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:

s a s ⊥ ⊥

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), az az 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 `PIROS`at ábrázolhatjuk a 0-val, a `ZÖLD`et 1-gyel, a `KÉK`et 2-vel, a `SÁRGÁ`t 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.

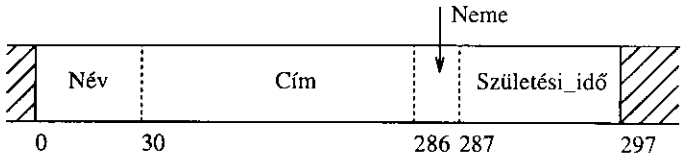
3.2.1. Rögzített hosszú rekordok építése

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.

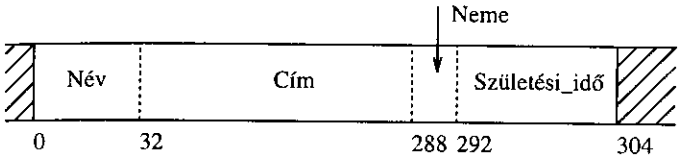
Igaz, hogy egy reláció sorait a lemezen tároljuk és nem a központi memóriában, de azért jó, ha tudunk erről. Ugyanis mikor egy blokkot a lemezről beolvasunk a központi memóriába, akkor a blokk első bájtja a központi memóriában biztosan egy olyan címen lesz, mely 4-nek többszöröse. Általában nem csak 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).

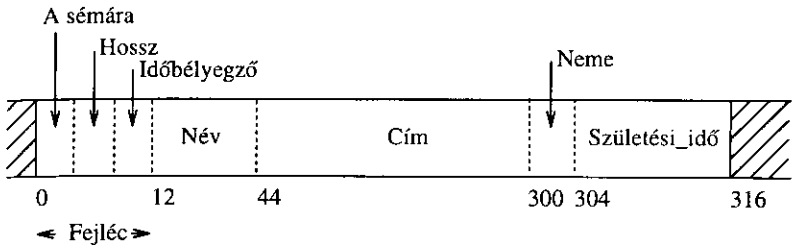
Az adatbázisrendszer tartja karban a *sémainformációt* (schema information). A sémainformáció lényegében az, amit a `CREATE TABLE` utasításban egy relációhoz megadunk:

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.

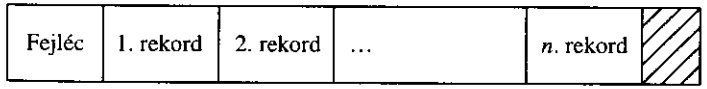
3.7. példa: Módosítsuk a 3.6. példában megadott rekordfelosztást úgy, hogy egy 12 bájtos fejlécet is tartalmazzon a rekord. Az első négy bájt a típus. Ez tulajdonképpen egy eltolási érték egy olyan területen, ahol az összes reláció sémáját tároljuk. A második a rekord hossza, egy 4 bájtos egész szám, 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.

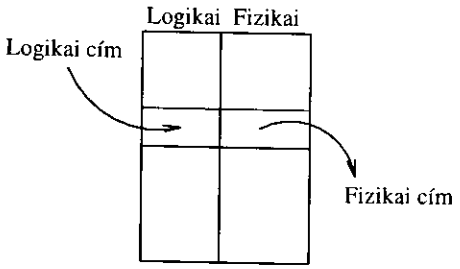
A szerver adatai az *adatbázis címterületén* (database address space) léteznek. Ennek a területnek a címei blokkokra vagy esetleg blokkon belüli eltolási értékekre vonatkoznak. Több mód is kínálkozik arra, ahogy ennek a címterületnek a címeit ábrázoljuk:

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

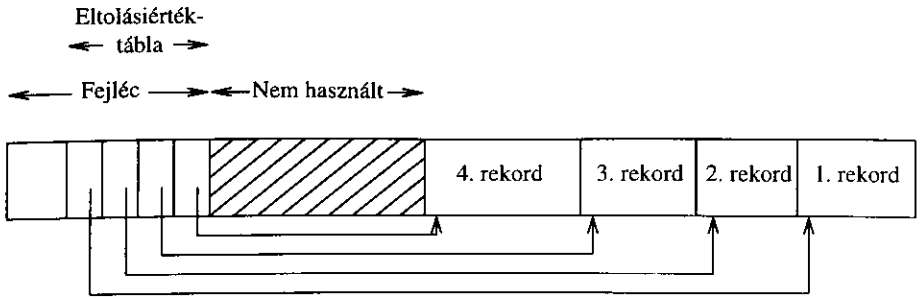
Most már biztos kíváncsiak vagyunk, hogy mire is kellhetnek a logikai címek. Minden olyan információ, ami szükséges egy fizikai címhez, a leképezési táblában található. Így ahhoz, hogy a logikai mutatókat követve eljussunk a rekordokhoz, először meg kell vizsgálnunk a leképezési táblát, és az innen kiolvasott 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

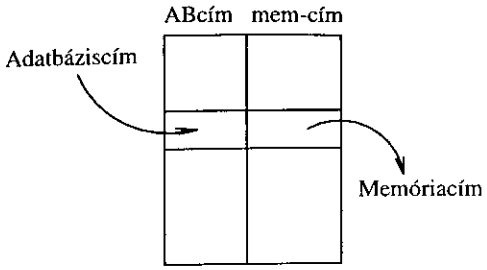
A mutatók és címek gyakran a rekordoknak részét képezik. Ez a helyzet általában nem azokra a rekordokra jellemző, melyek egy reláció sorait ábrázolják, hanem azokra a sorokra, amelyek objektumokat ábrázolnak. A modern objektumrelációs adatbázisrendszerekben megengedett a mutató típusú attribútumok használata 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:

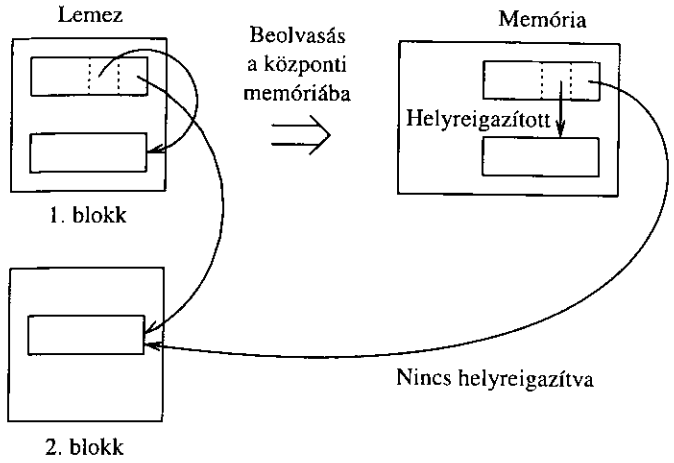
a) A logikai és a fizikai címek egyaránt az adatbáziscím reprezentációi. Ezzel szemben a fordítási táblában található memóriacímek a memóriába másolt megfelelő objektumra vonatkoznak.
b) Az adatbázis minden címezhető adattételéhez tartozik bejegyzés a leképezési táblában, míg a fordítási táblában csak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

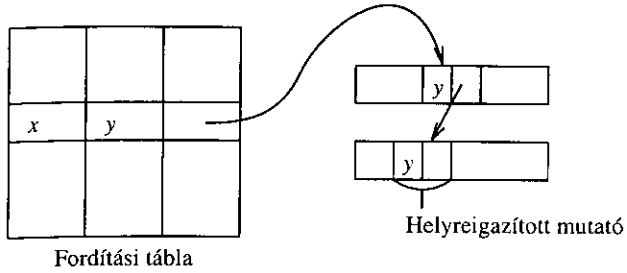
3.3.5. Feltűzött rekordok és blokkok

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:

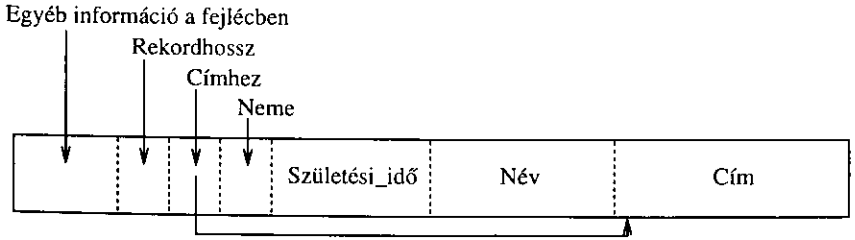
1. *Változó méretű adattételek.* Például a 3.1. ábrán láttunk egy `Filmszínész` relációt, amelynek van egy cím mezője, és ennek a hossza 255 bájtig bármekkora lehet. Noha lehetnek bizonyos címek ilyen hosszúak, de azért a címek nagy többsége valószínűleg 50 bájtos lesz vagy még rövidebb. Valószínűleg 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.



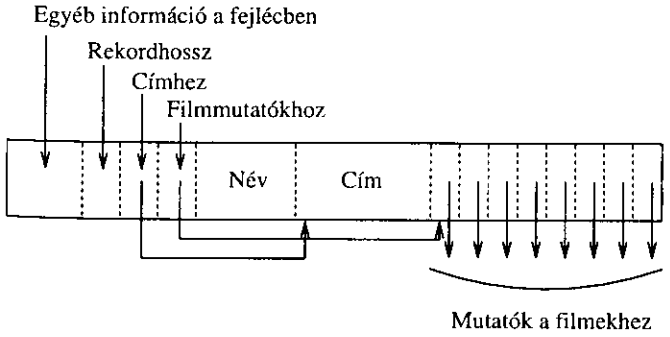
3.12. ábra. *Egy* `Filmszínész` *rekord, melyhez tartozó* `név` *és* `cím` *mezőket változó hosszú karakterláncokkal valósítjuk meg*

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

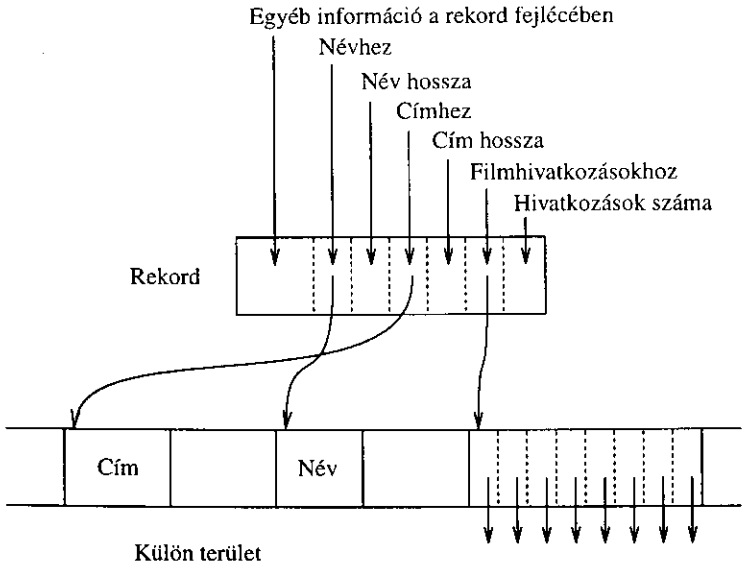
A soroknak gyakran lehetnek olyan mezőik, amelyek `nullértéket (NULL)` is tartalmazhatnak. A 3.12. ábra rekordformátuma egy kényelmes módot nyújt ahhoz, hogyan ábrázoljuk a `nullértékeket`. Ha egy mező, például a `cím` nullértékű, akkor nullmutatót (null pointer) teszünk arra a helyre, ahol a címre hivatkozó 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

nullértékünk lenne, és így jelentős helyet takaríthatunk meg a címkézéssel, és azzal, hogy csak azokat a mezőket listázzuk, amelyek értéke nem nullérték.

2. *Nagyon változékony sémával rendelkező rekordok.* Ha egy rekordban sok mező ismétlődhet és/vagy sok mező egyáltalán nem szerepel, akkor is hasznos lehet a címkézett mezők használata, még akkor is, ha ismerjük a sémát. Például a kórházi kartonok nagyon sok vizsgálatról tartalmazhatnak információt, akár több ezer vizsgálat is lehet, de egy betegnek viszonylag kevés vizsgálati eredménye van.

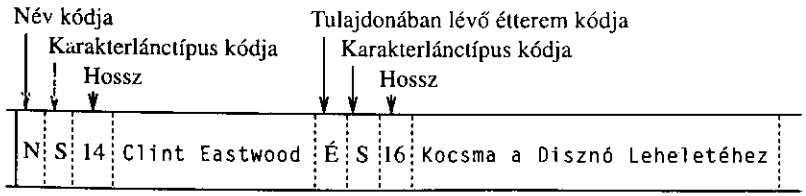
3.12. példa: Tegyük fel, hogy egyes filmszínészekről olyan információk is rendelkezésre állnak, hogy milyen filmeket rendezett az illető színész, kik voltak a korábbi házastársai, milyen éttermek tartoznak a tulajdonába, és még számos más rögzített, de nem szokásos információ is lehet. A 3.15. ábrán láthatjuk egy ilyen elképzelt filmszínész rekordnak az elejét, melyben címkézett mezőket használunk. Feltesszük, hogy a különböző lehetséges mezőneveket és -típusokat egybájtos kódokkal tudjuk ábrázolni. Két mezőhöz (történetesen mindkettő karakteres típusú) tartozó megfelelő kódok és a hosszak az ábrán láthatók. □

3.4.4. Olyan rekordok, amelyek nem férnek el egy blokkban

Most egy másik problémát célzunk meg, amelynek fontossága napjainkban egyre nő, mivel az adatbázis-kezelő rendszerek egyre gyakrabban szoktak kezelni olyan adattípusokat, amelyekhez olyan nagy értékek tartozhatnak, melyek sokszor nem férnek el egy blokkban. Tipikus példák erre a video- és audioklipek.

Ezeknek a nagy értékeknek gyakran változó a hossza, de még ha rögzített is lenne minden ilyen típusú érték hossza, akkor is speciális technikákat kellene használnunk az ilyen értékek ábrázolásához. Ebben a részben meg fogunk nézni egy technikát, amelyet „átnyúló rekordoknak" (spanned records) hívunk, és amely a blokkoknál nagyobb rekordok kezelésére használható. A különlegesen nagy értékek (megabájt vagy gigabájt méretűek) kezelésével a 3.4.5. részben foglalkozunk majd.

Az átnyúló rekordok az olyan esetekben is hasznosak lehetnek, mikor a rekordok ugyan kisebbek, mint a blokkok, de ha teljes rekordokat tennénk a blokkokba, akkor jelentős mennyiségű helyet tékozolnánk el. Például a 3.8. példában az elpazarolt hely csak 7%, de ha a rekordok csak kicsivel lennének nagyobbak, mint a blokk méretének a fele, akkor a veszteség megközelítheti az 50%-ot. Ennek az az oka, hogy csak egy rekordot tudunk egy blokkba tenni.



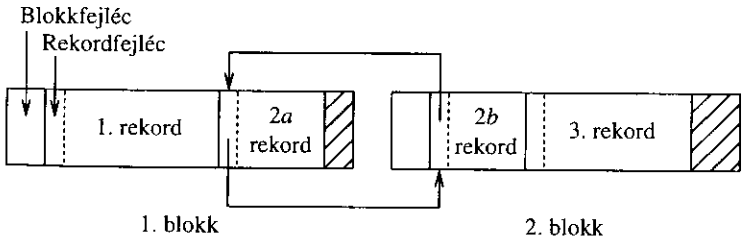
3.15. ábra. *Egy címkézett mezőket tartalmazó rekord*

A fenti okokból kifolyólag néha érdemes megengedni, hogy a rekordokat két vagy több blokkba eső részekre vágjuk szét. Egy rekordnak azt a részét, amely egy blokkba kerül, *rekordtöredéknek* (record fragment) hívjuk. Egy olyan rekordot pedig, amelynek két vagy több töredéke van, *átnyúló* (spanned) rekordnak nevezünk, azokat a rekordokat pedig, amelyek nem nyúlnak túl egy blokk határán, *át nem nyúló* (unspanned) rekordoknak hívunk.

Ha a rekordok átnyúlók is lehetnek, akkor több információt kell tárolnunk minden rekord és rekordtöredék fejlécében:

1. Minden rekordnak vagy töredéknek a fejlécében kell lennie egy olyan bitnek, amely megmondja, hogy ez egy töredék vagy nem.
2. Ha ez egy töredék, akkor ahhoz is kellenek bitek, hogy megmondják, hogy ez a rekord első vagy utolsó töredéke.
3. Ha ugyanannak a rekordnak következő és/vagy előző töredéke is létezik, akkor a töredékben szükséges mutatókat elhelyezni ezekre a további töredékekre is.

3.13. példa: A 3.16. ábrán látható, hogy hogyan lehet bármely két blokkban összesen három olyan rekordot tárolni, melyek mérete egyenként körülbelül a blokk méretének 60%-a. A 2*a* rekordtöredék fejléce tartalmaz egy jelzést, amely azt mutatja, hogy ez egy töredék, egy másik jelzést, hogy ez az első töredéke a rekordjának, és tartalmaz még egy mutatót a következő, vagyis a 2*b* töredékre. Hasonlóan a 2*b* fejléce jelzi, hogy ez az utolsó töredéke a rekordnak, és tartalmaz még egy visszamutató hivatkozást az előző, vagyis a 2*a* töredékre. □



3.16. ábra. *Blokkokon átnyúló rekordok tárolása*

3.4.5. Bináris, nagy objektumok (BLOB-ok)

Most nézzük meg, hogyan kell ábrázolni az igazán nagy értékeket, ha ezek rekordokhoz vagy rekordok mezőihez tartoznak. A szokásos példák erre a különböző formátumú képfájlok (például GIF vagy JPEG), filmek MPEG vagy hasonló formátumban, és mindenféle jelek (hangok, radarjelek stb.). Az ilyen értékeket gyakran *bináris, nagy objektumoknak* (binary, large object – *BLOB*) nevezik. Mikor egy mezőnek az értéke BLOB, akkor legalább két dolgot újra végig kell gondolnunk.

A BLOB-ok tárolása

Egy BLOB-ot blokkok sorozatán kell tárolni. Általában előnyben részesítjük, ha ezek a blokkok a lemez egy vagy több cilinderén egymás után következnek, azért hogy a BLOB-ot hatékonyan lehessen visszanyerni. Ennek ellenére az is lehetséges, hogy a BLOB-ot blokkok összeláncolt listáján tároljuk.

Ezenfelül az is előfordulhat, a BLOB-ot olyan gyorsan kellene visszanyerni (például egy film lejátszásának valós idejűnek kell lennie), hogy ha egy lemezen tárolnánk, akkor ez nem tenné lehetővé a megfelelő gyors beolvasást. Ekkor arra van szükség, hogy a BLOB-ot *részekre (csíkokra) szedjük* (stripe), és ezeket a részeket több lemezen helyezzük el, vagyis a BLOB blokkjai felváltva következnek ezeken a lemezeken. Ezzel tehát lehetővé tettük, hogy a BLOB-nak egyszerre több blokkját tudjuk visszanyerni. A visszanyerés sebességét ezzel a módszerrel körülbelül akkora tényezővel növeltük meg, ahány lemez vett részt a csíkozásban.

A BLOB-ok visszahozása

Az a feltevésünk, hogy mikor a kliensnek kell egy rekord, akkor az adatbázisszerver átadja a kliensnek azt a blokkot, amely ezt a rekordot tartalmazza, lehet, hogy nem teljesen igaz. Lehet, hogy csak a rekord „kis" mezőit akarjuk átadni, és emellett megengedjük, hogy a kliens egyesével igényelje a BLOB blokkjait, függetlenül a rekord többi részétől. Például ha a BLOB egy kétórás film, és a kliens kéri a film lejátszását, akkor a filmet át lehetne úgy küldeni a kliensnek, hogy egyszerre több blokkot is küldenénk, pont azzal a sebességgel, ami a film lejátszásához szükséges.

Sok alkalmazásban az is fontos, hogy a kliens egy BLOB belsejéből igényelhessen egy részt anélkül, hogy meg kellene kapnia a teljes BLOB-ot. Például legyen az az igény, hogy szeretnénk látni egy film 45. percét vagy egy hanganyagnak a végét. Ha az adatbázis-kezelő rendszer támogatja az ilyen műveleteket, akkor ehhez egy megfelelő indexstruktúra kell, például egy BLOB típusú film másodperceire épülő indexre van szükség.

3.4.6. Feladatok

* **3.4.1. feladat:** Egy beteg rekordja a következő rögzített (egyaránt 10 bájt) hosszú mezőket tartalmazza: a beteg születési dátuma, a TAJ-száma és egy betegazonosító. A rekordnak vannak változó hosszú mezői is: név, cím és a kórtörténet. Tegyük fel, hogy egy rekordon belüli mutató 4 bájt, és a rekord hossza egy 4 bájtos egész szám. A változó hosszú mezőkhöz szükséges helyet nem számítva, hány bájt szükséges a rekordhoz? Feltehetjük, hogy nem kell a mezőket igazítani (azaz nem szükséges, hogy például néggyel osztható helyen kezdődjenek).

* **3.4.2. feladat:** Tegyük fel, hogy ugyanolyanok a rekordjaink, mint a 3.4.1. feladatban, és a név, cím, kórtörténet változó hosszú mezők mindegyikének a hossza egyenletes

eloszlású a következő intervallumokon. A név hossza 10 és 50 bájt között változhat, a cím hossza 20 és 80 bájt között, a kórtörténet pedig 0 és 1000 bájt között. Mekkora egy beteg rekordjának átlagos hossza?

3.4.3. feladat: Tegyük fel, hogy a 3.4.1. feladat betegrekordjait kibővítjük egy ismétlődő mezővel, amely a koleszterinvizsgálatokat tartalmazza. Minden koleszterinvizsgálati eredmény egy dátumból és a teszteredményt meghatározó egész számból áll, melyekhez összesen 16 bájt szükséges. Mutassuk meg, hogy néz ki egy beteg rekordjának felosztása, ha

a) az ismétlődő vizsgálatokat magában a rekordban tároljuk,
b) a vizsgálatokat egy külön blokkban tároljuk, és a vizsgálati eredményekre a rekordban mutatók mutatnak.

3.4.4. feladat: Induljunk ki a 3.4.1. feladatban szereplő, beteghez tartozó rekordból. Tegyük fel, hogy különféle teszteket és teszteredményeket tartalmazó mezőkkel bővítjük a rekordot. Minden teszt egy tesztnévből, egy dátumból és egy teszteredményből áll. Tegyük fel, hogy ezek együttesen 40 bájt hosszúak. Tegyük fel továbbá, hogy bármelyik beteg és bármelyik teszt esetében p valószínűséggel tároljuk a betegnek egy ilyen teszteredményét.

a) Tegyük fel, hogy a mutatók és egész számok mindegyike 4 bájtos. Hány bájt szükséges átlagosan a teszteredményekhez egy beteg rekordjában, ha feltesszük, hogy minden teszteredményt magában a rekordban tárolunk, méghozzá egy változó hosszú mezőben?
b) Ismételjük meg a feladat a) részét, de most azt tegyük fel, hogy a teszteredményeket a rekordban mutatók ábrázolják, melyek a valahol máshol tárolt teszteredmény mezőkre mutatnak.
! c) Tegyük fel, hogy kevert sémát használunk, amelyben k számú teszteredményt a rekordban tárolunk, és a további teszteredményeket úgy tudjuk megtalálni, hogy követünk egy másik blokkhoz (vagy blokkláncholz) vezető mutatót, ahol ezeket az eredményeket tároljuk. Határozzuk meg a p függvényében, hogy milyen k értékre lesz minimális a teszteredményekhez szükséges tárolási hely.
!! d) Az ismétlődő teszteredmény mezőhöz szükséges tárterület fontos szempont, de nem ez az egyetlen. Tegyük fel, hogy a felhasznált bájtok számához büntetésből hozzáadunk 10 000-et, ha egy másik blokkot is használnunk kell egyes teszteredmények tárolásához (hiszen ilyenkor egy lemez I/O-művelet is kell majd sokszor a teszteredmények eléréséhez). Ilyen feltételezések mellett p függvényében mi lesz a k legjobb értéke?

*!! **3.4.5. feladat:** Tegyük fel, hogy a blokkokban 1000 bájt használható rekordok tárolására, és rögzített hosszú rekordokat akarunk tárolni a blokkokban. A rekordok hossza legyen r, ahol $500 < r \leq 1000$. Az r értékbe beszámítottuk a rekord fejlécét is, de egy rekordtöredék esetében további 16 bájt szükséges a töredék fejlécéhez. Milyen r értékek esetében lehet javítani a helykihasználáson az átnyúló rekordok módszerével?

! **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?

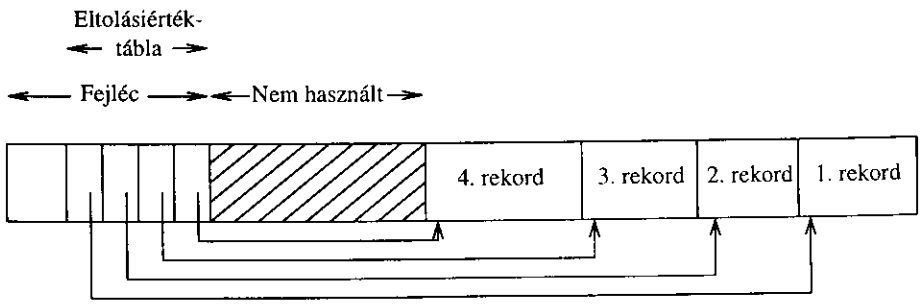
3.5. Rekordmódosítások

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*

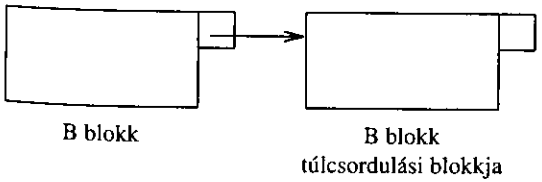
ahogy ezt majd a 4.1. részben látni fogjuk. Ha be kell szúrnunk egy új rekordot, akkor először meg kell keresnünk a rekordnak megfelelő blokkot. Ha véletlenül van még hely ebben a blokkban, akkor betesszük ide a rekordot. Mivel a blokkokat rendezve tároljuk, ezért lehet, hogy a blokkon belül a rekordokat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

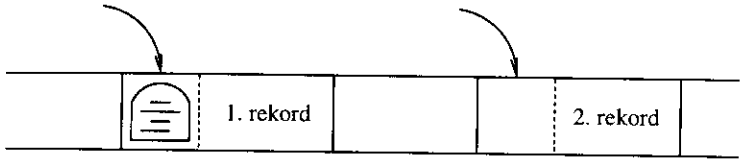
Van azonban még egy probléma a törléssel kapcsolatban, amire mindig figyelnünk kell, függetlenül attól, hogy milyen sémát használunk a blokkok szervezéséhez. Létezhetnek olyan mutatók, amelyek a törölt rekordra mutatnak. Ha vannak ilyen mutatók, akkor nem szeretnénk, hogy ezek a mutatók a törlés után 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 eltolásiértéktáblájában, 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 osztálykiterjedéseket. 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:

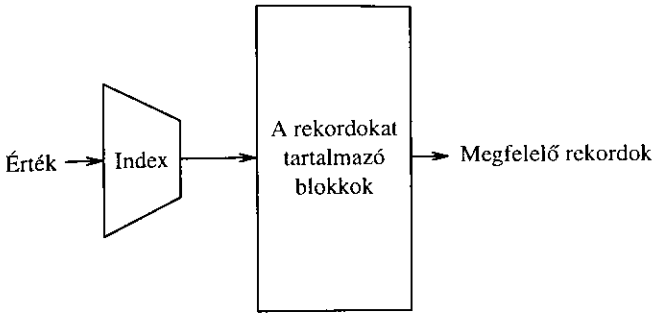
1. A blokkok fejléceiben tárolt információ arról, hogy hol kezdődnek az adott blokkban a rekordok.
2. A rekordok fejléceiben tárolt információ arról, hogy az adott rekord melyik relációhoz tartozik.

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.

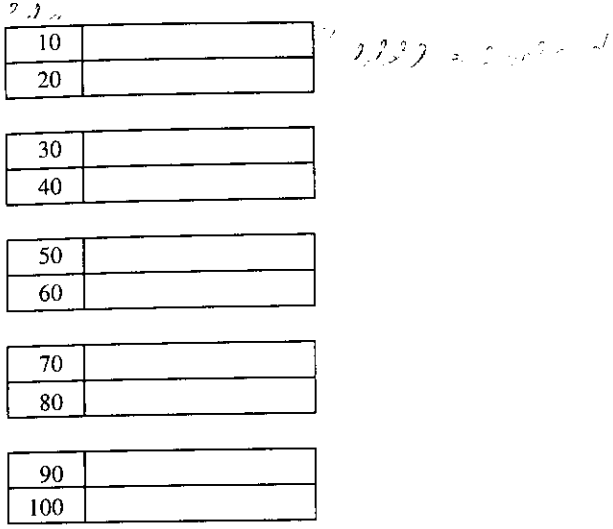
4.1.2. Sűrű indexek

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.

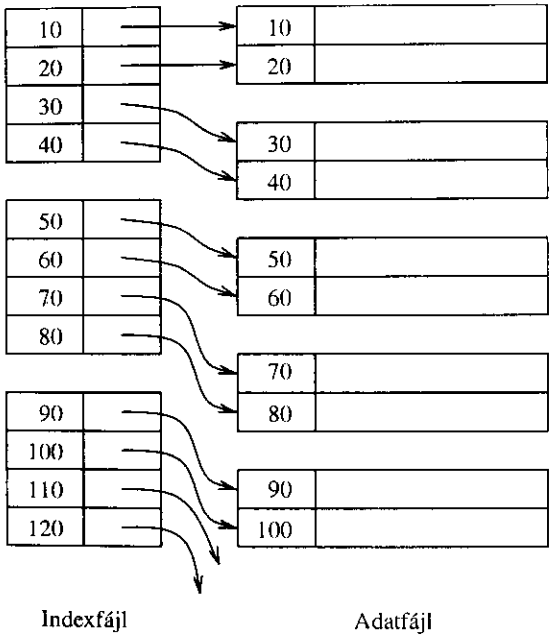
Kulcsok és még mindig kulcsok

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, ami-

kor 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

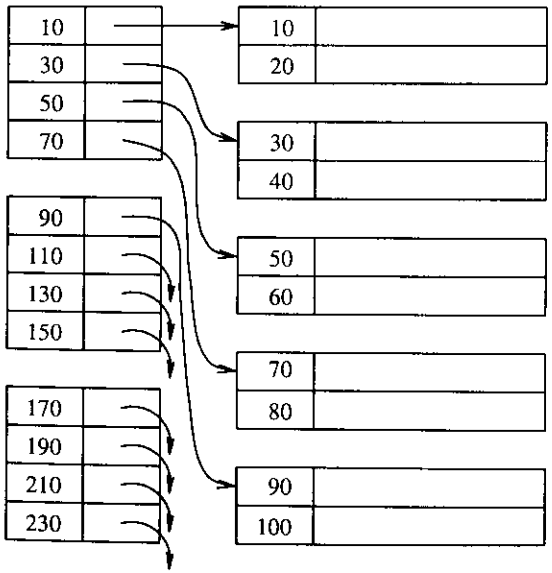
Indexblokkok helyének meghatározása

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 befémek 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.

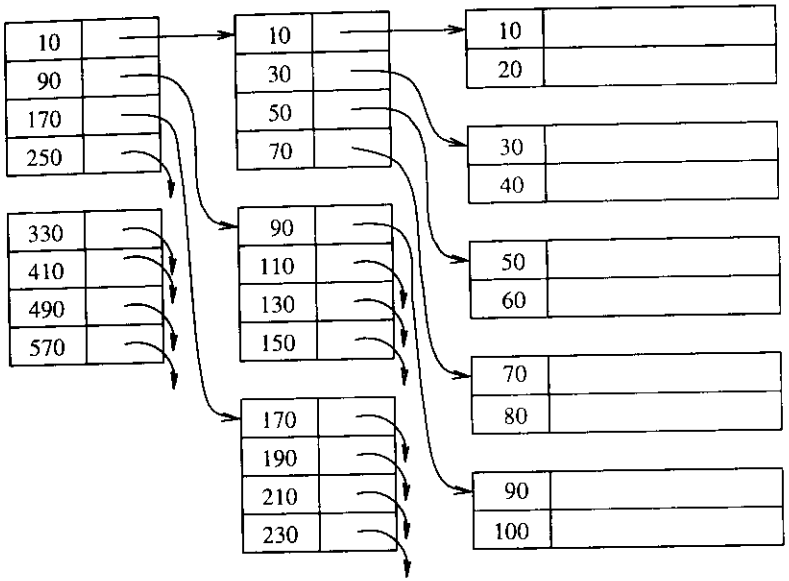
Másrészről azonban, egy sűrű index lehetővé teszi olyan típusú lekérdezések megválaszolását, mint „létezik-e K kulcsértékkel rendelkező rekord?" anélkül, hogy a rekordot tartalmazó blokkhoz hozzá kellene férni. Az a tény, hogy a K a sűrű indexben előfordul, garantálja a K kulcsértékű rekord létezését. 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

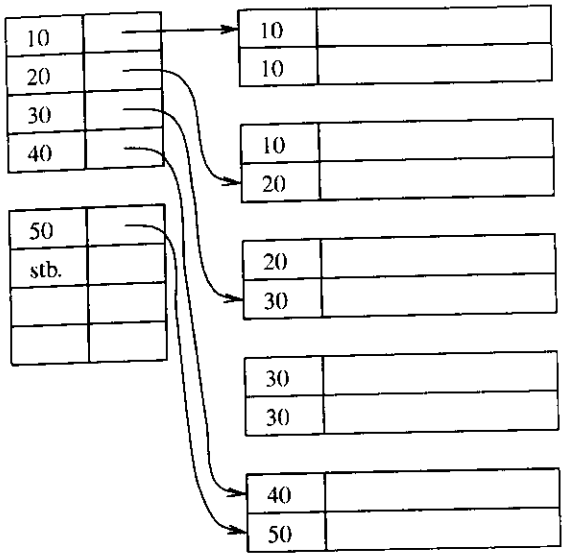
Mindeddig feltételeztük, hogy a keresési kulcs, amely az index alapját képezi, a reláció egy kulcsa, tehát adott kulcsértékhez legfeljebb egy rekord tartozhat. Gyakran használunk azonban indexeket nem kulcs attribútumokra is, így előfordulhat, hogy egy adott kulcsértékhez több mint egy rekord tartozik. 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.

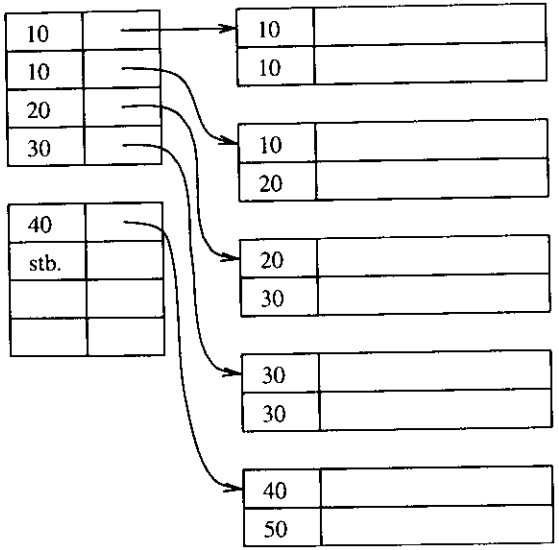
Némileg hatékonyabb az a megközelítés, amikor a sűrű indexben csak egy bejegyzés található valamennyi K keresési kulcshoz. A K kulcshoz olyan mutató tartozik, amely az első K értékkel rendelkező rekordra mutat. Ahhoz, hogy a többi K értékkel rendelkező rekordot is megtaláljuk, csupán el kell mozdulnunk 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*

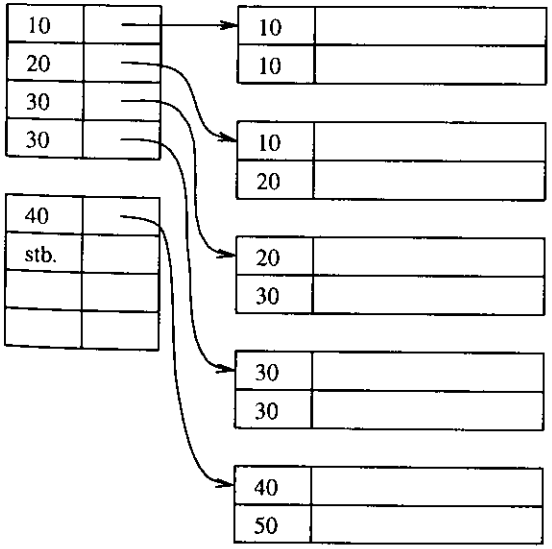
A 4.7. ábrán egy ritka indexet láthatunk, amely a 4.6. ábrán látható adatfájlra épült. A ritka index meglehetősen hagyományos; kulcs-mutató párokat tartalmaz, az adatfájl valamennyi blokkjának első keresési kulcsainak megfelelően.

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 $\leq$ 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.

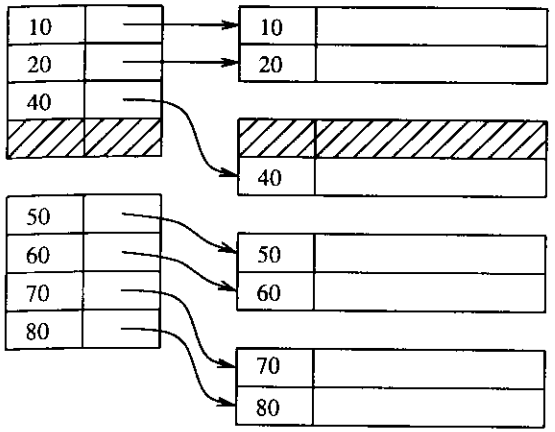
- Rekordok beszúrása és törlése éppolyan hatással van a sűrű indexre, mintha az adott rekordhoz egy kulcs-mutató párt szúrnánk be vagy törölnénk. Nincs azonban általában hatással a ritka indexre. Kivétel persze, ha az adott rekord éppen egy blokk első helyén áll, ilyenkor a ritka index megfelelő kulcsértékét 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.

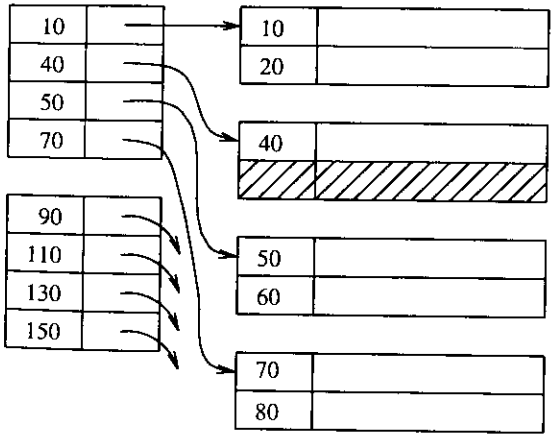
Először is töröljük a 30-as kulcsértéket tartalmazó rekordot a szekvenciális fájlból. Feltételezzük, hogy a blokk rekordjaira blokkon kívüli mutatók mutatnak, ily módon arra a döntésre jutottunk, hogy a blokk megmaradó rekordját nem csúsztatjuk előrébb a blokkban. Ehelyett inkább egy törlésre utaló jelet, 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.

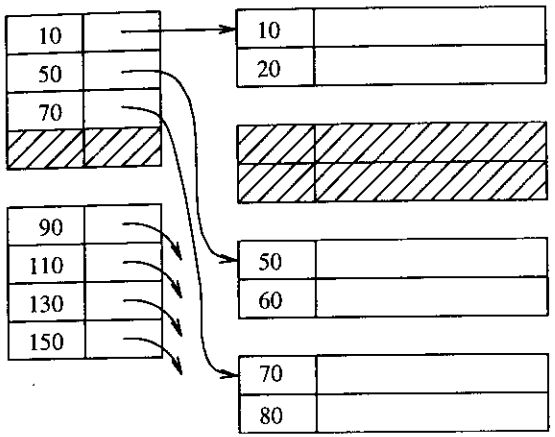


4.11. ábra. *A 30-as keresésikulcs-értékű rekord törlése ritka index esetén*

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

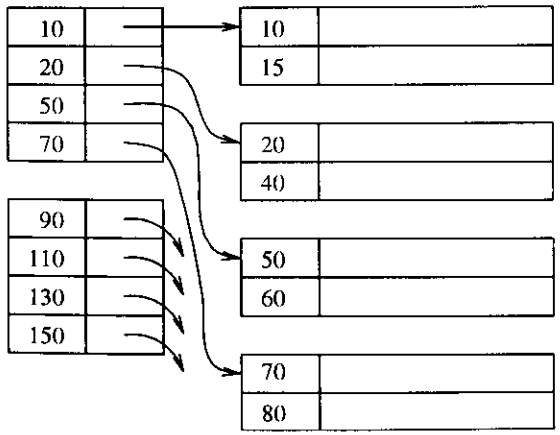
műveletnek az eredményét a 4.12. ábrán láthatjuk. A második adatblokkban nincs már egy rekord sem. Ha a szekvenciális fájl tetszőleges blokkokon van tárolva (és mondjuk nem egy adott cilinder egymás utáni blokkjain), akkor felfűzhetjük a nem használt blokkot a szabad helyek listájára.



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*

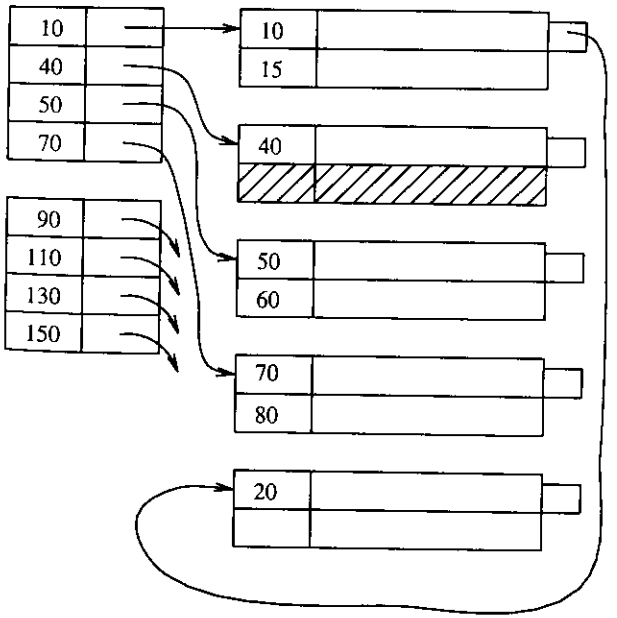
megmaradt. Most be fogunk szúrni egy 15-ös kulcsértékű rekordot. A ritka indexet megvizsgálva arra jutunk, hogy ez a rekord az első adatblokkhoz tartozik. Ez a blokk viszont tele van: a 10-es és 20-as rekordok vannak benne.

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 keresésikulcs-értékhez csak egy kulcs-mutató pár tartozik, amelynek mutatója az első olyan rekordra mutat, amely az adott kulcsot tartalmazza. Számítsuk ki egy adott K keresésikulcs-értékkel 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);
```

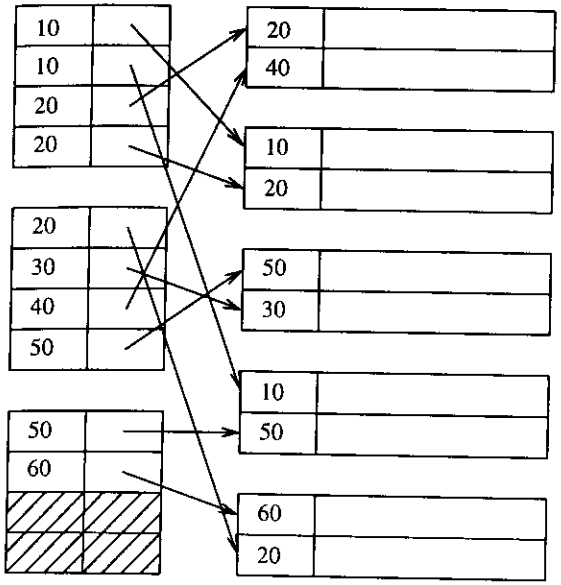
Egy másodlagos index megfelel bármely index céljainak: olyan adatszerkezet, amely megkönnyíti megadott értékű mezővel, illetve mezőkkel rendelkező rekordok megtalálását. A másodlagos index abban különbözik azonban az elsődleges indextől, hogy a másodlagos index nem határozza meg a rekordok adatfájlban 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.

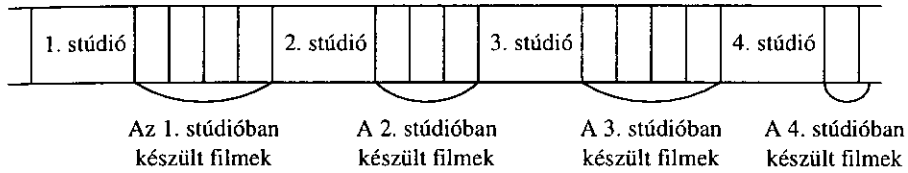
A `Film` reláció sorainak puszta rendezése egy, az elsődleges kulcstól különböző attribútum szerint, azonban nem segít abban az esetben, ha össze akarjuk kapcsolni a filmekről tárolt információkat a stúdiókról tárolt információkkal. Jó példa erre a következő lekérdezés, amelyben azt szeretnénk megtudni, 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*

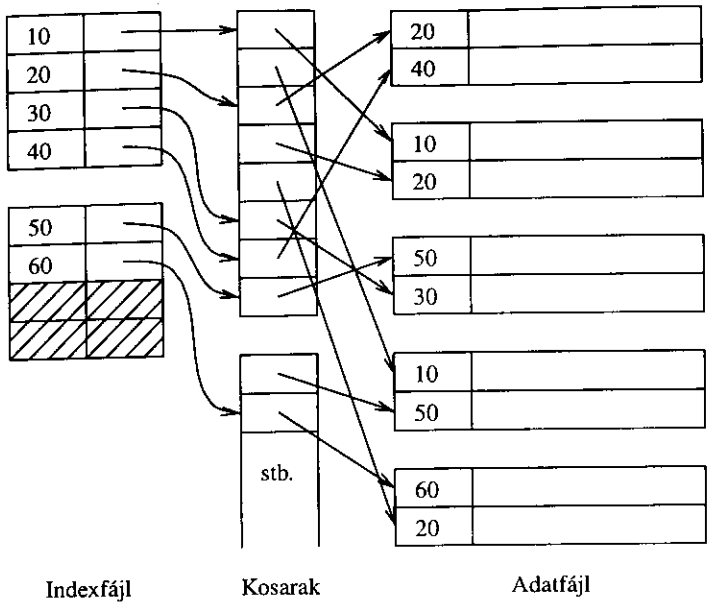
Ebben az esetben, ha annak a stúdiónak az elnökét keressük, amely egy adott filmet készített, jó esélyünk van arra, hogy a stúdióra és a filmre vonatkozó rekordok ugyanabban a blokkban találhatók, megtakarítva ezzel egy lemez I/O-műveletet. Ha azokat a filmeket keressük, amelyeket egy adott stúdió gyártott, 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. □

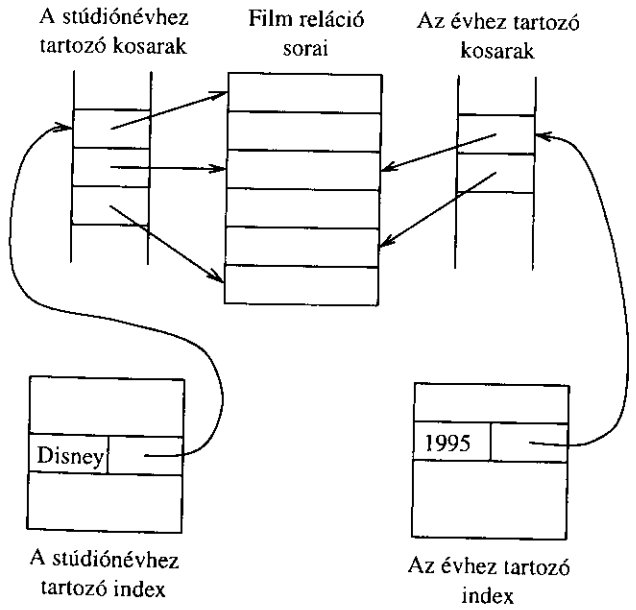
A 4.17. ábrán látható elrendezés mindaddig helymegtakarítást jelent, amíg a keresésikulcs-értékek több helyet foglalnak, mint a mutatók, és mindegyik kulcs átlagosan legalább kétszer megjelenik. A közvetett másodlagos indexek használatának akkor is van azonban egy fontos előnye, amikor a kulcsok és a 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

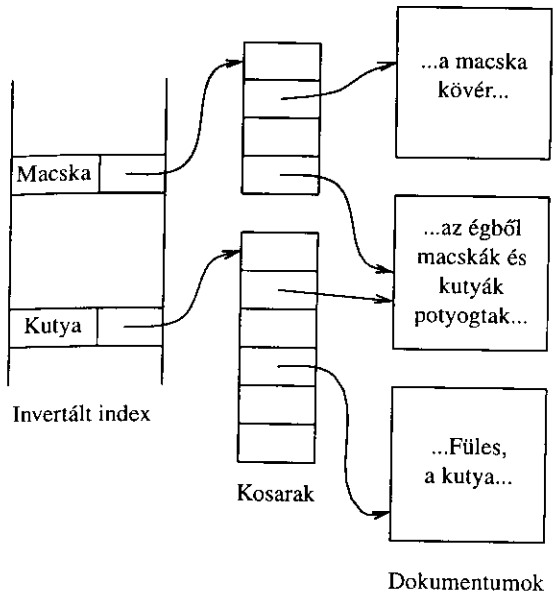
Az információszolgáltatók közössége több éve foglalkozik dokumentumok tárolásával és az adott kulcsszavakat tartalmazó dokumentumok hatékony visszakeresésével. A World Wide Web megjelenésével és a dokumentumok on-line elérhetőségének megvalósulásával az adott kulcsszavakat tartalmazó dokumentumok visszakeresése 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:

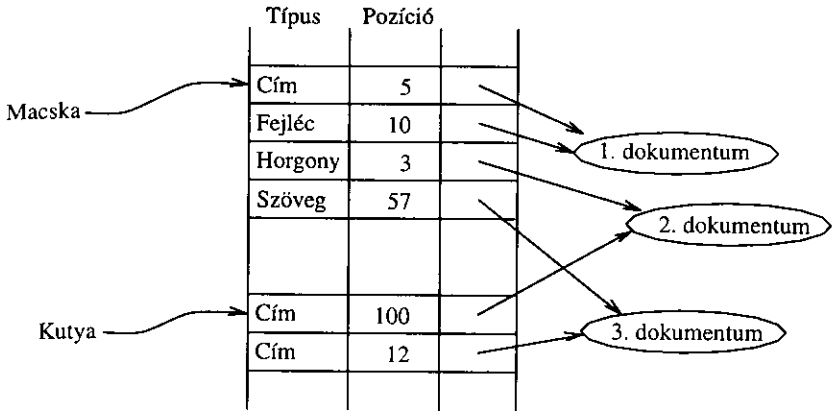
1. *Szótőképzés*. Mielőtt egy szó előfordulását bejegyeznénk az indexbe, eltávolítjuk a toldalékokat, hogy megkapjuk a „szótövet". A főnevek többes számát például úgy kezelhetjük, mintha egyes számban lennének. A 4.17. példában az invertált index természetesen szótőképzést használ, hiszen ha a „kutya" 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 $n = 340$. □

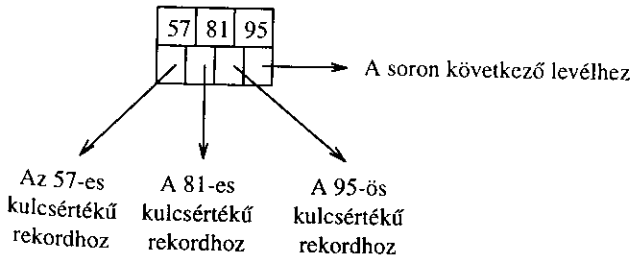
Van néhány olyan fontos szabály, amely megszorításokat jelent arra nézve, hogy egy B-fa blokkjai mit tartalmazhatnak.

- A gyökérben van legalább két használatban levő mutató.[3] Minden mutató a B-fa következő szintjének blokkjaira mutat.

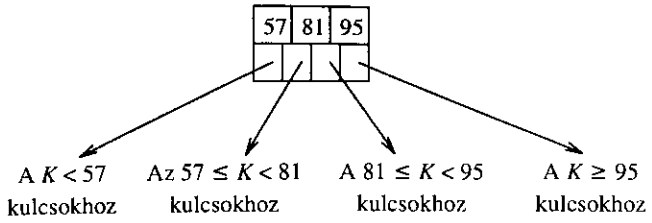
[3] Technikailag lehetséges, hogy a teljes B-fa mindössze egy mutatót tartalmazzon, akkor, ha az adatfájl egyetlen rekordból áll. Ebben az esetben az egész fa egy olyan gyökérblokk, amely egyben levél is, és ez a blokk egyetlen kulcsot és egyetlen mutatót tartalmaz. Az elkövetkezőkben eltekintünk ettől 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.

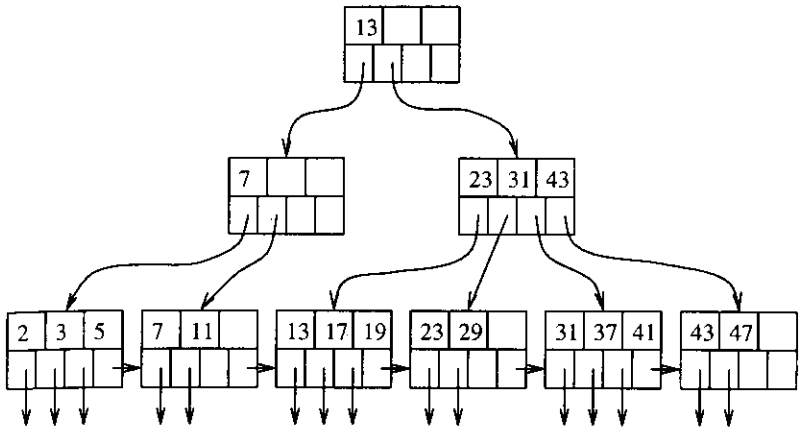
Éppúgy, mint a leveleknél, itt sem szükséges, hogy a kulcsok és mutatók tárolására fenntartott minden hely ki legyen töltve. Azonban $n = 3$, és legalább 1 kulcs és 2 mutató jelen kell legyen egy belső csúcsban. Ennek legszélsőségesebb esete az lenne, amikor az egyetlen kulcs az 57, és ekkor csak az 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*

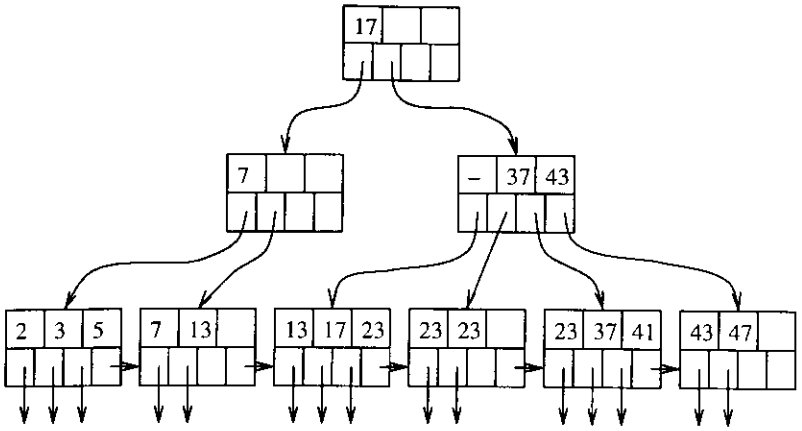
Ha megnézzük a gyökér első gyermekét, amelyben a 7-es kulcs található, ismét 2 mutatót találunk. Az egyik azokhoz a kulcsokhoz vezet, amelyek kisebbek, mint 7, a másik azokhoz, amelyek értéke nagyobb vagy egyenlő, mint 7. Vegyük észre, hogy ennek a csúcsnak a második mutatója csak a 7-es és 11-es kulcsokhoz 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

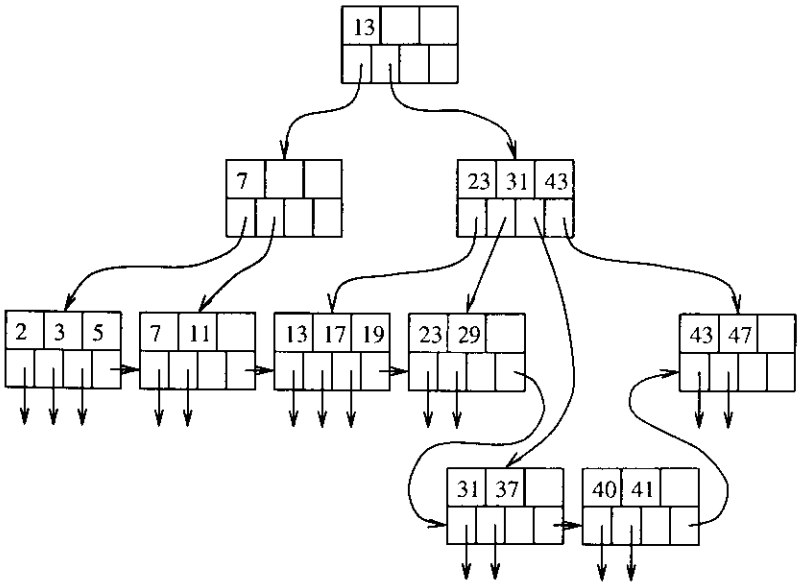
A B-fáknak vannak előnyei az egyszerűbb többszintű indexekkel szemben, ezek közül láthatunk néhányat, miközben áttekintjük, hogy miként kell beszúrni egy B-fába egy új kulcsot. A megfelelő rekordot a 4.1. részben bemutatott módszerek valamelyikével beszúrjuk a B-fával indexelt fájlba; itt most azt tekintjük 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*

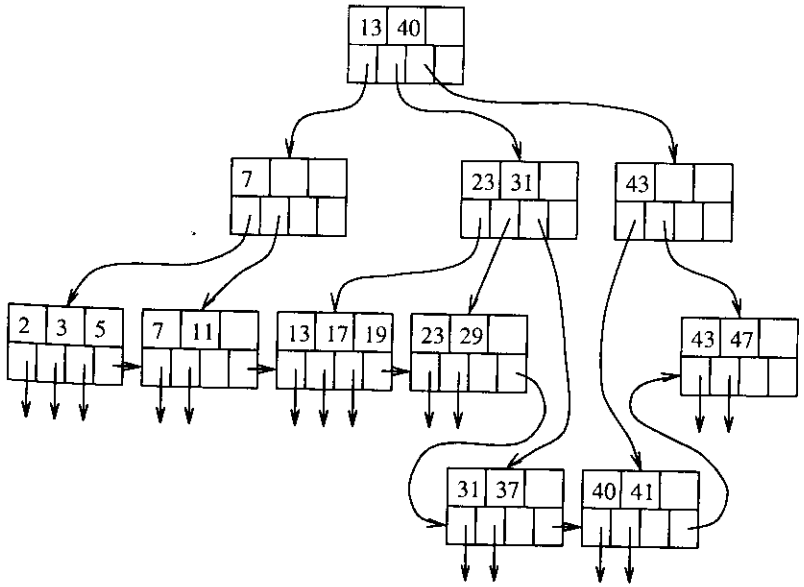
Figyeljük meg, hogy középen marad egy kulcs, amely nem jelenik sem az N, sem az M csúcsban. A maradék K kulcs azt a legkisebb kulcsot jelöli, amely az M első gyermekén keresztül elérhető. Habár ez a kulcs nem jelenik meg sem az N, sem az M csúcsban, mindamellett az M csúcshoz tartozik abban az értelemben, 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.

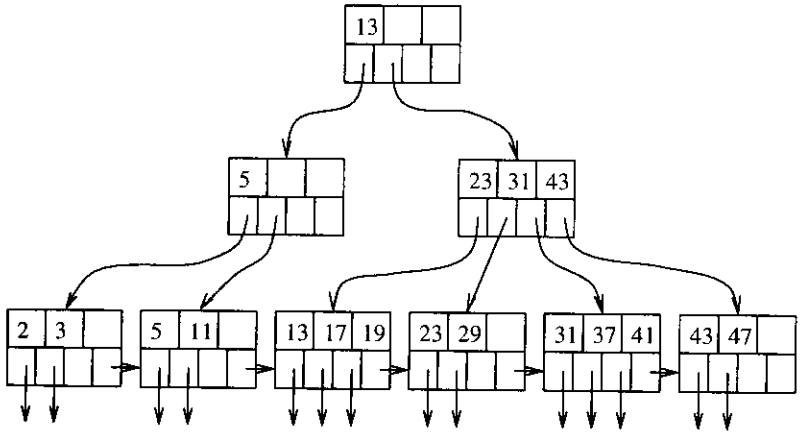
Ha a B-fának az a csúcsa, amelyben a törlés megtörtént, még így is tartalmaz legalább annyi kulcsot és mutatót, amennyit minimum tartalmaznia kell, akkor készen is vagyunk.[7] Lehetséges azonban, hogy a csúcs törlés előtti kihasználtsága minimális volt, így a törlés után a kulcsok számára vonatkozó megszorítás 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 jobb oldali testvére, amelyet nevezzünk 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.

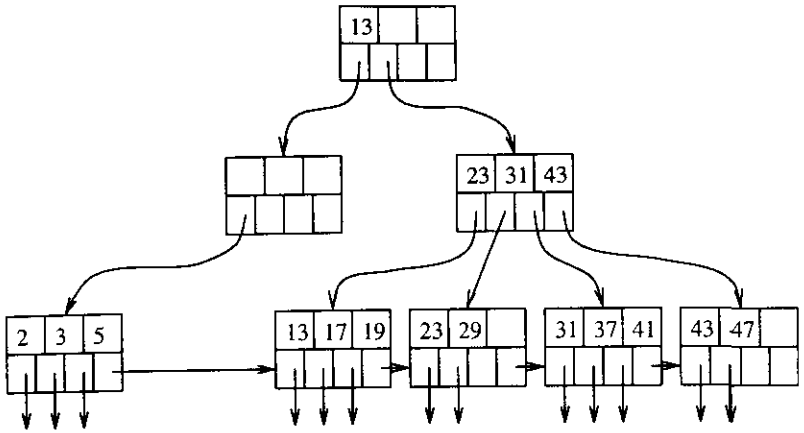


4.27. ábra. *A 7-es kulcs törlése*

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.

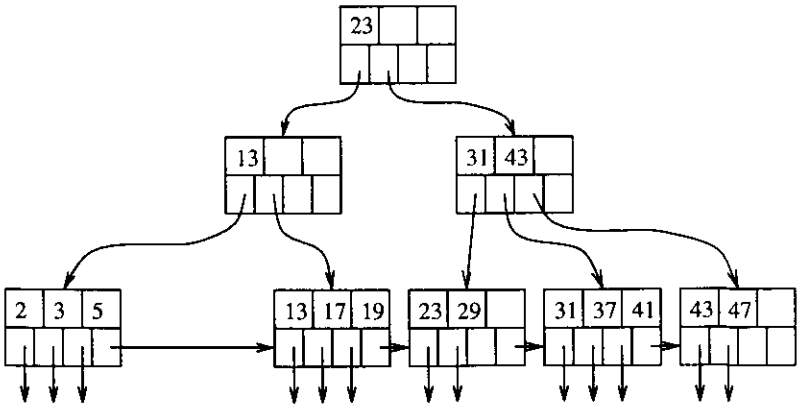
[8] Figyeljük meg, hogy a jobbra levő levél, amelynek kulcsai 13, 17 és 19 nem testvér, hiszen nem ugyanaz a két levél szülője. Ezzel együtt persze „kölcsönözhetünk" ettől a csúcstól, azonban a fa kulcsainak a helyzethez történő igazítása sokkal bonyolultabbá válik. Ezt a lehetőséget meghagyjuk feladatnak.



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. □

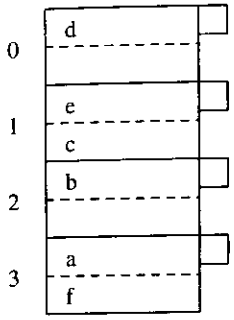
Figyeljük meg, hogy a 4.30. ábra valamennyi blokkjának jobb szélén egy „kinövés" látható. Ez a kinövés további információkat reprezentál a blokk fejlécében.

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

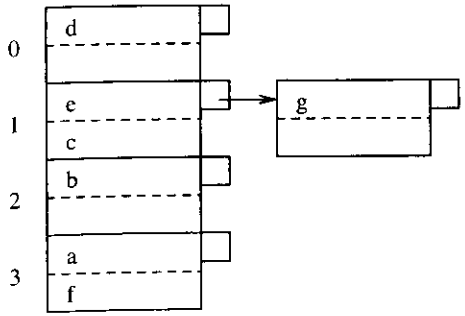
Ha egy K keresésikulcs-értékű új rekordot kell beszúrni, akkor kiszámoljuk a $h(K)$-t. Ha a $h(K)$ jelzőszámú kosárban van szabad hely, akkor a rekordot beszúrjuk a kosárhoz tartozó blokkba, vagy ha az első blokkban nincs hely, akkor a kosárhoz tartozó lánc valamelyik túlcsordulásblokkjába. Ha a $h(K)$ 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

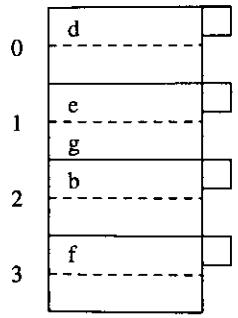
A K keresésikulcs-értékű rekord (illetve rekordok) törlése hasonló szabályszerűséget követ. Vesszük a $h(K)$ sorszámú kosarat, és megkeressük benne a megfelelő kulcsértékű rekordokat. Valamennyi megtalált rekordot töröljük. Ha van lehetőségünk a blokkok közötti rekordmozgatásra, akkor opcionálisan konszolidálhatjuk 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-műveletteljár, 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.

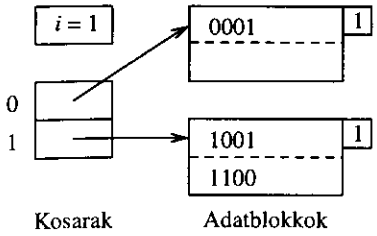
4.4.5. Kiterjeszthető tördelőtáblázatok

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 mindössze 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 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. □



4.33. ábra. *Kiterjeszthető tördelőtáblázat*

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.

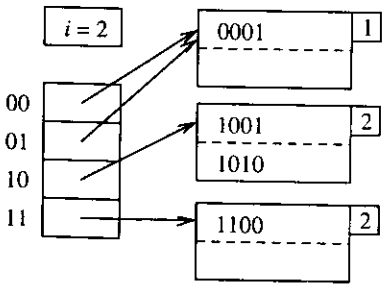
d) A kosártömb mutatóit az új helyzethez igazítjuk úgy, hogy azok a bejegyzések, amelyek azelőtt a B-re mutattak, most vagy a B-re vagy az új blokkra mutassanak, a $(j + 1)$-edik bittől függően.

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.

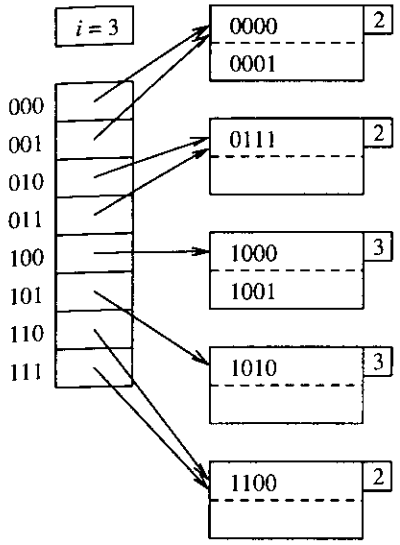
2. Ha $j = i$, akkor először meg kell növelnünk 1-gyel az i értékét. Megduplázzuk a kosártömb hosszát, hiszen most 2^{i+1} bemenetet tartalmaz. Tegyük fel, hogy a w egy i számú bitből álló sorozat, amely az előző kosártömb egyik bejegyzésének volt a jelzőszáma. Az új kosártömbben a $w0$ és $w1$ jelzőszámú 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:

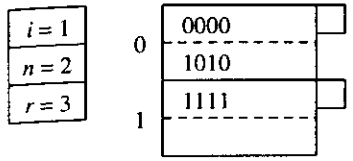
1. Amikor meg kell dupláznunk a kosártömböt, akkor tekintélyes mennyiségű munkát kell végezni (ha az i nagy). Ez a munka megszakítja az adatfájlhoz való hozzáférést, vagy igen lassúvá tesz bizonyos beszúrásokat.
2. Ha a kosártömb méretét megduplázzuk, lehet, hogy nem fér majd el az elsődleges memóriában, 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 $a_1a_2...a_i$. 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 jelző-



4.36. ábra. *Lineáris tördelőtáblázat*

számai 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.

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. □

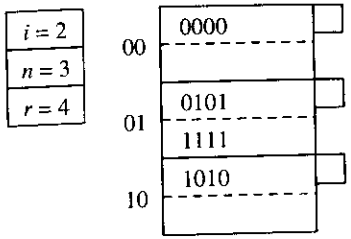
4.4.8. Beszúrás lineáris tördelőtáblázatokba

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*

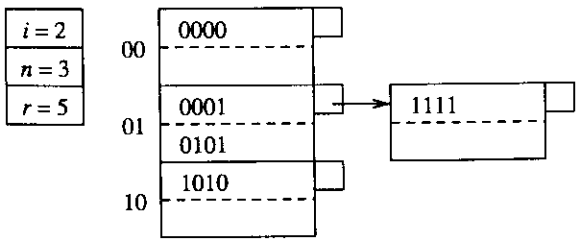
végződik, a rekord a 4.36. ábra második kosarába kerül. Van szabad hely a rekord számára, így nem kell túlcsordulásblokkot készíteni.

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.

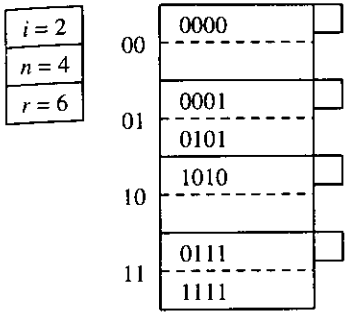
Most tegyük fel, hogy egy olyan rekordot akarunk beszúrni, amelynek keresési kulcsa 0001-et tördel. A két utolsó bit 01, így ebbe a kosárba tesszük, amely jelenleg létezik. Sajnos a kosár blokkja tele van, így hozzáadunk egy túlcsordulásblokkot. A három rekordot elosztjuk a kosár két blokkja között; 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.