nála megrendelt kocsik közül melyek találhatók meg valamelyik tőle nem messze működő forgalmazónál. Kicsit formálisabban, tekintsük Forgalmazó 1-et, aki a következő relációval rendelkezik:

```
IgényeltKocsik(modell, szín, autoSeb)
```

A reláció minden sora egy vásárló által igényelt kocsi leírása: milyen modell, milyen színű, és akarnak-e bele automata sebességváltót vagy nem. Forgalmazó 2 a leltárához a 11.1. példában bevezetett kétrelációs sémát használja:

```
Autók(sorsz, modell, szín)
Extrák(sorsz, extra)
```

Forgalmazó 1 ír egy programot, amely egy távoli lekérdezés segítségével olyan kocsikat keres Forgalmazó 2 adatbázisában, amelyek leírása megfelel valamelyik olyan

```
for(minden IgényeltKocsik-hoz tartozó (:m, :c, :a) sorra) {
   if(:a= TRUE) {/*automata sebességváltót akarnak*/
      SELECT sorsz
      FROM Autók, Extrák
      WHERE Autók.sorsz = Extrák.sorsz AND
            Extrák.extra = 'autoSeb' AND
            Autók.modell = :m AND
            Autók.szín = :c;
   }
   else { / *nem akarnak automata sebességváltót */
      SELECT sorsz
      FROM Autók
      WHERE Autók.modell = :m AND
         Autók.szín = :c AND
         NOT EXISTS (
            SELECT *
            FROM Extrák
            WHERE sorsz = Autók.sorsz AND
                  extra = 'autoSeb'
         );
   }
}
```

11.2. ábra. *Forgalmazó 1 lekérdezi Forgalmazó 2 adatbázisát*

autónak, amely Forgalmazó 1 `IgényeltKocsik` relációjában szerepel. A program vázlata a 11.2. ábrán látható.

A beágyazott SQL rész jelenti Forgalmazó 2 adatbázisának távoli lekérdezését. Ennek az eredményét kapja vissza Forgalmazó 1. A szabvány SQL-ben használatos konvenciónak megfelelően kettőspontot tettünk az olyan változók elé, amelyek az adatbázisból visszakapott konstans értékeket jelölik.

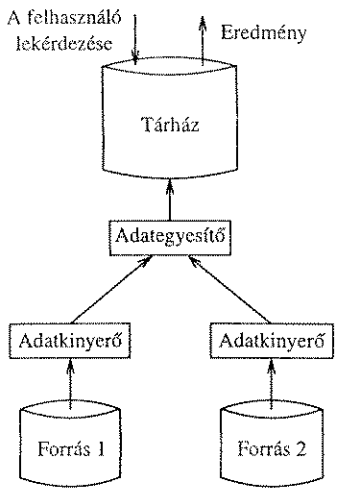
Ez a lekérdezés Forgalmazó 2 sémáját használja. Ha Forgalmazó 1 ugyanezt a lekérdezést Forgalmazó 3 adatbázisán is végre akarná hajtani, aki a 11.1. példában bemutatott

```
Kocsik(sorszám, modell, szín, autoSeb,...)
```

egyrelációs sémát használja, akkor a lekérdezés egész másképpen nézne ki. De mindegyik lekérdezés helyesen működik azon az adatbázison, amelyikre írták. □

11.1.3. Adattárházak

Az *adattárházban* mint információegyesítő architektúrában, a különböző információforrásokból nyert adatokat egy *globális* sémába illesztjük. Ezután az adatot a tárházban tároljuk, amely a felhasználó számára úgy néz ki, mint egy közönséges adatbázis.



11.3. ábra. *Az adattárház az egyesített információt egy külön adatbázisban tárolja*

A 11.3. ábrán látható egy adattárház felépítése, bár az ott szereplő két forrásnál sokkal többet is felvehettünk volna a rendszerbe.

Ha az adat egyszer már bekerült a tárházba, akkor a felhasználó ugyanolyan lekérdezéseket hajthat rajta végre, mint bármilyen más adatbázison. Azt viszont általában megtiltják, hogy a felhasználó módosítsa az itt tárolt adatokat, mivel ezek a változtatások nem jelennének meg a tárház adatforrásaiban, és így az adattárház inkonzisztenssé válna a forrásaival szemben.

A tárházban tárolt adat létrehozásának legalább három megközelítési módja van:

1. A tárházat a források aktuális adatai alapján újra és újra létrehozzuk. Ez az eljárás a leggyakoribb, általában az éjszakánként vagy akár hosszabb időközönként végrehajtott rekonstrukcióval (az éjszakai időpont azért nagyon kedvező, mert ilyenkor ki lehet kapcsolni a rendszert, így a tárház építése idején nincsenek lekérdezések). A módszer fő hátránya, hogy a rendszert le kell állítani, és hogy néha a tárház létrehozása hosszabb ideig tart, mint egy normális „éjszaka". Néhány alkalmazásban további hátrányt jelent, hogy a tárházban tárolt adat sokat veszthet az aktualitásából.
2. A tárházat időszakosan frissítjük (például minden éjjel) a forrásokon a tárház utolsó frissítése óta végrehajtott változtatások alapján. Ez az eljárás kisebb adattömeggel dolgozik, mint az előző módszer, és ez nagyon fontos szempont, amikor egy nagy tárházat (sok gigabájt, illetve terabájt nagyságúak vannak manapság használatban) rövid idő alatt kell módosítani. Hátrányt jelent azonban, hogy a tárházon végzendő változtatásokat számító eljárás, a *növekményes frissítés* (incremental update), nagyon összetett azokhoz az algoritmusokhoz képest, amelyek a semmiből építik újjá az adattárházat.

```
INSERT INTO AutóTház(sorszám, modell, szín, autoSeb, forgalmazó)
   SELECT sorszám, modell, szín, 'igen', 'forgalmazó2'
   FROM Autók, Extrák
   WHERE Autók.sorsz = Extrák.sorsz AND
         Extrák.extra = 'autoSeb';

INSERT INTO AutóTház(sorszám, modell, szín, autoSeb, forgalmazó)
   SELECT sorszám, modell, szín, 'nem', 'forgalmazó2'
   FROM Autók
   WHERE NOT EXISTS (
      SELECT *
      FROM Extrák
      WHERE sorsz = Autók.sorsz AND
            extra = 'autoSeb'
   );
```

11.4. ábra. *Az adatkinyerő Forgalmazó 2 adatait áthelyezi a tárházba*

3. A tárházat azonnal változtatjuk, mihelyst valami módosítás történik valamelyik adatforráson. Ez a megközelítés túl sok kommunikációt és feldolgozást igényel, hogy a gyakorlatban is alkalmazható legyen. Lassan változó forrásokból épített kis tárházak esetén azonban alkalmas lehet a használata. Mindezek ellenére ez a téma jó kutatási terület, és egy ilyen típusú tárház sikeres megvalósításának számos fontos alkalmazása lehetne, például az automatizált értékpapír-kereskedés területén.

11.3. példa: Az egyszerűség kedvéért tegyük fel, hogy a Süni rendszerben csak két forgalmazó szerepel, amelyek rendre a következő sémákat használják:

```
Kocsik(sorszám, modell, szín, autoSeb, cdJátszó,...),
```

illetve

```
Autók(sorsz, modell, szín)
Extrák(sorsz, extra)
```

Létre akarunk hozni egy adattárházat a következő séma szerint:

```
AutóTház(sorszám, modell, szín, autoSeb, forgalmazó)
```

A globális séma tehát hasonló ahhoz, amelyet az első forgalmazó használ, de a tárházban az extra felszerelések közül csak az automata sebességváltót tartjuk nyilván, és felveszünk egy új attribútumot is, amiből megtudhatjuk, hogy melyik forgalmazónál található meg az adott gépkocsi.

A szoftver, amely a második forgalmazó adatbázisából nyert adatokból a globális sémának megfelelően feltölti az adattárházat, SQL-lekérdezések segítségével is megírható. Az első forgalmazóhoz intézett lekérdezés egyszerű:

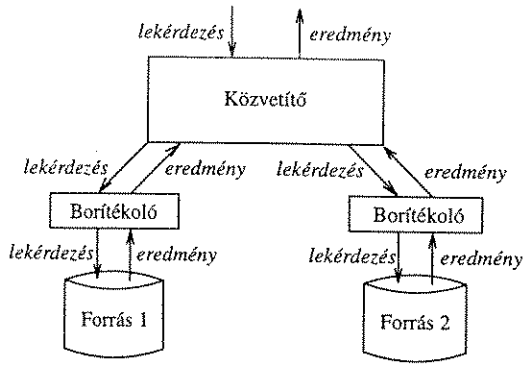

```
INSERT INTO AutóTház(sorszám, modell, szín, autoSeb, forgalmazó)
   SELECT sorszám, modell, szín, autoSeb, 'forgalmazó1'
   FROM Kocsik;
```

A második forgalmazóhoz írt adatkinyerő bonyolultabb, mert el kell döntenünk, hogy az adott autóban van-e automata sebességváltó vagy nincs. A nyilvánvaló jelentéssel bíró `'igen'`, illetve `'nem'` karakterláncokat használjuk az `autoSeb` attribútum értékeként. Az adatkinyerő SQL-kódja a 11.4. ábrán látható.

Ebben az egyszerű példában nincs adategyesítő. Mivel a tárház a forrásokból kiszűrt relációk uniója, az adatokat közvetlenül, előzetes feldolgozás nélkül töltöttük a tárházba. Sok tárház azonban a forrásokból nyert relációkon különböző műveleteket hajt végre. Például két relációt összekapcsolnak, és az eredményt teszik a tárházba, vagy több reláció unióján valamilyen összesítést hajtanak végre. Még általánosabban, minden forrásból több relációt is kiszűrhetnek, és különböző relációkat sokféle módon kombinálhatnak. □

11.1.4. Adatközvetítő

Az adatközvetítő egy vagy több virtuális nézettáblát támogat, amelyek az adattárházban testet öltött relációkhoz nagyon hasonló módon egyesítik az egyes forrásokat. Mivel azonban maga a közvetítő nem tárol adatokat, az adattárháztól elég eltérő módon működik. A 11.5. ábrán látható közvetítő két forrást koordinál. Ahogy azt az adattárház felépítésénél is említettük, kettőnél több forrás használata a tipikus. Első lépésként a felhasználó megad egy lekérdezést a közvetítőnek. Mivel maga a közvetítő nem tárol semmilyen adatot, az információforrásoktól kell megszereznie a szükséges adatokat, és ezek alapján kell választ adnia a felhasználónak.



11.5. ábra. *A közvetítő és a borítékolók átalakítják a lekérdezéseket a források számára, és összegyűjtik az eredményt*

A 11.5. ábrán látható, hogy a közvetítő mindegyik borítékolójának küld egy lekérdezést, és ezek a következő lépésben a saját adatforrásukhoz intéznek egy-egy lekérdezést. Valójában a közvetítő több lekérdezést is küldhet a borítékolónak, és lehet, hogy néhány borítékolónak egyáltalán nem küld lekérdezést. A visszakapott adatokat a közvetítő egyesíti. Nem tüntettünk fel egy kifejezett adategyesítő alkotóelemet, mint ahogy ezt a 11.3. ábrán az adattárház esetén tettük, mert az adatközvetítő esetén a forrásokból visszakapott eredmények egyesítése a közvetítő egyik feladata.

11.4. példa: Tekintsük a 11.3. példában tárgyalt helyzetet, de most adattárház helyett használjunk adatközvetítőt. A közvetítő tehát ugyanazt a két gépkocsi-adatforrást egyesíti, mint az előző példában, és egy nézettáblát hoz létre az alábbi sémával:

```
AutóKözv(sorszám, modell, szín, autoSeb, forgalmazó)
```

Tegyük fel, hogy a felhasználó a piros autók iránt érdeklődik, és a következő lekérdezést teszi fel:

```
SELECT sorszám, modell
FROM AutóKözv
WHERE szín = 'piros';
```

A közvetítő a felhasználó kérdésére reagálva továbbíthatja a lekérdezést mindkét borítékolójának. Hogy hogyan lehet ilyen és ehhez hasonló lekérdezéseket kezelő borítékolókat tervezni és megvalósítani, az a 11.2. rész témája. Bonyolultabb esetekben szükség lehet a lekérdezés alkotórészeinek az átalakítására és szétosztására is, de ebben az esetben az átalakítás műveletét egyedül a borítékolóra is lehet hagyni.

A Forgalmazó 1-hez tartozó borítékoló lefordítja a lekérdezést a forgalmazó sémájának megfelelően, amely emlékeztetőül a következő:

```
Kocsik(sorszám, modell, szín, autoSeb, cdJátszó,...)
```

Egy megfelelő fordítás:

```
SELECT sorszám, modell
FROM Kocsik
WHERE szín = 'piros';
```

A `sorszám-modell` párokból álló válaszhalmazt visszaküldi a borítékoló a közvetítőnek.

Eközben a Forgalmazó 2-höz tartozó borítékoló ugyanazt a lekérdezést a második forgalmazó sémájának megfelelően alakítja át. A séma:

```
Autók(sorsz, modell, szín)
Extrák(sorsz, extra)
```


és az ennek megfelelő lekérdezés:

```
SELECT sorsz, modell
FROM Autók
WHERE szín = 'piros';
```

A borítékoló válaszként visszaküldi a `sorsz-modell` párokból álló halmazt a közvetítőnek, esetleg előtte végrehajtva azt a nem túl megerőltető feladatot, hogy az eredménytáblázat sémájában a `sorsz` attribútumot `sorszám`-ra cserélje.

A közvetítő ezek után veheti a két reláció unióját, és ezzel megadhatja a választ a felhasználó kérdésére. Mivel a kocsisorszámtól elvárjuk, hogy „globális kulcsként" működjön, vagyis hogy ne legyen két autó, még különböző adatbázisokban sem, amely ugyanazzal a sorszámmal rendelkezik, vehetjük a két reláció zsák-unióját, feltételezve, hogy úgysem lesznek az eredményrelációban azonos sorok. □

A közvetítőnek számos olyan választási lehetősége van egy lekérdezés megválaszolása közben, amelyre a 11.4. példában nem került sor. Például megteheti, hogy először csak egy adatforrást kérdez le, majd ennek az eredménye alapján dönt a következő lekérdezés(ek) felől. Ez a módszer megfelelő lenne például, ha a felhasználót az érdekelte volna, hogy van-e Süni „Tüskés" sportkupé kékben. A közvetítő először csak Forgalmazó 1-et kérdezné le, és ha ennek az eredménye az üres halmaz lenne, csak akkor fordulna Forgalmazó 2-höz a következő lekérdezéssel.

11.1.5. Feladatok

11.1.1. feladat: Ellenőrizzük néhány on-line könyvkereskedés weboldalán, hogy milyen információk találhatók erről a könyvről. Hogyan hoznánk létre ezek alapján egy olyan globális sémát, amely megfelel egy adattárház vagy egy közvetítő számára?

! **11.1.2. feladat:** Az *A* számítógépekkel foglalkozó cég a következő sémának megfelelően tartja nyilván az árusított modelleket:

```
Szgépek(szám, proc, seb, memória, hd)
Monitorok(szám, képernyő, maxX, maxY)
```

A (123, PIII, 500, 128, 18,7) sor az `Szgépek` relációban például azt jelenti, hogy a 123-as számú modellhez egy 500 MH-es Pentium–III processzor, 128 Mb memória és 18,7 Gb merevlemez tartozik. A (456, 19, 1600, 1200) sor a `Monitorok` relációban azt jelenti, hogy a 456-os számú modell képernyője 19 inches és a maximális felbontása 1600 × 1200.

A *B* számítógépekkel foglalkozó cég csak teljes konfigurációkkal foglalkozik, számítógépet és monitort csak együtt ad el. Ez a következő sémát használja:

```
Konfig(azon, processzor, mem, lemez, képMéret)
```

A `processzor` attribútum értéke egész szám és a processzor sebességét adja meg. A processzor típusa (például Pentium III) és a monitor maximális felbontása nincs nyilvántartva. Az `azon`, `mem` és `lemez` attribútumok az *A* cég `szám`, `memória` és `hd` attribútumaihoz hasonló jelentéssel bírnak azzal a különbséggel, hogy a merevlemez mérete itt gigabájt helyett megabájtban van megadva.

a) Milyen SQL-utasítást kellene kiadnia az *A* cégnek, ha a *B* cég által árusított cikkekről szeretne információt tárolni a saját adatbázisában?

* b) Milyen SQL-utasítás lenne *B* számára a legmegfelelőbb, ha az *A* számítógépeiből és monitoraiból összeállítható konfigurációkról a lehető legtöbb információt akarja felvenni a saját `Konfig` relációjába?

*! **11.1.3. feladat:** Milyen globális séma segítségével tudnánk a lehető legtöbb információt nyilvántartani a 11.1.2. feladat *A* és *B* cégei által kínált termékekről?

11.1.4. feladat: Adjuk meg azt a lekérdezéssorozatot, amely az *A* és *B* cégek adataiból gyűjtött információt egy olyan adattárházba teszi, amely a 11.1.3. feladat megoldásaként adott globális sémát használja! Szükség esetén használható a szerzők által megadott séma is.

11.1.5. feladat: Tegyük fel, hogy egy közvetítő a 11.1.3. feladat sémáját (akár a saját megoldást, akár a szerzők megoldását) használja. Hogyan lehetne megadni az 500 MHz-es számítógépekkel együtt elérhető maximális merevlemez méretét?

! **11.1.6. feladat:** Adjunk meg két másik sémát, amelyek segítségével a számítógépes cégek a 11.1.2. feladatban leírtakhoz hasonló adatokat tarthatnak nyilván!

! **11.1.7. feladat:** A 11.3. példában szó volt a Forgalmazó 1 `Kocsik` relációjáról, amely tartalmazott egy kényelmesen használható `autoSeb` attribútumot, amelynek csak „igen"/„nem" értékei lehettek. Mivel a globális sémában ehhez az attribútumhoz ugyanezeket az értékeket használtuk, az `AutóTház` relációt nagyon könnyű volt létrehozni. Most tegyük fel, hogy a `Kocsik.autoSeb` attribútum egész értékeket vehet fel; 0 jelenti azt, hogy nincs automata sebességváltó beszerelve, $i > 0$ pedig azt, hogy i sebességes automata sebességváltó tartozik a kocsihoz. Adjuk meg azt az SQL-lekérdezést, amely segítségével a `Kocsik` reláció átültethető az `AutóTház` relációba!

11.1.8. feladat: Hogyan fordítaná a 11.4. példa közvetítője a következő lekérdezéseket:

* a) Adjuk meg az automata sebességváltós gépkocsik sorszámát!

b) Adjuk meg azon gépkocsik sorszámát, amelyek nem rendelkeznek automata sebességváltóval!

! c) Adjuk meg a Forgalmazó 1-nél elérhető kék gépkocsik sorszámát!

11.2. Borítékolók a közvetítő alapú rendszerekben

A 11.3. ábrához hasonló adattárház adatkinyerői a következő alkotóelemekből állnak:

1. Egy vagy több beépített lekérdezés, amelyek végrehajtásával az adatforrásból adatok állíthatók elő az adattárház számára.
2. Megfelelő kommunikációs mechanizmus, amely segítségével az adatkinyerő képes az alábbi feladatokra:

   a) ad hoc lekérdezéseket továbbítani a forrásnak,
   b) válaszokat fogadni a forrástól és
   c) információt átadni az adattárháznak.

A forrás számára beépített lekérdezések lehetnek SQL-lekérdezések, ha az adatforrás egy olyan SQL-adatbázis, mint amilyenekkel a 11.1. rész példáiban találkoztunk. Ha a forrás azonban nem hagyományos adatbázis, az adatkinyerőbe beépített lekérdezések akármilyen más, a forrás által értelmezhető műveletek is lehetnek. Például a borítékoló kitölthet egy weboldalon található űrlapot, lekérdezhet egy on-line bibliográfiát a rendszer saját, speciális nyelvén, vagy számtalan más eszközt is igénybe vehet, hogy a forrással meg tudja értetni a lekérdezést.

Az adatközvetítőnek azonban sokkal összetettebb borítékolókra van szüksége, mint a legtöbb adattárháznak. A közvetítő lekérdezések egész változatát intézheti a borítékolóhoz, és a borítékolónak képesnek kell lennie ezeket elfogadni és akármelyiket lefordítani a megfelelő adatforrás nyelvére. Ezek után természetesen a lekérdezés eredményét továbbítania kell a közvetítő felé, ahogy ezt a borítékoló a tárház alapú rendszerben is megteszi. Ebben a részben a közvetítők számára használható rugalmas borítékolók konstrukciójával foglalkozunk.

11.2.1. Sablonok lekérdezési formákhoz

Az adatforrás és közvetítő kapcsolatát alkotó borítékoló tervezésének létezik egy szisztematikus módja: a közvetítő által feltett lehetséges lekérdezéseket egy-egy *sablonnak* megfelelő osztályba soroljuk. A sablonok olyan paraméteres lekérdezések, ahol a paraméterek csak konstans értékeket vehetnek fel. A borítékoló a közvetítő által kínált konstansokkal végzi el a lekérdezést. A következő példán keresztül bemutatjuk a módszer lényegét. $T \Rightarrow S$ azt jelöli, hogy a borítékoló a T sablont a forrás által feldolgozható S lekérdezéssé, úgynevezett forráslekérdezéssé alakítja át.

11.5. példa: Olyan borítékolót akarunk létrehozni, amely a közvetítő és Forgalmazó 1 között teremti meg a kapcsolatot. Forgalmazó 1, illetve a közvetítő rendre a következő sémát használja:

```
Kocsik(sorszám, modell, szín, autoSeb, cdJátszó,…)
AutóKözv(sorszám, modell, szín, autoSeb, forgalmazó)
```

Gondoljuk meg, hogyan tudna a borítékoló adott színű kocsikat megadni a közvetítőnek. Ha a szín számára bevezetjük a `$c` paramétert, akkor az adott színtől függetlenül mindig használható lesz a 11.6. ábrán látható sablon. Hasonlóan, a borítékoló rendelkezhet egy másik sablonnal is, a modellnek megfelelő `$m` paraméterrel, sőt egy harmadikkal az automata sebességváltó paraméterezésével és így tovább. Ha a lekérdezés e három attribútum közül bármelyiket rögzítheti, akkor összesen nyolc sablonra van szükségünk.[4] Általában, ha n attribútum rögzítésére van lehetőség, a szükséges sablonok száma 2^n.[5] Másfajta lekérdezésekhez, mint például „bizonyos típusú autók száma" vagy „van-e raktáron egy bizonyos típusú autó?", más sablonok kellenek. A sablonok száma túlságosan is nagyra nőhet, de a borítékoló finomításával egyszerűsíthető a helyzet. Erről a 11.2.3. részben lesz szó. □

```
SELECT *
FROM AutóKözv
WHERE szín = '$c';
   =>
SELECT sorszám, modell, szín, autoSeb, 'forgalmazól'
FROM Kocsik
WHERE szín = '$c';
```

11.6. ábra. *Borítékoló sablon adott színű kocsik lekérdezésére*

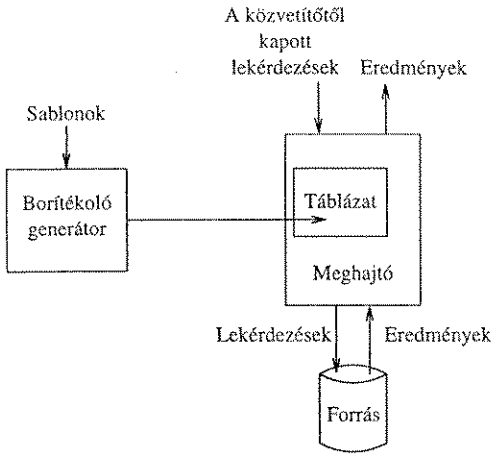
11.2.2. Borítékoló generátor

A borítékolót meghatározó sablonok alapján olyan programot kell írni, amely létrehozza magát a borítékolót. Ez a program a *borítékoló generátor*. Szellemében nagyon hasonló az elemző generátorokhoz (például YACC[6]), amelyek magas szintű specifikációk alapján hoznak létre fordítóprogram-komponenseket. A 11.7. ábrán bemutatott eljárás azzal kezdődik, hogy a specifikációt, vagyis a sablonkészletet, átadjuk a borítékoló generátornak.

[4] A három alapsablonon kívül szükségünk lesz még azokra is, amelyekben egyszerre adható meg a szín és modell, szín és automata sebességváltó és így tovább. A nyolcba beleszámoljuk az „üres" sablont is, amelyben a három paraméter közül egy sem szerepel, vagyis amely az egész egyesített adatbázist adja eredményül. *A fordító megjegyzése.*

[5] Ha az adatforrás ehhez a példához hasonlóan egy olyan adatbázis, amely lekérdezhető SQL-ben, jogosan elvárhatnánk, hogy egy sablon akárhány konstansértékű attribútumot képes legyen kezelni, egyszerűen úgy, hogy a `WHERE` záradékot paraméterré teszi. Ez a módszer működik az SQL-források és az olyan lekérdezések esetén, amelyekben az attribútumok értéke egy konstanshoz van kötve, de nem szükségszerű, hogy bármilyen adatforrás, például egy weboldal esetén is megfelel, ahol csak bizonyos formák használhatók interfészként. Általában nem tehetjük fel, hogy ahogy mi fordítunk le egy lekérdezést, az bármiben is hasonlít ahhoz, ahogy egy hasonló lekérdezés van lefordítva.

[6] *Yet Another Compiler-Compiler,* magyarul Ismét Egy Újabb Fordító Fordító. Értelmező- és fordítóprogramok fejlesztéséhez használt eszköz. *A fordító megjegyzése.*



11.7. ábra. *A borítékoló generátor létrehozza a táblázatokat a meghajtó számára; a meghajtó és a táblázatok alkotják a borítékolót*

A borítékoló generátor ez alapján táblázatot készít, amely a sablon és az annak megfelelő forráslekérdezési formapárokat tartalmazza. Minden borítékolóban használunk egy *meghajtót* (driver), amely általában lehet ugyanaz mindegyik generált borítékolóhoz. A meghajtó feladata a következő:

1. A közvetítőtől érkezett lekérdezések fogadása. Mivel a kommunikációs mechanizmus közvetítőfüggő lehet, a kommunikációs rutint mint „plug-in"-t adjuk a meghajtónak, így ugyanaz a meghajtó más plug-in-nel másféle kommunikációt használó rendszerekben is használható.
2. A lekérdezésnek megfelelő sablon keresése a táblázatban. Ha a keresés sikeres, akkor a lekérdezésből vett paraméterértékekkel kell végrehajtani a forráslekérdezést. Ha a keresés sikertelen, a borítékoló negatív választ küld a közvetítőnek.
3. A forráslekérdezés továbbítása az adatforrás felé, a „plug-in" kommunikációs rutin használatával. A lekérdezésre adott választ a borítékoló fogadja.
4. A válasz továbbítása a közvetítő felé, amelyen szükség esetén a borítékoló valamilyen feldolgozó eljárást hajt végre. Hogy ezzel a borítékolók hogyan támogathatják a lekérdezések egy nagyobb osztályát, azt a következő fejezetekben tárgyaljuk.

11.2.3. Szűrők

Tegyük fel, hogy az egyik forgalmazó adatbázisához tartozó borítékoló rendelkezik a 11.6. ábrán látható sablonnal, azonban az adatközvetítőtől modell *és* szín szerint kérnek tájékoztatást. A borítékoló talán fel van szerelve egy kicsit összetettebb sablonnal

is, amely segítségével kezelhetők a modellt és színt is rögzítő lekérdezések (11.8. ábra). De ahogy a 11.5. példa végén említettük, a gyakorlatban nem mindig valósítható meg, hogy az összes lehetséges lekérdezéshez megírjuk a megfelelő borítékoló sablont.

```
SELECT *
FROM AutóKözv
WHERE modell = '$m' AND szín = '$c';
   =>
SELECT sorszám, modell, szín, autoSeb, 'forgalmazól'
FROM Kocsik
WHERE modell = '$m' AND szín = '$c';
```

11.8. ábra. *Borítékoló sablon, amely az adott modellhez tartozó, adott színű kocsikat adja meg*

Létezik azonban egy másik megoldás is, amellyel növelhető a támogatott lekérdezések száma. Ennek az a lényege, hogy a borítékoló a forráslekérdezések eredményén egy *szűrést* (filter) hajt végre. Ha a borítékoló rendelkezik egy olyan sablonnal, amely (a paraméterek gondos behelyettesítése után) az eredeti lekérdezés eredményének egy bővebb részhalmazát adja, akkor lehetőség van arra, hogy a sablonnak megfelelő sorokból „kiszűrje" azokat, amelyek a kívánt lekérdezés eredményét adják, és csak ezeket küldje tovább a közvetítőnek. Annak eldöntése, hogy a közvetítő által igényelt lekérdezés végeredménye részhalmaza-e a borítékoló egyik sablonjának megfelelő lekérdezés végeredményének, általában nehéz probléma, bár az eddig látott példákhoz hasonló egyszerű esetekre az elmélet jól ki van dolgozva. A kérdés további tanulmányozásához jó kiindulópontok találhatók az irodalomjegyzékben.

11.6. példa: Tegyük fel, hogy egyetlen sablonnal rendelkezünk, azzal, amellyel az adott színű kocsikat kérdezhetjük le (11.6. ábra). A közvetítőnek azonban a kék „Tüskés" autókra van szüksége, azaz a következő lekérdezés végeredményét várja:

```
SELECT *
FROM AutóKözv
WHERE szín = 'kék' and modell = 'Tüskés';
```

A válaszadás egy lehetséges módja:

1. A 11.6. ábra sablonját `$c = 'kék'` helyettesítéssel használva megkeressük az összes kék autót.
2. Az eredményt az alábbi ideiglenes relációban tároljuk:

   ```
   TmpAutók(sorszám, modell, szín, autoSeb, forgalmazó)
   ```

3. Ebből a következő lekérdezés segítségével kiválasztjuk a Tüskéseket, az eredményt pedig visszaküldjük.

Hol végezzük a szűrést?

A példákban eddig mindig feltettük, hogy a szűrés műveletét a borítékoló végzi el. Az is lehetséges azonban, hogy a borítékoló továbbadja a közvetítőnek a még feldolgozatlan információt, és a szűrésre ott kerül sor. Ha viszont a sablon által visszaadott adatok többsége nem felel meg a közvetítő lekérdezésének, akkor a legjobb megoldás még a borítékoló szintjén elvégezni a szűrést, így kerülve el a szükségtelen sorok továbbításából eredő többletköltséget.

```
SELECT *
FROM TempAutók
WHERE modell = 'Tüskés';
```

A végeredmény a kívánt személygépkocsik halmaza. A gyakorlatban a `TempAutók` relációnak egyszerre csak egy sora jönne létre, amit rögtön meg is szűrnénk, csővezetékes jellegű feldolgozással. Inkább ezt a módszert használnánk, minthogy az egész relációt tároljuk és szűrjük a borítékolóban. □

11.2.4. A borítékoló más műveletei

A szűrésen kívül másféle átalakításokat is elvégezhet a borítékoló, amennyiben biztosak vagyunk benne, hogy az ehhez szükséges adatokat hiány nélkül visszakapja a sablon forráslekérdezés részétől. A közvetítőnek való továbbítás előtt elvégezhet például egy vetítést, de az is elképzelhető, hogy összekapcsolások vagy valamilyen összesítések után küldi csak tovább az eredményt.

11.7. példa: Tegyük fel, hogy a közvetítő az egyes forgalmazóknál fellelhető kék Tüskésekről akar információt szerezni, de csak a sorszámra, forgalmazóra és a sebességváltó típusára (automata-e) van szüksége, mivel a `modell` és `szín` mezők értéke a kérdésből nyilvánvaló. A borítékoló követheti ugyanazokat a lépéseket, mint a 11.6. példában, de az utolsó pontban, amikor az eredményeket kell visszaküldenie a közvetítőnek, el kell végeznie egy újabb „szűrést" is a `SELECT` záradékbeli vetítés képében a Tüskés modellek `WHERE` záradékbeli szűrése mellett:

```
SELECT sorszám, autoSeb, forgalmazó
FROM TempAutók
WHERE modell = 'Tüskés';
```

A 11.6. példához hasonlóan a `TempAutók` reláció itt sem jönne létre fizikailag a borítékolóban, a vetítést valószínűleg csővezetékes jellegű feldolgozás segítségével soronként végeznénk el. □

11.8. példa: Egy összetettebb példa kedvéért tegyük fel, hogy a közvetítőnek olyan forgalmazó-modell párokat kell találnia, ahol a forgalmazónak ebből a modellből van két piros példánya, az egyik automata sebességváltóval, a másik e nélkül. Még azt is tegyük fel, hogy Forgalmazó 1-hez csak a 11.6. ábráról már jól ismert sablon használható egyedül. A közvetítő tehát a 11.9. ábrán megadott lekérdezésre várja az eredményt a borítékolótól. Vegyük észre, hogy a forgalmazót nem kell megadnunk sem `A1` sem `A2` esetén, mert ez a borítékoló úgyis csak Forgalmazó 1 adataihoz fér hozzá. A közvetítő ugyanezt a lekérdezést küldi el az összes többi forgalmazó borítékolójához is.

```
SELECT A1.modell, A1.forgalmazó
FROM AutóKözv A1, AutóKözv A2
WHERE A1.modell = A2.modell AND
   A1.szín = 'piros' AND
   A2.szín = 'piros' AND
   A1.autoSeb = 'nem' AND
   A2.autoSeb = 'igen';
```

11.9. ábra. *Lekérdezés a közvetítőtől a borítékolóhoz*

Egy okosan tervezett borítékoló felismerné, hogy a közvetítő lekérdezését úgy is meg tudja válaszolni, ha először elkészíti a Forgalmazó 1-nél található összes piros autó adatait tartalmazó relációt:

```
PirosAutók(sorszám, modell, szín, autoSeb, forgalmazó)
```

Ehhez a 11.6. ábra sablonját használja, amelynek a segítségével az olyan lekérdezéseket lehet kezelni, amelyek csak egy színt adnak meg. Valójában a borítékoló úgy viselkedik, mintha a következő lekérdezésre válaszolna:

```
SELECT *
FROM AutóKözv
WHERE szín = 'piros';
```

A 11.6. ábra sablonját a `$c = 'piros'` behelyettesítéssel használva létre tudja hozni a `PirosAutók` relációt a Forgalmazó 1 adatbázisa alapján. Ahhoz, hogy a 11.9. ábrán látható lekérdezés végeredményét adja meg, a következő lépésben össze kell kapcsolnia a `PirosAutók` relációt saját magával, és el kell végeznie a szükséges kiválasztást. A 11.10. ábrán látható a borítékoló[7] által végrehajtott lekérdezés. □

```
SELECT DISTINCT A1.modell, A1.forgalmazó
FROM PirosAutók A1, PirosAutók A2
WHERE A1.modell = A2.modell AND
   A1.autoSeb = 'nem' AND
   A2.autoSeb = 'igen';
```

11.10. ábra. *A borítékoló (vagy a közvetítő) által végrehajtott lekérdezés a 11.9. ábra lekérdezésére ad választ*

[7] Néhány információegyesítő architektúrában ezt a feladatot inkább a közvetítő látná el.

11.2.5. Feladatok

* **11.2.1. feladat:** A 11.6. ábrán egy olyan borítékoló sablont láttunk, amely a közvetítő adott színű gépkocsikra vonatkozó lekérdezéseit a `Kocsik` relációval rendelkező forgalmazó számára értelmezhető lekérdezésekké alakította át. Most tegyük fel, hogy a közvetítő sémájában használt színkódolás más, mint amit a forgalmazó használ, de a következő konverziós táblázattal rendelkezünk: `GbólL(globSzín, LokSzín)`. Ennek megfelelően írjunk új sablont a borítékoló számára!

11.2.2. feladat: A 11.1.2. feladatban két számítógépes cégről, *A*-ról és *B*-ről beszéltünk, amelyek különböző sémákat használtak az adatbázisaikban. Tegyük fel, hogy létezik egy közvetítő a következő sémával:

```
PCKözv(gyártó, seb, mem, lemez, képernyő)
```

A reláció egy sora megadja a gyártót (*A* vagy *B*) és az annál a cégnél vásárolható összeállítás jellemzőit: a processzor sebességét, a központi memória, a merevlemez és a képernyő méretét. Írjunk a borítékolóban használható sablonokat a következő típusú lekérdezésekhez (lekérdezésenként összesen két sablont kell készíteni, egyet-egyet mindkét gyártó számára):

* a) Adjuk meg azokat a sorokat, amelyekben a sebesség nagysága egy előre meghatározott értékkel egyenlő!
b) Adjuk meg azokat a sorokat, amelyekben a képernyő mérete egy előre meghatározott értékkel egyenlő!
c) Adjuk meg azokat a sorokat, amelyekben a memória és a merevlemez mérete egy előre meghatározott értékkel egyenlő!

11.2.3. feladat: Tegyük fel, hogy mindkét információforrás (számítógépgyártó) borítékolója rendelkezik az előző feladatban leírt sablonokkal. Hogyan tudná ezeket a közvetítő felhasználni a következő lekérdezések megválaszolásában:

* a) Adjuk meg az összes olyan konfiguráció gyártóját, memóriájának és képernyőjének a méretét, amelynek a processzora 400 MHz-es és merevlemeze 12 Gb!
! b) Adjuk meg az 500 MHz-es konfigurációkban megtalálható maximális merevlemez-méretet!
c) Adjuk meg az összes olyan konfigurációt, amely 128 Mb memóriával rendelkezik, és a képernyő mérete (inchben) meghaladja a merevlemez méretét (gigabájtban)!

11.3. On-line analitikus feldolgozás

Ebben a fejezetben az egyesített információs rendszerek, különösen a tárház alapú rendszerek köré nőtt alkalmazások egy fontos osztályával ismerkedünk meg. Különböző cégek és szervezetek hoznak létre hatalmas adatbázisok felhasználásával olyan

Adattárházak és OLAP

Különböző okai vannak annak, hogy a tárházak miért játszanak olyan fontos szerepet az OLAP-alkalmazásokban. Egyrészt lehetséges, hogy az adataink kezdetben sok különböző adatbázisban vannak szétszórva, vagyis ahhoz, hogy OLAP-lekérdezéseket tudjunk végrehajtani, egy adattárházat kell létrehoznunk, amelyben megfelelően tudjuk szervezni és központosítani az egyesített adatokat. Másrészt viszont legtöbbször fontosabb annak a ténye, hogy az OLAP-lekérdezések végrehajtása – mivel ezek összetettek és az adatbázis nagy részét érintik – túl sok időt vesz igénybe az olyan tranzakciókezelő rendszerekben, amelyektől nagy teljesítményt várunk el. Az OLAP-lekérdezéseket gyakran tekinthetjük a 10.7. rész értelmében „hosszú tranzakcióknak".

Az egész adatbázist zároló hosszú tranzakciók miatt a szokásos OLTP-műveletek elvégzése lehetetlenné válna (például az eladásokból származó átlagos bevételt számító OLAP-lekérdezés futtatása közben nem vehetnénk fel az újabb vásárlások adatait az adatbázisba). Ezt a problémát rendszerint úgy oldjuk meg, hogy a még feldolgozatlan, nyers adatok másolatait összegyűjtjük egy adattárházba, itt futtatjuk az OLAP-lekérdezéseket, az adatváltoztató és az OLTP-lekérdezéseket pedig az adatforrásokon hajtjuk végre. A tárházat legtöbbször csak éjszakánként módosítjuk, napközben az elemzők a „befagyasztott" adatokon dolgoznak. Vagyis az adattárház adatai már 24 óra alatt elavulnak, ami az OLAP-lekérdezések eredményének az időszerűségét elég jól korlátozza, de ez az „időeltolódás" sok döntéstámogató alkalmazásban még a tűréshatáron belül marad.

tárházakat, amelyek alapján az arra kijelölt elemzők a szervezet számára fontos mintákat, irányvonalakat próbálnak megállapítani. Ez a tevékenység, az *on-line analitikus feldolgozás (OLAP)*, általában nagyon összetett, egy vagy több összesítő függvényt is felhasználó lekérdezéseket foglal magában. Ezeket a lekérdezéseket gyakran hívjuk *OLAP-* vagy *döntéstámogató lekérdezéseknek*. A 11.3.1. részben láthatunk ezekre néhány példát. Tipikus kérdés az olyan termékek keresése, amelyek iránt mindent egybevéve növekszik vagy éppen csökken a kereslet.

Az OLAP-alkalmazásokban használt lekérdezések jellemzően nagyon nagy mennyiségű adatot vizsgálnak át, még ha az eredmény nagyon kicsi is lesz. Ezzel szemben a mindennapos adatbázis-műveletek (például banki befizetés vagy repülőjegy-foglalás) az adatbázisnak ennél csak lényegesen kisebb hányadát érintik. Az ilyen típusú műveleteket gyakran nevezzük *on-line tranzakció-feldolgozásnak (OLTP)*.[8]

Az utóbbi időben olyan új lekérdezésfeldolgozó technikákat dolgoztak ki, amelyek különösen jól használhatók az OLAP-lekérdezések hatékony végrehajtásához. Az OLAP-lekérdezések bizonyos osztályainak különböző természete miatt pedig speciá-

[8] Az OLTP mozaikszó az angol *On-Line Transaction Processing* kifejezésnek felel meg. *A fordító megjegyzése.*

lis adatbázis-kezelő rendszereket, az *adatkocka*rendszereket fejlesztették ki és dobták piacra, hogy az OLAP-alkalmazásokat megfelelően támogassák. Ezek a rendszerek a 11.4. részben kerülnek tárgyalásra.

11.3.1. OLAP-alkalmazások

Az OLAP-alkalmazások általában fogyasztásra vonatkozó adatokból összeállított tárházat használnak. Nagyobb üzlethálózatok terabájtnyi információt halmoznak fel arról, hogy melyik üzletükben melyik cikkből mennyit adtak el. A cégre váró problémák vagy nagy lehetőségek előrejelzésében nagy szerepe lehet az olyan lekérdezéseknek, amelyek a fogyasztási adatok csoportosítása, összesítése alapján a valamilyen szempontból jelentős csoportokat azonosítani tudják.

11.9. példa: Tegyük fel, hogy a Süni cég létrehoz egy adattárházat, amely alapján elemezni tudja majd a gépkocsik iránti keresletet. A tárház sémája lehet a következő:[9]

```
Eladások(sorszám, dátum, forgalmazó, ár)
Autók(sorszám, modell, szín)
Forgalmazók(név, város, állam, tel)
```

Egy jellemző döntéstámogató lekérdezés az 1999. április 1-jén vagy azóta történt értékesítések alapján megállapíthatná, hogy az eladott járművek átlagára hogyan változik államonként. Egy ilyen lekérdezést mutatunk be a 11.11. ábrán.

```
SELECT állam, AVG(ár)
FROM Eladások, Forgalmazók
WHERE Eladások.forgalmazó = Forgalmazók.név AND
      dátum >= '1999-01-04'
GROUP BY állam;
```

11.11. ábra. *Államonkénti átlagos fogyasztói ár*

Figyeljük, meg, ahogy a lekérdezés végigveszi az adatbázis nagy részét, miközben az `Eladások` relációban tárolt vásárlási adatokat osztályozza a forgalmazó címe szerint. Ezzel szemben a „milyen áron adták el a 123-as sorszámú autót?" típusú gyakori OLTP-lekérdezések az adatbázis egyetlen sorát érintik csupán. □

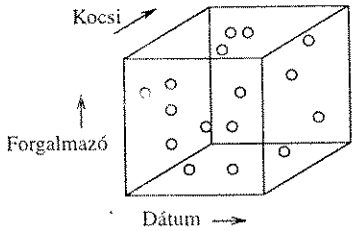
Hogy lássunk egy másik OLAP-példát is, vegyünk egy hitelkártya-kibocsátó céget, amely a kártyaigénylőiről akarja eldönteni, hogy hitelképesek lesznek-e. Készítenek egy adattárházat, amely a cég összes ügyfeléről és az eddigi befizetéseiről tartalmaz információkat. Az OLAP-lekérdezések olyan tényezőket (például életkor, jövedelem,

[9] Az Egyesült Államokban ésszerű a forgalmazó címében az államot is nyilvántartani, ezért szerepel a `Forgalmazók` relációban az `állam` attribútum. *A fordító megjegyzése.*

ingatlantulajdon, irányítószám) fognak keresni, amelyek segíthetnek annak az előrejelzésében, hogy egy adott ügyfél be fogja-e fizetni időben a számláit vagy nem. Ehhez hasonlóan kórházak is rendelkezhetnek betegek adatait (felvétel időpontja, elvégzett laborvizsgálatok, ezek eredménye, diagnózis, gyógykezelés leírása és így tovább) tartalmazó adattárházzal, hogy elemezni tudják a kezeléssel járó kockázatot, és ennek tükrében válasszák ki a legjobbnak ítélt módszert.

11.3.2. OLAP-adatok többdimenziós nézete

A tipikus OLAP-alkalmazásokban mindig létezik egy központi reláció vagy adatgyűjtemény: a *ténytáblázat* (fact table). A ténytáblázat olyan érdekes eseményeket vagy objektumokat tartalmaz, mint a 11.9. példában megismert vásárlási adatok. Gyakran segít, ha a ténytáblázatban tárolt objektumokra úgy gondolunk, mintha egy többdimenziós térben vagy „kockában" lennének elrendezve. A 11.12. ábrán háromdimenziós adatobjektumok láthatók, a kocka belső pontjaiként ábrázolva.[10] A dimenziók elnevezései: kocsi, forgalmazó, dátum, a korábbi személygépkocsi értékesítős példának megfelelően. A kocka minden egyes belső pontjára gondolhatunk úgy, mint egy-egy gépkocsi eladására, a dimenziókra pedig úgy, mintha ennek az eladásnak a körülményeit, jellemzőit írnák le.



11.12. ábra. *Az adatok többdimenziós kockába szervezése*

A 11.4. részben bevezetünk egy, az OLAP-adatok kezelésére alkalmas, speciális architektúrát, az „adatkockát". Ez a felépítés egy kicsit más szempontból közelíti meg a többdimenziós adatokat, ugyanis az adatkockában a pontok jelenthetnek összesített adatot is. Például ahelyett, hogy a „kocsi" dimenzió mentén minden egyes pont más és más kocsit jelölne, elképzelhető, hogy a dimenzió modellenként van összesítve, és a 11.12. ábra pontjai nem egyes autók, hanem egyes modellek összeladását jelentik a forgalmazó és a dátum dimenziók által meghatározott paraméterek mellett. A többdimenziós adat e kétfajta értelmezése közti különbség tükröződik a kockaszerkezetű OLAP-adatot támogató speciális rendszerek által vett két fő irányvonalban is:

[10] Háromnál több dimenzióval rendelkező objektumok esetén természetesen legalább négydimenziós kockát kellene felhasználnunk, de azt jóval nehezebb elképzelni és lerajzolni is. *A fordító megjegyzése.*

1. *ROLAP*[11] vagy *Relációs OLAP*. Ebben a megközelítésben az adatot relációkban tároljuk, de egy speciális szerkezetben, a „csillag sémában". A relációk egyike a ténytáblázat, amely a *nyers*, vagyis a „még nem összesített" adatokat tartalmazza. A rendszer lekérdezőnyelve és más képességei ezt az adatszervezési módot kihasználva alakíthatók. A csillag sémával a 11.3.3. részben foglalkozunk.
2. *MOLAP*[12] vagy *Többdimenziós OLAP*. Ebben az esetben a fent említett speciális szerkezetet, az „adatkockát" használjuk az adat tárolására. Ahogy már említettük, ez az adat gyakran már részben összesített. Az efféle adat OLAP-lekérdezéseit támogatandó, a rendszer nem relációs műveleteket is megvalósíthat.

11.3.3. A csillag séma

A *csillag séma* a ténytáblázat sémájából áll, amely több más relációhoz, a „dimenziótáblázatokhoz" kapcsolódik. A dimenziótáblázatokról kicsit később lesz részletesebben szó. A „csillag" központjában található a ténytáblázat, a csúcsaiban pedig a dimenziótáblázatok. A ténytáblázatnak általában van néhány dimenzió attribútuma – ezek az egyes *dimenziókat* jelölik –, és egy vagy több *függő* attribútuma, amelyek az adat, mint a többdimenziós tér egy pontjának, mint egésznek érdekes tulajdonságait jelölik. Például a vásárlásra vonatkozó adatok dimenziói között szerepelhet a vásárlás időpontja, helyszíne (melyik üzletben történt), a vásárolt cikk típusa, a fizetési mód (készpénz vagy hitelkártya) és így tovább. Függő attribútum(ok) lehet(nek) például a fogyasztói ár, a kereskedelmi ár vagy az adó mennyisége.

11.10. példa: Az előző példában bevezetett `Eladások` reláció

```
Eladások(sorszám, dátum, forgalmazó, ár)
```

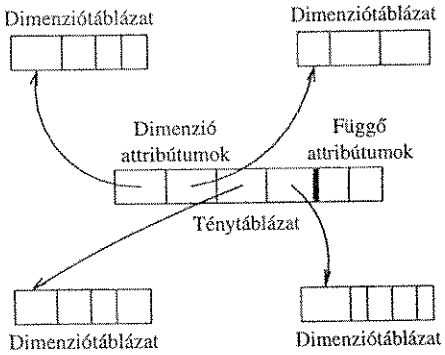
a ténytáblázat. A dimenziók a következők:

1. `sorszám`, az eladott gépkocsit jelöli, azaz az összes gépkocsi terében az ezzel a sorszámmal rendelkező autónak megfelelő pontot.
2. `dátum`, a vásárlás időpontját jelöli, azaz a vásárlás, mint esemény pozícióját az idő dimenzióban.
3. `forgalmazó`, az esemény pozícióját jelöli az összes forgalmazó terében.

Az egyetlen függő attribútum az `ár`, amire az adatbázis OLAP-lekérdezéseinek jellemzően valamilyen összesített formában lesz majd szüksége. Persze az összár vagy az átlagár megállapítása mellett a leszámlálós lekérdezéseknek is lehet értelmük. Például „adjuk meg, hogy 1999 májusában forgalmazóként összesen hány autót adtak el". □

[11] A ROLAP mozaikszó az angol *Relational OLAP* kifejezésnek felel meg. *A fordító megjegyzése.*

[12] A MOLAP mozaikszó az angol *Multidimensional OLAP* kifejezésnek felel meg. *A fordító megjegyzése.*



11.13. ábra. *A ténytáblázat dimenzió attribútumai a dimenziótáblázatok kulcsmezőire hivatkoznak*

A ténytáblázat kiegészítéseképpen használhatók a *dimenziótáblázatok*, amelyek az egyes dimenziók lehetséges értékeit írják le. A ténytáblázat mindegyik dimenzió attribútuma rendszerint idegen kulcsként működik a megfelelő dimenziótáblázatban, ahogy ezt a 11.13. ábra szemlélteti. A dimenziótáblázat attribútumai azt is leírják, hogy melyek azok a lehetséges csoportosítások, amelyeknek értelmük lenne egy SQL-lekérdezés `GROUP BY` záradékában. A következő példán keresztül bemutatjuk az új fogalmakat.

11.11. példa: A 11.9. példa gépkocsi-adatbázisában a dimenziótáblázatok közül kettő nyilvánvaló:

```
Autók(sorszám, modell, szín)
Forgalmazók(név, város, állam, tel)
```

A ténytáblázat

```
Eladások(sorszám, dátum, forgalmazó, ár)
```

`sorszám` attribútuma idegen kulcs, és az `Autók` dimenziótáblázatban[13] a sorszám attribútumra hivatkozik. Az `Autók.modell` és `Autók.szín` attribútumok pedig az adott autó tulajdonságait írják le. Szerepelhetne még sokkal több attribútum is ebben a relációban, igaz/hamis értékekkel jelölve, hogy a kocsiban van-e automata sebességváltó vagy akármilyen más extra felszerelés. Ha az `Eladások` ténytáblázatot összekapcsoljuk az `Autók` dimenziótáblázattal, akkor a `modell` és `szín` attribútumok szerint különböző érdekes módon lehet csoportosítani a vásárlásokat. Például lebonthatjuk az adatokat szín szerint, vagy a Tüskés modell értékesítéseket hónap és forgalmazó szerint.

[13] Most véletlenül a `sorszám` az `Eladások` relációban is kulcsként működik, de nem kell, hogy legyen olyan attribútum, amely a ténytáblázatnak is kulcsa és ideiglenes kulcs is egyben valamelyik dimenziótáblázatban.

Hasonlóan, az `Eladások` táblázat `forgalmazó` attribútuma idegen kulcs a `Forgalmazó` dimenziótáblázatban a `név` attribútumra hivatkozva. Ha az `Eladások` és `Forgalmazó` táblázatokat összekapcsoljuk, további lehetőségek adódnak az adatok csoportosítására. Például lebonthatjuk a vásárlásokat állam, város vagy forgalmazó szerint.

Eltűnődhetünk azon, hogy hol találjuk az időnek (az `Eladások` táblázat `dátum` attribútumának) megfelelő dimenziótáblázatot. Mivel az idő egy fizikai adottság, nincs értelme az adatbázisban időre vonatkozó tényeket tárolni, hiszen a „melyik évre esik 2000. július 5.?" típusú lekérdezések eredményét úgysem tudjuk megváltoztatni. De minthogy az elemzőknek gyakran van szükségük különböző időegységek, például hét, hónap, negyedév, év szerinti csoportosításra, segít, ha az idő fogalmát beépítjük az adatbázisba, éppen úgy, mintha a következő dimenziótáblázattal rendelkeznénk:

```
Napok(nap, hét, hónap, év)
```

A „reláció" egy jellegzetes, 2000. július 5-nek megfelelő sora lenne a következő:

```
(5, 27, 7, 2000)
```

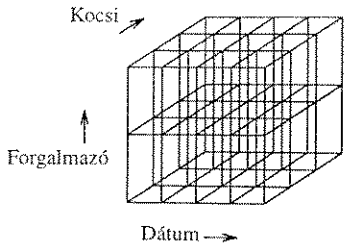
Ezt úgy értelmezzük, hogy ez a nap a 2000. év hetedik hónapjának az ötödik napjára esik, és történetesen a 2000. év 27-edik teljes hetéhez tartozik. Ez bizonyos fokig redundáns, mert a hetet ki lehet számolni a másik három attribútum alapján, viszont a hetek nem illeszthetők pontosan a hónapokhoz, azaz hetek szerinti csoportosításból nem nyerhetünk hónapok szerinti csoportosítást, és megfordítva sem. Tehát van értelme annak, ha úgy képzeljük el, hogy a hetek és a hónapok is jelölve vannak ebben a „dimenziótáblázatban". □

11.3.4. Szeletelés és kockázás

Az adatkockára úgy is gondolhatunk, mintha minden dimenzió mentén valamekkora finomsággal fel lenne osztva. Például az idő dimenziót feloszthatjuk (SQL-es szóhasználattal „csoportosíthatjuk"[14]) napok, hetek, hónapok, évek szerint, de azt is megtehetjük, hogy nem osztjuk fel egyáltalán. Az autó dimenziót feloszthatjuk modell szerint, szín szerint, modell és szín szerint, a forgalmazó dimenziót pedig a forgalmazó neve, a város vagy az állam szerint. Természetesen e két dimenzió esetén is megtehetjük, hogy egyáltalán nem csoportosítunk.

Ha minden dimenzióhoz választunk egy felosztást, ez „felkockázza" az adatkockát, ahogy ez a 11.14. ábrán látható. A kockázás eredményeképpen az adatkockában kisebb kockák jönnek létre. Az ezeket alkotó pontcsoportok jellemzőit az a lekérdezés

[14] Az angol *group by* kifejezésből. *A fordító megjegyzése.*



11.14. ábra. *A dimenziók felosztása darabolja (kockázza) a kockát*

összesíti, amely a `GROUP BY` záradékban a felosztásnak megfelelő csoportosítást hajtja végre. A `WHERE` záradékon keresztül a lekérdezésnek arra is lehetősége van, hogy csak bizonyos felosztásokra koncentráljon egy (vagy több) dimenzió mentén, azaz a kockának csak bizonyos „szeleteivel" foglalkozzon. Ennek az lesz az eredménye, hogy a lekérdezés a kockának csak bizonyos alterein végez összesítést.

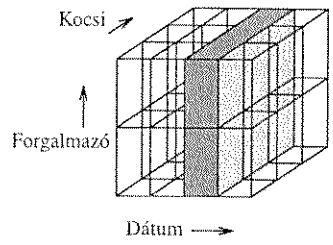
11.12. példa: A 11.15. ábra egy olyan lekérdezésre utal, amely a kockából az egyik dimenzió (dátum) mentén kivág egy szeletet, majd ezt a másik két dimenzió mentén (kocsi és forgalmazó) tovább kockázza. A dátum négy részre van felosztva, talán arra a négy évre, amelyek alatt ezek az adatok összegyűltek. Az ábra satírozása azt akarja kifejezni, hogy ebből a négy évből csak egyetlenegyre vagyunk kíváncsiak.

A kocsik három csoportra vannak osztva, például szedánokra (négyüléses, zárt autókra), kupékra (sportkocsikra) és a kabrioletekre (leereszthető tetejű kocsikra). A forgalmazók kettőre, például a keleti, illetve a nyugati országrészen működőkre. A lekérdezés eredménye egy olyan táblázat, amely a számunkra érdekes évben megadja a gépkocsi-értékesítésekből származó összbevételt mind a hat kategóriában. □

Az úgynevezett „szeletelős-kockázós" lekérdezés általános formája tehát a következő:

```
SELECT csoportosító attribútumok és összesítések
FROM ténytáblázat nulla vagy több dimenziótáblával
     összekapcsolva
WHERE bizonyos attribútumok konstans értékekkel vannak
     összehasonlítva
GROUP BY csoportosító attribútumok;
```

11.13. példa: Folytassuk a személygépkocsis példánkat, de a 11.11. példában megtárgyalt `Napok` fogalmi szintű dimenziótáblázattal együtt. Ha a Tüskés modellt nem vásárolják annyian, mint gondoltuk, megpróbálhatjuk kiszűrni azokat a színeket, amelyek nem mennek olyan jól. Ez a lekérdezés csak az `Autók` dimenziótáblázatot használja, és SQL-ben így nézhet ki:



11.15. ábra. *Egy szelet kiválasztása a darabolt (kockázott) kockából*

```
SELECT szín, SUM(ár)
FROM Eladások NATURAL JOIN Autók
WHERE modell = 'Tüskés'
GROUP BY szín;
```

Először szín szerint kockázza, majd modellenként szeleteli az adatkockát. Eközben a Tüskés modellre összpontosít, és a többi adatot figyelmen kívül hagyja.

Tegyük fel, hogy a lekérdezés eredménye nem mondott sokat, minden színből körülbelül ugyanakkora jövedelmünk származott. Mivel a lekérdezés az időt nem bontotta részekre, a színek szerinti végösszeget az egész időtartamra számította ki. Feltehetjük, hogy egy vagy több szín gyenge szereplése csak az utóbbi időre jellemző. Ekkor megpróbálkozhatunk a következő, immár bővített lekérdezéssel, amely hónapok szerint is particionál:

```
SELECT szín, hónap, SUM(ár)
FROM (Eladások NATURAL JOIN Autók) JOIN Napok ON dátum = nap
WHERE modell = 'Tüskés'
GROUP BY szín, hónap;
```

Fontos észben tartanunk, hogy a `Napok` reláció nem egy szokásos módon tárolt reláció, bár kezelhetjük úgy, mintha a következő sémával rendelkezne.

```
Napok(nap, hét, hónap, év)
```

Az adatkockarendszerek[15] többek között azért is tekinthetők speciális adatbázis-kezelő rendszereknek, mert képesek az efféle „relációk" használatára is.

Az előző lekérdezés alapján esetleg azt állapíthatjuk meg, hogy a piros Tüskések iránt az utóbbi időben nem volt túl nagy kereslet. Következő lépésként megpróbálhatnánk kideríteni, hogy ez általánosan az összes forgalmazóra igaz, vagy csak egy részüknél figyelhető meg ez a probléma. Az újabb lekérdezésben csak a piros Tüskésekre összpontosítunk, és a forgalmazó dimenzió szerint is csoportosítunk. Ily módon a lekérdezés:

[15] Adatkockarendszereknek az adatkocka adatmodellt támogató rendszereket hívjuk. Ezekről bővebben a 11.4. részben lesz szó. *A fordító megjegyzése.*

```
SELECT forgalmazó, hónap, SUM(ár)
FROM (Eladások NATURAL JOIN Autók) JOIN Napok ON dátum = nap
WHERE modell = 'Tüskés' AND szín = 'piros'
GROUP BY hónap, forgalmazó;
```

Ezen a ponton azt vesszük észre, hogy a piros Tüskésekből havonta olyan keveset adtak el, hogy ez alapján nem figyelhető meg könnyen semmilyen jellemző irányvonal. Úgy döntünk tehát, hogy hiba volt a hónapok szerinti felosztás. Jobb ötlet lenne csak évekre bontani az adatokat és csak az utolsó két évet vizsgálni (ebben a képzeletbeli példában 1999-et és 2000-et). A végleges lekérdezés a 11.16. ábrán látható. □

```
SELECT forgalmazó, év, SUM(ár)
FROM (Eladások NATURAL JOIN Autók) JOIN Napok ON dátum = nap
WHERE modell = 'Tüskés' AND
      szín = 'piros' AND
      (év = 1999 OR év = 2000)
GROUP BY év, forgalmazó;
```

11.16. ábra. *A végleges szeletelő-kockázó lekérdezés a piros Tüskés vásárlásokról*

11.3.5. Feladatok

* **11.3.1. feladat:** Egy on-line számítógép-értékesítő cég adatbázisban akarja nyilvántartani a vásárlók megrendeléseit. A vevők a PC-jükbe különböző processzorok, merevlemezek és CD-, illetve DVD-olvasók közül választhatnak, és megadhatják, hogy mekkora központi memóriára van szükségük. Az adatbázis ténytáblázata lehet a következő:

```
Megrendelések(vásárló, dátum, proc, memória, lemez, cd, menny, ár)
```

A `vásárló` attribútum a vevő azonosítójaként működik, és egyúttal idegen kulcs a vásárlókat nyilvántartó dimenziótáblázathoz. Ugyanez igaz a `proc`, `lemez` és `cd` attribútumokra is. A lemezazonosító szerepelhet például abban a dimenziótáblázatban, amely megadja a merevlemez gyártóját és más jellemzőit. A `memória` attribútum egy egész szám, a megrendelt memória méretét adja meg megabájtban. A `menny` attribútum a vásárló által megrendelt konfigurációk számát jelenti, az `ár` attribútum pedig ebből egy darab összköltségét.

a) Melyek a dimenzió attribútumok és melyek a függő attribútumok?
b) Néhány dimenzió attribútumhoz valószínűleg dimenziótáblázat szükséges. Adjunk meg ezekhez megfelelő sémákat!

! **11.3.2. feladat:** Tegyük fel, hogy az előző feladat adatbázisát vizsgáljuk annak érdekében, hogy a jellemző trendek alapján előre jelezhessük a cégnek, hogy milyen alkotórészekből kell majd többet rendelnie. Adjuk meg mélyre ásó és felgörgető lekérdezések egy olyan sorozatát, amely ahhoz a következtetéshez vezethet, hogy a vásárlók egyre inkább előnyben részesítik a DVD-meghajtót a CD-meghajtóval szemben!

Mélyre ásás és felgörgetés

A 11.13. példában az adatkockát szeletelő és kockázó lekérdezések sorozatában két gyakori mintával találkozunk.

1. A *mélyre ásás* az a folyamat, amely során a megfelelő dimenziókat egyre finomabb részekre osztjuk, és/vagy a dimenzió bizonyos értékeire koncentrálunk. A 11.13. példában az utolsó lépés kivételével mindegyik a mélyre ásás folyamatába illik.
2. A *felgörgetés* az egyre durvább particionálás folyamata. Az utolsó két lépést hozhatjuk fel erre példaként, amelyben hónapok helyett évek szerint csoportosítottunk, hogy az adatok esetlegességét kiküszöböljük.

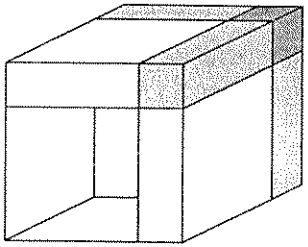
11.4. Adatkocka

A döntéstámogató lekérdezések eddig mindig ad hoc lekérdezések formájában jelentek meg. Ezzel szemben egy másik lehetőség az összes lehetséges összesítés módszeres kiszámítása, előre. Meglepő, de az ehhez szükséges tártöbblet mennyisége gyakran a még ésszerű határon belül marad, és amíg az adattárházban tárolt adat nem változik, az összesített adat frissítése sem jelent többletköltséget. Ebben a fejezetben az adatbázis-kezelő rendszerek egy családjával, az *adatkocka*rendszerekkel, más néven MOLAP- (többdimenziós OLAP) rendszerekkel foglalkozunk. Ezek közvetlenül a 11.12. ábrán bemutatott (adat)kocka adatmodellt támogatják, és a legfontosabb OLAP-műveletek elvégzését is biztosítják.

Az adatkockarendszerekben teljesen hétköznapi dolognak számít, ha még az adatkocka-tároló rendszerbe való felvétel előtt a ténytáblázat nyers adata néhány szempont szerint összesítve van. Az autós adatbázisunkban például a csillag sémában szereplő sorszám dimenziót kicserélhetjük a modell dimenzióra, így minden bejegyzés az adatkockában egy modell, egy forgalmazó, egy dátum, és ezzel együtt az adott modellből az adott napon az adott forgalmazónál történt vásárlások összegének leírásává válik. Az adatkocka pontjainak összességét továbbra is „ténytáblázatnak" nevezzük, még ha a pontok értelmezése egy kicsit el is tér a csillag séma ténytáblázatától.

11.4.1. A kockaművelet

Adott F ténytáblázat esetén definiálhatunk egy kibővített KOCKA(F) táblázatot, amely egy további, *-gal jelölt értéket ad minden dimenzióhoz. A *-nak a jelentése „bármi", és összesítést jelent annak a dimenziónak a mentén, ahol feltűnik. A 11.17. ábra szemlélteti azt az eljárást, amely során a kockához minden dimenzió mentén a * értéket, és az ennek megfelelő összesített értéket képviselő új határvonalakat adjuk hozzá. Az ábrán három dimenziót látunk, a leghalványabb részek jelölik az egy dimenzió mentén vett összesítéseket, a kicsit sötétebbek a két dimenzió mentén, a legsötétebb kis kocka a sarokban pedig a mind a három dimenzió mentén vett összesítéseket. Gondoljuk meg, hogy ha az egyes dimenziók lehetséges értékeinek száma elég nagy, de nem annyira, hogy a legtöbb pont a kocka belsejében szabad legyen (azaz, hogy ne tartozzon hozzá sor a ténytáblázatban), akkor a „határ" csak kis többletet jelent a kocka terjedelméhez (azaz a ténytáblázat sorainak számához) képest. Ebben az esetben a KOCKA(F) táblázat mérete nem sokkal nagyobb az F ténytáblázat méreténél.



11.17. ábra. *A kockaművelet a dimenziók minden lehetséges kombinációján kiszámított összesítéseknek megfelelő határvonalakkal bővíti az adatkockát*

A KOCKA(F) táblázat egy olyan sora, amelyben egy vagy több dimenzió attribútum mentén * található, a függő attribútumoknak megfelelő mezőkben egy összeget (vagy egy más összesítő függvénnyel számított mennyiséget) tartalmaz. Ez úgy számítható ki, hogy vesszük az összes olyan sort, amelyben a dimenzió attribútum értéke nem *, hanem egy valódi érték, és ezen sorok függő attribútumai mentén végezzük el az összesítést. Gyakorlatilag beépítjük az adatokba az összes lehetséges dimenzióhalmazon végzett összesítés eredményét. Vegyük azonban észre, hogy a KOCKA művelet nem támogatja a „köztes szinten" számított összesítéseket. Az adatokat például vagy hagyjuk napokra (illetve az idő dimenzió legfinomabb felbontásának megfelelő időközre) lebontva, vagy a teljes időtartamra nézve összesítünk. A KOCKA művelettel önmagában nem összesíthetünk hetek, hónapok vagy évek szerint.

11.14. példa: A 11.9. példa Süni adatbázisát vizsgáljuk meg a KOCKA művelet lehetőségeinek fényében. Emlékeztetőül, az ott használt ténytáblázat a következő:

```
Eladások(sorszám, dátum, forgalmazó, ár)
```


A `sorszám` attribútumnak megfelelő dimenzió azonban nem alkalmas arra, hogy a kockában használjuk, mert a sorszám egyértelműen azonosítja a gépkocsikat, és így a `sorszám` kulcsként működik az `Eladások` relációban. Vagyis ha a kocsik árát összegezzük a teljes időtartamra vagy az összes forgalmazóra nézve úgy, hogy közben a sorszám rögzítve marad, akkor ennek nem lesz hatása – az adott sorszámú (egyetlen) autó árának az összegét kapjuk. Használhatóbb az adatkocka, ha a sorszámot két másik attribútummal, a modellel és a színnel cseréljük fel. Ezek azok az attribútumok, amelyekhez a sorszám az `Autók` dimenziótáblázaton keresztül az `Eladások` relációt kapcsolja. Vegyük észre, hogy ha a `sorszám` attribútumot kicseréljük a `modell` és `szín` attribútumokra, akkor a kockának már egyetlen dimenziója sem működik kulcsként. Így a kocka egy-egy bejegyzése az olyan kocsik összértékét adja, amelyek adott színűek, adott modellek és az adott napon az adott forgalmazónál vásárolták meg őket.

Ha az `Eladások` ténytáblázatot az adatkockarendszerben akarjuk megvalósítani, egy további változtatás is hasznunkra válhat. Mivel a KOCKA operátor a függő attribútumokat általában összegezni szokta, ezért ha kíváncsiak vagyunk az átlagárra is, szükségünk lehet az eladott kocsik összértékére minden kategóriában (adott szín és modell az adott napon az adott forgalmazónál) és az abba a kategóriába eső vásárlások darabszámára is. Vagyis az `Eladások` reláció, amire a KOCKA műveletet alkalmazzuk, a következő:

```
Eladások(modell, szín, dátum, forgalmazó, összért, db)
```

Az `összért` attribútum az adott modellhez, színhez, dátumhoz és forgalmazóhoz tartozó eladott kocsik összértéke, a `db` pedig az ebbe a kategóriába eső autók darabszáma. Vegyük észre, hogy ebben az adatkockában az egyes autók külön-külön nem azonosíthatók, vagyis ilyen értelemben nincsenek jelen a kockában. Természetes viszont, hogy a saját kategóriájukon belül az `összért` és `db` attribútumok értékére kifejtik a hatásukat.

Vegyük szemügyre a KOCKA(`Eladások`) relációt. Ennek egy lehetséges sora, amely az `Eladások` relációban is megtalálható, a következő:

```
('Tüskés', 'piros', '1999-05-21', 'Undok Ubul', 45000, 2)
```

Ezt úgy értelmezzük, hogy 1999. május 21-én Undok Ubul eladott két piros Tüskést, összesen 45 000 dollár értékben. A következő sor:

```
('Tüskés', *, '1999-05-21', 'Undok Ubul', 152000, 7)
```

azt jelenti, hogy 1999. május 21-én Undok Ubul összesen hét Tüskést adott el, amelyek összértéke 152 000 dollár. Vegyük észre, hogy ez a sor előfordulhat a KOCKA(`Eladások`) relációban, de az `Eladások` biztosan nem tartalmazza.

A KOCKA(`Eladások`) reláció olyan sorokat is tartalmaz, amelyek egynél több attribútum mentén is tartalmaznak összesítést. Például a következő sor:

```
('Tüskés', *, '1999-05-21', *, 2348000, 100)
```

azt jelenti, hogy 1999. május 21-én a forgalmazók összesen 100 Tüskést adtak el, és ezek összértéke 2 348 000 dollár volt.

```
('Tüskés', *, *, *, 1339800000, 58000)
```

Ez a sor pedig azt jelenti, hogy a nyilvántartott egész időtartam alatt a forgalmazók összesen 58 000 Tüskést adtak el (mindenféle színből), összesen 1 339 800 000 dollár értékben. Végül, ebből a sorból:

```
(*, *, *, *, 3521727000, 198000)
```

azt tudhatjuk, meg, hogy a nyilvántartott időtartam alatt a Süni típusú személygépkocsikból összesen 198 000 példányt adtak el (színre és forgalmazóra való tekintet nélkül), és ezek értéke összesen 3 521 727 000 dollár. □

Nézzük meg, hogyan lehet válaszolni az olyan lekérdezésekre, amelyekben az `Eladások` reláció bizonyos attribútumaira feltételeket adunk meg, más attribútumok szerint csoportosítunk, eredményül pedig az összeget, darabszámot vagy az átlagos fogyasztói árat várjuk. A KOCKA(`Eladások`) relációban az olyan t sorokat keressük, amelyek rendelkeznek a következő tulajdonságokkal:

1. Ha a lekérdezés az a attribútum értékeként v-t adja meg, akkor a t sor a mezőjében v szerepel.
2. Ha a lekérdezés a attribútum szerint csoportosít, akkor a t sor a mezőjében tetszőleges nem-* érték szerepel.
3. Ha a lekérdezés nem ad meg a-nak értéket és nem is csoportosít a szerint, akkor a t sor a mezőjében * szerepel.

Minden t sor tartalmazza az összeget és a darabszámot a csoportosítások egyikére nézve. Ha az átlagárra van szükségünk, akkor minden t sorban az összeg és darabszám hányadosát kell venni.

11.15. példa: Ezt a lekérdezést

```
SELECT szín, AVG(ár)
FROM Eladások
WHERE modell = 'Tüskés'
GROUP BY szín;
```

úgy válaszoljuk meg, hogy kikeressük a KOCKA(`Eladások`) reláció minden olyan sorát, amely a következő formában írható:

```
('Tüskés', c, *, *, v, n)
```


A „kocka" különféle fogalmai

A 11.3. részben kezdtünk el „kockákkal" foglalkozni, de ott még ez a fogalom sokkal filozofikusabb jellegű volt. A relációs adatok bizonyos fajtáira hasznos úgy gondolnunk, mintha egy ténytáblázatból és dimenziótáblázatokból állnának. Ebben az esetben a „kocka" a ténytáblázat egy megjelenési formája.

A 11.4. részben bevezettünk egy formális kocka műveletet, amely összesített értékeket számol ki egy vagy több dimenzió mentén. Ez a művelet a 11.3. rész ténytáblázataira is alkalmazható, hogyha csak a mindent-vagy-semmit összegzéseknek van értelmük. A 11.14. példában azonban azt is láttuk, hogyan lehet egy dimenziót, mondjuk az „autók" dimenziót több, a dimenziótáblázatból nyert dimenzióval helyettesíteni, amelyek az autókat más nézőpontból közelítik meg (ebben a példában a kocsikat a sorszámuk helyett a színükkel és a modellel írtuk le). E változtatás után sokkal jobban ki tudtuk használni a KOCKA művelet lehetőségeit. Addig csak kétféle módon csoportosíthattunk: vagy minden autót összevontunk, vagy nem vontuk össze őket egyáltalán. A régi dimenzió lecserélése után azonban már bármely új dimenzió mentén tudtunk összesíteni (modell, szín vagy mindkettő szerint).

A 11.4.2. részben aztán olyan esetekkel ismerkedtünk meg, ahol a dimenzió több, független dimenzióra osztása (mint a kocsik színre és modellre osztása) nem volt elegendő. A dimenziók rendelkezhetnek ennél összetettebb struktúrával, ahol az egyedeket (például a forgalmazókat) különböző finomságú szinteken vagy még egy ennél is bonyolultabb rendszer szerint lehet csoportosítani, ahogy azt az időegységek példáján láttuk. Az olyan dimenziók esetén, amelyek nem oszthatók további dimenziókra, a KOCKA művelet lehetőségeit kevésbé lehet jól kihasználni, de a bizonyos felosztások szerinti előösszesítés a KOCKA művelet egy olyan általánosítása, amely hatékony lehet és számos forgalomban lévő rendszer is alkalmazza.

ahol c bármilyen jellemző szín lehet. Ebben a sorban v a megfelelő színű eladott Tüskések összértékét, n pedig az ilyen színű eladott Tüskések számát adja meg. A lekérdezés (c, v/n) formában írható sorokat vár eredményül. Bár a átlagár nem egy közvetlen attribútum az `Eladások` vagy a KOCKA(`Eladások`) relációban, az értéke kiszámítható az összérték és a darabszám hányadosaként. A lekérdezésre adott válasz tehát a ('Tüskés', c, *, *, v, n) sorokból nyert (c, v/n) párok halmaza. □

11.4.2. Kockaimplementáció megvalósított nézettáblákkal

A 11.17. ábra kapcsán azt állítottuk, hogy a kocka kibővítése az összesített értékekkel nem jár jelentős tártöbblettel, sőt időt takarítunk meg vele a leggyakrabban előforduló döntéstámogató lekérdezések tekintetében. Ez az elemzésünk azonban azon a feltevé-

sen alapult, hogy a lekérdezés vagy mindent átfogóan összesít egy dimenzióra nézve, vagy nem összesít egyáltalán. Néhány dimenzió esetén viszont a csoportosítás többféle finomsági fokon is elvégezhető.

Már említettük az idő dimenziót, ahol számos lehetőségünk van a csoportosításra. Az alapvető „mindent vagy semmit", azaz az egész időtartamra, illetve a napokra való lebontás mellett összesíthetünk például hetek, hónapok, negyedévek vagy évek szerint is. Egy másik példa kedvéért gondoljunk a személygépkocsis adatbázisra. Lehetőségünk volt a forgalmazók teljes vagy „egyáltalán nem" összevonására. Dönthetünk azonban a város, az állam vagy más kisebb-nagyobb területegység szerinti összegzés mellett is. Vagyis az idő szerinti csoportosítás esetén legalább hat, forgalmazók szerint pedig legalább négy választási lehetőségünk van.

Ha a választható csoportosítási szintek száma minden dimenzió mentén növekszik, egyre drágább az összes lehetséges csoportosítási kombináció összesített eredményeit tárolni. Nem csak az jelent gondot, hogy túl sok adatot kell tárolni, hanem az is, hogy nem olyan könnyen szervezhetők, mint a 11.17. ábrán a „mindent vagy semmit" esetén. Ezért a forgalomban lévő adatkockarendszerek segítenek a felhasználónak az adatkocka valamilyen *megvalósított nézettábláit* kiválasztani. A megvalósított nézettábla egy lekérdezés végeredménye, amelyet inkább az adatbázisban tárolunk, mint hogy újra és újra előállítsuk az egészet vagy bizonyos részeit egy lekérdezés megválaszolása közben. A megvalósított nézettáblák rendszerint az egész adatkocka különböző szempontok szerint számított összesítéseit tartalmazzák.

Minél durvább a csoportosításnak megfelelő felosztás, annál kevesebb helyet foglal a megvalósított nézettábla. Másrészről azonban, ha a nézettáblát fel akarjuk használni egy bizonyos lekérdezés megválaszolásához, akkor nem szabad egyetlen egy dimenziót sem durvábban osztani, mint ahogy azt a lekérdezés teszi. Vagyis, hogy maximálisan kihasználhassuk a megvalósított nézettáblákban rejlő lehetőségeket, nagy nézettáblákat veszünk, amelyek elég finoman osztják fel a dimenziókat csoportokra. Hogy milyen nézettáblákat valósítsunk meg, azt még az elemzőktől várt lekérdezések várható típusa is erősen befolyásolja. A következő példán keresztül bemutatjuk, hogy milyen problémákra kell odafigyelni.

11.16. példa: Térjünk vissza a 11.14. példa adatkockájához:

```
Eladások(modell, szín, dátum, forgalmazó, összért, db)
```

Egy lehetséges megvalósított nézettábla a dátumokat hónaponként, a forgalmazókat városonként csoportosítja. Ezt a nézettáblát, amelyet `EladásokN1`-nek nevezünk, a 11.18. ábrán látható lekérdezés hozza létre. Ez nem szigorú SQL-lekérdezés, mivel úgy képzeljük, hogy a dátumokat és a hozzá tartozó csoportosítási egységeket, mint például a hónap, az adatkockarendszer anélkül is értelmezni tudja, hogy az `Eladások` és a 11.11. példa képzetes `Napok` relációját összekapcsolná.

Egy másik lehetséges megvalósított nézettábla a színeket teljesen összevonja, a dátumokat hetenként, a forgalmazókat pedig államonként csoportosítja. Ezt a nézettáblát, az `EladásokN2`-t a 11.19. ábra lekérdezése adja meg. Mindkét nézettábla


```
INSERT INTO EladásokN1
   SELECT modell, szín, hónap, város, SUM(összért) AS összért,
      SUM(db) AS db
   FROM Eladások JOIN Forgalmazók ON forgalmazó = név
   GROUP BY modell, szín, hónap, város;
```

11.18. ábra. *Az* `EladásokN1` *megvalósított nézettábla*

```
INSERT INTO EladásokN2
   SELECT modell, hét, állam,
      SUM(összért) AS összért, SUM(db) AS db
   FROM Eladások JOIN Forgalmazók ON forgalmazó = név
   GROUP BY modell, hét, állam;
```

11.19. ábra. *Az* `EladásokN2` *egy másik megvalósított nézettábla*

használható az olyan lekérdezések megválaszolására, amelyek egyiknél sem bontják finomabb részekre a dimenziókat. Ez a lekérdezés tehát:

```
L1:  SELECT modell, SUM(összért)
FROM Eladások
GROUP BY modell;
```

megválaszolható így:

```
SELECT modell, SUM(összért)
FROM EladásokN1
GROUP BY modell;
```

de akár így is:

```
SELECT modell, SUM(összért)
FROM EladásokN2
GROUP BY modell;
```

Az L_2 lekérdezést

```
L2:  SELECT modell, év, állam, SUM(összért)
     FROM Eladások JOIN Forgalmazók ON forgalmazó = név
     GROUP BY modell, év, állam;
```

azonban csak az `EladásokN1` nézettábla felhasználásával válaszolhatjuk meg, a következő formában:

```
SELECT modell, év, állam, SUM(összért)
FROM EladásokN1
```

```
GROUP BY modell, év, állam;
```

Mellékesen jegyezzük meg, hogy ez a lekérdezés, hasonlóan azokhoz, amelyek valamilyen időegység szerint csoportosítanak, nem szigorú SQL-lekérdezés. Ez azt jelenti, hogy az `EladásokN1` nézettáblának az `állam` nem attribútuma, csak a `város`. Fel kell tennünk, hogy az adatkockarendszer tudja, hogyan lehet városokat államonként összevonni, valószínűleg a forgalmazókat leíró dimenziótáblázat segítségével.

Az L_2 lekérdezéshez nem használható az `EladásokN2` nézettábla. Ugyan a városokat össze tudtuk vonni az államok szerint úgy, hogy az `EladásokN1`-t használni tudjuk, ugyanezt a hetekkel nem tudjuk megtenni, hogy az `EladásokN2`-t is használhassuk, mert az évek nem egyenletesen vannak hetekre osztva. Az 1999. december 29-ével kezdődő hét adatainak egy része például még az 1999-es évhez tartozik, a másik része viszont már 2000-hez. Hogy pontosan hol lehetne a kettőt elválasztani egymástól, azt a hetenként összesített adatokból nem tudjuk megállapítani.

Végül egy olyan lekérdezés, mint:

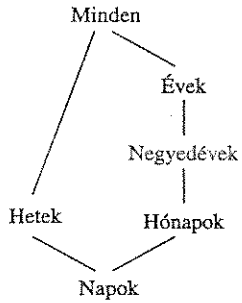
```
L3:  SELECT modell, szín, dátum, SUM(összért)
     FROM Eladások
     GROUP BY modell, szín, dátum;
```

nem válaszolható meg sem az `EladásokN1`, sem az `EladásokN2` segítségével. Nem használhatjuk az `EladásokN1`-t, mert a napok havonkénti összevonása túl durva ahhoz, hogy ez alapján a vásárlásokat napok szerint listázzuk. Nem használható az `EladásokN2` sem, mert ez a nézettábla nem csoportosít színek szerint. Az L_3 lekérdezést közvetlenül az egész adatkocka felhasználásával kellene megválaszolnunk. □

11.4.3. Nézetháló

Hogy a 11.16. példában tett megfigyeléseinket formalizálni tudjuk, hasznunkra válhat egy olyan háló, amely a kocka egyes dimenziói mentén az összes lehetséges csoportosítást tartalmazza. A háló pontjai az adott dimenzióhoz tartozó dimenziótáblázat egy vagy több attribútuma szerinti csoportosítási módot jelölik. Azt mondjuk, hogy a P_1 partíció a P_2 partíció alá esik, ha a P_1 partíció minden egyes csoportját tartalmazza P_2 valamely csoportja. Jelölés: $P_1 \leq P_2$.

11.17. példa: Az idő dimenzió felosztásához választhatnánk a 11.20. ábrán látható hálót. Ha létezik valamilyen P_2 csúcsból P_1-be vezető út, akkor ez azt jelenti, hogy $P_1 \leq P_2$. Az ábrán nem szerepel az összes lehetséges időegység, de mindenesetre jó példáját láthatjuk annak, hogy melyek azok az egységek, amelyeket egy rendszer támogathat. Vegyük észre, hogy a napok a hetek, illetve a hónapok alá esnek, de a hetek nem esnek a hónapok alá. Ennek az az oka, hogy egy adott napon bekövetkezett események biztosan egy megfelelő héten, illetve hónapon belül történtek, de az már nem igaz, hogy egy adott hét folyamán bekövetkezett események összessége szükségszerűen



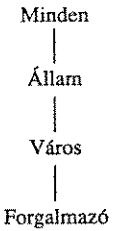
11.20. ábra. *Partíciók hálója az idő intervallumok felosztásához*

egy megfelelő hónapon belül történt volna. Hasonlóan, a hetek eseménycsoportjait nem lehet negyedévek vagy évek eseménycsoportjai alá sorolni. A háló legmagasabb pontjában a „minden" partíció áll. Ez azt jelenti, hogy az események egyetlenegy csoportba vannak beosztva, azaz a különböző időpillanatok között nem teszünk különbséget.

A 11.21. ábrán egy másik háló látható, ezúttal a gépkocsis példánk forgalmazó dimenziójához. Ez egy egyszerűbb háló, amely alapján megtudhatjuk, hogy az autóvásárlások forgalmazó szerinti csoportosítása finomabb, mintha a forgalmazóhoz tartozó város szerint csoportosítottunk volna, a város szerinti felosztás viszont finomabb az állam szerinti felosztásnál. A háló legtetején az a partíció áll, amely az összes forgalmazót egy csoportba osztja be. □

Most, hogy már minden dimenzióhoz meg tudunk adni egy hálót, definiálhatunk egy hálót az adatkocka azon megvalósított nézettábláihoz is, amelyek a dimenziók valamilyen felosztásának megfelelő csoportosítással jönnek létre. Ha N_1 és N_2 a dimenziók valamely partíciója (csoportosítása) alapján megvalósított nézettáblák, akkor $N_1 \leq N_2$ azt jelenti, hogy minden egyes dimenzió N_1-hez használt P_1 partíciója legalább olyan finom, mint ugyanennek a dimenziónak a N_2-höz használt P_2 partíciója, azaz $P_1 \leq P_2$.

A nézettáblák hálójába sok OLAP-lekérdezés is elhelyezhető. Tulajdonképpen az OLAP-lekérdezések gyakran ugyanolyan formájúak, mint az előbb leírt nézettáblák, azaz minden dimenzióhoz megadnak egy-egy felosztást (lehet, hogy a „minden" vagy a „semmi" partíciót). Más OLAP-lekérdezések egy ugyanilyen típusú csoportosítás

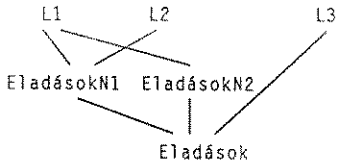


11.21. ábra. *Háló a személygépkocsi-forgalmazók felosztásához*

után még „szeletelik" is a kockát (ahogy a 11.15. ábrán láttuk), hogy az adatoknak csak egy részhalmazára fordítsák a figyelmüket. Az általános szabály a következő:

- Egy L lekérdezés pontosan akkor válaszolható meg a N nézettábla segítségével, ha $N \leq L$.

11.18. példa: A 11.22. ábrán a 11.16. példa nézettábláihoz és lekérdezéseihez adtunk meg egy hálót. Vegyük észre, hogy gyakorlatilag maga az `Eladások` adatkocka is egy nézettábla, ahol minden dimenzió mentén a lehető legfinomabb a felosztás. Ahogy azt az eredeti példában is megfigyeltük, L_1 lekérdezés az `EladásokN1` és `EladásokN2` nézettáblák bármelyikével megválaszolható. Természetesen az egész `Eladások` adatkockát is segítségül hívhattuk volna, de ha a többi nézettáblázat közül legalább az egyik megvalósított (azaz fizikailag jelen van az adatbázisban), akkor nincs okunk az adatkockát választani. L_2 megválaszolható akár az `EladásokN1`, akár az `Eladások` felhasználásával, míg L_3 lekérdezéshez csak az `Eladások` használható. Ezeket a kapcsolatokat a 11.22. ábrán a lekérdezéstől az őket támogató nézettáblákhoz vezető utak fejezik ki. □



11.22. ábra. *Nézettáblák és lekérdezések hálója a 11.16. példához*

Ha a nézettáblák hálójába a lekérdezéseket is beillesztjük, ez segít az adatkocka-adatbázisok tervezésében. Néhány újonnan fejlesztett adatkockarendszer-tervező eszköz az alkalmazás szempontjából „tipikus" lekérdezések halmazából indul ki. Ezután a megvalósítandó nézettáblák halmazát úgy választja ki, hogy minden lekérdezést legalább egy nézettábla támogasson (azaz a hálóban a lekérdezés legalább egy nézettábla fölé essen), de még jobb, ha a lekérdezés valamelyik nézettáblával azonos vagy majdnem azonos (vagyis a legtöbb dimenzió mentén a lekérdezés és a nézettábla ugyanazt a csoportosítást használja).

11.4.4. Feladatok

11.4.1. feladat: Adjuk meg a KOCKA(F) és az F ténytáblázat méretarányát, ha F a következő tulajdonságokkal rendelkezik:

* a) F-nek tíz dimenzió attribútuma van, mindegyik tíz különböző értékkel.
b) F-nek tíz dimenzió attribútuma van, mindegyik két különböző értékkel.

11.4.2. feladat: Ebben a feladatban a 11.14. példa KOCKA(`Eladások`) adatkockát használjuk, amelyet a következő reláció alapján hoztunk létre:


```
Eladások(modell, szín, dátum, forgalmazó, összért, db)
```

Határozzuk meg a kocka azon sorait, amelyeket a következő lekérdezések megválaszolásához használnánk:

* a) Adjuk meg, hogy az egyes forgalmazóknak mennyi bevételük származott a kék autók eladásából!
b) Adjuk meg, hogy „Kedves Kunigunda" a zöld Tüskésekből összesen hányat adott el!
c) Adjuk meg, hogy 2000 márciusában naponta az egyes forgalmazók a különböző színű Tüskésekből átlagosan hány darabot adtak el!

*! **11.4.3. feladat:** A 11.3.1. feladatban egy kockába rendezett adatbázisról volt szó, amelyet számítógép-rendelések tárolására használtunk. Ha alkalmazni akarnánk a KOCKA műveletet, kényelmesebb lenne, ha néhány dimenziót tovább osztanánk. Például az egyetlen processzor dimenzió helyett lehetne egy dimenziónk a processzor típusára (AMD K–6 vagy Pentium–III), egy másik pedig a sebességére. Adjunk meg olyan dimenzió és függő attribútumokat, amelyek sokféle hasznos összesítő lekérdezés végrehajtását támogatják! Mi a szerepe a vásárlónak? A 11.3.1. feladatban az `ár` attribútum egyetlen egy számítógép árát adta meg, holott egyszerre több, ugyanarra a konfigurációra vonatkozó megrendelést is nyilvántarthatnánk ugyanabban a sorban. Melyek lennének a függő attribútumok?

11.4.4. feladat: Határozzuk meg a 11.4.3. feladatban szereplő kocka azon sorait, amelyeket a következő lekérdezések megválaszolásához használnánk:

a) Adjuk meg, hogy 2000-ben havonként összesen mennyit rendeltek a különböző sebességű gépekből!
b) Adjuk meg, hogy összesen hány számítógépet rendeltek, az egyes processzor- és merevlemeztípusok (például SCSI vagy IDE) kombinációira lebontva!
c) Adjuk meg 1999 januárjától kezdve havonként a 400 MHz-es számítógépek átlagárát!

! **11.4.5. feladat:** A 11.4.3. feladatban leírt adatkocka nem tartalmazza a monitorok adatait. Milyen dimenziók segítségével írhatnánk le őket? Feltehetjük, hogy a monitor ára benne van a számítógép árában.

11.4.6. feladat: Tegyük fel, hogy egy kockának 10 dimenziója van, és ezek mindegyikét 5-féle finomsági szinten oszthatjuk fel (beleszámítva a két triviális „minden", illetve „semmi" partíciót is). Hány egymástól eltérő nézettábla nyerhető a dimenziók különböző finomságú felosztásával?

11.4.7. feladat: Mutassuk meg, hogyan illeszthetők be a 11.20. ábrán látható hálóba a következő időegységek: óra, perc, másodperc, két hét, évtized, évszázad!

11.4.8. feladat: Hogyan illesztenénk be a 11.21. ábra forgalmazó hálójába a következő „régiókat", ha:

a) Egy régió az államok egy halmaza.
* b) A régiókat nem lehet államok szerint megadni, de minden várost csak egy régió tartalmaz.
c) A régiók az irányítószámokhoz hasonlóak[16]. Minden régió egy-egy államon belül helyezkedik el, néhány város két vagy több régiónak is része, és néhány régiónak sok város is része lehet.

! **11.4.9. feladat:** A 11.4.3. feladatban egy olyan kockát terveztünk, amelyre alkalmazható a KOCKA művelet. Azonban néhány dimenzióhoz megadható egy nemtriviális hálóstruktúra is. A processzortípust szervezhetnénk például gyártó (SUN, Intel, AMD, Motorola), sorozat (SUN UltraSparc, Intel Pentium vagy Celeron, AMD K-sorozat, Motorola G-sorozat) és modell (Pentium–III, AMD K–6) szerint.

a) A megadott példák alapján tervezzünk hálót a processzortípus dimenzióhoz!
b) Adjuk meg azt a nézettáblát, amely a processzorokat sorozat szerint, a merevlemezeket típus szerint, a CD-meghajtókat sebesség szerint csoportosítja, ezeken kívül pedig minden mást összevon!
c) Adjuk meg azt a nézettáblát, amely a processzorokat gyártó szerint, a merevlemezeket sebesség szerint csoportosítja, ezeken kívül pedig minden mást összevon, kivéve a memóriaméretet!
d) Adjunk példákat olyan lekérdezésekre, amelyekhez

i) csak a b) nézettábla
ii) csak a c) nézettábla
iii) mindkét nézettábla
iv) egyik nézettábla sem

használható!

*!! **11.4.10. feladat:** Amennyiben az F ténytáblázat ritka (azaz sokkal kevesebb sora van, mint a dimenziók lehetséges értékei számának a szorzata), akkor a KOCKA(F) és F táblázatok méretaránya nagyon nagy lehet. Mennyire?

11.5. Adatbányászat

Az adatbázis-alkalmazások egy családja, az *adatbányászat* (data mining) vagy *tudásbányászat* (knowledge discovery in databases) iránt jelentős érdeklődés mutatkozott az utóbbi időben, mert segítségükkel létező adatbázisok alapján meglepő megfigyelések birtokába juthatunk. Az adatbányászó lekérdezésre úgy is gondolhatunk, mint a

[16] Természetesen az Egyesült Államokban használt irányítószámokról van szó. *A fordító megjegyzése.*

döntéstámogató lekérdezések egy kiterjesztett formájára, bár a kettő között formálisan nincs különbség (lásd „Adatbányászati lekérdezések és döntéstámogató lekérdezések" című doboz). Az adatbányászatban a hagyományos adatbázisrendszereknek mind a lekérdezésoptimalizáló, mind az adatkezelő komponense hangsúlyos szerephez jut. Emellett nagyon fontos téma az adatbázisnyelvek kibővítése is. Itt elsősorban olyan nyelvi építőelemekről (nyelvprimitívekről) van szó, amelyek támogatják a hatékony mintavételt. Ebben a részben bemutatjuk, hogy milyen fő irányokat vett az adatbányász alkalmazások fejlődése. Kicsit részletesebben is foglalkozunk a „társítási szabályok" problémakörrel, amely adatbázisos szempontból az utóbbi időben a legtöbb figyelmet kapta.

11.5.1. Adatbányászati alkalmazások

Nagy vonalakban körülírva, az adatbányászó lekérdezések az adatok egy „hasznos" összefoglalását várják eredményül, gyakran a paraméterként a célnak legjobban megfelelő értékre vonatkozó mindenféle útmutató nélkül. Emiatt újra át kell gondolnunk, hogy hogyan nyerhetünk efféle bepillantást az adatok mélyére az adatbázis-kezelő rendszerek segítségével. Ebben a részben néhány olyan alkalmazást és problémát tekintünk át, amelyek nagyon nagy mennyiségű adattal kapcsolatban kerülnek elő. Mivel sok esetben még nyitott kérdés, hogy hogyan használhatók legjobban az ABKR-ek az adott problémával kapcsolatban, a megoldásokat itt nem tárgyaljuk, csupán néhány szóban utalunk arra, hogy miből áll a feladat nehézsége. A 11.5.2. részben viszont egy olyan probléma tárgyalásába kezdünk, amellyel kapcsolatban figyelemre méltó haladás következett be. Az utolsó részben bemutatunk egy nem triviális, adatbázis-orientált megoldást.

Döntési fa építése

Egy adatbázis felhasználói az adatok alapján egy fontos kérdést akarnak eldönteni. Az 5.7. példában egy olyan kérdést tettünk fel, amely akár egy érdekes adatbányászós probléma alapja is lehetne: „Ki vásárol arany ékszereket?". Abban a példában a vásárlóknak csak két tulajdonságával foglalkoztunk: az életkorukkal és a jövedelmükkel. A mai adatbázisok azonban sokkal több információt is felvehetnek a vásárlókról, közvetlenül vagy legitim források adatait integrálva egy tárházba. Ilyen lehet például a vásárló irányítószáma, családi állapota, lakáshelyzete vagy az általa beszerzett árucikkek különböző jellemzői.

Az 5.7. példával ellentétben, ahol az adatbázis csak olyan embereket tart nyilván, akik már vásároltak arany ékszereket, a *döntési fa* (decision tree) egy olyan eszköz, amely segítségével az adatok két részre bonthatók, az „igen-halmazra" illetve a „nem-halmazra". Az arany ékszerek esetében ezek az adatok emberekről nyilvántartott információkat jelentenének. Az igen-halmazba sorolnánk azokat, akikről valószínűnek tartjuk, hogy vásárolnának arany ékszereket, a nem-halmazba pedig azokat,

Adatbányászati lekérdezések és döntéstámogató lekérdezések

A döntéstámogató lekérdezéseknek lehet, hogy az adatbázis nagy részét kell megvizsgálniuk és összesíteniük, cserébe viszont a kérdést feltevő elemző pontosan megadja a végrehajtandó lekérdezést, azaz tudtára adja a rendszernek, hogy az adat mely részeivel foglalkozzon. Az adatbányászati lekérdezés még egy lépést tesz előre. A rendszerre hagyja annak az eldöntését, hogy a lekérdezés szempontjából az adatbázis melyik része lehet fontos. Egy döntéstámogató lekérdezés például „összesítsük a Süni gépkocsik eladását szín és év szerint" az adatbányászat nyelvén így hangzana: „mely tényezők befolyásolták legjobban a Süni gépkocsik eladását?". Az adatbányászati lekérdezések naiv megvalósításai nagyszámú döntéstámogató lekérdezés végrehajtását jelentik, és ezért olyan sok időbe telhet a válaszadás, hogy ez a fajta megközelítés teljesen használhatatlanná válik.

akikről nem tartjuk valószínűnek, hogy vásárolnának arany ékszereket. Ha az előrejelzésünk megbízható, máris rendelkezésünkre áll egy megfelelő célcsoport, amely tagjainak például közvetlen levél útján küldhetünk reklám-összeállítást az aranyékszerkínálatunkról.

Maga a döntési fa valahogy úgy nézne ki, mint az 5.13. ábrán, azzal a különbséggel, hogy a levelek nem tartalmaznának számszerű értékeket. Minden belső csúcshoz tartozik egy attribútum és egy küszöbértékként szolgáló attribútumérték. Egy belső pont közvetlen leszármazottja vagy szintén belső pont vagy levél. A leveleket döntési csúcsnak nevezzük, és „igen" vagy „nem" érték tartozhat hozzájuk. Egy reláció adott sorában található adat kiértékelése úgy történik, hogy a döntési fa legtetejéről indulva minden lépésben a belső csúcsokhoz tartozó attribútumérték alapján hol a bal, hol a jobb oldali él mentén haladunk tovább, egészen addig, amíg egy döntési csúcshoz nem érkezünk.

A döntési fát az adatok egy olyan *mintahalmaza* (training set) alapján építjük, amelyekről tudjuk, hogy az igen-halmazba vagy a nem-halmazba tartoznak. Az arany ékszerek esetén vesszük a vásárlók adatbázisát, amely arról is tartalmaz információkat, hogy az egyes vevők vásároltak-e már arany ékszereket vagy sem. Adatbányászati probléma ezen adatok alapján egy olyan döntési fát létrehozni, amely a legnagyobb biztonsággal dönt egy adott tulajdonságokkal (életkor, jövedelem és így tovább) rendelkező új vásárló esetén annak a valószínűségéről, hogy fog-e arany ékszert venni vagy nem. Azaz meg kell határozni a legjobb A attribútumot és a hozzá tartozó legjobb v küszöbértéket, amit a gyökérre írhatunk, majd hasonlóan a gyökér bal- illetve jobboldali leszármazottjára írandó optimális attribútumot, és a küszöbértéket arra az esetre, ha a besorolandó új vásárlóhoz tartozó A attribútum értéke kisebb, illetve legalább akkora, mint v. Ugyanezt a problémát kell megoldanunk a döntési fa minden szintjén, egészen addig, amíg már nem érdemes újabb pontokat felvennünk a gráfba (mert a mintahalmazból túl kevés adat ér el egy adott csúcspontot, hogy ez alapján

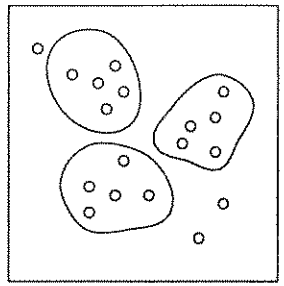
hasznos döntést hozhatnánk). A csúcspontok tervezése során használatos lekérdezések az adatok legnagyobb részét összesítik, így ezek alapján meg tudunk adni egy olyan attribútum-küszöbérték párosítást, amely a mintaadatok „igen" felének nagy részét elválasztja a mintaadatok „nem" felének nagy részétől.

Csoportosítás

Az adatbányászati problémák egy másik osztályába tartozik a „csoportosítás" kérdéskör. A feladat az, hogy az adatokat beosszuk néhány csoportba úgy, hogy az egyes csoportok elemeiben legyen valami lényeges vonás, ami közös. A 11.23. ábrán kétdimenziós adatok csoportosítása látható, a gyakorlatban természetesen a dimenziók száma sokkal nagyobb is lehet. Az ábrán a három legjobb csoport hozzávetőleges körvonalait rajzoltuk be, bár néhány pont mindegyik csoport központjától távol esik. Ezeket tekinthetjük „kívülállóknak", de az egyes pontokat hozzá is csaphatjuk a legközelebb eső csoporthoz.

Egy mintaalkalmazás kedvéért nézzük meg, hogyan működnek az internetes keresőprogramok. Ezek a programok gyakran százezrével találnak olyan weboldalakat, amelyek az adott keresési feltételnek megfelelnek. Hogy ezeket megfelelően lehessen rendszerezni, a keresőprogramok talán a használt szavak alapján csoportosítják a dokumentumokat. Vehetjük például azt a teret, amelyben minden szónak megfelel egy-egy dimenzió, talán a leggyakoribb szavakat, mint „és", „az", „a" *(leállító szavak)* kivéve, amelyek lényegében minden dokumentumban előfordulnak a tartalomtól függetlenül. Minden dokumentum elhelyezhető ebben a térben aszerint, hogy az egyes szavak milyen arányban fordulnak elő a szöveges részében. Ha például egy weboldalon 1000 szó található összesen és ebből kettő az „adatbázis", akkor ezt az „adatbázis" szónak megfelelő dimenzió mentén a 0,002 koordinátához igazítjuk. Ha ebben a térben csoportosítjuk az adott dokumentumokat, akkor a csoportok a dokumentumok témái szerint fognak szerveződni. Például az adatbázisokról szóló szövegekben előfordulhatnak olyan szavak, mint „adat", „lekérdezés", „zárolás" és így tovább, viszont a foci témájú szövegek aligha tartalmaznak ezekhez hasonlókat.

Itt az jelenti az adatbányászati problémát, hogy hogyan határozzuk meg a csoportok



11.23. ábra. *Kétdimenziós adatok három csoportja*

„középértékét" vagy középpontját az adott adatok alapján. Gyakran a csoportok száma előre rögzített, bár ezt az értéket az adatbányászati eljárás is megválaszthatja. Mindkét esetben azonban az a naiv algoritmus, amely úgy választja meg a csoportok középpontját, hogy egy tetszőleges pont átlagos távolsága a hozzá legközelebbi középponttól minimális legyen, számos bonyolult összesítést végrehajtó lekérdezést foglal magában.

11.5.2. Társítási szabály bányászat

Ebben a fejezetben egy olyan adatbányászati problémával ismerkedünk meg, amelyhez eredményesen fejlesztettek ki olyan algoritmusokat, amelyek a háttértárakat használják. A feladat, az úgynevezett „társítási szabályok" keresése, legkönnyebben a legfontosabb alkalmazásán, a *bevásárlókosár-adatok* elemzésén keresztül írható le. A mai áruházak gyakran tárolnak arról feljegyzéseket az adattárházban, hogy egyes vásárlók milyen árucikkeket vásároltak egyszerre. A vevő a teli „bevásárlókosarával" beáll a pénztárhoz, a pénztáros pedig az összes cikket, amit a kosárban talál, egyetlen tranzakció formájában rögzíti. Így aztán, ha magáról a vevőről nem is tudunk semmit, és azt sem tudjuk megmondani, hogy vásárol-e még valaha ebben az üzletben, bizonyos cikkekről *határozottan* tudjuk, hogy ezeket valaki egyszerre vásárolta.

Ha néhány cikk gyakrabban fordul elő együtt a kosarakban, mint az egyébként várható lenne, akkor az áruház ez alapján következtetéseket vonhat le arra vonatkozólag, hogy a vásárlók milyen útvonalon járják be az üzletet. Az árucikkek elrendezhetők úgy, hogy a vevők kénytelenek legyenek bizonyos útvonalakon végigmenni, és ezek mentén vonzó cikkeket lehet elhelyezni.

11.19. példa: Egy híres példa, sokak által megfigyelt jelenség, hogy azok az emberek, akik eldobható pelenkát vásárolnak, nagy valószínűséggel sört is vesznek. Hogy miért áll fenn ez a kapcsolat a két árucikk között, arra több elmélet is született. Többek között azzal is magyarázzák, hogy a pelenkavásárló emberek, mivel kicsi gyerekük van, ritkán járnak kocsmába, ezért otthon isszák a sört. Az áruházak aztán így vagy úgy kihasználhatják azt a tényt, hogy sok vásárló a pelenkáktól a sör felé veszi az útját, vagy fordítva. Az ötletes kereskedő burgonyaszirmot tesz a sör és a pelenka közé. Az állítás az, hogy ilyenkor mindhárom cikk iránt növekszik a kereslet. □

A bevásárlókosár-adatokat tárolhatjuk a következő relációban:

```
Kosarak(kosár, árucikk)
```

Az első attribútum egy „kosárazonosító", amely egyértelműen azonosít minden bevásárlókosarat, a második attribútum pedig a kosárban található cikk azonosítója. Vegyük észre, hogy a reláció szempontjából lényegtelen, hogy az adatok valódi bevásárlókosár-adatok legyenek. Bármilyen olyan adat megfelel, ahol a cikkek közötti kapcsolatokat akarjuk vizsgálni. A „kosarak" lehetnének például dokumentumok, a „cikkek" pedig szavak, ami azt jelenti, hogy valójában olyan szavakat keresünk, amelyek sok dokumentumban együtt fordulnak elő.

A *társítási szabály* legegyszerűbb formája bevásárlókosár-adatok esetén a cikkek egy halmaza. Az $\{i_1, i_2, \ldots, i_n\}$ cikkhalmazok jelentősége változó lehet. Egy halmaz legelemibb tulajdonsága, amelyet vizsgálhatunk, hogy azon kosarak száma, amelyekben a halmaz *minden* eleme szerepel, nagy. Egy cikkhalmaz *tartója* azon kosarak száma, amelyekben a halmaz minden eleme megtalálható. *Erős tartójú cikkhalmazok* felkutatása azt jelenti, hogy adott s küszöbértékhez meg kell találnunk az összes olyan cikkhalmazt, amely tartója legalább s.

Ha az adatbázisba felvett cikkek száma nagy, akkor még abban az esetben is, ha csak kis elemszámú cikkhalmazokkal, mondjuk cikkpárokkal foglalkozunk, az összes kérdéses cikkhalmaz tartóját kiszámítani nagyon sok időt vehet igénybe. Így a kézenfekvő megoldás az erős tartójú cikkhalmazok felkutatására még cikkpárok esetén sem működik. Ilyenkor, mint ahogy erre a 11.24. ábrán látható SQL-lekérdezés utal, minden $\{i, j\}$ árucikkpár tartóját ki kell számítanunk. A lekérdezés összekapcsolja a `Kosarak` relációt saját magával, az így kapott sorokat a bennük található két árucikk szerint csoportosítja, majd a csoportok közül kihagyja azokat, amelyekben a kosarak száma (azaz a tartó) nem éri el az s küszöbértéket. Érdemes végiggondolni, hogy a `WHERE` záradékban rögzített `I.árucikk < J.árucikk` feltétel mire használható. Ennek segítségével előzhető meg, hogy ugyanaz az $\{i, j\}$ pár fordított sorrendben, azaz $\{j, i\}$ alakban is előforduljon, továbbá, hogy az $\{i, i\}$ típusú, vagyis ugyanabból az egy elemből alkotott „párok" egyáltalán létrejöjjenek.

```
SELECT I.árucikk, J.árucikk, COUNT(I.kosár)
FROM Kosarak I, Kosarak J
WHERE I.kosár = J.kosár AND
      I.árucikk < J.árucikk
GROUP BY I.árucikk, J.árucikk
HAVING COUNT(I.kosár) >= s;
```

11.24. ábra. *Naiv megoldás az erős tartójú cikkpárok felkutatására*

11.5.3. Az előzetes algoritmus

A 11.24. ábrán látható lekérdezés futási ideje optimalizálható abban az esetben, ha az s küszöbérték elég magas ahhoz, hogy csak néhány pár feleljen meg a feltételnek. Ésszerű magas s értékkel dolgoznunk, hiszen cikkpárok százaival, ezreivel úgysem mennénk semmire. Az adatbányászati lekérdezéstől azt várjuk, hogy egy kis létszámú, de a lehető legjobb cikkpárokat tartalmazó halmazra hívja fel a figyelmünket. Az *előzetes* algoritmus a következő megfigyelésen alapul:

- Ha egy X cikkhalmaz tartója s, akkor minden $Y \subseteq X$ cikkhalmaz tartója is legalább s.

Például ha az $\{i, j\}$ cikkpár 1000 kosárban szerepel, akkor világos, hogy i is és j is feltűnik ugyanezekben a kosarakban, vagyis létezik legalább 1000 kosár az i cikkhez és 1000 kosár a j cikkhez is.

A fenti szabály megfordításából adódik, hogy ha legalább s tartójú cikkpárokat ke-

A társítási szabály más formái

A társítási szabály egy általánosabb formája a cikkhalmazt egy másik cikkel hozza kapcsolatba. A szabály $\{i_1, i_2, \ldots, i_n\} \Rightarrow j$ alakban írható. Ezzel az alakkal két tulajdonság adható meg:

1. *Bizonyosság:* Annak a valószínűsége, hogy egy $\{i_1, i_2, \ldots, i_n\}$ cikkeket tartalmazó kosárban j árucikk is megtalálható, egy bizonyos küszöbérték, például 50% felett van. Például „a pelenkavásárlók legalább 50%-a sört is vásárol".
2. *Érdekesség:* Annak a valószínűsége, hogy egy $\{i_1, i_2, \ldots, i_n\}$ cikkeket tartalmazó kosárban j árucikk is megtalálható, lényegesen magasabb vagy alacsonyabb annak a valószínűségénél, hogy j egy véletlenül választott kosárban előfordul. Statisztikai szóhasználattal j pozitívan vagy negatívan korrelál az $\{i_1, i_2, \ldots, i_n\}$ halmazzal. A 11.19. példa megfigyelése valójában az volt, hogy a {pelenka} ⇒ sör társítási szabály nagyon érdekes.

Vegyük észre, hogy egy nagy bizonyosságú vagy érdekességű szabály legtöbb esetben csak akkor lesz igazán használható, ha a kérdéses árucikkeknek is erős a tartójuk. Ez azért van így, mert ha a tartó nem erős, akkor a szabály előfordulásainak a száma sem magas, ami korlátozza az adott szabályt hasznosító stratégiával járó előnyöket.

resünk, akkor eleve kizárhatók azok a cikkek, amelyek – más cikkektől függetlenül – nem fordulnak elő legalább s kosárban. Ennek megfelelően az *előzetes algoritmus* a következő lépéseket hajtja végre:

1. Keressük meg a „jó" cikkeket – vagyis azokat, amelyek megtalálhatók elegendő számú kosárban, majd
2. Futtassuk csak a jó cikkeken a 11.24. ábra lekérdezését.

Az előzetes algoritmus az egymás után végrehajtott két SQL-lekérdezés formájában a 11.25. ábrán látható. Először feltölti a `JóKosarak` relációt az elég erős tartójú cikkekkel, azaz a `Kosarak` reláció egy megfelelő részhalmazával, majd a 11.24. ábra naiv algoritmusának mintájára összekapcsolja a `JóKosarak` relációt saját magával.

11.20. példa: Hogy érezzük, mennyit segít az előzetes algoritmus, nézzük meg, hogyan működik egy 10 000 árucikkes élelmiszer-áruház esetén. Tegyük fel, hogy átlagosan 20 cikk van egy-egy kosárban, és hogy az adatbázis 1 000 000 kosár adatait tárolja (a valósághoz képest még ez az érték is kicsiny). Ekkor a `Kosarak` reláció 20 000 000 sorból áll és a naiv algoritmus összekapcsolása után 190 000 000 pár jön létre, ugyanis az 1 000 000 kosárban $\binom{20}{2} = 190$-féle cikkpár szerepelhet összesen. A csoportosítást és a számlálást tehát 190 000 000 soron kell végrehajtani.


```
INSERT INTO JóKosarak
   SELECT *
   FROM Kosarak
   WHERE árucikk IN (
      SELECT árucikk
      FROM Kosarak
      GROUP BY árucikk
      HAVING COUNT(*) >= s
   );

SELECT I.árucikk, J.árucikk, COUNT(I.kosár)
FROM JóKosarak I, JóKosarak J
WHERE I.kosár = J.kosár AND
      I.árucikk < J.árucikk
GROUP BY I.árucikk, J.árucikk
HAVING COUNT(*) >= s;
```

11.25. ábra. *Az előzetes algoritmus az erős tartójú cikkpárok felkutatását az erős tartójú cikkek keresésével kezdi*

Tegyük fel azonban, hogy s = 10 000, azaz a kosarak számának 1%-a. Lehetetlen, hogy több, mint 20 000 000/10 000 = 2000 cikk legalább 10 000 kosárban szerepeljen, hiszen a `Kosarak` relációban csak 20 000 000 sor van, és minden cikk, amely 10 000 kosárban megjelenik, a reláció 10 000 sorában is feltűnik. Vagyis a 11.25. ábra előzetes algoritmusában az erős tartójú cikkeket kereső lekérdezés nem eredményezhet 2000-nél több cikket, de valószínűleg ennél sokkal kevesebbet fog találni.

Nem tudhatjuk előre, hogy a `JóKosarak` reláció mekkora lesz, a legrosszabb esetben ugyanis a `Kosarak` relációban előforduló *összes* árucikk megjelenik a kosarak 1%-ában is. Ha azonban s elég nagy, akkor a `JóKosarak` reláció a gyakorlatban lényegesen kisebb lesz, mint a `Kosarak` reláció. Tegyük fel például, hogy a `JóKosarak` relációban egy-egy kosár átlagosan 10 árucikket tartalmaz, azaz a reláció fele akkora, mint a `Kosarak` reláció. Ekkor a második lépésben az összekapcsolás után $1\,000\,000 \times \binom{10}{2} = 45\,000\,000$ sort kapunk, azaz kevesebb, mint a negyedét annak a sormennyiségnek, amit a `Kosarak` reláció önmagához kapcsolása eredményez. Ezek alapján az várható, hogy az előzetes algoritmus körülbelül negyedannyi időt vesz igénybe, mint a naiv algoritmus. Legtöbb esetben, ahol a `JóKosarak` reláció a `Kosarak` relációnak jóval kevesebb, mint a felét tartalmazza, a futási idő csökkenése még nagyobb mértékű. Általában, ha az összekapcsolásban részt vevő sorok számát n-ed részére csökkentjük, akkor a futási idő n^2-ed részére csökken. □

11.6. Összefoglalás

- *Információk egyesítése:* Gyakran előfordul, hogy többféle adatbázis vagy más információforrás egymással összefüggő adatokat tartalmaz. Megvan a lehetőség arra, hogy ezeket a forrásokat egyesítsük. Gyakran azonban heterogén forrásokkal van dolgunk. Az inkompatibilitás sokféle formában jelentkezhet: ugyanazokhoz az értékekhez különböző típus, kód, illetve konvenció tartozhat, azonos fogalmakhoz egymástól eltérő értelmezések adhatók, az egyes sémákban pedig megvannak a fogalmi különbségek.
- *Az információegyesítés megközelítési módjai:* A korai módszerek közé sorolható a „szövetség" létrehozása, ahol az egyes adatbázisok egymás nyelvén intézhetik egymáshoz a lekérdezéseket. Ennél újabb módszer az adattárház használata, ahol az adatokat egy globális sémának megfelelően alakítjuk át, és a tárházban tároljuk a másolatokat. Ennek egy alternatívája a közvetítő rendszer, ahol egy virtuális adattárháznak tehetők fel a globális sémának megfelelő lekérdezések, amelyeket az egyes adatforrások sémáihoz igazítva alakítunk át.
- *Adatkinyerő és borítékoló:* A tárház, illetve közvetítő rendszerek mindegyike tartalmaz egy, az adatforrásokhoz rendelt alkotóelemet, az adatkinyerőt illetve a borítékolót. Fő feladatuk a lekérdezések és az eredmények átalakítása a globális, illetve a lokális séma szerint.
- *Borítékoló generátor:* A borítékoló tervezésének egy módja a borítékoló sablonok használata, amelyek leírják, hogy egy speciális formájú globális sémának megfelelő lekérdezés hogyan alakítható át a lokális séma nyelvére. A sablonok táblázatba foglalása és értelmezése egy meghajtó feladata, amely majd az adott lekérdezéshez kiválasztja a neki megfelelő sablont, ha van ilyen. A meghajtó képes lehet arra, hogy a sablonokat különféle módon variálja, és/vagy összetettebb lekérdezések esetén további feladatokat is (például az adatok szűrését) elvégezzen.
- *OLAP:* Az adattárházak egy fontos alkalmazását jelenti az a lehetőség, hogy miközben az adatforrásokon a szokásos tranzakciófeldolgozó eljárások működnek, feltehetők a tárház egészét vagy nagy részét érintő bonyolult, általában összesítő jellegű lekérdezések. Ezek a lekérdezések az on-line analitikus adatfeldolgozó vagy OLAP-lekérdezések.
- *ROLAP és MOLAP:* Ha OLAP-alkalmazás céljából építünk adattárházat, gyakran hasznos, ha az adatra egy kocka képében gondolunk, ahol a kocka dimenziói mentén az adatot más és más megvilágításban látjuk. Az olyan rendszer, amely ezt az adatmodellt támogatja, vagy relációs szempontból tekint a kockára (ROLAP- vagy relációs OLAP-rendszerek), vagy az adatkockára specializálódik (MOLAP- vagy többdimenziós OLAP-rendszerek).
- *Csillag séma:* ROLAP-megközelítés esetén minden adatelemet (például egy árucikk eladását) egy relációban, a ténytáblázatban tárolunk, a dimenziók különböző értékeinek az értelmezéséhez (például az 1234 azonosítójú cikk milyen típusú termék?) szükséges információkat pedig az egyes dimenziókhoz tartozó dimenziótáblázatokban. Ezt a típusú adatbázissémát csillag sémának nevezzük. A ténytáblázat a csillag központja, a dimenziótáblázatok pedig a csillag csúcsai.

- *A KOCKA művelet:* MOLAP-megközelítés esetén hasznosnak bizonyul, ha a ténytáblázat dimenzióinak lehetséges részhalmazai mentén végrehajtunk egy előösszesítést. Az így kibővített táblázat kicsit több helyet foglal, mint az eredeti ténytáblázat, de segítségével nagymértékben csökkenthető az OLAP-lekérdezések futási ideje.
- *Dimenzióhálók és megvalósított nézettáblák:* Néhány adatkocka-megvalósítás a KOCKA műveletnél is hatékonyabb eszközt használ: minden dimenzióhoz létrehoz egy-egy hálót, amely az adott dimenzió – összesítésekhez használható – különböző finomsági fokait tartalmazza (például különböző időegységeket, mint nap, hónap, év). Az adattárházba aztán bizonyos megvalósított nézettáblák is bekerülnek, amelyek különféle módokon más és más dimenziók mentén összesítik az adatokat. Egy lekérdezés megválaszolására aztán a vele leginkább megegyező nézettábla használható.
- *Adatbányászat:* A tárházakon olyan tág értelmű lekérdezések is végrehajthatók, amelyek nem csak az előre meghatározott összesítéseket végzik el (miként az OLAP-lekérdezések), hanem keresik a „megfelelő" összesítést. Az adatbányászat gyakran előforduló típusai: az adatok hasonló csoportokba osztása; döntési fa tervezése, amely egy adott attribútum értékét jósolja meg a többi attribútum értéke alapján; sokféle érték között gyakran előforduló párokra vonatkozó társítási szabályok keresése.
- *Előzetes algoritmus:* Az előzetes algoritmus használatával hatékonyan tudunk társítási szabályok után kutatni. Ez a technika azt a tényt használja ki, hogy ha egy halmaz gyakran előfordul, akkor annak minden részhalmaza is gyakori.

11.7. Irodalomjegyzék

Az adattárház-rendszerekről és a kapcsolódó technológiákról ad áttekintést [10], [4] és [8]. Az adatbázis-szövetségekről [12]-ben van szó. A közvetítő fogalma [14]-ből származik. A közvetítő és a borítékoló megvalósításával, különös tekintettel a borítékoló generátor alkalmazására, [6] foglalkozik.

Manapság a legtöbb információegyesítő módszer a „felstrukturált" adatmodellen alapszik, amelynek segítségével megoldható a hiányzó értékek problémája, és a sémák közti egyéb különbségek is áthidalhatók. Az adatmodell ötlete [11]-ből származik; [1] és [13] áttekintést nyújt a témával kapcsolatban.

A KOCKA műveletet [7] vetette fel. A megvalósított nézettáblák segítségével megvalósított adatkockákról [9]-ben esik szó.

Adatbányászati technikák áttekintését adja [5]. Az előzetes algoritmussal [2] és [3] foglalkozik.

1. S. Abiteboul, „Querying semi-structured data," *Proc. Intl. Conf. on Database Theory* (1997), Lecture Notes in Computer Science 1187 (F. Afrati and P. Kolaitis, eds.), Springer-Verlag, Berlin, pp. 1–18.

2. R. Agrawal, T. Imielinski, and A. Swami, „Mining association rules between sets of items in large databases," *Proc. ACM SIGMOD Intl. Conf on Management of Data* (1993), pp. 207–216.
3. R. Agrawal, and R. Srikant, „Fast algorithms for mining association rule," *Proc. Intl. Conf. on Very Large Databases* (1994), pp. 487–499.
4. S. Chaudhuri and U. Dayal, „An overview of data warehousing and OLAP technology," *SIGMOD Record* **26**:1 (1997), pp. 65–74.
5. U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, *Advances in Knowledge Discovery and Data Mining*, AAAI Press, Menlo Park CA, 1996.
6. H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, V. Vassalos, J. D. Ullman, and J. Widom, The TSIMMIS approach to mediation: data models and languages, *J. Intelligent Information Systems* **8**:2 (1997), pp. 117–132.
7. J. N. Gray, A. Bosworth, A. Layman, and H. Pirahesh, „Data cube: a relational aggregation operator generalizing group-by, cross-tab, and subtotals," *Proc. Intl. Conf. on Data Engineering* (1996), pp. 152–159.
8. A. Gupta and I. S. Mumick, *Materialized Views: Techniques, Implementations, and Applications*, MIT Press, Cambridge MA (1999).
9. V. Harinarayan, A. Rajaraman, and J. D. Ullman, „Implementing data cubes efficiently," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1996), pp. 205–216.
10. D. Lomet and J. Widom (eds.), Special issue on materialized views and data warehouses, *IEEE Data Engineering Bulletin* **18**:2 (1995).
11. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom, „Object exchange across heterogeneous information sources." *Proc. Intl. Conf. on Data Engineering* (1995), pp. 251–260.
12. A. P. Sheth and J. A. Larson, „Federated databases for managing distributed, heterogeneous, and autonomous databases," *Computing Surveys* **22**:3 (1990), pp. 183-236.
13. D. Suciu (ed.), Special issue on management of semistructured data, *SIGMOD Record* **26**:4 (1997).
14. G. Wiederhold, „Mediators in the architecture of future information systems," *IEEE Computer* **C-25**:1 (1992), pp. 38–49.

Index