Hector Garcia-Molina
Jeffrey D. Ullman
Jennifer Widom

Adatbázisrendszerek megvalósítása

Panem–John Wiley & Sons

Hector Garcia-Molina
Jeffrey D. Ullman
Jennifer Widom

Adatbázisrendszerek megvalósítása

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

Translation copyright © 2001 by Panem Könyvkiadó Kft.

ISBN 963 545 280 2

A kiadásért felel a Panem Könyvkiadó Kft. ügyvezetője, 2001

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

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

Minden jog fenntartva. Jelen könyvet, illetve annak részeit tilos reprodukálni, adatrögzítő rendszerben tárolni, bármilyen formában vagy eszközzel – elektronikus úton vagy más módon – közölni a kiadók engedélye nélkül.

Tartalomjegyzék

Előszó a magyar kiadáshoz 19

Előszó 21

1. Bevezetés az adatbázis-kezelő rendszerek implementálásába 24

1.1. Bevezetés: a Megatron 2000 adatbázisrendszer 25
1.1.1. A Megatron 2000 implementálásának részletei 26
1.1.2. Hogyan hajtja végre a Megatron 2000 a lekérdezéseket? 27
1.1.3. Mi a baj a Megatron 2000-rel? 29

1.2. Egy adatbázis-kezelő rendszer áttekintése 30
1.2.1. Az adatdefiníciós nyelv parancsai 30
1.2.2. A lekérdezés feldolgozásának áttekintése 30
1.2.3. A központi memória pufferei és a pufferkezelő 32
1.2.4. A tranzakció feldolgozása 33
1.2.5. A lekérdezésfeldolgozó 34

1.3. A könyv vázlatos felépítése 35
1.3.1. Előismeretek 35
1.3.2. A tárkezelés áttekintése 36
1.3.3. A lekérdezésfeldolgozás áttekintése 37
1.3.4. A tranzakciófeldolgozó áttekintése 37
1.3.5. Az információintegráció áttekintése 38

1.4. Az adatmodellek és nyelvek áttekintése 38
1.4.1. A relációs modell áttekintése 38
1.4.2. Az SQL áttekintése 39
1.4.3. A relációs és objektumorientált adatok 42

1.5. Összefoglalás ... 44

1.6. Irodalomjegyzék ... 44

2. Adattárolás ... 46

2.1. A memóriahierarchia ... 47
2.1.1. Cache ... 47
2.1.2. A központi memória ... 48
2.1.3. Virtuális memória ... 49
2.1.4. Másodlagos tárolás ... 51
2.1.5. Harmadlagos tárolás ... 52
2.1.6. Felejtő és nem felejtő tárolás ... 54
2.1.7. Feladatok ... 55

2.2. Lemezek ... 55
2.2.1. A lemezek mechanikája ... 56
2.2.2. A lemezvezérlő ... 57
2.2.3. A lemeztárolók jellemzői ... 58
2.2.4. A lemezhozzáférés jellemzői ... 60
2.2.5. Blokkok írása ... 64
2.2.6. Blokkok módosítása ... 64
2.2.7. Feladatok ... 65

2.3. A másodlagos tárolók hatékony használata ... 66
2.3.1. A számítás I/O-modellje ... 66
2.3.2. Adatok rendezése a másodlagos tárolóban ... 67
2.3.3. Az összefésülő rendezés (Merge-Sort) ... 68
2.3.4. Kétfázisú, többutas, összefésülő rendezés ... 70
2.3.5. A többutas összefésülés kiterjesztése nagyobb relációkra ... 72
2.3.6. Feladatok ... 74

2.4. A másodlagos tároló hozzáférési idejének javítása ... 75
2.4.1. Az adatok cilinderes szervezése ... 77
2.4.2. Több lemez használata ... 78
2.4.3. Lemezek tükrözése ... 79
2.4.4. A lemez ütemezése és a lift algoritmus ... 80
2.4.5. Korai beolvasás és nagy léptékű pufferezés ... 84
2.4.6. A stratégiák előnyeinek és hátrányainak összegzése ... 86
2.4.7. Feladatok ... 87

2.5. Lemezhibák ... 89
2.5.1. Ideiglenes meghibásodás ... 90
2.5.2. Ellenőrző összegek ... 90

2.5.3. Stabil tárolás ... 92
2.5.4. A stabil tárolás hibakezelő képessége ... 92
2.5.5. Feladatok ... 93

2.6. Lemezhiba helyreállítása ... 94
2.6.1. A lemezek meghibásodási modelljei ... 94
2.6.2. A tükrözés mint redundanciatechnika ... 95
2.6.3. Paritásblokkok ... 96
2.6.4. Egy továbbfejlesztés: az 5. szintű RAID ... 100
2.6.5. Mi a teendő, ha több lemez is tönkremehet? ... 101
2.6.6. Feladatok ... 104

2.7. Összefoglalás ... 107

2.8. Irodalomjegyzék ... 109

3. Adatelemek ábrázolása ... 111

3.1. Adatelemek és mezők ... 111
3.1.1. Relációs adatbáziselemek ábrázolása ... 112
3.1.2. Objektumok ábrázolása ... 113
3.1.3. Adatelemek ábrázolása ... 114

3.2. Rekordok ... 119
3.2.1. Rögzített hosszú rekordok építése ... 119
3.2.2. Rekordfejlécek ... 121
3.2.3. Rögzített hosszú rekordok blokkokba pakolása ... 123
3.2.4. Feladatok ... 124

3.3. Blokkcímek és rekordcímek ábrázolása ... 125
3.3.1. Kliens-szerver rendszerek ... 125
3.3.2. Logikai és strukturált címek ... 127
3.3.3. Mutatók helyreigazítása ... 129
3.3.4. Blokkok visszaírása a lemezre ... 133
3.3.5. Feltűzött rekordok és blokkok ... 134
3.3.6. Feladatok ... 135

3.4. Változó hosszú adatok és rekordok ... 137
3.4.1. Változó hosszú mezőket tartalmazó rekordok ... 138
3.4.2. Ismétlődő mezőket tartalmazó rekordok ... 139
3.4.3. Változó formátumú rekordok ... 141
3.4.4. Olyan rekordok, amelyek nem férnek el egy blokkban ... 142
3.4.5. Bináris, nagy objektumok (BLOB-ok) ... 143
3.4.6. Feladatok ... 144

3.5. Rekordmódosítások ... 146
3.5.1. Beszúrás ... 146
3.5.2. Törlés ... 148
3.5.3. Módosítás ... 149
3.5.4. Feladatok ... 150

3.6. Összefoglalás ... 151

3.7. Irodalomjegyzék ... 152

4. Indexstruktúrák ... 153

4.1. Indexek szekvenciális fájlokon ... 154
4.1.1. Szekvenciális fájlok ... 155
4.1.2. Sűrű indexek ... 155
4.1.3. Ritka indexek ... 158
4.1.4. Többszintű indexelés ... 159
4.1.5. Indexelés ismétlődő keresésikulcs-érték esetén ... 161
4.1.6. Indexek kezelése adatmódosításkor ... 164
4.1.7. Feladatok ... 170

4.2. Másodlagos indexek ... 171
4.2.1. Másodlagos indexek tervezése ... 172
4.2.2. Másodlagos indexek alkalmazása ... 173
4.2.3. Közvetett másodlagos indexek ... 175
4.2.4. Dokumentumok visszakeresése és az invertált indexek ... 178
4.2.5. Feladatok ... 181

4.3. B-fák ... 184
4.3.1. B-fák szerkezete ... 184
4.3.2. B-fák alkalmazása ... 187
4.3.3. Keresés B-fában ... 189
4.3.4. Tartományra vonatkozó lekérdezések ... 190
4.3.5. Beszúrás B-fában ... 191
4.3.6. Törlés B-fában ... 194
4.3.7. B-fák hatékonysága ... 197
4.3.8. Feladatok ... 197

4.4. Tördelőtáblázatok ... 200
4.4.1. Másodlagos tárolón tárolt tördelőtáblázatok ... 201
4.4.2. Beszúrás tördelőtáblázatba ... 202
4.4.3. Törlés tördelőtáblázatban ... 202
4.4.4. Tördelőtáblázat-indexek hatékonysága ... 203
4.4.5. Kiterjeszthető tördelőtáblázatok ... 204

4.4.6. Beszúrás kiterjeszthető tördelőtáblázatokba ... 205
4.4.7. Lineáris tördelőtáblázatok ... 207
4.4.8. Beszúrás lineáris tördelőtáblázatokba ... 209
4.4.9. Feladatok ... 211

4.5. Összefoglalás ... 213

4.6. Irodalomjegyzék ... 214

5. Többdimenziós indexek ... 216

5.1. Többdimenziós alkalmazások ... 217
5.1.1. Térinformatikai rendszerek ... 217
5.1.2. Adatkockák ... 218
5.1.3. Többdimenziós lekérdezések SQL-ben ... 219
5.1.4. Tartománylekérdezések végrehajtása hagyományos indexekkel ... 221
5.1.5. Legközelebbi szomszéd-lekérdezések végrehajtása hagyományos indexekkel ... 222
5.1.6. A hagyományos indexek további korlátjai ... 224
5.1.7. A többdimenziós indexstruktúrák áttekintése ... 224
5.1.8. Feladatok ... 225

5.2. Tördelésen alapuló struktúrák többdimenziós adatokhoz ... 226
5.2.1. Rácsos állományok ... 227
5.2.2. Keresés rácsos állományban ... 227
5.2.3. Beszúrás rácsos állományba ... 229
5.2.4. A rácsos állományok hatékonysága ... 230
5.2.5. Particionált tördelőfüggvények ... 233
5.2.6. A rácsos állományok és a particionált tördelés összehasonlítása ... 234
5.2.7. Feladatok ... 235

5.3. Faszerű struktúrák többdimenziós adatokhoz ... 238
5.3.1. Többkulcsos indexek ... 238
5.3.2. A többkulcsos indexek hatékonysága ... 240
5.3.3. *kd*-fák ... 241
5.3.4. Műveletek a *kd*-fákon ... 243
5.3.5. A *kd*-fák alkalmazása másodlagos tárolók esetén ... 245
5.3.6. Quad-fák ... 246
5.3.7. R-fák ... 248
5.3.8. Műveletek az R-fákon ... 249
5.3.9. Feladatok ... 251

5.4. Bittérképindexek ... 253
5.4.1. Indítékok a bittérképindexekhez ... 254

5.4.2. Tömörített bittérképek ... 256
5.4.3. Műveletek szakaszhosszkódolt bitvektorokon ... 258
5.4.4. Bittérképindexek kezelése ... 259
5.4.5. Feladatok ... 260

5.5. Összefoglalás ... 261

5.6. Irodalomjegyzék ... 263

6. Lekérdezések végrehajtása ... 265

6.1. Algebrai megközelítés ... 267
6.1.1. Egyesítés, metszet és különbség ... 269
6.1.2. Kiválasztás ... 270
6.1.3. Vetítés ... 272
6.1.4. Relációk szorzata ... 273
6.1.5. Összekapcsolások ... 274
6.1.6. Ismétlődések kiküszöbölése ... 276
6.1.7. Csoportosítás és összesítés ... 277
6.1.8. Rendezés ... 279
6.1.9. Kifejezésfák ... 280
6.1.10. Feladatok ... 282

6.2. Bevezetés a fizikai lekérdezésterv-operátorok világába ... 285
6.2.1. Táblák átvizsgálása ... 285
6.2.2. Rendezés a táblák átvizsgálásakor ... 286
6.2.3. A fizikai operátorok kiszámításának modellje ... 287
6.2.4. A költségbecslés paraméterei ... 287
6.2.5. Az átvizsgáló operátorok I/O-költsége ... 289
6.2.6. Fizikai operátorok megvalósításához használatos iterátorok ... 290

6.3. Adatbázis-műveletek egymenetes algoritmusai ... 293
6.3.1. Soronkénti műveletek egymenetes algoritmusai ... 294
6.3.2. Unáris, teljes relációs műveletek egymenetes algoritmusai ... 295
6.3.3. Bináris műveletek egymenetes algoritmusai ... 299
6.3.4. Feladatok ... 302

6.4. Beágyazott ciklusú összekapcsolások ... 303
6.4.1. Sor alapú beágyazott ciklusú összekapcsolás ... 304
6.4.2. Egy iterátor a sor alapú beágyazott ciklusú összekapcsoláshoz ... 304
6.4.3. Egy algoritmus a blokk alapú beágyazott ciklusú összekapcsoláshoz ... 304
6.4.4. A beágyazott ciklusú összekapcsolás elemzése ... 307
6.4.5. Az eddigi algoritmusok összefoglalása ... 307
6.4.6. Feladatok ... 308

6.5. Rendezésen alapuló kétmenetes algoritmusok ... 308
6.5.1. Ismétlődések kiküszöbölése rendezés segítségével ... 309
6.5.2. Csoportosítás és összesítés rendezés segítségével ... 312
6.5.3. Az egyesítés egy rendezésen alapuló algoritmusa ... 312
6.5.4. A metszet és a különbség rendezésen alapuló algoritmusai ... 313
6.5.5. Egy egyszerű rendezésen alapuló összekapcsolási algoritmus ... 315
6.5.6. Az egyszerű rendezéses összekapcsolás elemzése ... 317
6.5.7. Egy hatékonyabb rendezésen alapuló összekapcsolás ... 317
6.5.8. A rendezésen alapuló algoritmusok összefoglalása ... 319
6.5.9. Feladatok ... 319

6.6. Tördelésen alapuló kétmenetes algoritmusok ... 321
6.6.1. Relációk particionálása tördeléssel ... 321
6.6.2. Egy tördelésen alapuló algoritmus az ismétlődések kiküszöbölésére ... 322
6.6.3. Egy tördelésen alapuló algoritmus a csoportosításra és az összesítésre ... 323
6.6.4. Az egyesítés, a metszet és a különbség tördelésen alapuló algoritmusai ... 323
6.6.5. A tördeléses összekapcsolási algoritmus ... 324
6.6.6. Lemez I/O-műveletek megtakarítása ... 325
6.6.7. A tördelésen alapuló algoritmusok összefoglalása ... 327
6.6.8. Feladatok ... 328

6.7. Index alapú algoritmusok ... 329
6.7.1. Nyalábolt és nem nyalábolt indexek ... 329
6.7.2. Index alapú kiválasztás ... 330
6.7.3. Összekapcsolás index segítségével ... 333
6.7.4. Összekapcsolások rendezett index segítségével ... 334
6.7.5. Feladatok ... 336

6.8. Pufferkezelés ... 337
6.8.1. A pufferkezelő működése ... 338
6.8.2. Pufferkezelő stratégiák ... 338
6.8.3. Kapcsolat a fizikai operátor kiválasztása és a pufferkezelés között ... 340
6.8.4. Feladatok ... 342

6.9. Több mint kétmenetes algoritmusok ... 343
6.9.1. Többmenetes, rendezésen alapuló algoritmusok ... 343
6.9.2. Többmenetes, rendezésen alapuló algoritmusok műveletigénye ... 344
6.9.3. Többmenetes, tördelésen alapuló algoritmusok ... 345
6.9.4. Többmenetes, tördelésen alapuló algoritmusok műveletigénye ... 345
6.9.5. Feladatok ... 346

6.10. Párhuzamos algoritmusok relációs műveletekre ... 347
6.10.1. A párhuzamosság modelljei ... 347
6.10.2. Soronkénti műveletek párhuzamos megvalósítása ... 350
6.10.3. Teljes relációs műveletek párhuzamos algoritmusai ... 351

6.10.4. A párhuzamos algoritmusok hatékonysága ... 352
6.10.5. Feladatok ... 355

6.11. Összefoglalás ... 356

6.12. Irodalomjegyzék ... 358

7. **A lekérdezésfordító** ... 359

7.1. Elemzés ... 360
7.1.1. Szintaktikus elemzés és elemzőfák ... 360
7.1.2. Egy leegyszerűsített SQL-részletet leíró nyelvtan ... 361
7.1.3. Az előfeldolgozó ... 366
7.1.4. Feladatok ... 367

7.2. Algebrai szabályok lekérdezéstervek javítására ... 367
7.2.1. Kommutatív és asszociatív szabályok ... 368
7.2.2. Kiválasztással kapcsolatos szabályok ... 371
7.2.3. Kiválasztások tologatása ... 374
7.2.4. Vetítéssel kapcsolatos szabályok ... 375
7.2.5. Összekapcsolásra és szorzatra vonatkozó szabályok ... 379
7.2.6. Ismétlődések elhagyására vonatkozó szabályok ... 379
7.2.7. Csoportosításra és összesítésre vonatkozó szabályok ... 380
7.2.8. Feladatok ... 382

7.3. Elemzőfák átalakítása logikai lekérdezéstervekké ... 384
7.3.1. Átfordítás relációs algebrába ... 384
7.3.2. Alkérdések eltávolítása feltételekből ... 386
7.3.3. Logikai lekérdezéstervek javítása ... 391
7.3.4. Asszociatív/kommutatív operátorok csoportosítása ... 393
7.3.5. Feladatok ... 394

7.4. Műveletek költségének becslése ... 395
7.4.1. Közbülső relációk méretének becslése ... 396
7.4.2. Vetítés méretének becslése ... 397
7.4.3. Kiválasztás méretének becslése ... 398
7.4.4. Összekapcsolás méretének becslése ... 401
7.4.5. Természetes összekapcsolás több összekapcsolási attribútummal ... 403
7.4.6. Sok reláció összekapcsolása ... 405
7.4.7. Egyéb műveletek méretének becslése ... 406
7.4.8. Feladatok ... 409

7.5. Bevezetés a költség alapú tervválasztásba ... 410
7.5.1. Méretre vonatkozó paraméterek becslése ... 411

7.5.2. Statisztikák növekményes kiszámítása ... 414
7.5.3. Logikai lekérdezéstervek költségének csökkentésére irányuló heurisztikák ... 416
7.5.4. Fizikai tervek felsorolásának lehetőségei ... 418
7.5.5. Feladatok ... 421

7.6. Összekapcsolások sorrendjének megválasztása ... 423
7.6.1. Összekapcsolások bal és jobb oldali argumentumainak jelentősége ... 423
7.6.2. Összekapcsolási fák ... 424
7.6.3. Bal-mély összekapcsolási fák ... 425
7.6.4. Dinamikus programozás az összekapcsolási sorrend és csoportosítás megválasztására ... 428
7.6.5. Dinamikus programozás részletesebb költségfüggvényekkel ... 433
7.6.6. Egy mohó algoritmus az összekapcsolási sorrend kiválasztására ... 434
7.6.7. Feladatok ... 435

7.7. A fizikai lekérdezésterv kiválasztásának befejezése ... 437
7.7.1. Kiválasztási eljárás megválasztása ... 437
7.7.2. Összekapcsolási eljárás megválasztása ... 440
7.7.3. Futószalagosítás és materializáció ... 441
7.7.4. Unáris műveletek futószalagosítása ... 442
7.7.5. Bináris műveletek futószalagosítása ... 443
7.7.6. Fizikai lekérdezéstervekkel kapcsolatos jelölések ... 445
7.7.7. Fizikai operátorok sorrendbe állítása ... 449
7.7.8. Feladatok ... 449

7.8. Összefoglalás ... 451

7.9. Irodalomjegyzék ... 452

8. **A rendszerhibák kezelése** ... 454

8.1. A helyreállítható beavatkozások példái és modelljei ... 454
8.1.1. A hibák fajtái ... 455
8.1.2. Részletesebben a tranzakciókról ... 457
8.1.3. A tranzakciók korrekt végrehajtása ... 458
8.1.4. A tranzakciók alaptevékenységei ... 460
8.1.5. Feladatok ... 463

8.2. Semmisségi (undo) naplózás ... 463
8.2.1. Naplóbejegyzések ... 465
8.2.2. A semmisségi naplózás szabályai ... 466
8.2.3. Helyreállítás a semmisségi naplózás használatával ... 468
8.2.4. Az ellenőrzőpont-képzés ... 471

8.2.5. Ellenőrzőpont-képzés a rendszer működése közben 473
8.2.6. Feladatok 476

8.3. Helyrehozó naplózás (redo logging) 477
8.3.1. A helyrehozó naplózás szabályai 478
8.3.2. Helyreállítás a helyrehozó naplózás használatával 479
8.3.3. Helyrehozó naplózás ellenőrzőpont-képzés használatával 480
8.3.4. Visszaállítás az ellenőrzőponttal kiegészített helyrehozó típusú naplózással 482
8.3.5. Feladatok 483

8.4. A semmisségi/helyrehozó (undo/redo) naplózás 484
8.4.1. A semmisségi/helyrehozó (undo/redo) naplózás szabályai 484
8.4.2. Helyreállítás a semmisségi/helyrehozó (undo/redo) naplózás használatakor 485
8.4.3. Semmisségi/helyrehozó naplózás ellenőrzőpont-képzéssel 487
8.4.4. Feladatok 489

8.5. Az eszközök meghibásodása elleni védekezés 490
8.5.1. Az archívmentés 490
8.5.2. Archiválás működés közben 491
8.5.3. Helyreállítás az archívmentés és a napló használatával 494
8.5.4. Feladatok 495

8.6. Összefoglalás 495

8.7. Irodalomjegyzék 497

9. Konkurenciavezérlés 498

9.1. Soros és sorba rendezhető ütemezések 499
9.1.1. Ütemezések 499
9.1.2. Soros ütemezések 500
9.1.3. Sorba rendezhető ütemezések 501
9.1.4. A tranzakció szemantikájának hatása 503
9.1.5. A tranzakciók és ütemezések jelölése 504
9.1.6. Feladatok 505

9.2. Konfliktus-sorbarendezhetőség 505
9.2.1. Konfliktusok 506
9.2.2. Megelőzési gráfok és teszt a konfliktus-sorbarendezhetőségre 507
9.2.3. Miért működik a megelőzési gráfon alapuló tesztelés? 510
9.2.4. Feladatok 511

9.3. A sorbarendezhetőség biztosítása zárakkal 513
9.3.1. Zárak 514
9.3.2. A zárolási ütemező 516
9.3.3. A kétfázisú zárolás 517
9.3.4. Miért működik a kétfázisú zárolás? 517
9.3.5. Feladatok 519

9.4. Különböző zármódú zárolási rendszerek 521
9.4.1. Osztott és kizárólagos zárak 521
9.4.2. Kompatibilitási mátrixok 523
9.4.3. Zárak felminősítése 524
9.4.4. Módosítási zárak 526
9.4.5. Növelési zárak 527
9.4.6. Feladatok 529

9.5. A zárolási ütemező felépítése 532
9.5.1. Zárolási műveleteket beszúró ütemező 532
9.5.2. A zártábla 534
9.5.3. Feladatok 537

9.6. Adatbáziselemekből álló hierarchiák kezelése 538
9.6.1. Többszörös szemcsézettségű zárak 538
9.6.2. A figyelmeztető zárak 539
9.6.3. Fantomok és a beszúrások helyes kezelése 542
9.6.4. Feladatok 543

9.7. Faprotokoll 544
9.7.1. Fa alapú zárolások idítékai 544
9.7.2. Faszerkezetű adatok hozzáférési szabályai 545
9.7.3. Miért működik a faprotokoll? 546
9.7.4. Feladatok 549

9.8. Konkurenciavezérlés időbélyegzőkkel 550
9.8.1. Időbélyegzők 551
9.8.2. Fizikailag nem megvalósítható viselkedések 552
9.8.3. Piszkos adatok problémái 553
9.8.4. Az időbélyegzőn alapuló ütemezések szabályai 554
9.8.5. Többváltozatú időbélyegzők 556
9.8.6. Az időbélyegzők és zárolások 558
9.8.7. Feladatok 559

9.9. Konkurenciavezérlés érvényesítéssel 560
9.9.1. Érvényesítésen alapuló ütemező felépítése 560
9.9.2. Az érvényesítési szabályok 561
9.9.3. Három konkurenciavezérlés működésének összehasonlítása 564

9.9.4. Feladatok ... 565

9.10. Összefoglalás ... 565

9.11. Irodalomjegyzék ... 568

10. Bővebben a tranzakciókezelésről ... 569

10.1. Tranzakciók, melyek nem véglegesített adatokat olvasnak ... 569
10.1.1. A piszkos adat probléma ... 570
10.1.2. Továbbgyűrűző visszagörgetés ... 572
10.1.3. A visszagörgetés kezelése ... 573
10.1.4. Csoportos véglegesítés ... 574
10.1.5. Logikai naplózás ... 576
10.1.6. Feladatok ... 579

10.2. Nézet-sorbarendezhetőség ... 580
10.2.1. Nézetekvivalencia ... 580
10.2.2. Poligráfok és nézet-sorbarendezhetőségi teszt ... 582
10.2.3. A nézet-sorbarendezhetőség tesztelése ... 585
10.2.4. Feladatok ... 585

10.3. Holtpontkezelés ... 586
10.3.1. Holtpontérzékelés időkorláttal ... 586
10.3.2. A várakozási gráf ... 587
10.3.3. Holtpontmegelőzés az elemek sorbarendezésével ... 589
10.3.4. Holtpontérzékelés időbélyegzővel ... 591
10.3.5. A holtpontkezelő módszerek összehasonlítása ... 593
10.3.6. Feladatok ... 594

10.4. Osztott adatbázisok ... 595
10.4.1. Osztott adatok ... 596
10.4.2. Osztott tranzakciók ... 597
10.4.3. Adattöbbszörözés ... 598
10.4.4. Osztott lekérdezésoptimalizálás ... 599
10.4.5. Feladatok ... 600

10.5. Osztott véglegesítés ... 606
10.5.1. Az osztott atomosság támogatása ... 601
10.5.2. Kétfázisú véglegesítés ... 601
10.5.3. Az osztott tranzakciók helyreállítása ... 604
10.5.4. Feladatok ... 606

10.6. Osztott zárolás ... 607
10.6.1. Központosított zárolási rendszerek ... 607
10.6.2. Költségmodell az osztott zárolási algoritmusokhoz ... 608
10.6.3. Többszörözött elemek zárolása ... 609
10.6.4. Az elsődleges példány zárolása ... 610
10.6.5. A lokális zártól a globálisig ... 611
10.6.6. Feladatok ... 612

10.7. Hosszú tranzakciók ... 613
10.7.1. A hosszú tranzakciók problémái ... 613
10.7.2. Regék ... 616
10.7.3. Kiegyenlítő tranzakciók ... 617
10.7.4. Miért működnek jól a kiegyenlítő tranzakciók? ... 619
10.7.5. Feladatok ... 619

10.8. Összefoglalás ... 620

10.9. Irodalomjegyzék ... 622

11. Információk egyesítése ... 624

11.1. Az információegyesítés módjai ... 624
11.1.1. Az egyesítés problémái ... 625
11.1.2. Adatbázis-szövetség ... 627
11.1.3. Adattárházak ... 628
11.1.4. Adatközvetítő ... 631
11.1.5. Feladatok ... 633

11.2. Borítékolók a közvetítő alapú rendszerekben ... 635
11.2.1. Sablonok lekérdezési formákhoz ... 635
11.2.2. Borítékoló generátor ... 636
11.2.3. Szűrők ... 637
11.2.4. A borítékoló más műveletei ... 639
11.2.5. Feladatok ... 641

11.3. On-line analitikus feldolgozás ... 641
11.3.1. OLAP-alkalmazások ... 643
11.3.2. OLAP-adatok többdimenziós nézete ... 644
11.3.3. A csillag séma ... 645
11.3.4. Szeletelés és kockázás ... 647
11.3.5. Feladatok ... 650

11.4. Adatkocka ... 651
11.4.1. A kockaművelet ... 652

11.4.2. Kockaimplementáció megvalósított nézettáblákkal ... 655
11.4.3. Nézetháló ... 658
11.4.4. Feladatok ... 660

11.5. Adatbányászat ... 662
11.5.1. Adatbányászati alkalmazások ... 663
11.5.2. Társítási szabály bányászat ... 666
11.5.3. Az előzetes algoritmus ... 667

11.6. Összefoglalás ... 670

11.7. Irodalomjegyzék ... 671

Index ... 675

Előszó a magyar kiadáshoz

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

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

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

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

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

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

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

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

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

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

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

Dr. Benczúr András

Előszó

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

A könyv használata

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

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

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

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

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

Előismeretek

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

Feladatok

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

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

Támogatás a World Wide Weben

A könyv honlapja:

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

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

Köszönetnyilvánítások

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

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

1. fejezet

Bevezetés az adatbázis-kezelő rendszerek implementálásába

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

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

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

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

A legfontosabb szakkifejezések áttekintése

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

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

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

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

1.1. Bevezetés: a Megatron 2000 adatbázisrendszer

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

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

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

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

1.1.1. A Megatron 2000 implementálásának részletei

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

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

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

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

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

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

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

```
dbhost> megatron2000
```

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

Erre a következő választ kapjuk:

```
ÜDVÖZÖL A MEGATRON 2000!
```

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

```
& SELECT *
  FROM Hallgatók #
```

válaszul a következő táblát adja meg:

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

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

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

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

```
Johnson#522#VM
```
□

1.1.2. Hogyan hajtja végre a Megatron 2000 a lekérdezéseket?

Tekintsük az alábbi általános formájú SQL-lekérdezést:

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

A Megatron 2000 erre a következőket teszi:

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

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

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

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

   Ahhoz, hogy a Megatron 2000 a

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

utasítást végrehajtsa, a következőt fogja tenni:

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

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

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

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

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

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

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

□

1.1.3. Mi a baj a Megatron 2000-rel?

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

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

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

1.2. Egy adatbázis-kezelő rendszer áttekintése

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

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

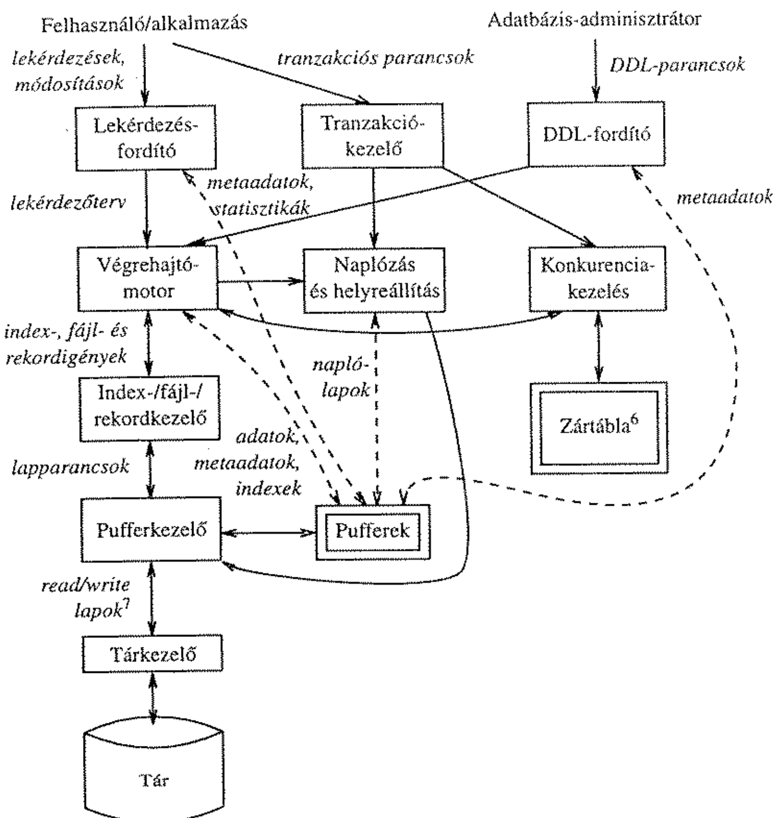
1.2.1. Az adatdefiníciós nyelv parancsai

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

1.2.2. A lekérdezés feldolgozásának áttekintése

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

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



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

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

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

[6] A szakirodalomban igen elterjedt a lock table kifejezés is. *A fordító megjegyzése.*
[7] Azok az adategységek, amelyeket beolvasunk (*read*), illetve kiírunk (*write*). *A fordító megjegyzése.*

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

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

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

Ezeket az alkotórészeket az 1.2.4. részben fogjuk tovább vizsgálni.

1.2.3. A központi memória pufferei és a pufferkezelő

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

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

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

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

1.2.4. A tranzakció feldolgozása

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

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

A tranzakciók ACID-tulajdonságai

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

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

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

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

1.2.5. A lekérdezésfeldolgozó

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

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

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

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

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

1.3. A könyv vázlatos felépítése

Az adatbázisrendszerek implementálásának témaköre nagy vonalakban három részre osztható fel:

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

A fenti témák mindegyikét a könyv több fejezeten keresztül tárgyalja.

1.3.1. Előismeretek

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

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

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

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

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

1.3.2. A tárkezelés áttekintése

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

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

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

1.3.3. A lekérdezésfeldolgozás áttekintése

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

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

1.3.4. A tranzakciófeldolgozó áttekintése

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

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

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

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

1.3.5. Az információintegráció áttekintése

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

1.4. Az adatmodellek és nyelvek áttekintése

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

1.4.1. A relációs modell áttekintése

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

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

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

Ennek a relációnak a sémája az alábbi:

```
Film(filmcím, év, hossz)
```

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

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

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

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

1.4.2. Az SQL áttekintése

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

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

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

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

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

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

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

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

[9] Szokás még mondatrésznek, klauzulának vagy klóznak is hívni. *A fordító megjegyzése.*
[10] Ezeket a statisztikai, összesítő függvényeket aggregátoroknak is szokták fordítani. *A fordító megjegyzése.*


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

1.3. ábra. *A Paramount színészeinek megkeresése*

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

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

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

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

Alkérdések

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

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

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

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

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

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

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

Nézettáblák

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

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

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

1.6. ábra. *A Paramount filmjeit meghatározó nézettábla*

1.4.3. A relációs és objektumorientált adatok

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

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

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

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

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

Hasznos lehet, ha összegezzük a következő hasonlóságokat:

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

1.5. Összefoglalás

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

1.6. Irodalomjegyzék

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

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

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

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

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

2. fejezet

Adattárolás

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

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

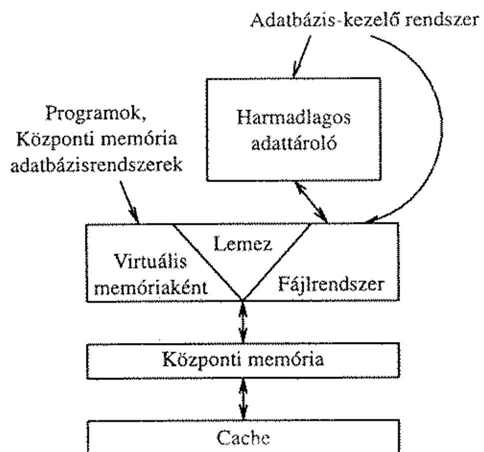
Az első kérdést ez a fejezet fedi le, míg a második a következő három fejezet témája lesz.

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

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

2.1. A memóriahierarchia

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



2.1. ábra. *A memóriahierarchia*

2.1.1. Cache[1]

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

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

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

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

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

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

2.1.2. A központi memória

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

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

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

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

A számítógépes mennyiségek 2 hatványai

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

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

2.1.3. Virtuális memória

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

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

Moore törvénye

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

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

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

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

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

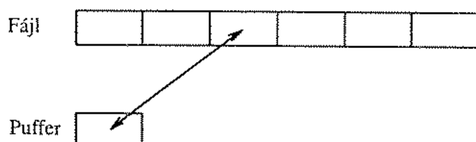
2.1.4. Másodlagos tárolás

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

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

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

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



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

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

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

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

2.1.5. Harmadlagos tárolás

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

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

1. *Ad hoc szalagos tárolás.* A legegyszerűbb – és az elmúlt években sokáig az egyetlen – megvalósítása a harmadlagos tárolónak az, hogy az adatokat orsós vagy kazettás szalagokra mentjük ki, és ezeket utána tárolókba helyezzük. Amikor valamilyen információra van szükség a harmadlagos tárolóról, akkor a kiszolgáló személyzetből egy operátor megkeresi, és felteszi a szükséges szalagot a szalagolvasóra. Az információ megkeresése úgy történik, hogy a szalagot a megfelelő pozícióra tekerjük, és az információt a szalagról bemásoljuk a másodlagos vagy a központi memóriába. A harmadlagos tárolóra írás ennek a megfordítottja, azaz a megfelelő szalagon a megfelelő helyet keressük meg, és a lemezről kimásoljuk az információt a szalagra.

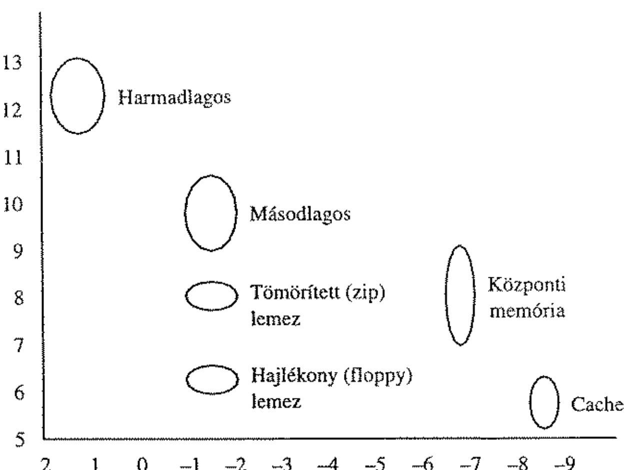
2. *Többlemezes optikai lemeztár* (juke box[4]). A lemeztár CD-ROM-tárolókból áll. (CD = kompaktlemez [compact disk]; ROM = csak olvasható memória [read-only memory]. Ezek azok az optikai lemezek, melyeken általában a szoftvereket forgalmazzák.) Az optikai lemezeken a biteket fekete vagy fehér kis területek reprezentálják, így a biteket úgy lehet elolvasni, hogy lézerrel megvilágítjuk az adott helyet, és megnézzük, hogy a fény visszaverődik-e. Egy robotkar is része a lemeztárnak, amely gyorsan ki tudja emelni bármelyik CD-ROM-ot, és be tudja helyezni egy CD-olvasóba. A CD tartalmát vagy annak egy részét így be lehet olvasni a másodlagos memóriába. Speciális eszközök nélkül általában nem lehet írni a CD-kre. Már kaphatók viszonylag olcsó CD-írók, és valószínűleg hamarosan gazdaságos lesz olyan harmadlagos tárolókat készíteni, amelyek írni és olvasni is tudják az optikai lemezeket.
3. *Szalagsilók.* A siló egy szoba nagyságú eszköz, mely szalagtárolókat tartalmaz. A szalagokat robotkarokkal lehet elérni, melyek aztán a kiválasztott szalagot a szalagolvasók valamelyikéhez tudják vinni. Ekképpen a siló a korábbi ad hoc szalagtárolásnak egy automatizált formája. Mivel a berendezés számítógéppel vezérelt és a szalag megkeresése is automatikus, ezért a működés legalább egy nagyságrenddel gyorsabb, mint az emberi személyzettel működtetett ad hoc rendszeré.

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

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

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

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



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

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

2.1.6. Felejtő és nem felejtő tárolás

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

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

A központi memória viszont általában felejtő típusú. Kiderült, hogy egy memóriachipet egyszerűbb áramkörökből lehet megtervezni, ha az is megengedett, hogy egy bit értéke bizonyos idő, például egy perc múlva megváltozzon. Ez az egyszerűsítés csökkenti a chip egy bitjére eső költségét. Tulajdonképpen az történik, hogy a bitet reprezentáló elektromos töltés lassan elfolyik abból a tartományból, mely ennek a bit-

nek volt kijelölve. Emiatt egy úgynevezett *dinamikus véletlen hozzáférésű memóriachipre*, DRAM-ra (dynamic random-access memory) van szükség, amely periodikusan olvassa és újraírja a teljes tartalmát. Ha kimegy az áram, akkor ez a frissítés nem működik, és a chip hamarosan elveszti, amit tárolt.

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

2.1.7. Feladatok

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

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

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

2.2. Lemezek

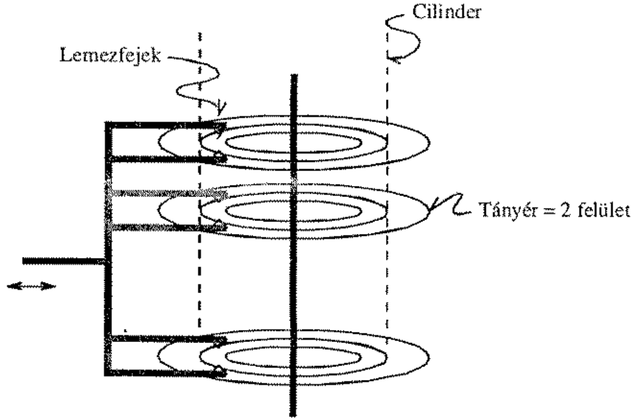
Az adatbázis-kezelő rendszerek egyik fontos jellemzője a másodlagos tárolók használata. A másodlagos tárolás szinte kizárólag mágneses lemezeken alapul. Így tehát ahhoz, hogy az adatbázis-kezelő rendszerek implementálásában használt ötleteket megindokolhassuk, először meg kell vizsgálnunk részletesen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

2.2.1. A lemezek mechanikája

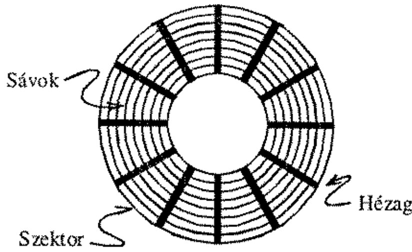
A 2.4. ábrán egy lemezmeghajtó két alapvető fontosságú mozgó részét láthatjuk, ezek a *lemezgyűjtemény* és a *fejszerelvény*. A lemezgyűjtemény egy vagy több kör alakú *tányérból* áll, melyek egy központi tengely körül forognak. A tányérok felső és alsó felülete is vékonyan be van vonva egy mágneses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

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

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



2.4. ábra. *Egy tipikus lemez*



2.5. ábra. *Egy lemez felszínének felülnézete*

Szektorok kontra blokkok

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

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

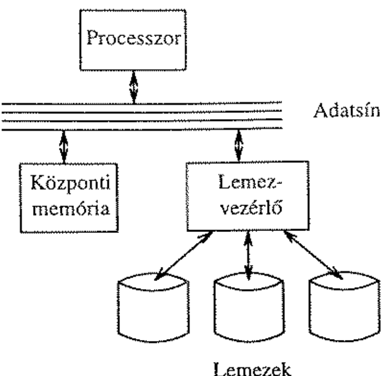
A 2.4. ábrán látható második mozgatható darab a fejszerelvény, amely a *lemezfejeket* tartja, áll. Mindegyik felülethez egy fej tartozik, mely igen-igen közel kerül a felülethez, de sohasem érintkezik vele (mert ha belezuhan a fej, akkor a lemez tönkremegy, és minden elvész, amit rajta tároltunk). A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

2.2.2. A lemezvezérlő

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

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

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



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

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

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

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

2.2.3. A lemeztárolók jellemzői

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

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

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

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

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

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

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

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

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

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

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

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

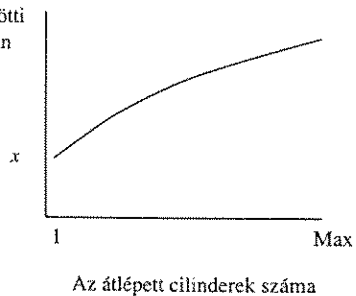
2.2.4. A lemezhozzáférés jellemzői

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

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

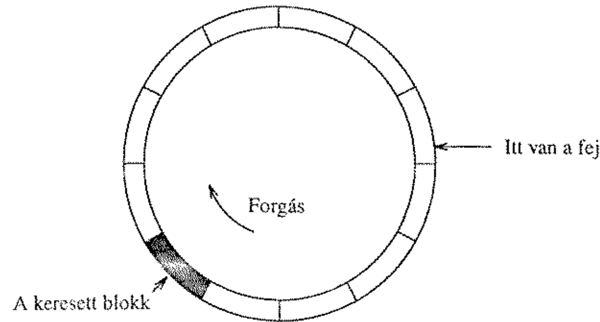
A blokkolvasási parancs kiadásának időpontja és a blokk tartalmának központi memóriába kerülésének időpontja közt eltelt időt a lemez ***késésének*** (latency) hívjuk. Ez a következő komponensekből áll össze:

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



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

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



2.8. ábra. *A rotációs késés oka*

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

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

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

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

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

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

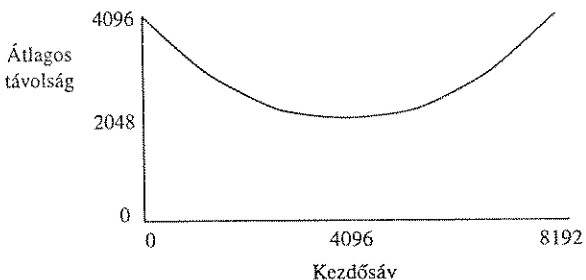
$$36 \times \frac{7}{256} + 324 \times \frac{8}{256} = 11{,}109.$$

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

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

Irányzatok a lemezvezérlők felépítésében

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



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

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

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

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

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

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

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

2.2.5. Blokkok írása

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

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

2.2.6. Blokkok módosítása

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

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

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

---

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

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

2.2.7. Feladatok

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

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

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

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

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

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

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

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

2.3. A másodlagos tárolók hatékony használata

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

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

2.3.1. A számítás I/O-modellje

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

Fel fogjuk tenni, hogy a lemez *Megatron 747* típusú, 4 Kbájt a blokkméret, és az időtényezők megegyeznek a 2.3. példában meghatározott értékekkel. Speciálisan az átlagos blokkírási vagy olvasási idő körülbelül 15 milliszekundum. Mivel sok fel-

használó van, és minden felhasználó rendszeresen lemez I/O-igényeket ad ki, ezért a lemezvezérlőnek gyakorta az igények sorát kell kielégíteni. A kiszolgálásról kezdetben azt tesszük fel, hogy mindig az elsőnek beérkezett igényt szolgálja ki először a lemezvezérlő. Ennek a stratégiának az a következménye, hogy egy adott felhasználó minden igénye véletlenszerűnek fog tűnni (vagyis a lemez feje véletlenszerű helyen fog állni az igény előtt), még akkor is, ha ez a felhasználó csak egyetlen relációhoz tartozó blokkokat olvas, és ez a reláció ráadásul a lemez egyetlen cilinderén van eltárolva. Ebben a fejezetben azt is megvizsgáljuk, hogyan lehet különböző módszerekkel a rendszer működését javítani. A *számítás I/O-modelljére* vonatkozó következő szabályt azonban végig igaznak tételezzük fel:

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

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

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

2.3.2. Adatok rendezése a másodlagos tárolóban

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

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

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

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

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

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

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

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

2.3.3. Az összefésülő rendezés (Merge-Sort)

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

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

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

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

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

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

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

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

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

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

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

2.3.4. Kétfázisú, többutas, összefésülő rendezés

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

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

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

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

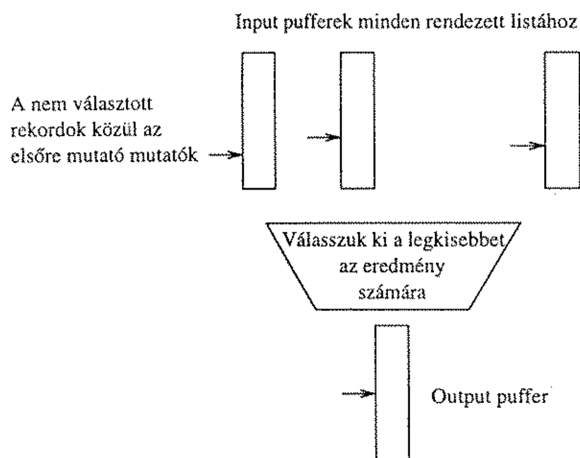
Az ily módon megadott *első fázis* végén az eredeti reláció összes rekordját egyszer olvastuk be a központi memóriába, minden rekord egy központi memória méretű rendezett részlistának lesz része, és ezeket a rendezett részlistákat kiírtuk a lemezre.

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

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

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

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



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

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

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

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

2.3.5. A többutas összefésülés kiterjesztése nagyobb relációkra

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

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

Milyen nagynak kell egy blokknak lennie?

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

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

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

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

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

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

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

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

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

2.3.6. Feladatok

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

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

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

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

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

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

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

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

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

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

2.4. A másodlagos tároló hozzáférési idejének javítása

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

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

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

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

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

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

2.4.1. Az adatok cilinderes szervezése

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

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

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

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

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

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

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

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

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

2.4.2. Több lemez használata

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

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

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

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

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

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

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

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

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

2.4.3. Lemezek tükrözése

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

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

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

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

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

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

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

2.4.4. A lemez ütemezése és a lift algoritmus

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

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

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

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

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

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

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

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

2.12. ábra. *Hat blokkhozzáférési igény érkezési sorrendje*

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

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

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

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

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

A lift algoritmus tényleges késése

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

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

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

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

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

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

Várakozás két blokk közül az utolsóra

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

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

2.4.5. Korai beolvasás és nagy léptékű pufferezés

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

2.13. példa: Ahhoz, hogy egy példát lássunk a dupla pufferezés használatára, nézzük meg újra a kétfázisú, többutas, összefésülő rendezés második fázisát, amit a 2.3.4. részben vázoltunk. Emlékezzünk vissza arra, hogy úgy fésültünk össze 20 rendezett részlistát, hogy minden listáról egy blokkot hoztunk be a központi memóriába. Ha annyi rendezett részlistát kell összefésülni, hogy a listákról behozott blokkok teljesen kitöltik a központi memóriát, akkor nem tudunk semmit sem jobbá tenni. Igen ám, de példánkban bőséges mennyiségű memória maradt meg. Például megtehetjük, hogy minden listához kétblokkos puffert rendelünk, és az egyik puffert feltöltjük, amíg a másikból az összefésüléshez válogatjuk a rekordokat. Ha kimerül az egyik puffer, akkor késlekedés nélkül átkapcsolunk ugyannak a listának a másik pufferére. □

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

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

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

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

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

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

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

2.4.6. A stratégiák előnyeinek és hátrányainak összegzése

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

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

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

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

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

Cilinder alapú szervezés

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

Több lemez használata

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

Tükrözés

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

Lift algoritmus

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

Korai beolvasás/Dupla pufferezés

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

2.4.7. Feladatok

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

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

2.15. ábra. *Négy blokkelérési igény beérkezési ideje*

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

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

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

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

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

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

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

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

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

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

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

2.5. Lemezhibák

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

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

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

2.5.1. Ideiglenes meghibásodás

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

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

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

2.5.2. Ellenőrző összegek

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

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

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

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

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

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

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

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

2.5.3. Stabil tárolás

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

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

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

A stabil tárolás olvasási elve a következő:

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

2.5.4. A stabil tárolás hibakezelő képessége

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

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

2.5.5. Feladatok

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

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

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

2.6. Lemezhiba helyreállítása

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

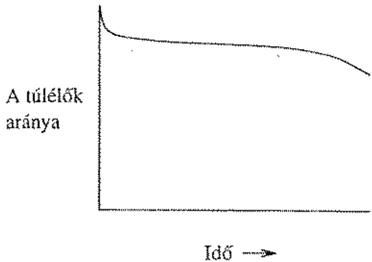
Sok különböző sémát fejlesztettek ki, hogy csökkentsék a lemezhibából származó adatveszteséget. Ezek általában redundanciát okoznak, például a 2.5.2. részben tárgyalt paritás-ellenőrzés ötletének kiterjesztésével, vagy a 2.5.3. részben leírtakhoz hasonló megduplázott szektorok használatával. Ezeknek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

2.6.1. A lemezek meghibásodási modelljei

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

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

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



2.16. ábra. *Lemezekre vonatkozó meghibásodási ráta görbéje*

latból eredő kopás, a parányi porszemcsék összegzett hatása) növeli a meghibásodás esélyét.

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

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

2.6.2. A tükrözés mint redundanciatechnika

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

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

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

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

2.6.3. Paritásblokkok

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

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

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

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

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

4. lemez: 01100010

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

Olvasás

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

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

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

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

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

1. lemez: 11110000

ami pontosan megegyezik az első lemez első blokkjával. □

Írás

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

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

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

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

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

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

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

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

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

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

A modulo-2 összegzések algebrája

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

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

Hibajavítás

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

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

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

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

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

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

2.6.4. Egy továbbfejlesztés: az 5. szintű RAID

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Olvasás

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

Írás

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

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

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

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

2.18. ábra. *A hét lemez első blokkja*

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

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

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

Hibajavítás

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

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

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

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

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

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

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

2.6.6. Feladatok

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

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

a) Ha tükrözött lemezeket használunk, akkor F és H függvényében mennyi az adatvesztési idő várható értéke?

b) Ha 4. vagy 5. szintű RAID-sémát használunk N számú lemezzel, akkor mennyi az adatvesztési idő várható értéke?

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

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

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

További észrevételek a 6. szintű RAID-del kapcsolatban

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

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

* a) 01010110, 11000000, 00111011 és 11111011.

b) 11110000, 11111000, 00111111 és 00000001.

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

* a) A 2., 3. és 4. lemez tartalma 01010110, 11000000 és 00111011, a redundáns lemez tartalma pedig 11111011.

b) A 2., 3. és 4. lemez tartalma 11110000, 11111000 és 00111111, a redundáns lemez tartalma pedig 00000001.

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

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

a) Mik a redundáns lemezek megfelelő blokkjai?

Hibajavító kódok és a 6. szintű RAID

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

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

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

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

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

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

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

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

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

a) Fejezzük ki ezt a helyzetet azzal, hogy megadunk egy 2.17. ábrához hasonló paritás-ellenőrző mátrixot.

!! b) Bizonyos esetekben, de nem az összes lehetséges esetben, ez a rendszer is képes a helyreállításra, ha két lemez egyszerre megy tönkre. Határozzuk meg, hogy milyen párokra lehetséges a helyreállítás, milyen párokra nem lehetséges.

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

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

2.7. Összefoglalás

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

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

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

2.8. Irodalomjegyzék

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

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

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

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

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

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

3. fejezet

Adatelemek ábrázolása

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

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

3.1. Adatelemek és mezők

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

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

3.1.1. Relációs adatbáziselemek ábrázolása

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

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

3.1. ábra. *Egy SQL-tábla deklarálása*

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

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

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

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

3.1.2. Objektumok ábrázolása

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

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

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

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

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

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

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

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

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

3.1.3. Adatelemek ábrázolása

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

Rögzített hosszú karakterláncok

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

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

```
sas⊥⊥
```


Egy megjegyzés a terminológiával kapcsolatban

A fájlrendszerekben, hagyományos C-hez hasonló programozási nyelvekben, relációs adatbázisnyelvekben (különös tekintettel az SQL-re) vagy az objektumorientált nyelvekben (például Smalltalk, C++ vagy objektumorientált adatbázis-nyelv, 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 OO:PP:MM (óra:perc:másodperc) formátumú 8 karakteres láncként ábrázolja. Tehát az első két karakter az órát ábrázolja. Ennek értéke egy egész szám 0-tól 23-ig úgy, hogy az egyjegyű számokat egy 0 előzi meg. A délelőtt 7 órát tehát a 07, az esti 7 órát pedig a 19 számjegyekkel ábrázoljuk. A kettőspont utáni két számjegy a perceket ábrázolja,

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

Több mező egyetlen bájtba csomagolása

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

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

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

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

Bitek

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

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

Felsorolási típusok

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

Egy felsorolási típus értékeit egész kódokkal ábrázolhatjuk, de csak annyi bájtot használunk, amennyi feltétlenül szükséges. Például a `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.

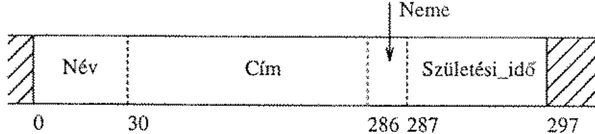
3.2.1. Rögzített hosszú rekordok építése

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.

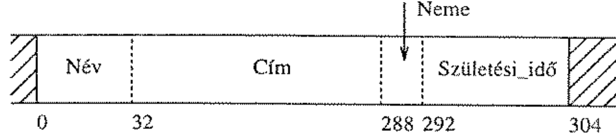
Igaz, hogy egy reláció sorait a lemezen tároljuk és nem a központi memóriában, de azért jó, ha tudunk erről. Ugyanis mikor egy blokkot a lemezről beolvasunk a központi memóriába, akkor a blokk első bájtja a központi memóriában biztosan egy olyan címen lesz, mely 4-nek többszöröse. Általában nem csak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

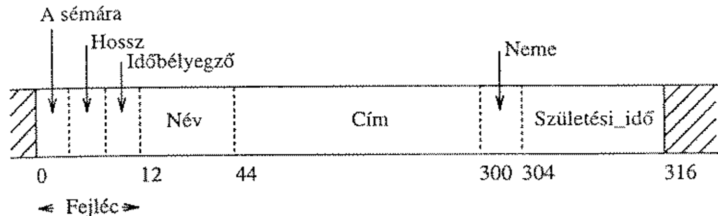
Az adatbázisrendszer tartja karban a *sémainformációt* (schema information). A sémainformáció lényegében az, amit a `CREATE TABLE` utasításban egy relációhoz megadunk:

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.

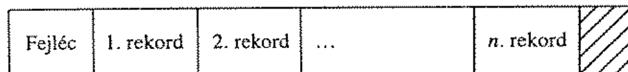
3.7. példa: Módosítsuk a 3.6. példában megadott rekordfelosztást úgy, hogy egy 12 bájtos fejlécet is tartalmazzon a rekord. Az első négy bájt a típus. Ez tulajdonképpen egy eltolási érték egy olyan területen, ahol az összes reláció sémáját tároljuk. A második a rekord hossza, egy 4 bájtos egész szám,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

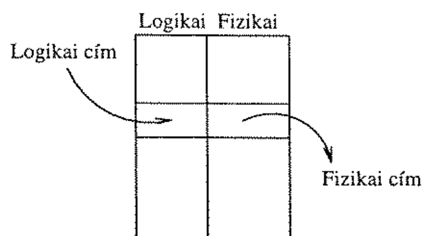
A szerver adatai az *adatbázis címterületén* (database address space) léteznek. Ennek a területnek a címei blokkokra vagy esetleg blokkon belüli eltolási értékekre vonatkoznak. Több mód is kínálkozik arra, ahogy ennek a címterületnek a címeit ábrázoljuk:

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 objektumokat 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.



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

3.3.2. Logikai és strukturált címek

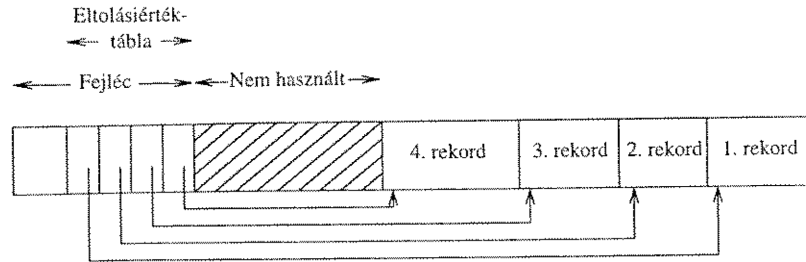
Most már biztos kíváncsiak vagyunk, hogy mire is kellhetnek a logikai címek. Minden olyan információ, ami szükséges egy fizikai címhez, a leképezési táblában található. Így ahhoz, hogy a logikai mutatókat követve eljussunk a rekordokhoz, először meg kell vizsgálnunk a leképezési táblát, és az innen kiolvasott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

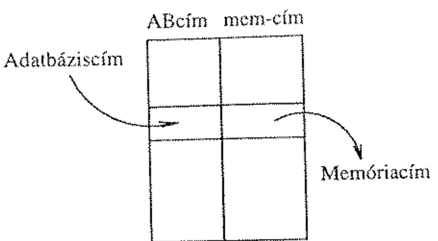
A mutatók és címek gyakran a rekordoknak részét képezik. Ez a helyzet általában nem azokra a rekordokra jellemző, melyek egy reláció sorait ábrázolják, hanem azokra a sorokra, amelyek objektumokat ábrázolnak. A modern objektumrelációs adatbázisrendszerekben megengedett a mutató típusú attribútumok használata 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 puffereltük, akkor van egy címe a virtuális memóriában is. Ezek a címek tipikusan négybájtosak. Az ilyen címeket az adattétel *memóriacímének* (memory address) hívjuk.

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

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



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

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

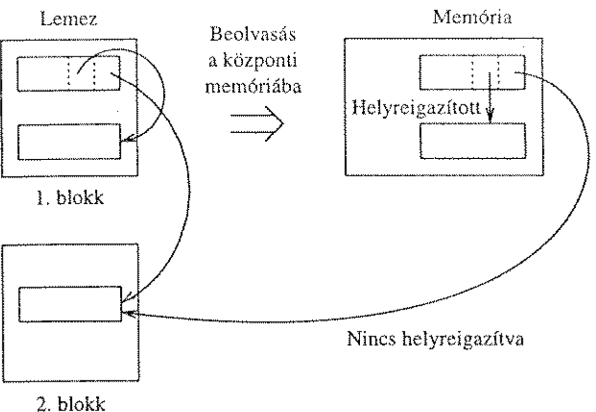
a) A logikai és a fizikai címek egyaránt az adatbáziscím reprezentációi. Ezzel szemben a fordítási táblában található memóriacímek a memóriába másolt megfelelő objektumra vonatkoznak.
b) Az adatbázis minden címezhető adattételéhez tartozik bejegyzés a leképezési táblában, míg a fordítási táblában csak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

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

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

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

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



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

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

Számos stratégia használható a helyreigazító mutató meghatározására. □

Automatikus helyreigazítás

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

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

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

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

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

Igény szerinti helyreigazítás

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

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

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

Nincs helyreigazítás

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

Programozó által vezérelt helyreigazítás

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

3.3.4. Blokkok visszaírása a lemezre

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

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

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

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

Ha az ellentétes lekérdezést is támogatni akarjuk,

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

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

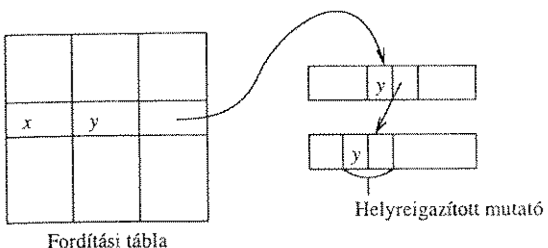
3.3.5. Feltűzött rekordok és blokkok

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

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

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

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



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

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

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

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

3.3.6. Feladatok

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.4. Változó hosszú adatok és rekordok

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

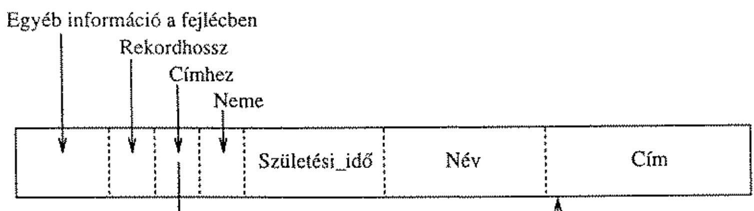
1. *Változó méretű adattételek.* Például a 3.1. ábrán láttunk egy `Filmszínész` relációt, amelynek van egy cím mezője, és ennek a hossza 255 bájtig bármekkora lehet. Noha lehetnek bizonyos címek ilyen hosszúak, de azért a címek nagy többsége valószínűleg 50 bájtos lesz vagy még rövidebb. Valószínűleg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

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

3.4.1. Változó hosszú mezőket tartalmazó rekordok

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

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



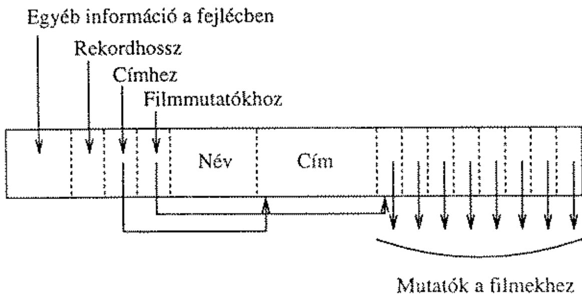
3.12. ábra. *Egy* Filmszínész *rekord, melyhez tartozó* név *és* cím *mezőket változó hosszú karakterláncokkal valósítjuk meg*

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

3.4.2. Ismétlődő mezőket tartalmazó rekordok

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

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



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

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

Nullértékek ábrázolása

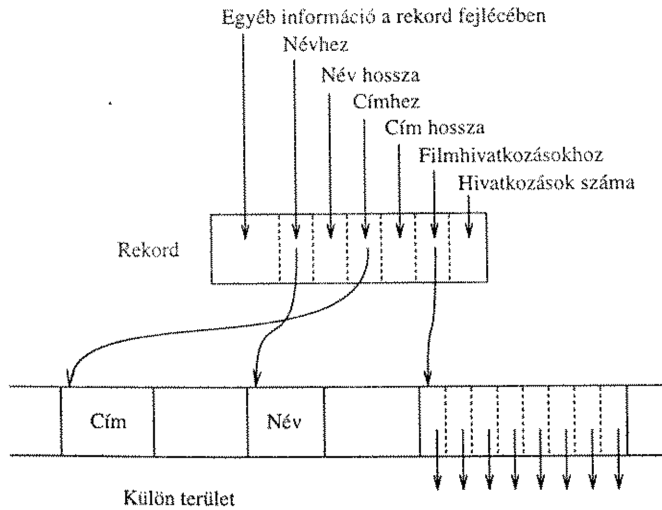
A soroknak gyakran lehetnek olyan mezőik, amelyek `nullértéket (NULL)` is tartalmazhatnak. A 3.12. ábra rekordformátuma egy kényelmes módot nyújt ahhoz, hogyan ábrázoljuk a `nullértékeket`. Ha egy mező, például a `cím` nullértékű, akkor nullmutatót (null pointer) teszünk arra a helyre, ahol a címre hivatkozó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

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

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

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

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



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

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

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

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

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

3.4.3. Változó formátumú rekordok

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

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

Legalább két okból értelmes a címkézett mezők használata.

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

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

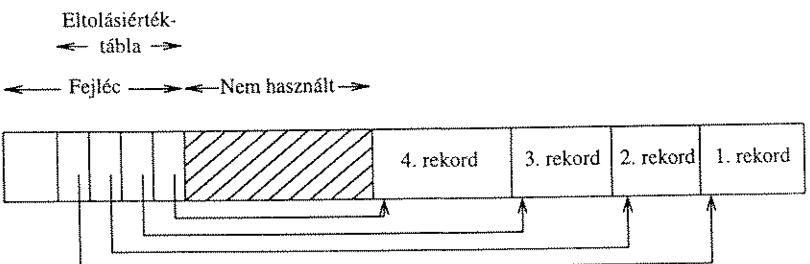
3.5. Rekordmódosítások

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

3.5.1. Beszúrás

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

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



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

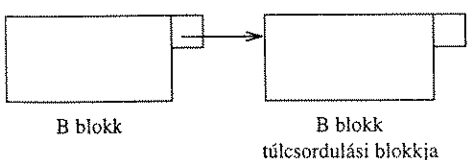
ahogy ezt majd a 4.1. részben látni fogjuk. Ha be kell szúrnunk egy új rekordot, akkor először meg kell keresnünk a rekordnak megfelelő blokkot. Ha véletlenül van még hely ebben a blokkban, akkor betesszük ide a rekordot. Mivel a blokkokat rendezve tároljuk, ezért lehet, hogy a blokkon belül a rekordokat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

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

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

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

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



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

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

3.5.2. Törlés

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

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

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

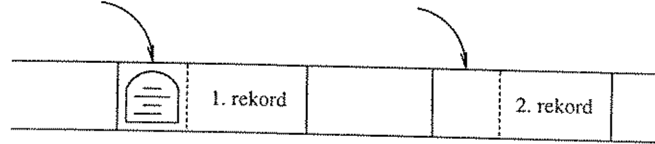
Van azonban még egy probléma a törléssel kapcsolatban, amire mindig figyelnünk kell, függetlenül attól, hogy milyen sémát használunk a blokkok szervezéséhez. Létezhetnek olyan mutatók, amelyek a törölt rekordra mutatnak. Ha vannak ilyen mutatók, akkor nem szeretnénk, hogy ezek a mutatók a törlés után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

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

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

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

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



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

3.5.3. Módosítás

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

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

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

3.5.4. Feladatok

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

i) Ha túlcsordulás fordul elő, akkor vágjuk ketté a blokkot, és állítsuk be megfelelően a blokkhoz tartozó, rendezési kulcsra vonatkozó tartományokat.
ii) Tároljuk az egy blokkhoz tartozó rendezési kulcsok tartományát, és szükség esetén használjunk túlcsordulási blokkokat. Minden blokkhoz és túlcsordulási blokkhoz tároljunk egy eltolásiérték-táblát, mely az abban a blokkban szereplő rekordokra vonatkozik.
iii) Ugyanaz, mint ii), de most egy blokkhoz és az összes túlcsordulási blokkjához egy eltolásiérték-táblát tároljunk, méghozzá az első blokkban (vagy a túlcsordulási blokkokban, ha az eltolásiérték-táblához több hely szükséges). Megjegyezzük, hogy ha több hely kell az eltolásiérték-tábla számára, akkor az első blokkból átmozgathatunk rekordokat egy túlcsordulási blokkba, hogy több helyet teremtsünk.
iv) Ugyanaz, mint ii), de most egy mutatóval együtt a rendezési kulcsot is tároljuk az eltolásiérték-táblában.
v) Ugyanaz, mint iii), de most egy mutatóval együtt a rendezési kulcsot is tároljuk az eltolásiérték-táblában.

Válaszoljuk meg a következő kérdéseket:

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

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

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

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

3.6. Összefoglalás

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

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

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

3.7. Irodalomjegyzék

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

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

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

4. fejezet

Indexstruktúrák

Az eddigiekben láthattuk, hogy milyen lehetőségek állnak rendelkezésre a rekordok tárolására, nézzük most meg, hogy miként lehet tárolni teljes relációkat vagy 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:

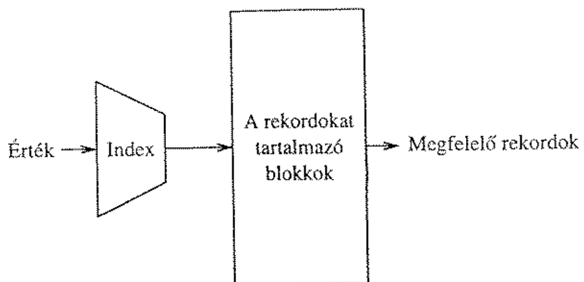
1. A blokkok fejléceiben tárolt információ arról, hogy hol kezdődnek az adott blokkban a rekordok.
2. A rekordok fejléceiben tárolt információ arról, hogy az adott rekord melyik relációhoz tartozik.

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.

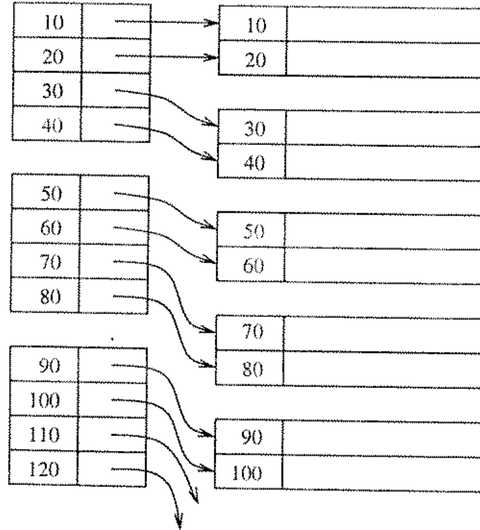
Kulcsok és még mindig kulcsok

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.

10	
20	
30	
40	
50	
60	
70	
80	
90	
100	

4.2. ábra. *Egy szekvenciális fájl*



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

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

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

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

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

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

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

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

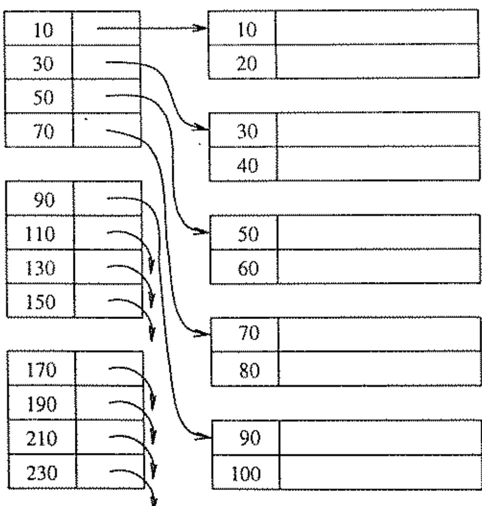
Indexblokkok helyének meghatározása

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

módon, ha nem is engedhetjük meg magunknak, hogy a teljes indexet a memóriában tartsuk, de a legfontosabb indexblokkok beférnek a memóriába, nos, még akkor is jelentősen kevesebb mint 14 lemez I/O-művelettel megtalálhatunk egy adott kulcsértékkel rendelkező rekordot. □

4.1.3. Ritka indexek

Ha a sűrű index túl nagy, akkor használhatjuk a *ritka indexnek* (sparse index) nevezett hasonló adatszerkezetet, amely kevesebb helyet foglal, de ennek az az ára, hogy valamivel több időt vesz igénybe egy adott kulcsértékkel rendelkező rekord megtalálása.



4.4. ábra. *Ritka index szekvenciális fájlon*

Egy ritka index egy adatblokkhoz csak egy kulcs-mutató párt tartalmaz, amint azt a 4.4. ábrán is láthatjuk. A kulcs az adatblokk első rekordjának a kulcsa.

4.3. példa: Ahogyan azt a 4.1. példában is tettük, feltételezzük, hogy az adatfájl rendezett és a kulcsok a tíz összes többszörösei valamely nagy számig bezáróan. Tegyük fel továbbá, hogy négy kulcs-mutató pár fér el egy indexblokkban. Ily módon az első indexblokk olyan bejegyzéseket tartalmaz, amelyek az első négy adatblokk első kulcsaihoz tartoznak, jelesül a 10-es, 30-as, 50-es és 70-es értékekhez. A második indexblokk a negyediktől nyolcadikig terjedő adatblokkok első kulcsaihoz tartozó bejegyzéseket tartalmazza, azaz, a kulcsok feltételezett szabályszerűségét folytatva, a 90-es, 110-es, 130-as és 150-es kulcsértékeket. Feltüntettük a harmadik indexblokkot is, amely a feltételezett kilencediktől tizenkettedikig terjedő adatblokkok első kulcsértékeit tartalmazza. □

4.4. példa: Egy ritka index sokkal kevesebb blokkot igényelhet, mint egy sűrű index. Ha a 4.2. példa sokkal életszerűbb paramétereit használjuk, azaz, hogy adott 100 000 adatblokk és 100 kulcs-mutató pár fér el egyetlen indexblokkban, akkor ritka index használata esetén mindössze 1000 indexblokkra van szükségünk. Ekkor az index mindössze négy megabájtot foglal le, ami jó eséllyel elhelyezhető majd az elsődleges memóriában.

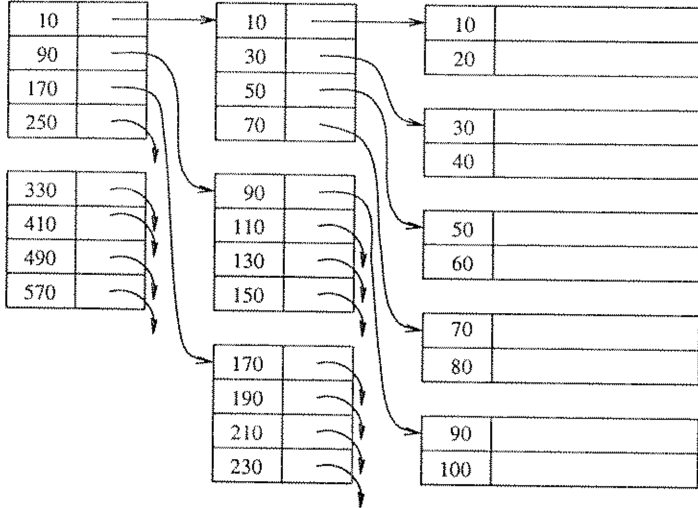
Másrészről azonban, egy sűrű index lehetővé teszi olyan típusú lekérdezések megválaszolását, mint „létezik-e *K* kulcsértékkel rendelkező rekord?" anélkül, hogy a rekordot tartalmazó blokkhoz hozzá kellene férni. Az a tény, hogy a *K* a sűrű indexben előfordul, garantálja a *K* kulcsértékű rekord létezését. Ugyanez a lekérdezés azonban ritka index használatával szükségessé tesz egy lemez I/O műveletet, amellyel beolvassuk azt a blokkot, amelyben a *K előfordulhat*. □

Ritka index esetén, egy *K* kulcsértékkel rendelkező rekord megtalálásához megkeressük azt a legnagyobb indexértéket, amely kisebb vagy egyenlő, mint a keresett *K*. Mivel az indexfájl rendezett a kulcs szerint, ismét használhatunk bináris keresést a megfelelő bejegyzés megtalálásához. Követjük az adatblokkra irányuló mutatót. Ekkor meg kell találnunk ebben a blokkban a *K* kulcsértékű rekordot. Természetesen a blokknak rendelkeznie kell elegendő információval a formátumra vonatkozóan ahhoz, hogy a rekordokat és azok tartalmát azonosítani lehessen. A 3.2. és 3.4. részekben bemutatott technikák bármelyikét használhatjuk, a helyzettől függően.

4.1.4. Többszintű indexelés

Amint azt a 4.2. és 4.4. példákban is láthattuk, egy index több blokkot is elfoglalhat. Ha ezek a blokkok nem egy előre rögzített helyen találhatók, például a háttértároló bizonyos cilinderein, akkor külön adatszerkezetre van szükségünk ahhoz, hogy megtaláljuk őket. Ha meg is tudjuk határozni az indexblokkok helyét, és bináris kereséssel a kívánt bejegyzést meg is találjuk, még akkor is igen sok lemez I/O-műveletre lehet szükség ahhoz, hogy a keresett rekordhoz hozzáférjünk.

Ha az indexre újabb indexet készítünk, akkor az első szintű index használatát még hatékonyabbá tehetjük. A 4.5. ábra kiterjeszti a 4.4. ábrát azzal, hogy egy második indexszintet ad hozzá (továbbra is feltételezzük, hogy a kulcsok a 10 többszörösei). Hasonló ötlettől vezérelve készíthetünk egy harmadik szintű indexet is a második szintre és így tovább. Ennek az ötletnek azonban megvannak a maga korlátai, így inkább a 4.3. részben ismertetett B-fákat részesítjük előnyben a többszintű indexek készítésekor.



4.5. ábra. *Második szintű ritka index készítése*

Ebben a példában az első szintű index ritka, habár választhattunk volna sűrű indexet is. A második és annál magasabb szintű indexek azonban kötelezően ritkák. Ennek oka az, hogyha az indexre egy sűrű indexet készítenénk, az ugyanannyi kulcs-mutató párt tartalmazna, mint az első szintű index, ezáltal ugyanakkora helyet is foglalna, mint az első szintű index. Ily módon a második szintű index egy újabb, de haszontalan adatszerkezet lenne csupán.

4.5. példa: Folytassuk a 4.4. példában használt reláció vizsgálatát. Tegyük fel, hogy készítünk egy második szintű indexet az első szintű ritka indexre. Az első szintű index 1000 blokkot foglal el és 100 kulcs-mutató pár fér el egy blokkban, ily módon a második szintű indexhez mindössze 10 blokkra van szükség.

Igen valószínű, hogy ez a 10 blokk elfér a memóriapufferben. Ha ez így van, akkor adott K kulcsértékű rekord megtalálásához a második szintű indexben megkeressük azt a legnagyobb kulcsértéket, ami kisebb vagy egyenlő, mint K. A megtalált mutatóval eljutunk az első szintű index egy olyan B blokkjához, amely minden bizonnyal elvezet a keresett rekordhoz. A B blokkot beolvassuk a memóriába, feltéve, hogy még nem olvastuk be. Ez az első lemez I/O-művelet. A B blokkban megkeressük azt a legnagyobb kulcsértéket, ami kisebb vagy egyenlő, mint K, és a megtalált kulcsérték megadja nekünk azt az adatblokkot, amely tartalmazza a K kulcsértékű rekordot, persze csak akkor, ha egyáltalán létezik ilyen rekord. Az adatblokk beolvasásához szükséges egy újabb lemez I/O-művelet. Ily módon mindössze két I/O-műveletet használtunk, és készen is vagyunk. □

4.1.5. Indexelés ismétlődő keresésikulcs-érték esetén

Mindeddig feltételeztük, hogy a keresési kulcs, amely az index alapját képezi, a reláció egy kulcsa, tehát adott kulcsértékhez legfeljebb egy rekord tartozhat. Gyakran használunk azonban indexeket nem kulcs attribútumokra is, így előfordulhat, hogy egy adott kulcsértékhez több mint egy rekord tartozik. Ha rendezzük a rekordokat a keresési kulcs szerint, az egyenlő kulcsértékkel rendelkező rekordokat tetszőleges sorrendben hagyva, akkor alkalmazhatjuk a korábbiakban bemutatott ötleteket olyan keresési kulcsokra is, amelyek nem kulcsai a relációnak.

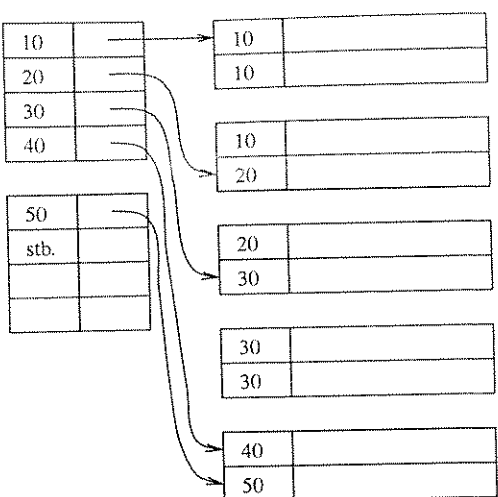
Az előző ötletek talán legegyszerűbb kiterjesztése az, ha olyan sűrű indexet készítünk, amelyben az adatfájl valamennyi K keresésikulcs-értékkel rendelkező rekordjához tartozik egy K kulcsot tartalmazó bejegyzés. Ezzel tulajdonképpen engedélyeztük az ismétlődő keresési kulcsokat az indexfájlban. Az adott keresésikulcs-értékkel rendelkező valamennyi rekord megtalálása ily módon igen egyszerű: megkeressük az indexfájlban az első K értéket, ezáltal megtaláljuk a többi K értéket is, hiszen rögtön az első után helyezkednek el. Ezután követjük a megtalált kulcsokhoz tartozó mutatókat, eljutva ezzel a K keresésikulcs-értékkel rendelkező rekordokhoz.

Némileg hatékonyabb az a megközelítés, amikor a sűrű indexben csak egy bejegyzés található valamennyi K keresési kulcshoz. A K kulcshoz olyan mutató tartozik, amely az első K értékkel rendelkező rekordra mutat. Ahhoz, hogy a többi K értékkel rendelkező rekordot is megtaláljuk, csupán el kell mozdulnunk előrefelé az adatfájlban, hiszen az adatfájl rendezett, és ezek a rekordok közvetlenül az első K érékű rekord után helyezkednek el. A 4.6. ábra ezt az ötletet mutatja be.

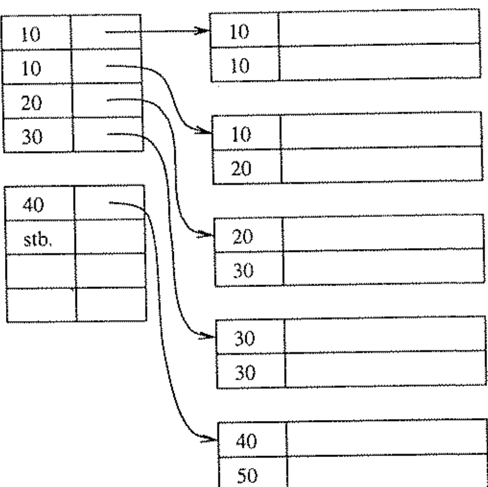
4.6. példa: Tegyük fel, hogy meg akarjuk találni a 4.6. ábrán az összes olyan rekordot, amelynek a keresési kulcsa 20. Megkeressük az indexben a 20-as értékhez tartozó bejegyzést, és követjük a hozzá tartozó mutatót, amely az első olyan rekordhoz vezet, amelyben a keresési kulcs értéke 20. Ezután elkezdünk előrefelé keresni az adatfájlban. Mivel a második blokk utolsó rekordján állunk, továbblépünk a harmadik blokkra.[1] Azt találjuk, hogy ennek a blokknak az első rekordja tartalmazza a 20-as kulcsértéket, a második rekord kulcsa azonban 30. Ezért aztán nem szükséges tovább keresni, megtaláltuk a 20-as kulcsértékkel rendelkező mindkét rekordot. □

---

[1] Ahhoz, hogy az adatfájl következő blokkját megtaláljuk, felfűzhetjük a blokkokat egy láncolt listára, például úgy, hogy valamennyi blokk végén elhelyezünk egy olyan mutatót, amelyik a következő blokkra mutat. Visszaléphetünk azonban az indexhez is, és követhetjük azt a mutatót, amely az adatfájl következő blokkjára mutat.



4.6. ábra. *Sűrű index, az ismétlődő keresési kulcsok megengedettek*



4.7. ábra. *Ritka index, amely a blokkok legkisebb keresési kulcsát tartalmazza*

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

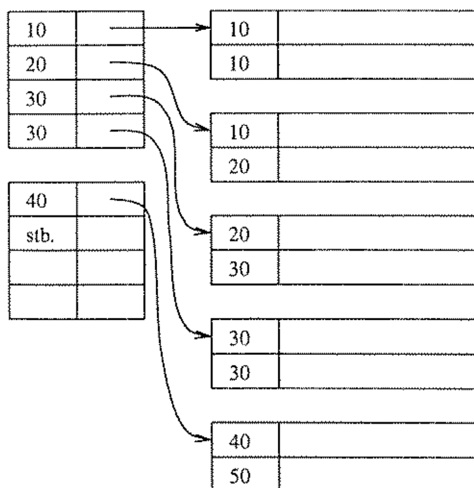
Ahhoz, hogy ezzel az adatszerkezettel megtaláljuk a K keresési kulccsal rendelkező rekordokat, meg kell keresnünk azt az utolsó bejegyzést az indexben, amelyben a

kulcs kisebb vagy egyenlő mint K. Nevezzük ezt a bejegyzést E_1-nek. Induljunk el az index eleje felé, és menjünk mindaddig, amíg el nem érünk az első bejegyzésig, vagy nem találunk egy olyan E_2 bejegyzést, amelyben a kulcs értéke szigorúan kisebb, mint K. Mindazon adatblokkok, amelyek tartalmazhatnak K keresési kulccsal rendelkező rekordot, elérhetők az E_2 és E_1 közötti bejegyzések mutatói segítségével, az E_1 bejegyzést is beleértve.

4.7. példa: Tegyük fel, hogy a 4.7. ábrán szemléltetett esetben szeretnénk megtalálni a 20-as kulcsértéket. Az első indexblokk harmadik bejegyzése az E_1; ez az utolsó olyan bejegyzés, ahol a kulcs ≤ 20. Amikor elkezdünk visszafelé keresni, rögtön találunk egy olyan bejegyzést, amelyben a kulcs kisebb, mint 20. Így, az E_2-nek az első indexblokk második bejegyzése felel meg. A bejegyzések két mutatója a második, illetve a harmadik adatblokkra mutat, és ez az a két blokk, amely tartalmaz 20-as keresésikulcs-értékkel rendelkező rekordokat.

Ha azonban például $K = 10$, akkor az E_1 az első indexblokk második bejegyzése, és az E_2 nem létezik, mivel nincs kisebb kulcsérték. Ily módon követjük az index valamennyi bejegyzésének mutatóit egészen a második bejegyzésig, azt is beleértve. Ezek a mutatók az első két adatblokkhoz vezetnek, ahol megtaláljuk az összes 10-es kulcsértékkel rendelkező rekordot. □

A 4.8. ábra egy némileg különböző helyzetet mutat be. Itt az egy adatblokkhoz tartozó indexbejegyzés a legkisebb *új* keresési kulcsot tartalmazza, azaz azt a legkisebb kulcsot, amely az előző blokkban nem szerepel. Ha egy blokkban nincs új keresési kulcs, akkor a hozzá tartozó indexbejegyzés az adott blokkban található keresési kulcs értékét tartalmazza. Ilyen feltételek mellett, a K keresésikulcs-értékkel rendelke-



4.8. ábra. *Ritka index, amely a blokkok legkisebb új keresési kulcsát tartalmazza*

ző rekordok megtalálásához meg kell keresnünk azt az első bejegyzést az indexben, amely:

a) egyenlő K-val, vagy
b) kisebb, mint K, de a következő kulcs nagyobb, mint K.

Követjük a bejegyzéshez tartozó mutatót, és ha a blokkban találunk legalább egy K keresésikulcs-értékkel rendelkező rekordot, akkor tovább keresünk előrefelé a következő blokkokban, egészen addig, amíg meg nem találjuk az összes K keresésikulcs-értékkel rendelkező rekordot.

4.8. példa: Tegyük fel, hogy a 4.8. ábrán látható esetben $K = 20$. A fenti szabály alapján az index második bejegyzését találjuk meg, amelynek mutatója elvezet minket az első olyan blokkhoz, amely tartalmaz 20-as kulcsértéket. Tovább kell keresnünk előrefelé, hiszen a következő blokkban is szerepel a 20.

Ha $K = 30$, a szabály alapján a harmadik bejegyzést találjuk meg. A bejegyzés mutatója a harmadik adatblokkhoz vezet, ahol a 30-as keresésikulcs-értéket tartalmazó rekordok kezdődnek. És végül, ha $K = 25$, akkor a kiválasztási szabály b) pontja alapján az index második bejegyzését találjuk meg. Ekkor a második adatblokkhoz jutunk. Ha léteznének olyan rekordok, amelyek keresési kulcsa 25, akkor ezen rekordok közül legalább az egyiknek ebben a blokkban kellene lenni a 20-as kulcsértékű rekordokat követve, hiszen tudjuk, hogy a harmadik adatblokk első új kulcsa 30. Mivel nincs 25-ös kulcsértékű rekord, keresésünk sikertelen. □

61

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.

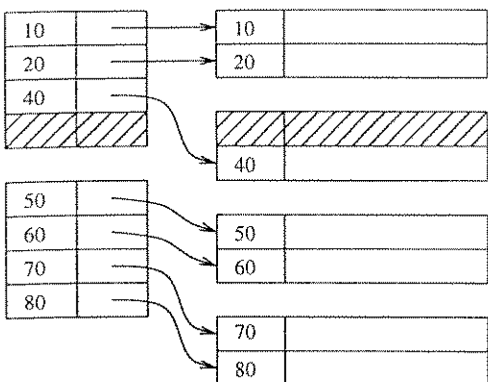
- Rekordok beszúrása és törlése éppolyan hatással van a sűrű indexre, mintha az adott rekordhoz egy kulcs-mutató párt szúrnánk be vagy törölnénk. Nincs azonban általában hatással a ritka indexre. Kivétel persze, ha az adott rekord éppen egy blokk első helyén áll, ilyenkor a ritka index megfelelő kulcsértékét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

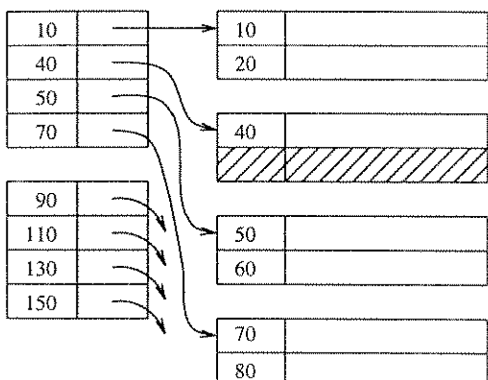
Először is töröljük a 30-as kulcsértéket tartalmazó rekordot a szekvenciális fájlból. Feltételezzük, hogy a blokk rekordjaira blokkon kívüli mutatók mutatnak, ily módon arra a döntésre jutottunk, hogy a blokk megmaradó rekordját nem csúsztatjuk előrébb a blokkban. Ehelyett inkább egy törlésre utaló jelet,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

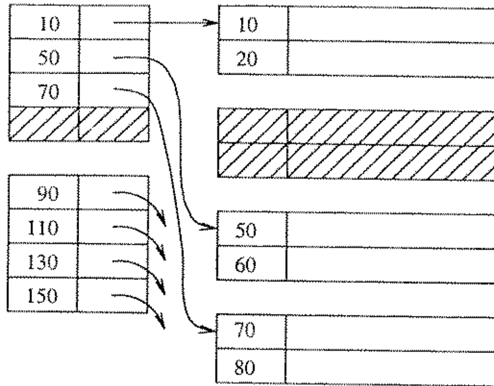


4.11. ábra. *A 30-as keresésikulcs-értékű rekord törlése ritka index esetén*

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

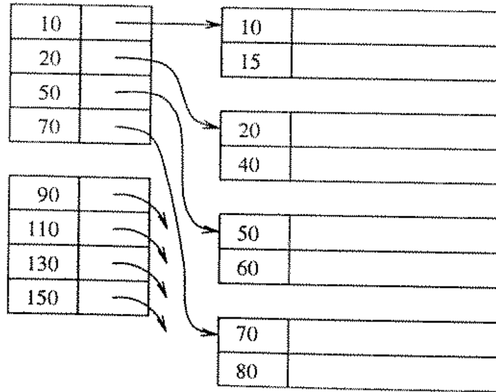
műveletnek az eredményét a 4.12. ábrán láthatjuk. A második adatblokkban nincs már egy rekord sem. Ha a szekvenciális fájl tetszőleges blokkokon van tárolva (és mondjuk nem egy adott cilinder egymás utáni blokkjain), akkor felfűzhetjük a nem használt blokkot a szabad helyek listájára.



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*

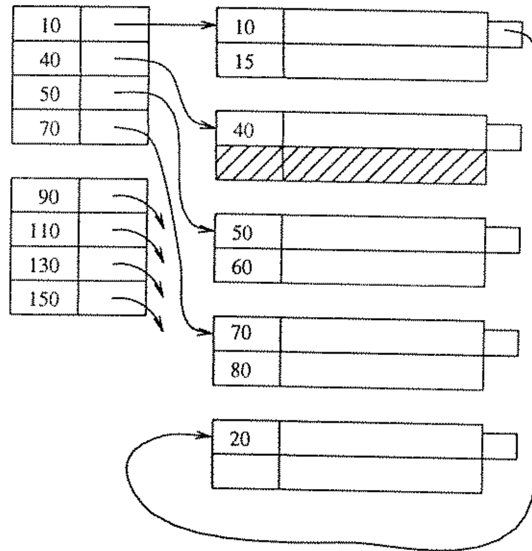
megmaradt. Most be fogunk szúrni egy 15-ös kulcsértékű rekordot. A ritka indexet megvizsgálva arra jutunk, hogy ez a rekord az első adatblokkhoz tartozik. Ez a blokk viszont tele van: a 10-es és 20-as rekordok vannak benne.

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);
```

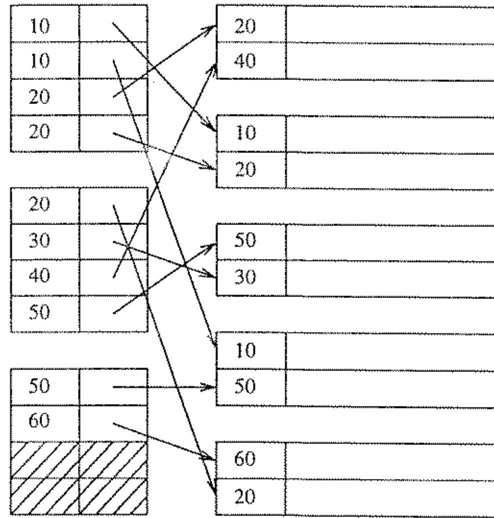
Egy másodlagos index megfelel bármely index céljainak: olyan adatszerkezet, amely megkönnyíti megadott értékű mezővel, illetve mezőkkel rendelkező rekordok megtalálását. A másodlagos index abban különbözik azonban az elsődleges indextől, hogy a másodlagos index nem határozza meg a rekordok adatfájlban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

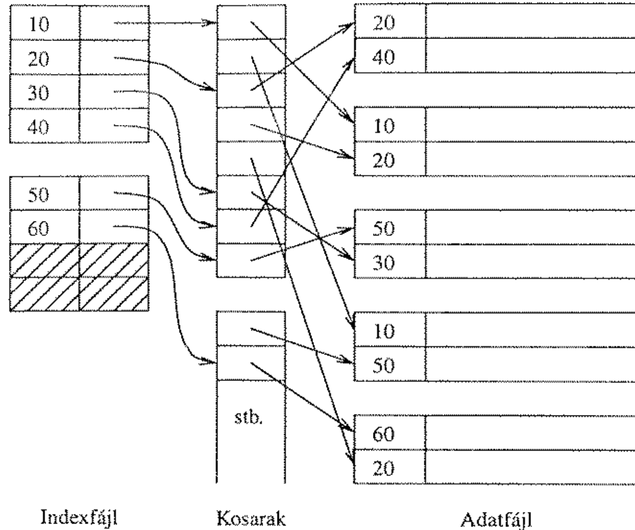
Ebben az esetben, ha annak a stúdiónak az elnökét keressük, amely egy adott filmet készített, jó esélyünk van arra, hogy a stúdióra és a filmre vonatkozó rekordok ugyanabban a blokkban találhatók, megtakarítva ezzel egy lemez I/O-műveletet. Ha azokat a filmeket keressük, amelyeket egy adott stúdió gyártott,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

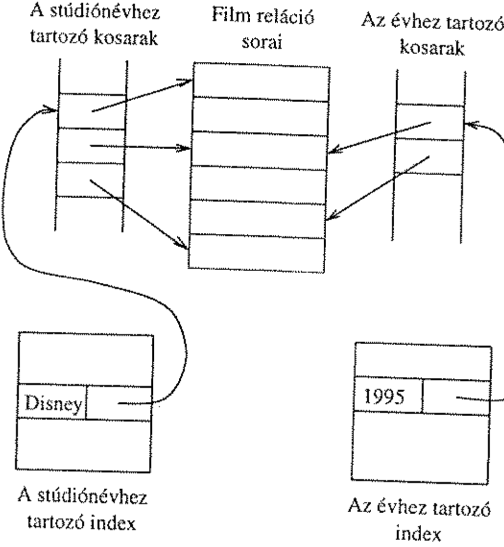
A 4.17. ábrán látható elrendezés mindaddig helymegtakarítást jelent, amíg a keresésikulcs-értékek több helyet foglalnak, mint a mutatók, és mindegyik kulcs átlagosan legalább kétszer megjelenik. A közvetett másodlagos indexek használatának akkor is van azonban egy fontos előnye, amikor a kulcsok és a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

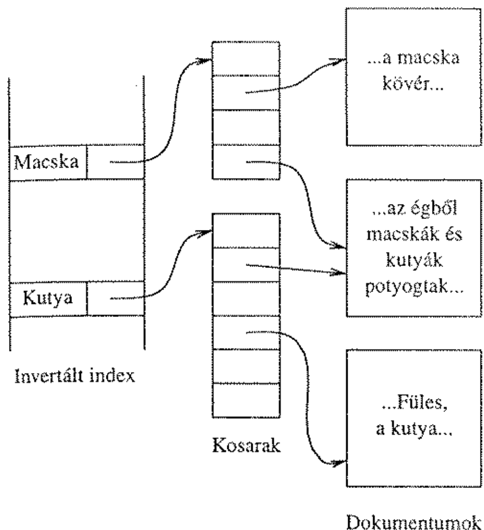
Az információszolgáltatók közössége több éve foglalkozik dokumentumok tárolásával és az adott kulcsszavakat tartalmazó dokumentumok hatékony visszakeresésével. A World Wide Web megjelenésével és a dokumentumok on-line elérhetőségének megvalósulásával az adott kulcsszavakat tartalmazó dokumentumok visszakeresése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

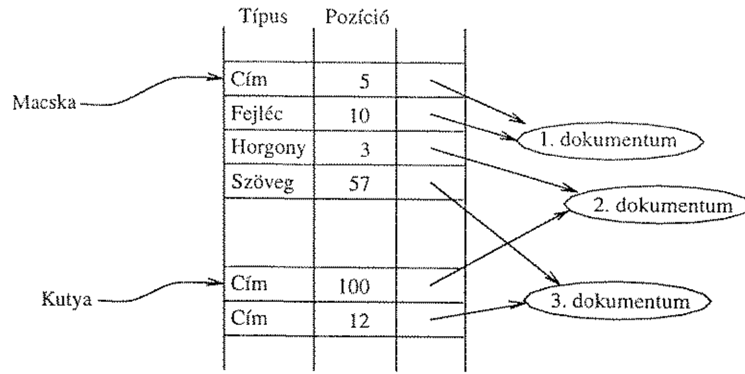
1. *Szótőképzés.* Mielőtt egy szó előfordulását bejegyeznénk az indexbe, eltávolítjuk a toldalékokat, hogy megkapjuk a „szótövet". A főnevek többes számát például úgy kezelhetjük, mintha egyes számban lennének. A 4.17. példában az invertált index természetesen szótőképzést használ, hiszen ha a „kutya" 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 $i = 1, 2, \ldots, 10\,000$.

* 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 $n = 340$. □

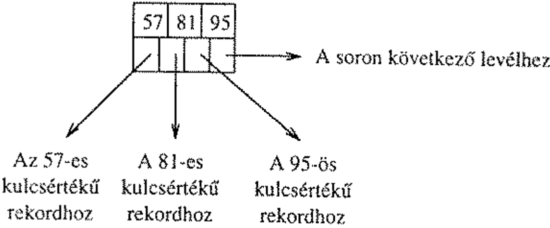
Van néhány olyan fontos szabály, amely megszorításokat jelent arra nézve, hogy egy B-fa blokkjai mit tartalmazhatnak.

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

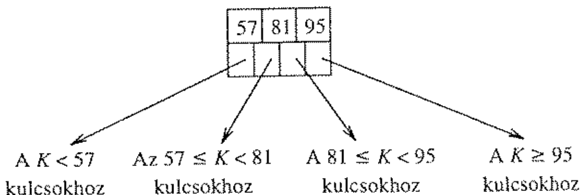
[3] Technikailag lehetséges, hogy a teljes B-fa mindössze egy mutatót tartalmazzon, akkor, ha az adatfájl egyetlen rekordból áll. Ebben az esetben az egész fa egy olyan gyökérblokk, amely egyben levél is, és ez a blokk egyetlen kulcsot és egyetlen mutatót tartalmaz. Az elkövetkezőkben eltekintünk ettől 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 $\left\lfloor \frac{n+1}{2} \right\rfloor$ 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.

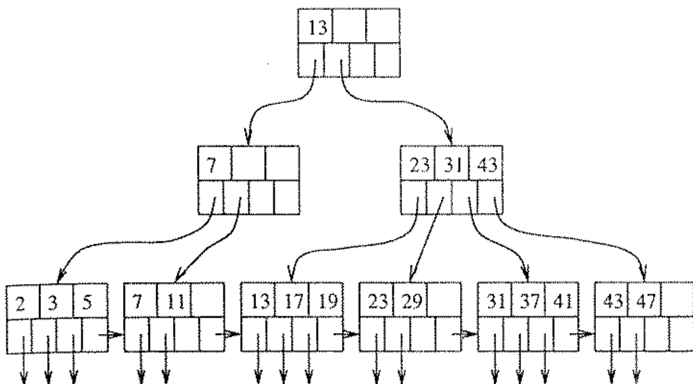
Éppúgy, mint a leveleknél, itt sem szükséges, hogy a kulcsok és mutatók tárolására fenntartott minden hely ki legyen töltve. Azonban $n = 3$, és legalább 1 kulcs és 2 mutató jelen kell legyen egy belső csúcsban. Ennek legszélsőségesebb esete az lenne, amikor az egyetlen kulcs az 57, és ekkor csak az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

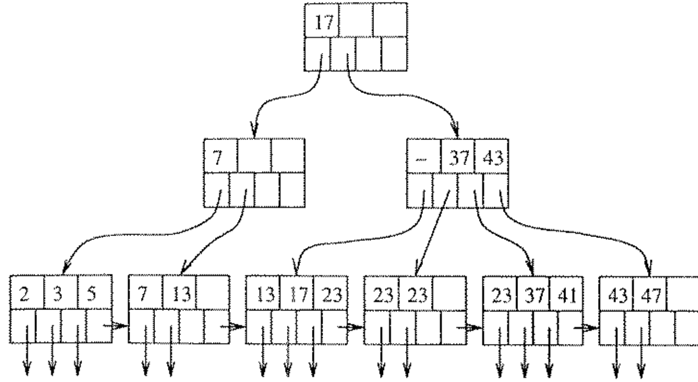
Ha megnézzük a gyökér első gyermekét, amelyben a 7-es kulcs található, ismét 2 mutatót találunk. Az egyik azokhoz a kulcsokhoz vezet, amelyek kisebbek, mint 7, a másik azokhoz, amelyek értéke nagyobb vagy egyenlő, mint 7. Vegyük észre, hogy ennek a csúcsnak a második mutatója csak a 7-es és 11-es kulcsokhoz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

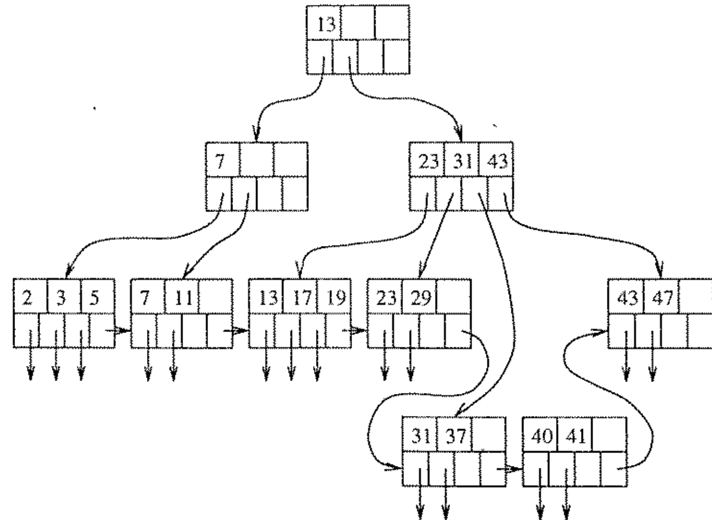
A B-fáknak vannak előnyei az egyszerűbb többszintű indexekkel szemben, ezek közül láthatunk néhányat, miközben áttekintjük, hogy miként kell beszúrni egy B-fába egy új kulcsot. A megfelelő rekordot a 4.1. részben bemutatott módszerek valamelyikével beszúrjuk a B-fával indexelt fájlba; itt most azt tekintjük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

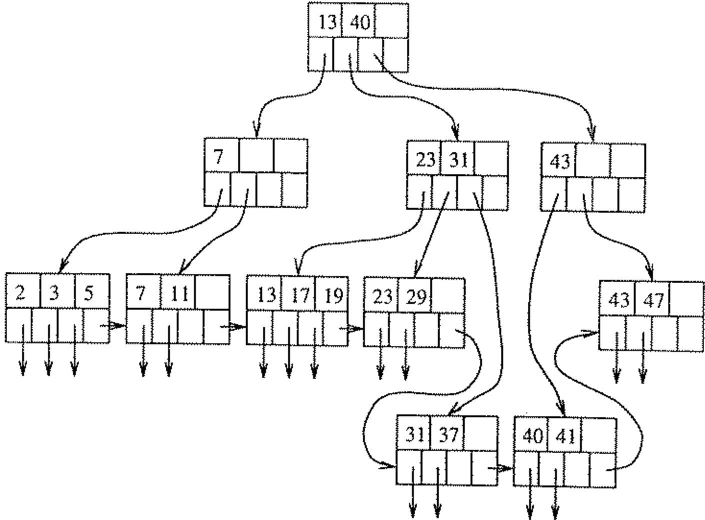
Figyeljük meg, hogy középen marad egy kulcs, amely nem jelenik sem az N, sem az M csúcsban. A maradék K kulcs azt a legkisebb kulcsot jelöli, amely az M első gyermekén keresztül elérhető. Habár ez a kulcs nem jelenik meg sem az N, sem az M csúcsban, mindamellett az M csúcshoz tartozik abban az értelemben,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

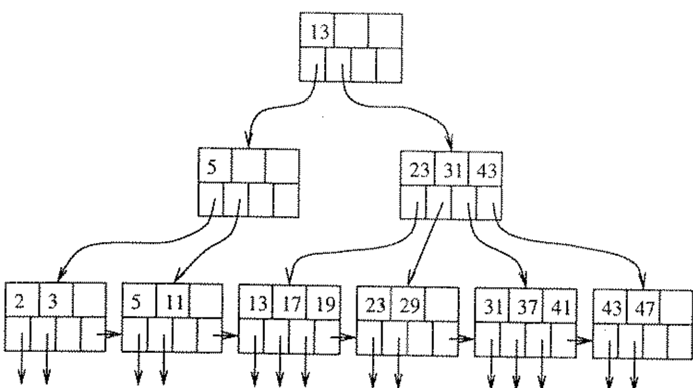
1. Ha az N csúcs egyik szomszédos testvére több kulcsot és mutatót tartalmaz, mint amennyi minimum szükséges, akkor egy kulcs-mutató párt áttehetünk az N csúcsba, miközben a kulcsok sorrendjét érintetlenül hagyjuk. Lehetséges, hogy az N csúcs szülőjében levő kulcsokat az új helyzethez kell igazítanunk. 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 töröl-

[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.

nünk 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.

4.26. példa: Kezdjünk a 40-es kulcs beillesztése előtti eredeti B-fával, amelyet a 4.23. ábrán láthatunk. Tegyük fel, hogy töröljük a 7-es kulcsot. Ez a kulcs a második levélben található. Kitöröljük a kulcsot, a hozzá tartozó mutatót és a megfelelő rekordot.

Sajnos, a második levél már csak egy kulcsot tartalmaz, és nekünk legalább két kulcsra van szükségünk valamennyi levélben. De meg vagyunk mentve, hiszen a balra lévő testvér, az első levél tartalmaz egy fölösleges kulcs-mutató párt. Ily módon áttehetjük a legnagyobb kulcsot és a hozzá tartozó mutatót a második levélbe. Az eredményül kapott B-fát a 4.27. ábrán láthatjuk. Figyeljük meg, hogy mivel a második levél legkisebb kulcsa most 5, ezért a két első levél szülőjében a 7-es kulcs 5-re változott.

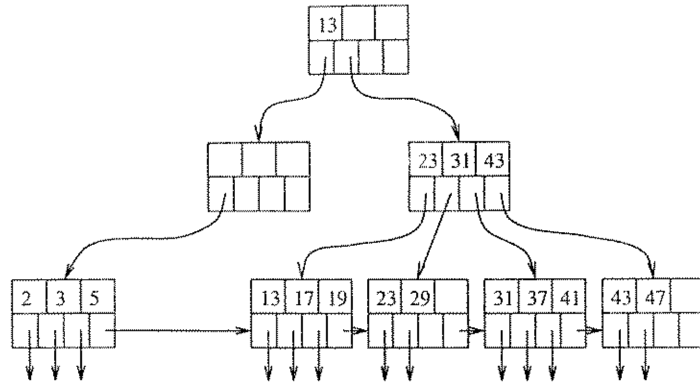


4.27. ábra. *A 7-es kulcs törlése*

A következőkben tegyük fel, hogy a 11-es kulcsot töröljük. Ennek a törlésnek hasonló hatása van a második levélre; ezzel a kulcsok száma ismét a minimum alá csökkent. Ezúttal azonban nem kérhetünk kölcsön az első levéltől, hiszen abban pontosan a minimum számú kulcs található. Ráadásul jobbról nincs is testvér, akitől kölcsönkérhetnénk.[8] Ily módon össze kell olvasztanunk a második levelet egy testvérével, nevezetesen az első levéllel.

Az első két levél három megmaradó kulcs-mutató párja befér egyetlen levélbe, így áttesszük az 5-ös kulcsot az első levélbe, és a második levelet kitöröljük. A szülőben levő kulcsokat és mutatókat a gyermekek új helyzetéhez igazítjuk, azaz a két mutatót egy mutató váltja fel (amely a megmaradt levélre mutat), és az 5-ös kulcs többé már nem fontos, ezért kitöröljük. Ezt a helyzetet a 4.28. ábrán mutatjuk be.

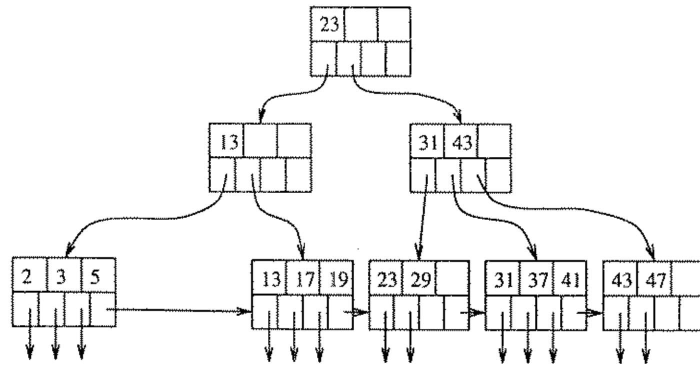
[8] Figyeljük meg, hogy a jobbra levő levél, amelynek kulcsai 13, 17 és 19 nem testvér, hiszen nem ugyanaz a két levél szülője. Ezzel együtt persze „kölcsönözhetünk" ettől a csúcstól, azonban a fa kulcsainak a helyzethez történő igazítása sokkal bonyolultabbá válik. Ezt a lehetőséget meghagyjuk feladatnak.



4.28. ábra. *A 11-es kulcs törlésének kezdete*

Sajnos, a levél törlése kedvezőtlenül befolyásolta a szülőt, amely a gyökér bal oldali gyermeke. Ez a csúcs ily módon nem is tartalmaz kulcsot, és mutatóból is csak egy van neki, amint a 4.28. ábrán láthatjuk. Ezért megpróbálunk szerezni egy fölösleges kulcsot és mutatót az egyik szomszédos testvértől. Ebben az esetben könnyű dolgunk van, hiszen a gyökér másik gyermeke megengedheti magának, hogy átadja a legkisebb kulcsot és egy mutatót.

A változás a 4.29. ábrán látható. Az a mutató, amely a 13, 17 és 19 kulcsokat tartalmazó levélre mutat, átköltözött a gyökér második gyermekéből az első gyermekbe. A belső csúcsokban is megváltoztattunk azonban néhány kulcsot. A 13-as kulcsot, amely eddig a gyökérben volt, és azt jelölte, hogy ez a legkisebb kulcs, amelyet az átköltöztetett mutatón keresztül el lehet érni, át kellett tenni a gyökér első gyermekébe. Másrészről, a 23-as kulcs, amely eddig a gyökér második gyermekének első és második gyermekét volt hivatott szétválasztani, most a gyökér második gyermekéből elérhető legkisebb kulcsot jelöli. Ily módon a 23-as kulcs átkerült magába a gyökérbe. □



4.29. ábra. *A 11-es kulcs törlésének befejezése*

4.3.7. B-fák hatékonysága

A B-fák lehetővé teszik rekordok keresését, beszúrását és törlését oly módon, hogy nagyon kevés az egy fájlra eső lemez I/O-műveletek száma. Először is, megfigyelhetjük, hogy ha az n, az egy blokkban tárolható kulcsok száma megfelelően nagy, mondjuk 10 vagy több, akkor ritkán lesz szükség blokkok szétvágására, illetve összeolvasztására. Továbbá, ha ilyen műveletre szükség van, akkor az majdnem mindig a levelekre korlátozódik, azaz mindössze két levél és azok szülője érintett. Ily módon lényegében elhanyagolható a B-fák újrarendezésének I/O-költsége.

Azonban adott keresési kulccsal rendelkező rekord(ok) megkereséséhez a gyökérből kell elindulnunk, és el kell jutnunk egy levélhez, hogy megtaláljuk a keresett rekordhoz tartozó mutatót. Mivel a B-fa blokkjait csak beolvassuk, ezért a lemez I/O-műveletek száma meg fog egyezni a B-fa szintjeinek a számával, plusz a rekord manipulásához szükséges lemez I/O-műveletek számával, ami egy vagy kettő, attól függően, hogy keresésről van-e szó avagy beszúrásról, illetve törlésről. Joggal kérdezhetjük: hány szintje van egy B-fának? A kulcsok, mutatók és blokkok tipikus méretével a legnagyobb adatbázisokat kivéve is elegendő három szint. Ily módon általában 3-nak fogjuk venni egy B-fa szintjeinek a számát. A következő példából láthatjuk ennek az okát.

4.27. példa: Idézzük fel a 4.19. példa elemzését, ahol meghatároztuk, hogy a példa adatai esetén 340 kulcs-mutató pár fér el egyetlen blokkban. Tegyük fel, hogy egy átlagos blokk telítettségi középértéke valahol a minimum és a maximum között van, azaz egy tipikus blokk 255 mutatót tartalmaz. Egy gyökér, 255 gyermek és $255^2 =$ $= 65\,025$ levél esetén a levelekből 255^3 mutató indulhat ki, ami körülbelül 16,6 millió rekordra utaló mutató. Ily módon egy 3 szintű B-fa akár 16,6 millió rekordot tartalmazó fájlba is alkalmazható. □

Azonban a B-fában történő kereséshez háromnál kevesebb lemez I/O-műveletet is használhatunk. A B-fa gyökérblokkjának állandóan az elsődleges memóriában történő tárolása egy nagyon jó választás. Ilyen esetben a keresés egy 3 szintű B-fában mindössze két lemezolvasást igényel. Valójában, bizonyos körülmények között a B-fa második szintű csúcsait is tarthatjuk az elsődleges memóriában, egyre csökkentve ily módon a kereséshez szükséges lemez I/O-műveletek számát, ehhez jön természetesen esetenként az adatfájl blokkjainak manipulálásához szükséges lemez I/O-műveletek száma.

4.3.8. Feladatok

4.3.1. feladat: Tegyük fel, hogy egy blokkban elfér tíz rekord vagy 99 kulcs és 100 mutató. Tegyük fel továbbá, hogy a B-fa átlagos csúcsának a telítettsége 70%; azaz 69 kulcsot és 70 mutatót tartalmaz. A B-fákat felhasználhatjuk több különböző adatszerkezet részeként. Az alábbiakban bemutatott valamennyi adatszerkezetre határozzuk meg i) a szükséges adatblokkok számát egy olyan fájl esetén, amelyik 1 000 000 re-

Szükséges-e törölnünk B-fákból?

Vannak olyan B-fa-megvalósítások, ahol a törlést egyáltalán nem szervezik meg. Ha egy levélben túl kevés kulcs és mutató van, megengedett, hogy így maradjon. Az igazság az, hogy a legtöbb fájl egyenletesen nő, és ha alkalmanként elő is fordul olyan törlés, amellyel egy levél a minimum alá csökken, a levél valószínűleg hamarosan ismét visszahízik, és ismét eléri a kulcs-mutató párok számára vonatkozó alsó határt.

Továbbá, ha rekordokra a B-fa indexen kívül máshonnan is mutatnak mutatók, akkor a rekordot egyszerűen egy ún. „sírkővel" helyettesítjük, és nem kell feltétlenül kitörölni B-fának azt a mutatóját, amely erre a rekordra mutat. Bizonyos körülmények között, ha garantált, hogy a kitörölt rekordra csak a B-fán keresztül történik hozzáférés, akkor a B-fa levelében a rekordra utaló mutató helyére sírkövet tehetünk. Ily módon, a rekord által elfoglalt hely újra felhasználható.

kordot tartalmaz, valamint ii) adott keresésikulcs-értékkel rendelkező rekord megtalálásához szükséges átlagos lemez I/O-műveletek számát. Feltételezhetjük, hogy kezdetben semmi sincs a memóriában, és hogy a keresési kulcs a rekordok elsődleges kulcsa.

* a) Az adatfájl egy szekvenciális fájl, amely a keresési kulcs alapján rendezett, és 10 rekord van egy blokkban. A B-fa egy sűrű index.

b) Ugyanaz, mint az a) esetben, viszont az adatfájl rekordjai nem rendezettek, és 10 rekord van egy blokkban.

c) Ugyanaz, mint az a) esetben, de a B-fa egy ritka index.

! d) Ahelyett, hogy a B-fa leveleiben olyan mutatók lennének, amelyek az adatrekordokra mutatnak, a B-fa levelei magukat a rekordokat tartalmazzák. Egy blokkban tíz rekord fér el, viszont átlagosan egy levélblokk 70% telítettségű; azaz 7 rekord van egy levélblokkban.

* e) Az adatfájl szekvenciális fájl, és a B-fa egy ritka index, viszont az adatfájl valamennyi elsődleges blokkja rendelkezik egy túlcsordulásblokkal. Átlagosan az elsődleges blokk tele van, és a túlcsordulásblokk félig telített. A rekordok azonban rendezetlenek az elsődleges és a túlcsordulásblokkokban.

4.3.2. feladat: Ismételjük meg a 4.3.1. feladatot arra az esetre, ha a lekérdezés olyan tartományt eredményez, amelybe 1000 rekord tartozik.

4.3.3. feladat: Tegyük fel, hogy a mutatók 4 bájt hosszúak, és a kulcsok 12 bájt hosszúak. Hány kulcsot és mutatót fog tartalmazni egy 16 384 bájtból álló blokk?

4.3.4. feladat: Mennyi a kulcsok, illetve a mutatók minimális száma egy B-fa i) belső csúcsaiban, illetve ii) leveleiben, ha:

* a) $n = 10$; azaz egy blokk 10 kulcsot és 11 mutatót tartalmaz.

b) $n = 11$; azaz egy blokk 11 kulcsot és 12 mutatót tartalmaz.

4.3.5. feladat: Hajtsuk végre a következő műveleteket a 4.23. ábrán. Írjuk le a változásokat azoknál a műveleteknél, amelyek módosítják a fát.

a) A 41-es kulcsértékű rekord megkeresése.

b) A 40-es kulcsértékű rekord megkeresése.

c) Az összes olyan rekord megkeresése, amelyek halmaza a 20 és 30 közötti tartományba tartozik.

d) Az összes olyan rekord megkeresése, amelynek kulcsa kisebb, mint 30.

e) Az összes olyan rekord megkeresése, amelynek kulcsa nagyobb, mint 30.

f) Az 1-es kulcsértékű rekord beszúrása.

g) A 12-es, 15-ös és 16-os kulcsértékű rekordok beszúrása.

h) A 23-as kulcsértékű rekord törlése.

i) A 23-as és annál nagyobb kulcsértékű rekordok törlése.

! **4.3.6. feladat:** Említettük, hogy a 4.21. ábrán látható levél és a 4.22. ábrán látható belső csúcs soha nem jelenhet meg ugyanabban a B-fában. Magyarázzuk meg, hogy miért.

4.3.7. feladat: Ha egy B-fában megengedettek az ismétlődő kulcsok, akkor szükség van néhány módosításra azokban az algoritmusokban, amelyeket ebben a fejezetben mutattunk be a keresésre, beszúrásra, illetve törlésre. Adjuk meg a módosításokat:

* a) keresésre,

b) beszúrásra,

c) törlésre.

! **4.3.8. feladat:** A 4.26. példában említettük, hogy lehetőség lenne bal (vagy jobb) oldali nem testvértől is kulcsot kölcsönözni, amennyiben a belső csúcsok kulcsainak karbantartására egy jóval bonyolultabb algoritmust használnánk. Adjunk meg egy olyan algoritmust, amely a kiegyenlítést az ugyanazon a szinten levő szomszédos csúcsoktól való kölcsönzéssel oldja meg, függetlenül attól, hogy az a csúcs, amelytől a kölcsönzés történt, testvére-e vagy sem a túl sok vagy túl kevés kulcs-mutató párt tartalmazó csúcsnak.

4.3.9. feladat: Ha 3 kulcsos, 4 mutatós csúcsokat használunk a fejezetben levő példákhoz, akkor hány különböző B-fa létezik, ha az adatfájlban:

*! a) 6 rekord van,

!! b) 10 rekord van,

!! c) 15 rekord van.

*! **4.3.10. feladat:** Tegyük fel, hogy olyan B-fa csúcsaink vannak, amelyekben 3 kulcs és 4 mutató számára van hely, éppúgy, mint a fejezet példáiban. Tegyük fel továbbá, hogy amikor szétvágunk egy levelet, akkor a mutatókat 2 és 2 arányban osztjuk meg, míg ha egy belső csúcsot vágunk szét, akkor az első 3 mutató az első (bal oldali) csúcshoz kerül, az utolsó 2 mutató pedig a második (jobb oldali) csúcshoz kerül. Egyetlen levéllel kezdünk, amely az 1, 2 és 3 kulcsokat tartalmazza. Ezután sorban beszúrjuk a 4, 5, 6 kulcsokat és így tovább. Melyik kulcs beszúrásánál éri el a B-fa szintjeinek száma először a négyet?

!! **4.3.11. feladat:** Adott egy B+-fa szerkezetű index. A levél csúcsok mutatói összesen N rekordra mutatnak, és valamennyi, az indexet felépítő blokk m mutatót tartalmaz. Szeretnénk megválasztani az m értékét úgy, hogy az minimalizálja a keresési időket azon a lemezen, amely a következő sajátosságokkal rendelkezik:

i) Egy adott blokk memóriába történő beolvasásának ideje körülbelül $70 + 0{,}5m$ milliszekundum. A 70 milliszekundum a beolvasás keresési és lappangási idejét jelenti, míg a $0{,}5m$ milliszekundum az átviteli idő. Ily módon, ha az m nő, a blokk mérete is nő, és több időbe kerül a memóriába való beolvasás.

ii) Ha a blokk egyszer már a memóriában van, akkor bináris keresést használunk a megfelelő mutató megtalálásához. Ily módon, egy blokk feldolgozási ideje az elsődleges memóriában $a + b \log_2 m$ milliszekundum, bizonyos a és b konstansokra.

iii) Az elsődleges memória a konstansa sokkal kisebb, mint a 70 milliszekundum, ami a lemez keresési és lappangási ideje.

iv) Az index tele van, így keresésenként $\log_m N$ számú blokkot kell megvizsgálni.

Adjunk feleletet a következőkre:

a) Milyen m érték minimalizálja egy adott rekord keresési idejét?

b) Mi történik, ha a lemez keresési és lappangási ideje (70 ms) csökken? Például, ha ez a konstans a felére csökken, hogyan változik az m optimális értéke?

4.4. Tördelőtáblázatok

Igen sok indexként hasznos olyan adatszerkezet létezik, amelyik magában foglal egy tördelőtáblázatot. Feltételezzük, hogy az olvasó találkozott már a tördelőtáblázattal, mint elsődleges memóriában használt adatszerkezettel. Egy ilyen szerkezetben van egy *tördelőfüggvény,* amely argumentumként megkap egy keresési kulcsot (amelyet *tördelőkulcsnak* is nevezhetünk), és eredményül ad egy 0 és $B-1$ közötti egész számot, ahol B a *kosarak* száma. A *kosártömb* egy olyan tömb, amelynek indexei 0 és $B-1$ között vannak, és B számú, a tömb valamennyi kosarához egy-egy, a láncolt lista fejlécét tartalmazza. Ha egy rekord keresési kulcsa K, akkor a rekordot a $h(K)$-val számozott kosárnál láncoljuk a kosárlistához, ahol h a tördelőfüggvény.

4.4.1. Másodlagos tárolón tárolt tördelőtáblázatok

Egy tördelőtáblázat, amely olyan sok rekordot tartalmaz, hogy többnyire másodlagos tárolón kell tartani, apró, de fontos dolgokban különbözik az elsődleges memóriában tárolt változattól. Először is, a kosártömb blokkokból áll, és nem a listák fejléceire mutató mutatókból. Azok a rekordok, amelyeket a h tördelőfüggvény egy bizonyos kosárba tördel, az adott kosárhoz tartozó blokkban vannak. Ha egy kosár *túlcsordul,* azaz nem képes befogadni az összes hozzá tartozó rekordot, akkor egy *túlcsordulásblokkokból* álló láncot lehet hozzáadni a kosárhoz, hogy több rekordot be tudjon fogadni.

Feltételezzük, hogy bármely i kosár első blokkja megtalálható, adott i érték esetén. Például, az elsődleges memóriában lehet egy olyan tömbünk, amelyik a kosarak sorszámával indexelt és a blokkokra mutató mutatókból áll. Egy másik lehetőség, hogy valamennyi kosár első blokkját rögzített, egymás utáni lemezterületekre tesszük, így kiszámolható az i kosár első blokkja, adott i érték esetén.

4.28. példa: A 4.30. ábrán egy tördelőtáblázatot láthatunk. A példa átláthatóságának megőrzése érdekében feltételezzük, hogy egy blokkban csak két rekord fér el, és, hogy $B = 4$; azaz a h tördelőfüggvény 0 és 3 közötti értékeket ad vissza. A tördelőtáblázatot benépesítő rekordokat feltüntettük. A 4.30. ábrán látható kulcsok a és f közötti betűk. Feltételezzük, hogy $h(d) = 0$, $h(c) = h(e) = 1$, $h(b) = 2$ és $h(a) = h(f) = 3$. A hat rekord ily módon megoszlik a blokkok között, amint az látható is. □

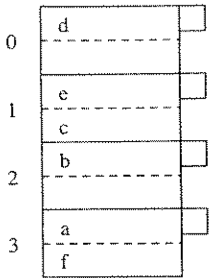
Figyeljük meg, hogy a 4.30. ábra valamennyi blokkjának jobb szélén egy „kinövés" látható. Ez a kinövés további információkat reprezentál a blokk fejlécében.

Tördelőfüggvény megválasztása

A tördelőfüggvénynek úgy kell „tördelnie" a kulcsot, hogy az eredményül kapott egész szám látszólag véletlenszerűen függjön a kulcstól. Ily módon a kosarak közel egyenlő számú rekordot fognak tartalmazni, ami javítja a rekordhozzáférések átlagos idejét, amint arról a 4.4.4. részben olvashatunk majd. Mindamellett a tördelőfüggvény könnyen kiszámítható kell legyen, mivel sokszor kerül majd kiszámításra.

- Ha a kulcsok egész számok, akkor a tördelőfüggvényt általában úgy választjuk meg, hogy a K/B maradékát számolja ki, ahol K a kulcsérték és B a kosarak száma. A B-t általában úgy választjuk meg, hogy prímszám legyen, habár indokolt lehet a B-t 2 valamilyen hatványának választani, ahogyan arról a 4.4.5. résztől kezdve olvashatunk.
- Ha a keresési kulcsok karakterláncok, akkor valamennyi karaktert kezelhetjük úgy, mint egész számot, összegezzük ezeket az egész számokat, és vegyük az összeg B-vel történő osztásának a maradékát.

Használhatjuk túlcsordulásblokkok összeláncolásához, és a 4.4.5. résztől kezdődően használjuk egyéb, blokkra vonatkozó kritikus információk tárolására.



4.30. ábra. *Tördelőtáblázat*

4.4.2. Beszúrás tördelőtáblázatba

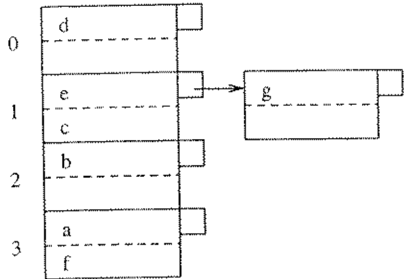
Ha egy K keresésikulcs-értékű új rekordot kell beszúrni, akkor kiszámoljuk a $h(K)$-t. Ha a $h(K)$ jelzőszámú kosárban van szabad hely, akkor a rekordot beszúrjuk a kosárhoz tartozó blokkba, vagy ha az első blokkban nincs hely, akkor a kosárhoz tartozó lánc valamelyik túlcsordulásblokkjába. Ha a $h(K)$ sorszámú kosárhoz tartozó lánc egyik blokkjában sincs szabad hely, akkor hozzáadunk a lánchoz egy újabb túlcsordulásblokkot, és ide szúrjuk be az új rekordot.

4.29. példa: Tegyük fel, hogy a 4.30. ábrán látható tördelőtáblázathoz szeretnénk hozzáadni a g kulcsértékű rekordot, és $h(g) = 1$. Így az új rekordot az 1-es sorszámú kosárba kell helyeznünk, amely felülről a második kosár. Az ehhez a kosárhoz tartozó blokkban azonban már van két rekord. Ezért hozzáadunk egy új blokkot, és az 1-es kosárhoz tartozó eredeti blokkhoz láncoljuk. A g kulcsértékű rekord ebbe a blokkba kerül, ahogyan azt a 4.31. ábrán is láthatjuk. □

4.4.3. Törlés tördelőtáblázatban

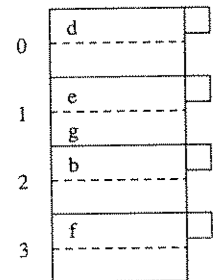
A K keresésikulcs-értékű rekord (illetve rekordok) törlése hasonló szabályszerűséget követ. Vesszük a $h(K)$ sorszámú kosarat, és megkeressük benne a megfelelő kulcsértékű rekordokat. Valamennyi megtalált rekordot töröljük. Ha van lehetőségünk a blokkok közötti rekordmozgatásra, akkor opcionálisan konszolidálhatjuk egy lánc blokkjait.[9]

[9] Ha egy lánc blokkjainak konszolidálása lehetséges, az azzal a veszéllyel jár, hogy egy oszcillálás, amikor felváltva szúrunk be és törlünk rekordokat a kosárból, azt fogja okozni, hogy mindegyik lépésnél egy blokkot létre kell hozni vagy ki kell törölni.



4.31. ábra. *A tördelőtáblázat egyik kosarához hozzáadunk egy további blokkot*

4.30. példa: A 4.32. ábrán láthatjuk a c kulcsértékű rekord 4.31. ábrából történő törlésének eredményét. Emlékezzünk vissza, hogy $h(c) = 1$, tehát az 1-es jelzőszámú (azaz második) kosárhoz kell mennünk, és végig kell keresnünk a hozzá tartozó valamennyi blokkot, hogy megtaláljuk a c kulcsértékű rekordot (illetve rekordokat, ha a keresési kulcs nem elsődleges kulcs). Az 1-es jelzőszámú kosárhoz tartozó lánc első blokkjában megtaláljuk. Most van hely arra, hogy a második blokk g kulcsértékű rekordját áttegyük a lánc első blokkjába, és ily módon töröljük a második blokkot.



4.32. ábra. *Tördelőtáblázatból történő törlések eredménye*

Bemutatjuk az a kulcsértékű rekord törlését is. Ezt a kulcsot a 3-as jelzőszámú kosárban találjuk, kitöröljük, és a megmaradó rekordot a blokk elejére „konszolidáljuk". □

4.4.4. Tördelőtáblázat-indexek hatékonysága

Ideális esetben elég sok kosarunk van ahhoz, hogy a legtöbb kosár egy blokkból álljon. Ha ez így van, akkor a tipikus keresés csak egy lemez I/O-művelettel jár, míg a beszúrás és a törlés két lemez I/O-műveletet igényel. Ez a szám jelentősen jobb, mint a hagyományos ritka vagy sűrű indexeknél, illetve a B-fa-indexeknél (habár a tördelőtáblázatok a B-fákkal ellentétben nem támogatják a 4.3.4. részben bemutatott tartományt eredményező lekérdezéseket).

Ha azonban a fájl nő, előbb-utóbb eljutunk egy olyan helyzethez, amikor egy átlagos kosárhoz tartozó lánc sok blokkot tartalmaz. Ha ez így van, akkor blokkok hosszú listáit kell végignéznünk, amely legalább egy lemez I/O-műveletet jelent blokkonként. Jó okunk van tehát rá, hogy az egy kosárra eső blokkok számát alacsonyan tartsuk.

Az eddig vizsgált tördelőtáblázatokat *statikus tördelőtáblázatoknak* nevezzük, mivel a B, a kosarak száma soha nem változik. Léteznek azonban különböző *dinamikus tördelőtáblázatok* is, ahol a B változhat, azaz a B megközelíti azt a számot, amelyet úgy kapunk, ha elosztjuk a rekordok számát azon rekordok számával, amelyek elférnek egy blokkban; ez azt jelenti, hogy körülbelül egy blokk tartozik egy kosárhoz. Két ilyen módszert fogunk bemutatni:

1. a kiterjeszthető tördelést a 4.4.5. részben és
2. a lineáris tördelést a 4.4.7. részben.

Az első úgy növeli a B értékét, hogy megduplázza azt, valahányszor túl kevésnek bizonyul, és a második mindig 1-gyel növeli a B értékét, valahányszor a fájl statisztikái alapján növelésre van szükség.

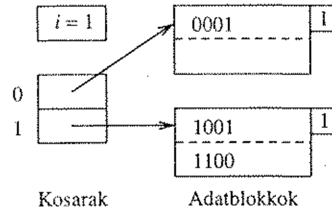
4.4.5. Kiterjeszthető tördelőtáblázatok

A dinamikus tördelés első megközelítését *kiterjeszthető tördelőtáblázatoknak* nevezzük. Az egyszerűbb statikus tördelőtáblázathoz képest a főbb kiegészítések:

1. A kosarakhoz egy közvetett szintet vezetünk be. Ez azt jelenti, hogy a kosarakat egy olyan, mutatókból álló tömb reprezentálja, ahol a mutatók blokkokra mutatnak ahelyett, hogy a tömb magukat az adatblokkokat tartalmazná.
2. A mutatókból álló tömb nőhet. Hossza mindig a 2 valamilyen hatványa, tehát egy növekedési lépésben a kosarak száma megduplázódik.
3. Mindamellett nem kell valamennyi kosárnak rendelkeznie egy adatblokkal; bizonyos kosarak osztozhatnak egy blokkon, ha a kosarakban levő rekordok elférnek ebben a blokkban.
4. A h tördelőfüggvény valamennyi kulcs esetén egy k bitből álló sorozatot számol ki, ahol a k elég nagy, mondjuk 32. A kosárjelzőszámok azonban mindenkor kevesebb számú bitet használnak, mondjuk i bitet a sorozat elejéről. Ily módon a kosártömbnek 2^i bejegyzése lesz, ahol i a felhasznált bitek száma.

4.31. példa: A 4.33. ábrán egy kisméretű, kiterjeszthető tördelőtáblázatot láthatunk. Az egyszerűség kedvéért tegyük fel, hogy $k = 4$; azaz a tördelőfüggvény egy mindöszsze négy bitből álló sorozatot ad vissza. Jelenleg ezen bitekből csak egy használatos, ahogyan azt fel is tüntettük a kosártömb fölötti dobozban. A kosártömbnek ily módon mindössze két bejegyzése van, egy a 0-hoz és egy az 1-hez.

A kosártömb bejegyzései két blokkra mutatnak. Az első tartalmazza az összes olyan aktuális rekordot, amelynek keresési kulcsa 0-val kezdődő bitsorozatot tördel, a



4.33. ábra. *Kiterjeszthető tördelőtáblázat*

második pedig azokat tartalmazza, amelyek keresési kulcsa 1-gyel kezdődő sorozatot tördel. A kényelem kedvéért a rekordok kulcsait úgy ábrázoljuk, mintha azok megegyeznének azokkal a teljes bitsorozatokkal, amelyekké a tördelőfüggvény konvertálja őket. Ily módon az első blokk egy rekordot tartalmaz, amelynek kulcsa a 0001-et tördeli, a második blokk azokat a rekordokat tartalmazza, amelyek kulcsai az 1001-et és az 1100-t tördelik. □

A 4.33. ábrán megfigyelhetjük, hogy valamennyi blokk kinövésében megjelenik az 1-es szám. Ez a szám, amely tulajdonképpen a blokk fejlécében is megjelenhet, azt jelzi, hogy a tördelőfüggvény által visszaadott sorozatból hány bit használatos annak eldöntésére, hogy egy rekord az adott blokkhoz tartozik-e vagy sem. A 4.31. példában valamennyi blokk és rekord esetén egy bit használatos, de amint azt majd látni fogjuk, a különböző blokkokhoz használatos bitek száma változhat, ahogyan a tördelőtáblázat növekszik. Ily módon a kosártömb mérete az aktuálisan használt bitek száma által meghatározott, de előfordulhat, hogy bizonyos blokkok kevesebbet használnak.

4.4.6. Beszúrás kiterjeszthető tördelőtáblázatokba

Egy kiterjeszthető tördelőtáblázatba történő beszúrás ugyanúgy kezdődik, mint egy statikus tördelőtáblázatba történő beszúrás. A K keresésikulcs-értékű rekord beszúrásához kiszámoljuk a $h(K)$ bitsorozatot, ennek vesszük az első i bitjét, és a kosártömb azon bejegyzéséhez megyünk, amelynek jelzőszáma ez az i bit. Megjegyzendő, hogy az i-t azért tudjuk meghatározni, mert a tördelő-adatszerkezetben el van tárolva.

Követjük a kosártömb ezen bejegyzésének mutatóját, és elérkezünk egy B blokkhoz. Ha van szabad hely a B-ben az új rekord elhelyezésére, akkor ezt megtesszük, és készen is vagyunk. Ha nincs szabad hely, akkor a j értékétől függően két lehetőségünk van; a j azt jelzi, hogy a tördelőfüggvény által kiszámolt érték hány bitje használatos annak eldöntésére, hogy a rekord a B blokkhoz tartozik vagy nem (ne feledjük, hogy a j érték az ábrán a blokkok kinövéseiben található).

1. Ha $j < i$, akkor a kosártömbbel semmit nem kell tennünk. Amit meg kell tennünk:

   a) A B blokkot kettévágjuk.

   b) A B rekordjait szétosztjuk a két blokk között, a $(j + 1)$-edik bit értéke alapján –

azok a rekordok, amelyek kulcsa 0 az adott bitnél, maradnak a *B* blokkban, míg azok a rekordok, melyek ezen a helyen 1-et tartalmaznak, az új blokkba kerülnek.

c) Mindegyik blokk „kinövésébe" $j + 1$ kerül, jelezvén, hogy ennyi bit használatos az odatartozás eldöntésére.

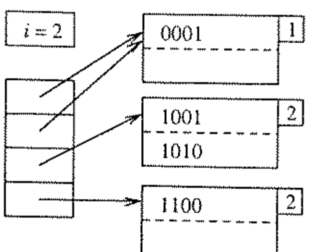
d) A kosártömb mutatóit az új helyzethez igazítjuk úgy, hogy azok a bejegyzések, amelyek azelőtt a *B*-re mutattak, most vagy a *B*-re vagy az új blokkra mutassanak, a $(j + 1)$-edik bittől függően.

Megjegyzendő, hogy a *B* blokk szétvágása nem biztos, hogy megoldja a problémát, hiszen a véletlen hozhatja úgy, hogy a *B* valamennyi rekordja a két blokk egyikébe kerül a szétvágás után. Ha ez így van, akkor meg kell ismételnünk az eljárást a *j* következő értékére, és arra a blokkra, amelyik még mindig tele van.

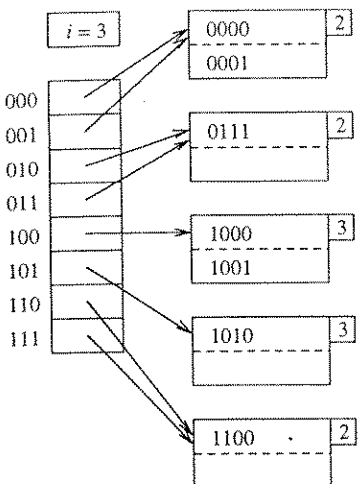
2. Ha $j = i$, akkor először meg kell növelnünk 1-gyel az *i* értékét. Megduplázzuk a kosártömb hosszát, hiszen most 2^{i+1} bemenetet tartalmaz. Tegyük fel, hogy a *w* egy *i* számú bitből álló sorozat, amely az előző kosártömb egyik bejegyzésének volt a jelzőszáma. Az új kosártömbben a *w*0 és *w*1 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:

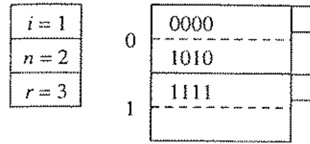
1. Amikor meg kell dupláznunk a kosártömböt, akkor tekintélyes mennyiségű munkát kell végezni (ha az i nagy). Ez a munka megszakítja az adatfájlhoz való hozzáférést, vagy igen lassúvá tesz bizonyos beszúrásokat.
2. Ha a kosártömb méretét megduplázzuk, lehet, hogy nem fér majd el az elsődleges memóriában,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

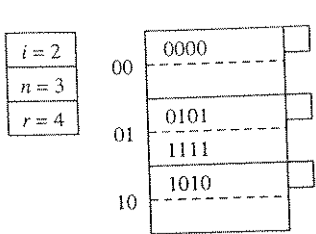
4.4.8. Beszúrás lineáris tördelőtáblázatokba

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*

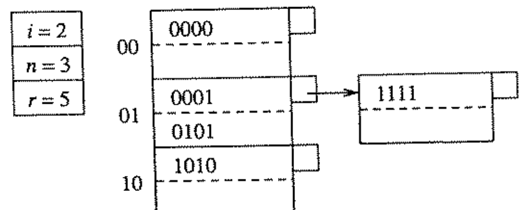
végződik, a rekord a 4.36. ábra második kosarába kerül. Van szabad hely a rekord számára, így nem kell túlcsordulásblokkot készíteni.

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.

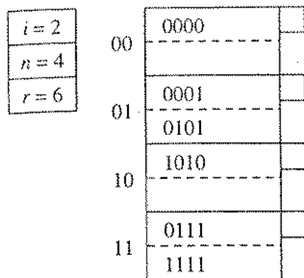
Most tegyük fel, hogy egy olyan rekordot akarunk beszúrni, amelynek keresési kulcsa 0001-et tördel. A két utolsó bit 01, így ebbe a kosárba tesszük, amely jelenleg létezik. Sajnos a kosár blokkja tele van, így hozzáadunk egy túlcsordulásblokkot. A három rekordot elosztjuk a kosár két blokkja között;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

hető tördelés kidolgozása a [4]-ben szerepel, míg a lineáris tördelés a [7]-ből való. A Knuth-könyv [6] sok információt tartalmaz az adatszerkezetekről, a tördelőfüggvények megválasztásától és a tördelőtáblázatok tervezésétől kezdve egészen a különböző B-fákkal foglalkozó ötletekig. A B+-fa változat (kulcsértékek nélküli belső csúcsok) a [6] 1973-as kiadásában jelent meg.

A másodlagos indexeket és a dokumentumok visszakeresésének egyéb technikáit a [9] mutatja be. Az [5] és az [1] szintén a szöveges dokumentumok indexelési módszereinek áttekintését tartalmazzák.

1. R. Baeza-Yates, „Integrating contents and structure in text retrieval," *SIGMOD Record* **25**:1 (1996), pp. 67–79.
2. R. Bayer and E. M. McCreight, „Organization and maintenance of large ordered indexes," *Acta Informatica* **1**:3 (1972), pp. 173–189.
3. D. Comer, „The ubiquitous B-tree," *Computing Surveys* **11**:2 (1979), pp. 121–137.
4. R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, „Extendible hashing – a fast access method for dynamic files," *ACM Trans. on Database Systems* **4**:3 (1979), pp. 315–344.
5. C. Faloutsos, „Access methods for text," *Computing Surveys* **17**:1 (1985), pp. 49–74.
6. D. E. Knuth, *The Art of Computer Programming, Vol. III, Sorting and Searching, Third Edition,* Addison-Wesley, Reading MA, 1998.
7. W. Litwin, „Linear hashing: a new tool for file and table addressing," *Proc. Intl. Conf. on Very Large Databases* (1980) pp. 212–223.
8. W. W. Peterson, „Addressing for random access storage," *IBM J. Research and Development* **1**:2 (1957), pp. 130–146.
9. G. Salton, *Introduction to Modern Information Retrieval,* McGraw-Hill, New York, 1983.

5. fejezet

Többdimenziós indexek

Az eddig tárgyalt indexstruktúrák *egydimenziósak* voltak; azaz mindegyik egyetlen keresési kulcsot tartalmazott, és olyan rekordokat adott vissza, amelyek megfeleltek az adott keresésikulcs-értéknek. Azt gondolhatjuk, hogy a keresési kulcs mindig egyetlen attribútum vagy mező. Azonban olyan index is lehet egydimenziós, amelynek keresési kulcsa több mező egyesítése. Ha egy egydimenziós indexet akarunk, amelynek a keresési kulcsa az $(F_1, F_2, \ldots, F_k)$ mezőkből áll, akkor képezhetjük a keresésikulcs-értéket az egyes mezőértékek egymás után helyezésével, elsőnek az F_1 értékét, másodiknak az F_2 értékét és így tovább. Ezeket az értékeket valami speciális jellel elválaszthatjuk, hogy a képzett érték és az $F_1, F_2, \ldots F_k$ értékek listája közötti megfeleltetést egyértelművé tegyük.

5.1. példa: Ha az F_1 és F_2 mező értéke szöveg, illetőleg egész szám, és a # karakter nem fordulhat elő a szöveg mezőben, akkor az F_1 = `'abcd'` és az F_2 = 123 értékek egyesítését az `'abcd#123'` karaktersorozattal ábrázolhatjuk. □

A 4. fejezetben az egydimenziós kulcstér előnyeit többféle módon is kihasználtuk:

- A szekvenciális állományok és a B-fák indexeinél feltételeztük, hogy a kulcsok rendezett sorrendben vannak.
- A tördelőtáblák használata megköveteli, hogy a keresési kulcs teljes egészében ismert legyen bármely kereséskor. Ha egy kulcs több mezőből áll, és azokból akár csak egy is ismeretlen, már nem tudjuk a tördelőfüggvényt alkalmazni, hanem helyette végig kell keresni az összes kosarat.

Sok alkalmazás azt igényli tőlünk, hogy az adatokat kétdimenziós vagy néha magasabb dimenziós térben ábrázolhatónak tekintsük. Némelyeket ezek közül ki tud szolgálni egy szokásos adatbázis-kezelő rendszer, de vannak speciális rendszerek, amelyeket eleve többdimenziós alkalmazásokhoz terveztek. Egy fontos dolog, ami megkülönbözteti ezeket a szakosított rendszereket a többitől az, hogy olyan adatstruktúrákat használnak, amelyek támogatnak bizonyosfajta lekérdezéseket, amelyek nem általánosak az SQL-alkalmazásokban. Az 5.1. rész bemutat tipikus lekérdezéseket, amelyek olyan indexeket használnak, amiket arra terveztek, hogy többdimenziós

adatokat és többdimenziós lekérdezéseket támogassanak. Azután az 5.2. és az 5.3. részben a következő adatstruktúrákat tárgyaljuk:

1. *Rácsos állományok,* amelyek az egydimenziós tördelőtáblák egyfajta többdimenziós kiterjesztései.
2. *Particionált tördelőfüggvények,* ez egy másik módszer, amellyel többdimenziós adatokra alkalmazzuk a tördelőtáblák ötletét.
3. *Többkulcsos indexek,* ahol egy A attribútum indexe elvezet egy másik B attribútum indexeihez, az A minden lehetséges értékére.
4. *kd-fák,* amelyek a B-fák általánosításai ponthalmazokra.
5. *Quad-fák,* olyan fák, amelyben egy csomópont minden gyereke egy többdimenziós kockának felel meg.
6. *R-fák,* a B-fák olyan általánosítása, amely alkalmas területgyűjtemények kezelésére.

Végül, az 5.4. részben a *bittérképindexnek* nevezett struktúrát tárgyaljuk. Ezekben az indexek tömör kódok, amelyek adott mezőben adott értéket tartalmazó rekordok elérésére használhatók. Ezek a megoldások napjainkban kezdenek megjelenni a nagy kereskedelmi adatbázis-kezelő rendszerekben, és időnként kiváló választásnak bizonyulnak egydimenziós indexek kezelésére. Azonban bizonyosfajta többdimenziós lekérdezések megválaszolására is hatékony eszközök lehetnek.

5.1. Többdimenziós alkalmazások

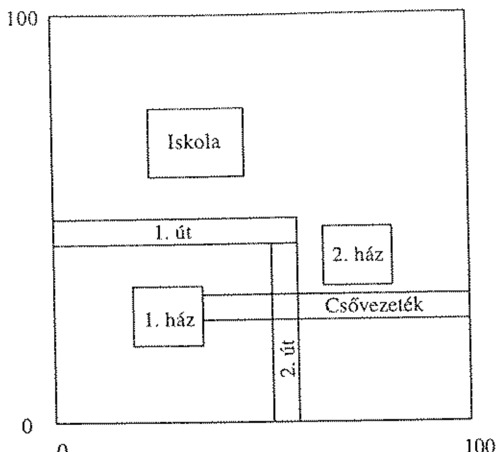
A többdimenziós alkalmazások két általános osztályát fogjuk áttekinteni. Az egyik *földrajzi* természetű, ahol az adatok a két- vagy néha a háromdimenziós világ elemei. A másik a dimenziók egy elvontabb fogalmát foglalja magában. Kissé pongyolán fogalmazva, tekinthetjük egy reláció minden attribútumát egy-egy dimenziónak, és minden sorát egy pontnak abban a térben, amelyet ezek a dimenziók meghatároznak.

Ebben a részben elemezzük azt is, hogyan használhatók a hagyományos indexek, például a B-fák, a többdimenziós lekérdezések támogatására. Bár bizonyos esetekben megfelelőek, arra is vannak példák, ahol a speciális struktúrák messze felülmúlják őket.

5.1.1. Térinformatikai rendszerek

Egy *térinformatikai rendszer* az objektumokat (tipikusan) egy kétdimenziós térben tárolja. Az objektumok lehetnek pontok vagy alakzatok. Ezek az adatbázisok gyakran térképek, ahol a tárolt objektumok házakat, utakat, hidakat, csővezetékeket és sok más fizikai tárgyat ábrázolhatnak. Egy ilyen térképre látunk mintát az 5.1. ábrán.

Azonban számos másfajta felhasználás is lehetséges. Például egy integrált áramkör terve is területek kétdimenziós térképe, ezek gyakran meghatározott anyagból készült téglalapok, úgynevezett rétegek. Hasonlóan, az ablakokat és ikonokat egy képernyőn szintén tekinthetjük kétdimenziós objektumok gyűjteményének.



5.1. ábra. *Néhány objektum egy kétdimenziós térben*

A térinformatikai rendszerek lekérdezései nem tipikus SQL-lekérdezések, de némi erőfeszítéssel sok kifejezhető SQL-ben. Ilyen típusú lekérdezésekre példák az alábbiak:

1. *Lekérdezések részleges egyezéssel.* Értékeket adunk meg egy vagy több dimenzióra, és az olyan pontokat keressük, amelyek megfelelnek a megadott értékeknek, a megadott dimenziókban.
2. *Tartománylekérdezések.* Tartományokat adunk meg egy vagy több dimenzióra, és az olyan pontok halmazát kérjük, amelyek ezeken a tartományokon belül vannak, vagy ha alakzatokat ábrázolunk, az olyan alakzatok halmazát, amelyek részben vagy teljesen a tartományon belül vannak. Ezek a lekérdezések a 4.3.4. részben tárgyalt egydimenziós tartománylekérdezések általánosításai.
3. *Legközelebbi szomszéd-lekérdezések.* Egy adott ponthoz legközelebbi pontot keressük. Például, ha egy pont egy várost jelent, és mi egy adott kisvároshoz legközelebbi 100 000-nél nagyobb lélekszámú várost akarjuk megtalálni.
4. *Hol-vagyok-én-lekérdezések.* Egy adott pontban vagyunk, és tudni akarjuk, hogy melyik alakzatban van benne ez a pont, ha van egyáltalán olyan alakzat, amiben benne van. Egy ismert példa erre, ami akkor történik, amikor az egérrel kattintunk, és a rendszer meghatározza, hogy mely látható elemre kattintottunk.

5.1.2. Adatkockák

Az utóbbi időkben megjelent az adatbázis-kezelő rendszerek azon családja, amit időnként *adatkocka*rendszereknek neveznek, amelyek az adatot többdimenziós térben létezőnek tekintik. Ezeket a 11.4. részben tárgyaljuk majd részletesebben, de a következő példa rávilágít az alapötletükre.

Többdimenziós adatokat sok vállalat gyűjt a *döntéstámogató* alkalmazások számára, amelyekkel elemzik az információkat, olyanokat mint az eladások, hogy jobban megértsék a vállalat működését. Például egy áruházlánc feljegyezheti minden eladásáról az alábbiakat:

1. a dátumot és időt,
2. az áruházat, amelyben az eladás történt,
3. a vásárolt árucikket,
4. az árucikk színét,
5. az árucikk méretét,

és esetleg az eladás egyéb tulajdonságait.

Szokásos az adatokat egy relációnak tekinteni, ahol minden tulajdonság a reláció egy attribútuma. Ezeket az attribútumokat egy többdimenziós tér, az „adatkocka" dimenzióinak tekinthetjük. Minden sor egy pont ebben a térben. Az elemzők aztán olyan kérdéseket tesznek fel, amelyek rendszerint csoportosítják az adatokat néhány dimenzió mentén, és a csoportokat összegzik valamilyen összesítő függvénnyel (aggregátorral). Egy jellemző példa lehet: „add meg a rózsaszínű ingek 1998-as eladásait áruházanként, havi bontásban".

5.1.3. Többdimenziós lekérdezések SQL-ben

Lehetséges a fent említett alkalmazások mindegyikét mint hagyományos, relációs adatbázist megvalósítani, és a felmerült lekérdezéseket SQL-ben megfogalmazni. Nézzünk néhány példát.

5.2. példa: Tegyük fel, hogy a legközelebbi szomszéd-lekérdezést akarjuk megválaszolni egy kétdimenziós térben lévő ponthalmazon. A pontokat ábrázolhatjuk valós számpárokból álló relációként.

```
Pontok(x, y)
```

Ennek két attribútuma van, x és y, amelyek a pont x és y koordinátái. A `Pontok` reláció további – itt nem mutatott – attribútumai esetleg a pontok egyéb jellemzőit ábrázolják.

Tegyük fel, hogy a (10.0, 20.0) ponthoz legközelebbi pontot keressük. Az 5.2. ábrán szereplő lekérdezés megtalálja a legközelebbi pontot, illetve pontokat, ha több ilyen is van. Minden egyes p pontra megnézi, létezik-e olyan másik q pont, amelyik közelebb van a (10.0, 20.0) ponthoz. A távolságok összehasonlítása úgy történik, hogy a (10.0, 20.0) pont és a lekérdezésben szereplő pont x, illetve y koordinátáinak különbségét négyzetre emeli és összeadja. Megjegyezzük, hogy nem kell gyököt vonni az összegből, hogy megkapjuk a tényleges távolságot, mert a távolság négyzetének összehasonlítása ugyanazt az eredményt adja, mintha a távolságértékeket magukat hasonlítanánk össze. □

```
SELECT *
FROM PONTOK p
WHERE NOT EXISTS(
    SELECT *
    FROM PONTOK q
    WHERE (q.x-10.0)*(q.x-10.0)+(q.y-20.0)*(q.y-20.0)<
          (p.x-10.0)*(p.x-10.0)+(p.y-20.0)*(p.y-20.0)
);
```

5.2. ábra. *Azon pontok keresése, amelyeknél nincs közelebbi a (10.0, 20.0) ponthoz*

5.3. példa: A téglalapok szokásos alakzatok a térinformatikai rendszerekben. Egy téglalapot többféleképpen ábrázolhatunk. Egyik népszerű forma a bal alsó sarok és a jobb felső sarok koordinátáinak megadása. Ezután a `Téglalapok` relációval ábrázolhatjuk a téglalapok gyűjteményét, ennek attribútumai a téglalap azonosítója, a négy koordináta, ami leírja a téglalapot, és esetleg bármi egyéb jellemzője a téglalapnak, amit rögzíteni szeretnénk. A következő relációt fogjuk használni ebben a példában:

```
Téglalapok(az, xba, yba, xjf, yjf)
```

Az attribútumok sorrendben a téglalap azonosítója, a bal alsó sarok x koordinátája, a bal alsó sarok y koordinátája, illetve a jobb felső sarok két koordinátája.

Az 5.3. ábrán szereplő lekérdezés azokat a téglalapokat keresi, amelyek tartalmazzák a (10.0, 20.0) pontot. A `WHERE` feltétel egyszerű. A (10.0, 20.0) pontot akkor tartalmazza a téglalap, ha a bal alsó sarkának x koordinátája 10.0 vagy attól balra van, és az y koordinátája 20.0 vagy az alatti. A jobb felső sarokra egyúttal az $x = 10.0$ vagy attól jobbra, és $y = 20.0$ vagy a feletti érték kell legyen. □

```
SELECT az
FROM Téglalapok
WHERE xba  <= 10.0  AND  yba <=    20.0    AND
      xjf  >= 10.0  AND  yjf >=    20.0;
```

5.3. ábra. *Téglalap(ok) keresése, amely(ek) tartalmaz(nak) egy adott pontot*

5.4. példa: Az adatkockarendszereknek megfelelő adatok általában *ténytáblába* – ezekben az alapadatok vannak rögzítve (pl. minden eladás) – és *dimenziótáblákba* – ezek tartalmazzák az egyes értékekhez tartozó jellemzőket az egyes dimenziókban – vannak szervezve. Például, ha az áruház, amely eladott valamit, az egy dimenzió, akkor az áruházhoz tartozó dimenziótábla megadhatja az áruház vezetőjének a címét, a telefonszámát és a nevét.

Ebben a példában csak a ténytáblával foglalkozunk, amelynek feltevésünk szerint az 5.1.2. részben javasolt dimenziói vannak. Tehát a ténytábla a következő reláció:

```
Eladások(nap, áruház, cikk, szín, méret)
```


Az „összegezd a rózsaszínű ingek eladását áruházanként, napi bontásban" lekérdezés az 5.4. ábrán látható. A lekérdezés csoportosítja az eladásokat a `nap` és az `áruház` dimenzió értékei szerint, miközben összefogja a többi dimenziót a `COUNT` összesítő függvénnyel. Az adatkocka csak azon részeire figyelünk, ami bennünket érdekel, így a `WHERE` feltétel csak a rózsaszín ingek sorait választja ki. □

```
SELECT nap, áruház, COUNT(*) AS összEladás
FROM Eladások
WHERE cikk  =  'ing' AND
      szín  =  'rózsaszín'
GROUP BY nap, áruház;
```

5.4. ábra. *A rózsaszínű ingek eladásának összegzése*

5.1.4. Tartománylekérdezések végrehajtása hagyományos indexekkel

Most tekintsük át, hogy a 4. fejezetben leírt indexek milyen mértékben segíthetik a tartománylekérdezések megválaszolását. Az egyszerűség kedvéért tegyük fel, hogy két dimenziónk van. Az x és az y dimenziók mindegyikére tehetünk egy másodlagos indexet. Mindkettőhöz B+-fát használva különösen könnyen lehet értékek egy tartományát visszanyerni bármelyik dimenzió esetén.

Ha mindkét dimenzióra adott egy tartomány, akkor kezdhetjük az x-hez tartozó B-fával, hogy visszanyerjük az összes olyan rekord mutatóját, amely az x-hez megadott tartományba esik. Azután az y B-fáját használva megkapjuk az összes olyan ponthoz tartozó rekord mutatóját, amelynek az y koordinátája az y-hoz megadott tartományba esik. Végül vesszük a mutatók metszetét, a 4.2.3. rész ötletét használva. Ha a mutatók elférnek a központi memóriában, akkor a szükséges lemez I/O-műveletek száma megegyezik a két B-fában a megvizsgálandó levélcsomópontok számával, plusz még néhány lemez I/O-művelet, amíg megtaláljuk a B-fákban lefelé az utat (lásd a 4.3.7. részt). Ehhez kell még hozzáadni a megfelelő rekordok eléréséhez szükséges lemez I/O-műveletek számát.

5.5. példa: Tekintsük 1 000 000 pontnak egy feltételezett halmazát, amely pontok véletlenszerűen oszlanak el a kétdimenziós térben, és az x, illetve y koordinátáik egyaránt 0 és 1000 közé esnek. Tegyük fel, hogy 100 pont rekordja fér el egy blokkban, és egy átlagos B-fa-levél körülbelül 200 kulcs-mutató párt tartalmaz (emlékeztetünk rá, hogy egy B-fa-blokk nem feltétlen minden bejegyzése foglalt minden időpontban). Feltételezzük továbbá, hogy van B-fa-index x-hez és y-hoz is.

Képzeljünk el egy olyan tartománylekérdezést, amely egy a tér közepén levő 100 egység oldalú négyzetbe eső pontok számát kérdezi le: $450 \le x \le 550$ és $450 \le y \le 550$. Az x-hez tartozó B-fát használva megkaphatjuk az összes rekord mutatóját, amely az x szerinti tartományba esik, ez körülbelül 100 000 mutató lehet, és ez elférhet a központi memóriában. Hasonlóan, az y-hoz tartozó B-fát használva megkaphatjuk az összes olyan rekord mutatóját is, amely y szerint a kívánt tartományba esik, ezekből ismét körülbelül 100 000 lesz. E két halmaz metszete körülbelül 10 000 mutató, és ezzel

a metszetben szereplő 10 000 mutatóval elérhető rekordok alkotják a választ a lekérdezésünkre.

Most becsüljük meg a tartománylekérdezés megválaszolásához szükséges lemez I/O-műveletek számát. Először is, ahogy a 4.3.7. részben rámutattunk, általában megvalósítható, hogy mindegyik B-fa gyökerét a központi memóriában tartsuk. Mivel keresésikulcs-értékek egy tartományát keressük a B-fákban, és a mutatók a levelekben e szerint a keresési kulcs szerint rendezettek, így mindössze annyit kell tennünk ahhoz, hogy dimenziónként hozzáférjünk a kb. 100 000 mutatóhoz, hogy átvizsgálunk egy köztes szintű csomópontot, valamint az összes levelet, amely a kívánt mutatókat tartalmazza. Mivel feltételeztük, hogy egy levél körülbelül 200 kulcs-mutató párt tartalmaz, így mindkét B-fa esetén körülbelül 500 levélblokkot kell megnéznünk. Ha ehhez hozzáadjuk B-fánként az egy köztes szintű csomópontot, akkor összesen 1002 lemez I/O-műveletet kapunk.

Végül vissza kell nyerni a blokkokat, amelyek a 10 000 kívánt rekordot tartalmazzák. Ha ezek véletlenszerűen vannak tárolva, azt várhatjuk, hogy ezek közel 10 000 különböző blokkban helyezkednek el. Mivel az egymilliós teljes állomány – a feltevés szerint 100 rekord tölt meg egy blokkot – 10 000 blokkban tárolódik, ezért nagyjából az adatállomány minden blokkját végig kell néznünk. Így, legalábbis ebben a példában, a hagyományos indexek nem, vagy csak alig segítették a tartománylekérdezés megválaszolását. Természetesen, ha a tartomány kisebb lett volna, a mutatóhalmazok metszetének létrehozása lehetővé tehette volna számunkra, hogy a keresés az adatállomány blokkjainak töredékére korlátozódjon. □

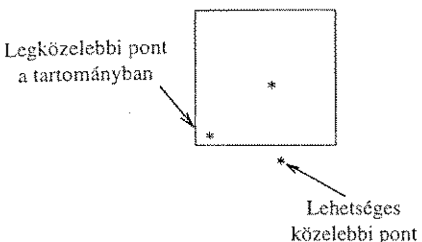
5.1.5. Legközelebbi szomszéd-lekérdezések végrehajtása hagyományos indexekkel

Majdnem minden általunk használt adatstruktúra lehetővé teszi a legközelebbi szomszéd-lekérdezések megválaszolását azáltal, hogy kiválasztunk egy tartományt minden dimenzióhoz, megválaszoljuk a tartománylekérdezést, majd kiválasztjuk a célhoz legközelebbi pontot a tartományon belül. Két probléma merülhet fel:

1. Nincs pont a kiválasztott tartományban.
2. A tartományon belüli legközelebbi pont lehet, hogy nem a legközelebbi.

Vizsgáljuk meg mindkét esetet az 5.2. példa legközelebbi szomszéd-lekérdezésével kapcsolatban, használva az 5.5. példában bemutatott x és y dimenzióra feltételezett indexeket. Ha jó okunk van feltételezni, hogy a (10.0, 20.0)-hoz d távolságon belül létezik pont, akkor használhatjuk az x-hez tartozó B-fát, hogy visszanyerjük azon pontok mutatóit, amelyek x koordinátája $10 - d$ és $10 + d$ között van. Azután használhatjuk az y-hoz tartozó B-fát, hogy visszanyerjük azon pontok mutatóit, amelyek y koordinátája $20 - d$ és $20 + d$ között van.

Ha van egy vagy több pont a metszetben, és minden mutatóhoz feljegyeztük az x, illetve az y koordinátát is (ami a keresési kulcs volt az indexhez), akkor a metszetben



5.5. ábra. *Legközelebbi pont a tartományban, ám lehet közelebbi pont a tartományon kívül*

rendelkezésünkre áll minden egyes pont koordinátája. Ezért meghatározhatjuk, hogy melyik a (10.0, 20.0)-hoz legközelebbi pont, és visszanyerhetjük annak a rekordját. Sajnos nem lehetünk biztosak benne, hogy van pont az adott ponthoz d távolságon belül, ezért lehet, hogy meg kell ismételni az eljárást egy nagyobb d értékkel.

Azonban, ha van is pont az átnézett tartományban, bizonyos esetekben az adott ponthoz – mely példánkban a (10.0, 20.0) – legközelebbi pont a tartományon belül, lehet, hogy a d távolságnál messzebb van. Egy ilyen helyzetet mutat az 5.5. ábra. Ha ez a helyzet, akkor meg kell növelnünk a tartományt, és újra kell keresnünk, hogy meggyőződjünk arról, hogy nincs közelebbi pont. Ha az eddig talált legközelebbi pont és az adott pont távolsága d', és $d' > d$, akkor meg kell ismételnünk a keresést d'-t használva d helyett.

5.6. példa: Tekintsük az 5.5. példában szereplő adatokat és indexeket. Ha a legközelebbi szomszédját keressük az adott $P = (10.0, 20.0)$ pontnak, választhatjuk a $d = 1$ értéket. Átlagosan egy pont jut a terület egy egységére, és $d = 1$ esetén megtalálunk minden olyan pontot, amely a P pont körüli 2.0 oldalú négyzetbe esik, ezeknek a várható száma 4.

Ha megvizsgáljuk az x koordinátához tartozó B-fát a $9.0 \leq x \leq 11.0$ tartománylekérdezés esetén, akkor a lekérdezés körülbelül 2000 pontot fog találni, így legalább 10 levelet be kell járnunk, de valószínűbb, hogy 11-et (mivel valószínűtlen, hogy az $x = 9.0$ értékű pontok éppen egy levél elején kezdődnek). Mint az 5.5 példában, várhatóan a B-fák gyökereit a központi memóriában tudjuk tartani, így csak egy lemez I/O-műveletre van szükségünk egy köztes szintű csomópont eléréséhez, és 11 lemez I/O-műveletre a levelekhez. További 12 lemez I/O-művelet kell, hogy az y koordináta B-fájában megkeressük azokat a pontokat, amelyeknek az y koordinátája 19.0 és 21.0 között van.

Ha a megközelítőleg 4000 mutató metszetét képezzük a központi memóriában, várhatóan négy rekordunk lesz, amelyek közül kikerülhet a (10.0, 20.0) pont legközelebbi szomszédja. Feltéve, hogy van legalább egy ilyen rekord, a mutatókhoz tartozó x és y koordinátákból meghatározhatjuk melyik a legközelebbi szomszéd. Még kell egy lemez I/O-művelet, hogy visszanyerjük a kívánt rekordot, így összesen 25 lemez I/O-művelet kellett, hogy befejezzük a lekérdezést. Azonban, ha $d = 1$ értékhez tartozó négyzetben nincs pont, vagy a legközelebbi pont távolsága az adott ponttól nagyobb, mint 1, meg kell ismételnünk a lekérdezést egy nagyobb d értékkel. □

Azt a következtetést vonhatjuk le az 5.6. példából, hogy a hagyományos indexek nem feltétlenül rosszak a legközelebbi szomszéd-lekérdezésekhez, de lényegesen több lemez I/O-műveletet igényelnek, mint amennyit használnánk, ha mondjuk egy rekordot adott kulcs és a kulcshoz tartozó B-fa alapján (amely valószínűleg csak két vagy három lemez I/O-művelettel járna) keresnénk meg. Az ebben a fejezetben ajánlott módszerek általában jobb teljesítményt nyújtanak, és ezeket használják a többdimenziós adatokat támogató szakosított adatbázis-kezelő rendszerek is.

5.1.6. A hagyományos indexek további korlátjai

A fent említett struktúrák költsége nem jobb tartománylekérdezésekre sem, mint a legközelebbi szomszéd-lekérdezések esetén. Gyakorlatilag az 5.6. példában úgy közelítettünk a legközelebbi szomszéd-lekérdezés megoldásához, hogy tulajdonképpen – egy kisméretű tartománnyal minden dimenzióra – tartománylekérdezéssé alakítottuk át azt, és reméltük, hogy a tartomány mérete elégséges ahhoz, hogy legalább egy pont beleessen. Következésképpen, ha egy tartománylekérdezéshez nagyobb tartományokat választanánk, és az adatstruktúrák indexek lennének minden dimenzióra, akkor a megfelelő rekordok mutatóinak eléréséhez szükséges lemez I/O-műveletek száma minden dimenzióban több lenne, mint amennyit az 5.6. példában találtunk.

Az 5.4. ábrán látható lekérdezés többdimenziós összegzése szintén nem szerencsés. Ha van indexünk a cikk és a szín attribútumra, akkor megtalálhatjuk a rózsaszínű ingek eladásához tartozó összes rekordot, a metszetüket véve, ahogyan az 5.6. példában tettük. Azonban az olyan lekérdezések esetében, amelyeknél a szín és a cikk attribútumok mellett más attribútumok is adottak, inkább az azokra az attribútumokra megadott indexekre lenne igény.

Sőt amíg az adatállományt rendezetten tudjuk tartani az öt attribútum valamelyike szerint, már nem tudjuk két attribútum szerint sorrendben tartani, nem is szólva az ötről. Így az 5.4. példában mutatott alakú lekérdezések többségénél szükség lenne az adatállomány minden vagy majdnem minden blokkjának az elérésére. Az ilyen típusú lekérdezések végrehajtása rendkívül költséges lenne, különösen ha az adatok háttértárolón vannak.

5.1.7. A többdimenziós indexstruktúrák áttekintése

A többdimenziós adatok lekérdezését támogató legtöbb adatstruktúra a következő két kategória egyikébe tartozik:

1. tördelőtábla alapú,
2. fastruktúrájú.

Mindkét fenti struktúránál feladunk valamit abból, amivel rendelkeztünk a 4. fejezet egydimenziós struktúráinál.

- A tördelőtábla alapú elgondolásoknál – rácsos állományok és particionált tördelőfüggvények az 5.2. részben – a továbbiakban nem lesz meg az az előnyünk, hogy a lekérdezésünkre adott válasz pontosan egy kosárban van. Azonban minden ilyen elgondolásnál a keresés a kosarak egy részhalmazára korlátozódik.
- A fastruktúrájú elgondolásoknál feladunk legalább egyet a következő B-fa-tulajdonságok közül:

1. A fa kiegyensúlyozottságát, ahol az összes levél ugyanazon a szinten van.
2. A facsomópontok és a lemezblokkok közötti megfeleltetést.
3. A sebességet, amellyel az adatok módosítását végre lehet hajtani.

Amint látni fogjuk az 5.3. részben, a fák gyakran mélyebbek lesznek bizonyos részeken, mint máshol, és gyakran a mélyebb részek sok pontot tartalmazó területeknek felelnek meg. Szintén látni fogjuk, hogy gyakran egy fa csomópontjához tartozó információ lényegesen kisebb méretű annál, mint ami elfér egy blokkban. Így valamilyen hasznos módon blokkokba célszerű csoportosítani a csomópontokat.

5.1.8. Feladatok

5.1.1. feladat: Írjunk SQL-lekérdezéseket az 5.3. példából vett

```
Téglalapok(az, xba, yba, xjf, yjf)
```

relációt használva, amelyek megválaszolják a következő kérdéseket:

* a) Keressük meg azon téglalapok halmazát, amelyeknek van közös része azzal a téglalappal, amelynek bal alsó sarka a (10.0, 20.0), a jobb felső sarka pedig a (40.0, 30.0) pont.
b) Keressük meg azon téglalappárokat, melyek átfedik egymást.
c) Keressük meg azokat a téglalapokat, amelyek teljesen tartalmazzák az a)-ban említett téglalapot.
d) Keressük meg azokat a téglalapokat, melyek teljesen benne vannak az a)-ban említett téglalapban.
! e) Keressük meg azokat a „téglalapokat" a `Téglalapok` relációban, amelyek valójában nem téglalapok, azaz fizikailag nem létezhetnek.

Mindegyik lekérdezésnél mondjuk meg, mely indexek – ha vannak ilyenek – segíthetnek a kívánt sorok elérésében.

5.1.2. feladat: Az 5.4. példából vett

```
Eladások(nap, áruház, cikk, szín, méret)
```

relációt használva adjuk meg a következő lekérdezéseket SQL-ben:

* a) Listázzuk ki az ingek színeit, és az eladások összesített számát az olyan színekre, amelyekből 1000-nél többet adtak el.
b) Listázzuk ki az ingek eladásait áruházanként és színenként.
c) Listázzuk ki az összes árucikk eladásait áruházanként és színenként.
! d) Listázzuk ki minden árucikkre és színre azt az áruházat, amely a legtöbbet adta el, és listázzuk ki ezeknek az eladásoknak a számát is.

5.1.3. feladat: Oldjuk meg újra az 5.5. példát azzal a feltételezéssel, hogy a tartománylekérdezés egy középen lévő n egység oldalú négyzetre vonatkozik, ahol n egy tetszőleges 1 és 1000 közötti érték. Hány lemez I/O-művelet szükséges? Milyen n értékeknél segítenek az indexek ténylegesen?

* **5.1.4. feladat:** Ismételjük meg az 5.1.3. feladatot úgy, hogy a rekordok adatállománya rendezett x-re.

!! **5.1.5. feladat:** Tegyük fel, hogy egy négyzetben véletlenszerűen elosztott pontjaink vannak (ahogy az 5.6. példában), és egy legközelebbi szomszéd-lekérdezést akarunk végrehajtani. Választunk egy d távolságot, és megkeressük az összes olyan pontot, ami abba a $2d$ oldalú négyzetbe esik, amelynek a középpontja az adott pont. A keresésünk akkor sikeres, ha találunk a négyzetben legalább egy pontot, amelynek a távolsága az adott ponttól d vagy annál kisebb.

* a) Ha egységnyi területen átlagosan egy pont van, adjuk meg annak a valószínűségét d függvényében, hogy sikeresek leszünk.
b) Ha sikertelenek vagyunk, meg kell ismételnünk a keresést egy nagyobb d-vel. Tegyük fel az egyszerűség kedvéért, hogy minden esetben, amikor sikertelenek vagyunk, akkor megduplázzuk a d-t, és kétszer annyi a költségünk, mint amennyi az előző kereséskor volt. Ismét csak feltesszük, hogy egységnyi területen átlagosan egy pont van. Melyik az a d kezdőérték, ami a minimális várható keresési költséget adja?

5.2. Tördelésen alapuló struktúrák többdimenziós adatokhoz

Ebben a részben áttekintünk két olyan adatstruktúrát, amely általánosítja az egyetlen kulcs használatára épülő tördelőtáblákat. Mindkét esetben a ponthoz rendelt kosár az összes attribútum, illetve dimenzió függvénye. Az egyik szerkezet az ún. „rácsos állomány", ez általában nem tördeli az értékeket a dimenzió mentén, hanem inkább felosztja a dimenziót, rendezve az értékeket a dimenzió mentén. A másik az ún. particionált tördelés, ami tényleg tördeli a különböző dimenziókat, és minden egyes dimenzió részt vesz a kosársorszám kialakításában.

5.2.1. Rácsos állományok

Az egyik legegyszerűbb adatstruktúra, amely gyakran felülmúlja teljesítményben az egydimenziós indexeket a többdimenziós adatokat magában foglaló lekérdezéseknél, a *rácsos állomány* (grid file). Gondoljunk egy pontokból álló térre, amelyet rácsok osztanak fel. Minden egyes dimenzióban *rácsvonalak* (grid lines) osztják fel a teret *sávokra* (stripes). Azokat a pontokat, amelyek egy rácsvonalra esnek, ahhoz a sávhoz tartozónak tekintjük, amelynek a rácsvonal az alsó határa. A különböző dimenziókhoz különböző számú rácsvonal tartozhat, és a szomszédos rácsvonalak között különböző lehet a távolság, még azonos dimenzión belül is.

5.7. példa: Bevezetjük e fejezet állandó példáját: a lekérdezésünk legyen „ki vásárol arany ékszert?". Képzeljük el az arany ékszerek vásárlóinak adatbázisát, amely számos dolgot mond nekünk minden egyes vásárlóról – a nevét, címét stb. Azonban, hogy a dolgokat egyszerűbbé tegyük, feltételezzük, hogy csak a vásárló életkora és fizetése a lényeges attribútum. A példa adatbázisunkban 12 vásárló van, akiket a következő életkor-fizetés párokkal ábrázolhatunk.

(25, 60)	(45, 60)	(50, 75)	(50, 100)
(50, 120)	(70, 110)	(85, 140)	(30, 260)
(25, 400)	(45, 350)	(50, 275)	(60, 260)

Az 5.6. ábrán látható a 12 pont elhelyezkedése egy kétdimenziós térben. Néhány rácsvonalat is választottunk mindegyik dimenzióban. Ehhez az egyszerű példához két rácsvonalat választottunk mindegyik dimenzióban, ezzel a teret 9 téglalap alakú területre osztottuk, de persze semmi nem indokolja, hogy ugyanannyi vonalat használjunk minden dimenzióban. Azt szintén megengedtük, hogy a vonalak között különböző távolságok legyenek. Például az életkor dimenziónál, a három terület – amire a két függőleges vonal osztja a teret – szélessége 40, 15 és 45.

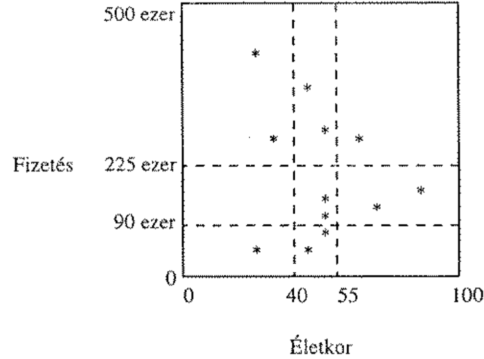
Ebben a példában nem esik pont a rácsvonalakra. De általánosságban a pont a téglalaphoz tartozik, ha az alsó vagy a bal oldali élére esik, és nem tartozik hozzá, ha a felső vagy a jobb oldali élén van. Például az 5.6. ábrán lévő középső téglalap azokat a pontokat tartalmazza, amelyekre $40 \leq életkor < 55$ és $90 \leq fizetés < 225$. □

5.2.2. Keresés rácsos állományban

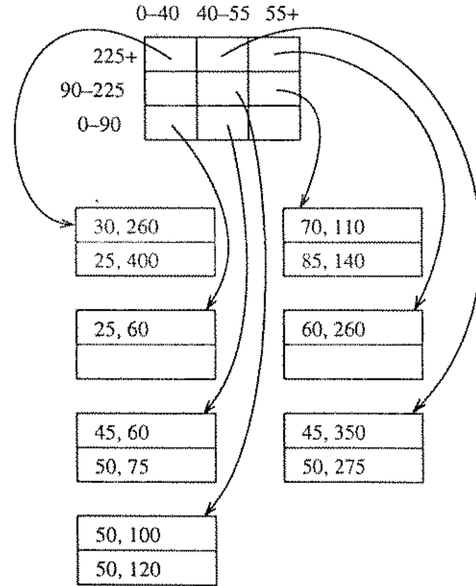
Mindegyik terület – amire a teret felosztottuk – úgy tekinthető, mint egy kosár egy tördelőtáblában, és minden ezen a területen lévő pontnak megvan a rekordja az ahhoz a kosárhoz tartozó valamely blokkban. Túlcsordulásblokkok használhatók – ha szükséges – a kosár méretének növelésére.

A kosarak egydimenziós tömbje helyett – ahogyan azt a hagyományos tördelőtábláknál láttuk – a rácsos állomány olyan tömböt használ, amelynek a dimenziószáma megegyezik az adatállományéval. Ahhoz, hogy egy ponthoz a megfelelő kosarat meg-

határozzuk, tudnunk kell minden dimenzióra az értékek listáját, ahol a rácsvonalak vannak. Egy pont hasítása így némileg különbözik attól, mint amikor egy tördelőfüggvényt alkalmazunk a változóinak értékeire. Vesszük a pont komponenseit, és meghatározzuk a pont helyzetét az ahhoz a dimenzióhoz tartozó rácson. A pont dimenziónként meghatározott helyzetei együtt határozzák meg magát a kosarat.



5.6. ábra. *Egy rácsos állomány*



5.7. ábra. *Egy rácsos állomány, amely az 5.6. ábra pontjait ábrázolja*

5.8. példa: Az 5.7. ábrán láthatjuk az 5.6. ábra adatainak elhelyezését a kosarakban. Mivel a rácsok a teret mindkét dimenziónál három területre osztják, a kosarak tömbje egy 3 × 3 -as mátrix. Kettő a kosarak közül üres:

1. A fizetés 90 ezer és 225 ezer dollár között, és az életkor 0 és 40 év között,

valamint

2. A fizetés 90 ezer dollár alatt, és az életkor 55 év fölött.

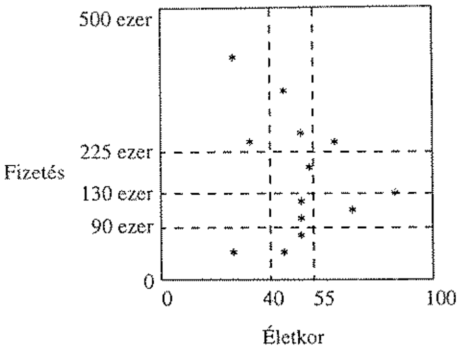
Ezért nem is jelölünk blokkot ehhez a két kosárhoz. A többi kosárhoz tartozó blokkban maximum két pont szerepelhet, ami egy mesterségesen alacsonyan tartott érték. Ebben az egyszerű példában nincs olyan kosár, amelynek kettőnél több tagja lenne, így túlcsordulásblokkokra sincs szükség. □

5.2.3. Beszúrás rácsos állományba

Amikor rekordot szúrunk be egy rácsos állományba, először a rekordkeresés eljárását követjük, és az új rekordot az így kapott kosárba tesszük. Ha van hely a kosár blokkjában, nincs is más teendőnk. Akkor van gond, ha nincs hely a kosárban. Két általános megközelítés lehetséges:

1. Adjunk túlcsordulásblokkokat a kosárhoz, ha szükséges. Ez a megoldás jól működik, egészen addig, amíg a kosár blokkjainak lánca nem válik túl naggyá. Ez utóbbi esetben a kereséshez, beszúráshoz vagy a törléshez szükséges lemez I/O-műveletek száma végül elfogadhatatlanul nagyra nőhet.
2. Szervezzük újra a struktúrát rácsvonalak hozzáadásával vagy elmozgatásával. Ez a megoldás hasonló a 4.4. részben tárgyalt dinamikus tördelési technikához, de vannak további problémái, mert a kosarak tartalma függ a dimenziótól. Azaz egy rácsvonal hozzáadása szétvágja az adott vonal mentén lévő összes kosarat. Ennek következtében elképzelhető, hogy nem lehetséges olyan rácsvonalat választani, amely mindegyik kosár számára a legjobb lenne. Például, ha egy kosár túl nagy, lehet hogy nem tudunk egy szétvágandó dimenziót vagy egy vágási pontot választani anélkül, hogy sok üres kosarat ne képeznénk vagy számos telit ne hagynánk meg.

5.9. példa: Tegyük fel, hogy valaki, aki 52 éves és 200 ezer dollár jövedelme van, arany ékszert vásárol. Ez a vásárló az 5.6. ábrán a középső téglalapba tartozik. Azonban, most már három rekord van a kosárban. Egyszerűen hozzáadhatunk egy túlcsordulásblokkot. Ha szét akarjuk vágni a kosarat, ki kell választanunk vagy az életkor, vagy a fizetés dimenziót, és választanunk kell egy új rácsvonalat, ami létrehozza a felosztást. Csak három lehetőség van, hogy egy olyan rácsvonalat vezessünk be, amely úgy vágja ketté a középső kosarat, hogy két pont kerüljön az egyik oldalra, és egy a másikra, amely esetünkben a legegyenletesebb lehetséges szétvágás.



5.8. ábra. *Az (52, 200) pont beszúrása a kosarak szétvágásával*

1. Egy függőleges vonal, mint például az életkor = 51, ami elválasztja a két 50 évest az 52-től. Ez a vonal nem vág szét semmit az alatta és felette lévő kosarakból, mivel mindkét pontja ennek a két további – 40–55 életkorhoz tartozó – kosárnak az életkor = 51 vonal bal oldalára esik.
2. Egy vízszintes vonal, ami elválasztja a középső kosárban a fizetés = 200 pontot a másik két ponttól. Választhatunk olyan számot, mondjuk a 130-at, amely szétvágja a jobbra lévő kosarat is (azaz az életkor 55–100 és fizetés 90–225 kosarat).
3. Egy vízszintes vonal, ami elválasztja a fizetés = 100 pontot a másik két ponttól. Ismét javasolhatunk olyan számot, mondjuk a 115-öt, amely szétvágja a tőle jobbra lévő kosarat is.

Az 1. lehetőség valószínűleg nem javasolt, mert nem vág szét egyetlen további kosarat sem; több üres kosarunk lesz, és egyetlen foglalt kosárnak sem csökkenti a méretét. A 2. és a 3. lehetőség egyformán jó, de mi a 2.-at választanánk, mivel a hozzáadott vízszintes fizetés = 130 rácsvonal közelebb van a 90 és 225 alsó, illetve felső határ közepéhez, mint a 3. választásakor. A kosarak eredményül kapott felosztását az 5.8. ábra mutatja. □

5.2.4. A rácsos állományok hatékonysága

Tekintsük át mennyi lemez I/O-műveletet igényel a rácsos állomány a különféle típusú lekérdezések esetén. Eddig a rácsos állomány kétdimenziós változatára koncentráltunk, bár használható tetszőleges számú dimenzió esetén. Egyik fő probléma a sokdimenziós eseteknél, hogy a kosarak száma exponenciálisan nő a dimenziók számával. Ha a tér nagy darabjai üresek, akkor sok üres kosár lesz. Megvilágíthatjuk a problémát akár két dimenzióban is. Tegyük fel, hogy szoros az összefüggés az életkor és a fizetés között, akkor az 5.6. ábra összes pontja az átló mentén fekszik, így mindegy hová helyezzük a rácsvonalakat, az átlótól távolabb lévő kosarak üresek lesznek.

Rácsos állomány kosarainak elérése

Míg egy háromszor hármas rácsban – amilyen az 5.7. ábrán látható – könnyű megtalálni egy pont megfelelő koordinátáit, ne feledjük: egy rácsos állománynak nagyon sok sávja lehet mindegyik dimenziójában. Ha így van, akkor indexet kell létrehoznunk minden egyes dimenzióhoz. Egy ilyen index keresési kulcsa az adott dimenzió felosztási értékeinek halmaza.

Adott egy v érték valamely koordinátára, és keressük azt a legnagyobb w kulcsértéket, amely kisebb vagy egyenlő mint v. Az indexben a w-hez rendelt érték a mátrix azon sora, vagy oszlopa lesz, amelyikbe v esik. Megadva az értékeket mindegyik dimenzióhoz, megtalálhatjuk azt a helyet, ahova a mátrixban a kosár mutatója esik. Ezzel a mutatóval aztán közvetlenül elérhetjük a blokkot.

Szélsőséges esetekben a mátrix olyan nagy, hogy a kosarak nagy része üres, és nem áll módunkban tárolni az üres kosarakat. Ekkor úgy kell kezelnünk a mátrixot, mint egy relációt, amelynek az attribútumai a nem üres kosarak sarkai, és egy záró attribútum ábrázolja a mutatót a kosárra. Az ilyen relációban való keresés is többdimenziós, de a mérete kisebb, mint magának az adatállománynak.

Azonban, ha az adatok eloszlása jó, és az adatállomány maga nem túl nagy, akkor ki tudjuk választani a rácsvonalakat úgy, hogy:

1. Kellően kisszámú kosár van ahhoz, hogy a kosármátrixot a központi memóriában tartsuk, így nem kell külön lemez I/O-művelet ahhoz, hogy megnézzük a mátrixot, vagy új sorokat, illetve oszlopokat adjunk hozzá, amikor új rácsvonalat veszünk fel.
2. A rácsvonalak értékeinek indexét is a memóriában tudjuk tartani mindegyik dimenzióban (lásd a „Rácsos állomány kosarainak elérése" című bekeretezett részt), vagy el tudjuk kerülni az indexeket, és a dimenziók rácsvonalait meghatározó értékeken bináris keresést tudunk alkalmazni a központi memóriában.
3. Egy tipikus kosárnak legfeljebb néhány túlcsordulásblokkja lehet, tehát ez nem okoz túl sok további lemez I/O-műveletet, amikor végignézzük a kosarat.

Ezen előfeltevések mellett, bemutatjuk a rácsos állományok viselkedését a lekérdezések néhány fontos osztályára.

Adott pont keresése

A megfelelő kosárhoz vezet bennünket, így a szükséges lemez I/O-művelet csak ennek a kosárnak a beolvasása. Beszúrás vagy törlés esetén egy további lemezírás szükséges. Ha a beszúrás túlcsordulásblokk létrehozásához vezet, az egy további írási műveletet jelent.

Lekérdezések részleges egyezéssel

Az ilyen lekérdezésre példák: „keresd meg az összes 50 éves vásárlót", vagy „keresd meg az összes vásárlót, akinek a fizetése 200 ezer dollár". Most meg kell néznünk az összes kosarat a kosármátrix egy sorában vagy oszlopában. A lemez I/O-műveletek száma igen magas lehet, ha sok kosár van abban a sorban, illetve oszlopban.

Tartománylekérdezések

Egy tartománylekérdezés a rács egy téglalap alakú területét határozza meg, és az összes pont, amit a területen belüli kosarakban találunk a lekérdezés eredményéhez tartozik, néhány olyan pont kivételével, amelyek a kijelölt terület határára eső kosarakban találhatók. Például, ha meg akarjuk találni a 35–45 éves vásárlókat, akiknek a fizetése 50 ezer–100 ezer dollár, akkor négy kosarat kell megnéznünk az 5.6. ábra bal alsó részén. Ebben az esetben minden kosár a határokon van, így jó sok pontot meg kell néznünk, amelyek esetleg nem tartoznak a kérdésre adott válaszhoz. Azonban ha olyan területet vizsgálunk, amely sok kosarat tartalmaz, akkor ezek nagy része szükségképpen belső, így minden pontjuk a válaszhoz tartozik. Tartománylekérdezéseknél a lemez I/O-műveletek száma nagy is lehet, mivel esetleg sok kosarat kényszerülünk megvizsgálni. Bár a tartománylekérdezések hajlamosak nagy eredményhalmazt produkálni, általában nem kell sokkal több blokkot megvizsgálnunk, mint a minimális blokkszám, amennyibe az eredmény egyáltalán elhelyezhető tetszőleges szervezés esetén.

Legközelebbi szomszéd-lekérdezések

Adott egy P pont, a keresést azzal a kosárral kezdjük, amelyikhez ez a pont tartozik. Ha találunk legalább egy pontot abban, akkor van egy Q jelöltünk a legközelebbi szomszédra. Azonban lehetnek a szomszédos kosarakban olyan pontok, amelyek közelebb vannak P-hez mint a Q; a helyzet hasonló, mint amit az 5.5. ábra mutat. Meg kell vizsgálnunk, vajon a P és az őt tartalmazó kosár határai közötti távolság kisebb-e, mint P és Q távolsága. Ha vannak ilyen határok, akkor minden ilyen határ túloldalán lévő szomszédos kosarat szintén át kell néznünk. Valójában, ha a kosarak olyan téglalapok, amelyek sokkal hosszabbak az egyik dimenzió mentén, mint a másikban, akkor szükséges lehet megvizsgálni még olyan kosarakat is, amelyek nem is szomszédosak a P pontot tartalmazóval.

5.10. példa: Tegyük fel, hogy az 5.6. ábrán, a $P = (45, 200)$ ponthoz legközelebbi pontot keressük. A kosárban az (50, 120) pont van hozzá a legközelebb, a távolsága 80.2. Az alsó három kosárban nem lehet egy pont sem közelebb a (45, 200)-hoz, mivel a fizetés részük legfeljebb 90, így ezeket kihagyhatjuk a keresésből. Azonban a másik öt kosarat át kell néznünk, és azt találjuk, hogy valójában két egyformán közeli pont van: (30, 260) és (60, 260), P-től 61.8 távolságra. Általában, a legközelebbi szomszéd keresését néhány kosárra, és így néhány lemez I/O-műveletre lehet korlátozni. Azonban, mivel a P ponthoz legközelebbi kosarak lehetnek üresek, nem tudunk könnyen felső korlátot adni a keresés költségére. □

5.2.5. Particionált tördelőfüggvények

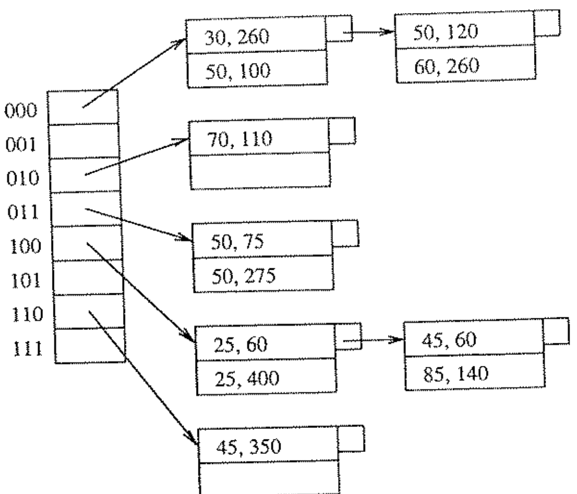
A tördelőfüggvényeknek az argumentuma lehet attribútumértékek egy listája, bár a tipikus az, hogy csak egyetlen attribútumból képzik a tördelőértékeket. Például, ha a egy egész értékű attribútum, b pedig egy szöveg értékű attribútum, akkor hozzáadhatjuk a értékéhez b minden egyes karakterének ASCII kód értékét, majd az így kapott értéket elosztjuk a kosarak számával, és vesszük ennek maradékát. Az eredményt használhatjuk egy tördelőtábla kosárszámaként, mint egy indexet az (a, b) attribútumpáron.

Azonban az ilyen tördelőtábla csak olyan lekérdezésekhez használható, amelyekben az a és a b értéke egyaránt adott. Célszerűbb választás úgy tervezni a tördelőfüggvényt, hogy állítson elő, mondjuk k számú bitet. Ezt a k bitet felosztjuk az n darab attribútum között úgy, hogy a tördelő érték k_i darab bitjét készítjük el az i-edik attribútumból, $i = 1, 2, \ldots, n$ értékekre, ahol $\sum_{i=1}^{n} k_i = k$. Pontosabban, a h tördelőfüggvény valójában a $(h_1, h_2, \ldots, h_n)$ tördelőfüggvények olyan listája, ahol a h_i-t az i-edik attribútum értékére alkalmazzuk, és az k_i hosszú bitsorozatot állít elő. Azt a kosarat, amelybe a tördelőérték képzésében részt vevő n attribútumra a $(v_1, v_2, \ldots, v_n)$ értéket felvevő sort kell helyezni, úgy kapjuk meg, hogy egymás után összekapcsoljuk a $h_1(v_1)h_2(v_2)\ldots h_n(v_n)$ bitsorozatokat.

5.11. példa: Ha van egy tördelőtáblánk 10 bites kosárszámokkal (1024 kosár), akkor felhasználhatunk ebből négy bitet az a, és a maradék hat bitet a b attribútumhoz. Tegyük fel, hogy van egy sorunk, amelyben az a értéke A és a b értéke B, esetleg még vannak más attribútumok is, amelyek nem vesznek részt a tördelésben. Az A értéket leképezzük – az a attribútumhoz tartozó – h_a tördelőfüggvénnyel, és így kapunk négy bitet, mondjuk: 0101. Azután a B-t képezzük le a h_b tördelőfüggvénnyel, legyen a kapott hat bit: 111000. Ehhez a sorhoz rendelt kosárszám ezért 0101111000, azaz a két bitsorozat egymás után helyezésével kapott érték.

A tördelőfüggvényt ilyen módon felosztva hasznos számunkra, ha bármely – egy vagy több – olyan attribútumértékét ismerjük, amelyik szerepel a tördelőfüggvényben. Például, ha adott az a attribútum egy A értéke, és erre $h_a(A) = 0101$, akkor tudjuk, hogy azok a sorok, amelyeknél az a értéke A, csak abban a 64 kosárban lehetnek, amelyeknek a száma 0101... alakú, ahol a ... tetszőleges hat bitet jelölhet. Hasonlóan, ha a b attribútum B értéke adott, kiszűrhetjük azt a 16 kosarat, amely az ilyen sorokat tartalmazhatja, mert a kosárszámuk a $h_b(B)$ hat hosszú bitsorozattal végződik. □

5.12. példa: Vegyük az 5.7. példa „arany ékszer" adatait, amelyeket egy particionált tördelőtáblában akarunk tárolni, amelynek nyolc kosara van (azaz hárombitesek a ko-



5.9. ábra. *Egy particionált tördelőtábla*

sárszámok). Feltesszük, mint korábban is, hogy mindössze két rekord fér egy blokkba. Egy bitet szánunk az életkor attribútumnak, és a maradék két bitet a fizetés attribútumnak.

Az életkor tördelőfüggvényének az életkort 2-vel osztva keletkező maradékot választjuk, azaz a páros életkor olyan kosárba kerül, aminek a száma $0xy$ alakú, ahol x és y tetszőleges bit lehet. A páratlan életkort tartalmazó rekord olyan kosárba kerül, amelynek száma $1xy$ alakú. A fizetés tördelőfüggvénye legyen a fizetés (ezresekben) maradéka 4-gyel osztva. Például az 57 ezer dollár fizetés maradéka 1, ha 4-gyel osztjuk, ez olyan kosárba kerül, amelynek a száma $z01$, ahol z tetszőleges bitérték.

Az 5.9. ábrán az 5.7. példa adatait látjuk, ebben a tördelőtáblában elhelyezve. Figyeljük meg, mivel leginkább a 10-zel osztható életkorokat és fizetéseket használtunk, a tördelőfüggvény nem osztja el a pontokat túl jól. A nyolc kosárból kettőben négy-négy rekord van, így ezekhez túlcsordulásblokk szükséges, miközben három másik kosár üres. □

5.2.6. A rácsos állományok és a particionált tördelés összehasonlítása

Az ebben a részben tárgyalt két adatstruktúra hatékonysága teljesen eltérő. Íme az összehasonlítás főbb pontjai.

- A particionált tördelőtáblák gyakorlatilag teljesen használhatatlanok legközelebbi szomszéd- vagy tartománylekérdezések esetén. A gond az, hogy a pontok fizikai távolságát a kosárszámok közelsége nem tükrözi. Természetesen tervezhetünk egy tördelőfüggvényt úgy, hogy egy a attribútum legkisebb értékéhez az első bitsorozatot rendelje (csupa 0), a sorrendben következő értékhez a következő bitsorozatot (00...01) és így tovább. Ha így tennénk, akkor újra feltalálnánk a rácsos állományt.
- Egy jól megválasztott tördelőfüggvény véletlenszerűen osztja el a pontokat a kosarak között, így a kosarak foglaltsága nagyjából egyenletes lesz. Azonban a rácsos állományok – különösen, ha a dimenziók száma nagy – várhatóan sok kosarat hagynak üresen vagy majdnem üresen. A sejthető oka ennek az, hogy ha sok attribútum van, valószínű, hogy bizonyos összefüggés van legalább néhányuk között. Például, ahogy az 5.2.4. részben említettük, az életkor és a fizetés közötti összefüggés azt okozhatja, hogy az 5.6. ábra legtöbb pontja az átlóhoz közel fekszik, miközben a téglalap nagy része üres. Következésképpen kevesebb kosarat használhatunk, és/vagy kevesebb túlcsordulásblokkra van szükség egy particionált tördelőtáblában, mint egy rácsos állományban.

Ebből következően, ha csak a lekérdezések részleges egyezéssel típusú lekérdezést kell támogatnunk, amelyeknél néhány attribútumértéke adott, a többi teljesen meghatározatlan, akkor a particionált tördelőfüggvény valószínűleg jobb teljesítményt nyújt, mint a rácsos állomány. Megfordítva, ha legközelebbi szomszéd- vagy tartománylekérdezéseket kell gyakran végrehajtanunk, akkor előnyben részesíthetjük a rácsos állományok használatát.

5.2.7. Feladatok

5.2.1. feladat: Az 5.10. ábrán 12 PC leírása látható. Tegyük fel, hogy csak a sebességre és a merevlemez méretére akarunk indexet tervezni.

* a) Válasszunk öt rácsvonalat (összesen a két dimenzióhoz) úgy, hogy ne kerüljön kettőnél több pont egyik kosárba sem.

Modell	*Sebesség*	*Memória*	*Merevlemez*
A	300	32	6,0
B	333	64	4,0
C	400	64	12,7
D	350	32	10,8
E	450	96	14,0
F	400	128	12,7
G	450	128	18,1
H	233	32	4,0
I	266	64	6,0
J	300	64	6,0
K	350	64	12,0
L	400	128	6,0

5.10. ábra. *Néhány PC és a tulajdonságai*

Kis kosarak kezelése

Általában úgy gondolunk a kosárra, mint ami körülbelül egyblokknyi adatot tartalmaz. Azonban vannak helyzetek, amikor annyi kosarat kellene létrehoznunk, hogy egy átlagos kosár csak töredékét tárolja annak a rekordszámnak, ami elfér egy blokkban. Például, sokdimenziós adatok esetén sok kosárra lesz szükségünk, ha sok részre készülünk osztani mindegyik dimenzió mentén. Így, e rész struktúráinál és az 5.3. rész fa alapú szerkezeteinél is, lehet, hogy azt választjuk, hogy több kosarat (vagy facsomópontot) rakunk egy blokkba. Ha ezt tesszük, néhány fontos dologra emlékeznünk kell.:

- A blokkfej információjának tartalmaznia kell, hogy melyik rekord hol van, és hogy melyik kosárhoz tartozik.
- Ha egy rekordot szúrunk be a kosárba, lehet, hogy nincs hely abban a blokkban, amely a kosarat tartalmazza. Ha így van, szét kell vágnunk a blokkot valamilyen módon. El kell döntenünk, melyik kosár melyik blokkba kerüljön, meg kell találni a kosarak rekordjait, és a megfelelő blokkba tenni, valamint beállítani a kosártáblát, hogy a megfelelő blokkra mutasson.

! b) Szét tudjuk-e választani a pontokat úgy, hogy legfeljebb kettő lehet egy kosárban, ha csak négy rácsvonalat használunk? Mutassuk meg hogyan, vagy bizonyítsuk be, hogy ez nem lehetséges.

! c) Javasoljunk egy particionált tördelőfüggvényt, amely szétosztja ezeket a pontokat négy kosárba úgy, hogy legfeljebb négy pont lehet egy kosárban.

! **5.2.2. feladat:** Tegyük fel, hogy az 5.10. ábra adatait egy háromdimenziós rácsos állományban akarjuk elhelyezni, amelyik a sebesség, a memória és a merevlemez attribútumokon alapul. Ajánljunk egy felosztást minden egyes dimenzióra, ami jól osztja el az adatokat.

5.2.3. feladat: Válasszunk egy particionált tördelőfüggvényt, amelyik egy-egy bitet használ a sebesség, a memória és a merevlemez attribútumokhoz, és jól osztja el az 5.10. ábra adatait.

5.2.4. feladat: Tegyük fel, hogy az 5.10. ábra adatait egy rácsos állományba tesszük, amelynek csak a sebesség és a memória a dimenziói. A felosztások a 310, 375 és 425 sebességeknél, és a 40 és 75 memóriaértékeknél vannak. Tegyük fel azt is, hogy csak két pont fér egy kosárba. Javasoljunk jó vágást arra az esetre, ha az alábbi pontot szúrjuk be:

* a) Sebesség = 250 és memória = 48.

b) Sebesség = 333 és memória = 48.

5.2.5. feladat: Tegyük fel, hogy az $R(x, y)$ relációt egy rácsos állományban tároljuk. Mindkét attribútum 0 és 1000 közötti értékeket vehet fel. Ennek a rácsos állománynak a felosztása történetesen egyenletes, x-re minden 20 egységnél van egy felosztás, azaz az osztópontok 20, 40, 60 és így tovább, y-ra pedig 50 egységenként, azaz 50, 100, 150 és így tovább.

a) Hány kosarat kell megvizsgálnunk, hogy megválaszoljuk a következő tartománylekérdezést?

```
SELECT *
FROM R
WHERE 310 < x AND x < 400 AND 520 < y AND y < 730;
```

*! b) Legközelebbi szomszéd-lekérdezést akarunk végrehajtani a (110, 205) pontra. Azzal a kosárral kezdjük a keresést, amelyiknek a bal alsó sarka (100, 200) és a jobb felső sarka (120, 250), és azt találjuk, hogy a legközelebbi pont ebben a kosárban, a (115, 220). Mely kosarakat kell még átnéznünk ahhoz, hogy megbizonyosodjunk róla, hogy ez a legközelebbi pont?

! **5.2.6. feladat:** Tegyük fel, hogy van egy rácsos állományunk három rácsvonallal (azaz négy sávval) minden dimenziójában. Azonban az (x, y) pontoknak történetesen van valami különleges tulajdonságuk. Mondjuk meg a nem üres kosarak lehetséges legnagyobb számát, ha:

* a) A pontok egy vonalon vannak, azaz létezik két konstans, a és b, úgy, hogy $y = ax + b$ minden egyes (x, y) pontra.

b) A pontok között másodfokú összefüggés van, azaz létezik három konstans, a, b és c, úgy hogy $y = ax^2 + bx + c$ minden egyes (x, y) pontra.

5.2.7. feladat: Tegyük fel, hogy az $R(x, y, z)$ relációt egy particionált tördelőtáblában tároljuk, aminek 1024 kosara van (azaz 10 bites a kosárcím). Az R-re vonatkozó lekérdezések mindegyike pontosan egyet ad meg az attribútumok közül, és bármelyiket a három attribútum közül egyforma valószínűséggel adhatják meg. Ha a tördelőfüggvény 5 bitet készít az x alapján, 3 bitet az y alapján és 2 bitet a z alapján, akkor mennyi a kosarak átlagos száma, amelyeket meg kell vizsgálni egy lekérdezés megválaszolásához?

!! **5.2.8. feladat:** Tegyük fel, hogy van egy tördelőtáblánk, amelynek a kosarai 0-tól $2^n - 1$-ig vannak számozva, azaz a kosárcím n bit hosszú. A táblában egy olyan relációt akarunk tárolni, amelynek az attribútumai x és y. Egy lekérdezés vagy az x-et, vagy az y-t határozza meg, de sohasem mind a kettőt egyszerre. Legyen p annak a valószínűsége, hogy az x értéke a megadott.

a) Tegyük fel, hogy a tördelőfüggvényt úgy osztjuk fel, hogy m bitet használunk x-re, és a maradék $n - m$ bitet y-ra. Mi a megvizsgálandó kosarak várható száma m, n és

p függvényében, ami szükséges ahhoz, hogy megválaszoljunk egy véletlenszerű lekérdezést?

b) Mely m értékre (mint n és p függvénye) lesz minimális a kosarak várható száma? Ne aggódjunk, hogy ez az m valószínűleg nem egész szám lesz.

*! **5.2.9. feladat:** Tegyük fel, hogy van egy $R(x, y)$ relációnk, 1 000 000 véletlenszerűen elosztott ponttal. x és y egyaránt 0 és 1000 közötti értékeket vehet fel. Az R relációnak 100 sora fér egy blokkba. Úgy döntünk, hogy egy rácsos állományt használunk, mindegyik dimenzióban egyforma lépésközzel elhelyezett rácsvonalakkal, ahol a sávok szélessége m. Olyan m-et akarunk választani, amelyre minimális a lemez I/O-műveletek száma, ami ahhoz kell, hogy beolvassuk az összes olyan kosarat, amely egy 50 egység oldalú négyzetre vonatkozó tartománylekérdezés megválaszolásához kell. Feltehető, hogy a négyzet oldalai sosem esnek egybe a rácsvonalakkal. Ha az m-et túl nagyra választjuk, akkor sok túlcsordulásblokkunk lesz minden kosárhoz, és sok pontja a kosárnak a lekérdezés tartományán kívül lesz. Ha az m-et túl kicsire választjuk, akkor túl sok kosár lesz, és valószínűleg nem lesznek tele adattal a blokkok. Mi az m legjobb értéke?

5.3. Faszerű struktúrák többdimenziós adatokhoz

Most további négy struktúrát fogunk áttekinteni, amelyek hasznosak többdimenziós adatokra vonatkozó tartomány- vagy legközelebbi szomszéd-lekérdezések esetén. E célból tárgyaljuk a:

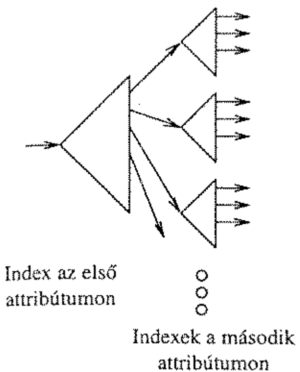
1. többkulcsos indexeket,
2. *kd*-fákat,
3. Quad-fákat,
4. R-fákat.

Az első hármat ponthalmazokhoz szánták. Az R-fa általában területek halmazának ábrázolására használatos, de pontokhoz is hasznos lehet.

5.3.1. Többkulcsos indexek

Tegyük fel, hogy van valahány attribútumunk, amik az adatpontjaink dimenzióit képviselik, és támogatni akarjuk a tartománylekérdezéseket vagy a legközelebbi szomszéd-lekérdezéseket ezeken a pontokon. Egy egyszerű faszerű szerkezet ezen pontok eléréséhez az indexek indexe, vagy általánosabban egy fa, amelyben a csomópontok, minden egyes szinten valamelyik attribútum indexei.

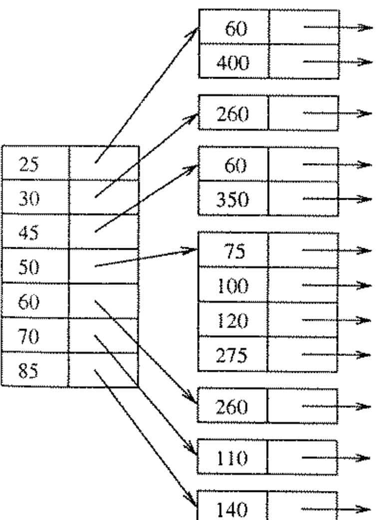
Az ötletet az 5.11. ábra mutatja két attribútum esetén. A „fa gyökere" egy index, a



5.11. ábra. *Egymásba ágyazott indexek használata különböző kulcsokon*

két attribútum közül az elsőre. Ez az index lehet bármelyik fajta szokásos index, mint a B-fa vagy a tördelőtábla. Az index hozzárendel minden egyes keresésikulcs-értékhez – azaz, az első attribútum értékeihez – egy mutatót a másik indexre. Ha V az első attribútum egy értéke, akkor az index, amit elérünk, a V értéket és annak mutatóját követve, az egy olyan pontok halmazának indexe, amelyeknél az első attribútum értéke V, a második attribútum értéke pedig tetszőleges.

5.13. példa: Az 5.12. ábra egy többkulcsos indexet mutat a megszokott „arany ékszer" példánkhoz, ahol az életkor az első attribútum, a második pedig a fizetés. A



5.12. ábra. *Többszintű indexek az életkor/fizetés adatokhoz*

gyökérindex az életkoron, az 5.12. ábra bal oldalán látható. Még nem mutattuk meg azt, hogy hogyan működik az index. Például, a kulcs-mutató párok, amelyek az index hét sorát alkotják, a B-fa levelein szétterjedhetnek. Azonban lényeges, hogy csak azok a kulcsok szerepelnek az életkorok között, amelyekhez van egy vagy több adatpont, és az index egyszerűvé teszi egy adott értékhez rendelt mutató megtalálását.

Az 5.12. ábra jobb oldalán hét index van, amelyek magukhoz a pontokhoz való hozzáférést teszik lehetővé. Például, ha követjük azt a mutatót, amely a gyökérindexben az 50 életkor értékhez van rendelve, elérünk egy kisebb rekordszámú indexhez, amelyben a fizetés a kulcs, és a négy kulcsérték az a négy fizetésérték, ami az életkor = 50 értékhez tartozó pontokban van. Megint csak nem jelöltük az ábrán, hogyan van megvalósítva az index, csak a kulcs-mutató párosítást adtuk meg. Amikor követjük a mutatókat, amelyek az egyes értékekhez (75, 100, 120 és 275) vannak rendelve, megkapjuk az ábrázolt egyedi rekordokat. Például, ha a 100 értékhez rendelt mutatót követjük, megtaláljuk azt a személyt, akinek az életkora 50, és a fizetése 100 ezer dollár. □

Többkulcsos index esetén a második vagy további szintű indexek nagyon kicsik lehetnek. Például az 5.12. ábrán található négy második szintű index, amiben csak egyetlen pár van. Ezért célszerű úgy megvalósítani ezeket, mint egyszerű táblákat, amelyekből többet berakhatunk egy blokkba, ahogyan azt az 5.2.5. részben a „Kis kosarak kezelése" című keretes rész javasolta.

5.3.2. A többkulcsos indexek hatékonysága

Nézzük meg milyen hatékony egy többkulcsos index a különféle többdimenziós lekérdezések esetén. Két attribútum esetére koncentrálunk, bár az általánosítás kettőnél több attribútum esetére magától értetődő.

Lekérdezések részleges egyezéssel

Ha az első attribútum van megadva, az elérés igen hatékony. A gyökérindexet használjuk annak az egy alindexnek a megtalálására, amely az elérni kívánt pontokhoz vezet. Például, ha a gyökér egy B-fa-index, akkor két vagy három lemezműveletet végzünk, amíg megkapjuk a megfelelő alindexet, és aztán a többi lemez I/O-művelet ahhoz szükséges, hogy elérjük teljes egészében az indexet, és magukat a pontokat az adatállományban. Másrészt azonban, ha az első attribútum értéke nem adott, akkor végig kell nézni az összes alindexet, ami időigényes eljárás lehet.

Tartománylekérdezések

A többkulcsos index nagyon jó a tartománylekérdezéshez, amennyiben az egyes indexek maguk támogatják a tartománylekérdezést a saját attribútumukon (azaz, ha B-fa-indexek). Egy tartománylekérdezés megválaszolásához használjuk a gyökérindexet és az első attribútum tartományát, így megtalálhatjuk az összes olyan alindexet, amely

tartalmazhat megfelelő pontokat. Azután ezeket az alindexeket vizsgáljuk meg a második attribútumhoz adott tartományt használva.

5.14. példa: Tegyük fel, hogy az 5.12. ábra szerinti többkulcsos indexünk van, és a tartománylekérdezésünk a $35 \leq$ életkor ≤ 55, és a $100 \leq$ fizetés ≤ 200. Megvizsgálva a gyökérindexet, a 45 és az 50 kulcsértékeket találjuk az életkori határok között. Kövessük a kapcsolódó mutatókat a két alindexhez a fizetésen. A 45 éves életkorhoz nincs fizetés a 100–200 tartományban, míg az 50 éves korhoz tartozó indexben van két ilyen fizetés: a 100 és a 120. Tehát csak két pont van az adott tartományban: az (50, 100) és az (50, 120). □

Legközelebbi szomszéd-lekérdezések

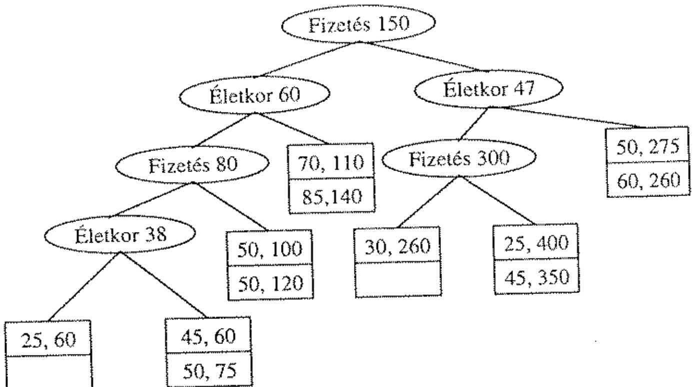
A legközelebbi szomszéd-lekérdezés megválaszolása többkulcsos index segítségével ugyanazt az eljárást követi, mint a legtöbb adatstruktúra esetén ebben a fejezetben. Az (x_0, y_0) pont legközelebbi szomszédjának megkereséséhez választunk egy d távolságot, amelyre várható, hogy találunk néhány pontot az (x_0, y_0) ponttól d távolságon belül. Azután megválaszoljuk az $x_0 - d \leq x \leq x_0 + d$ és az $y_0 - d \leq y \leq y_0 + d$ tartománylekérdezést. Ha az jön ki, hogy nincs pont ebben a tartományban, vagy ha van is pont, de a legközelebbi pont távolsága is nagyobb (x_0, y_0)-tól, mint d (azért még lehet közelebbi pont a tartományon kívül, ahogyan megtárgyaltuk az 5.1.5. részben), akkor meg kell növelnünk a tartományt, és újra kell keresnünk. Viszont tudjuk olyan sorrendben végezni a keresést, hogy a közelebbi helyeket nézzük át először.

5.3.3. *kd*-fák

Egy *kd*-fa (*k* dimenziós keresőfa) egy központi memóriabeli adatstruktúra, amely a bináris keresőfa általánosítása többdimenziós adatokra. Bemutatjuk az ötletet, majd megtárgyaljuk, hogyan lehet alkalmazni blokk módú tárolókra. A *kd*-fa egy bináris fa, amelyben minden belső csomóponthoz hozzá van rendelve egy a attribútum, és egy V érték, ami szétválasztja az adatpontokat két részre: az egyik rész, azon adatpontokból áll, amelyekre az a érték kisebb V-nél, és a másik rész, amelyben az a érték nagyobb vagy egyenlő mint V. A fa egymás alatti szintjein az attribútumok különböznek, miközben a szintek váltogatják az attribútumokat, körbe járva az összes dimenzión.

A klasszikus *kd*-fában, az adatpontok a csomópontokban vannak elhelyezve csakúgy, mint a bináris keresőfában. Azonban az alapötleten két módosítást fogunk végrehajtani annak érdekében, hogy a blokk módú tárolók bizonyos, korlátozott előnyeit kihasználhassuk.

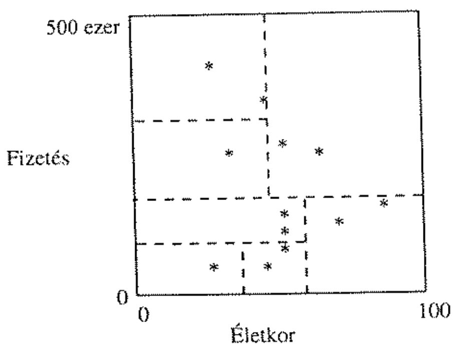
1. A belső csomópontokban csak egy attribútum lesz, egy elhatároló érték ehhez az attribútumhoz, és a mutatók a bal és a jobb gyerekre.
2. A levelek blokkok lesznek, annyi rekordnak biztosítva helyet, amennyi a blokkban elfér.



5.13. ábra. *Egy* kd-*fa*

5.15. példa: Az 5.13. ábrán egy *kd*-fa látható a már megszokott „arany ékszer" példánk tizenkét pontjával. Az egyszerűség kedvéért olyan blokkokat használunk, amelyekbe csak két rekord fér el, ezeket a blokkokat és a tartalmukat a négyzet alakú levelek mutatják. A belső csomópontok ovális alakúak egy attribútummal – vagy életkor, vagy fizetés – és egy értékkel. Például a gyökér a fizetés szerint vág ketté, a bal oldali részfában minden rekordban a fizetés kisebb, mint 150 ezeR dollár, míg a jobb oldali részfában miden rekord legalább 150 ezer dollár fizetés értékű.

A második szinten a vágás életkor szerinti. A gyökér bal oldali gyereke a 60 éves életkornál választ el, tehát minden rekordra ennek a bal oldali részfájában az életkor kisebb mint 60, és a fizetés 150 ezer dollárnál kevesebb. A jobb oldali részfájában az életkor legalább 60, és a fizetés 150 ezer dollárnál kevesebb. Az 5.14. ábra mutatja, hogy a különböző belső csomópontok hogyan osztják fel a pontok terét levélblokkokra. Például, a vízszintes vonal a fizetés = 150 értéknél a gyökérnél lévő vágást mutatja. A vonal alatti rész függőlegesen a 60 éves életkornál válik ketté, míg a felette lévő rész – a gyökér jobb oldali gyerekében lévő határoló értéknek megfelelően – a 47 éves életkornál. □



5.14. ábra. *Az 5.13. ábrán látható fának megfelelő felosztások (partíciók)*

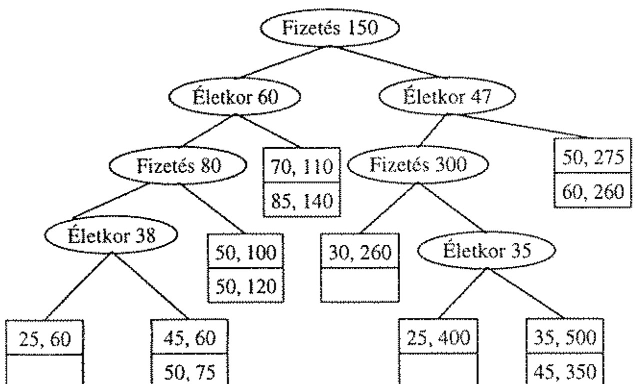
5.3.4. Műveletek a *kd*-fákon

Egy sor keresése, ha adott minden dimenzió értéke, ugyanúgy történik, mint a bináris keresőfákban. Minden belső csomópontnál eldöntjük, hogy merre kell továbbmenni, és így eljutunk egy levélhez, amelynek a blokkját keresük.

Egy beszúrás végrehajtásához úgy járunk el, mint a keresésnél. Végül egy levélhez jutunk, és ha annak a blokkjában van hely, az új adatpontot betesszük oda. Ha nincs hely, kettévágjuk a blokkot, és megosztjuk a tartalmát, aszerint, hogy azon a szinten, amin a szétvágandó levél van, melyik a megfelelő attribútum. Létrehozunk egy új belső csomópontot, amelynek a gyerekei a két új blokk, és úgy állítjuk be az elhatároló értéket a belső csomópontban, hogy megfeleljen a vágásnak, amit épp most készítettünk.[1]

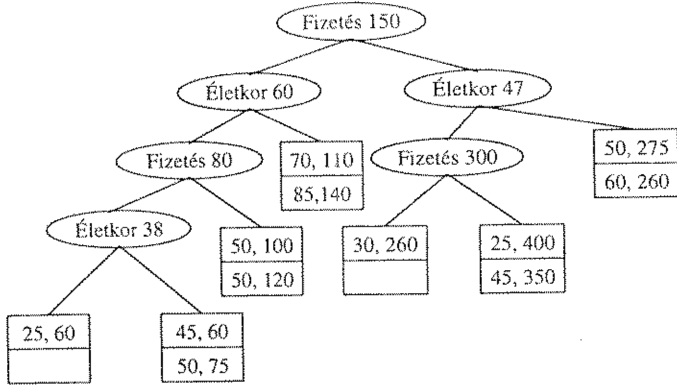
5.16. példa: Tegyük fel, hogy valaki, aki 35 éves, és 500 ezer dollár a fizetése, arany ékszert vásárol. A gyökérnél kezdünk, tudjuk, hogy a fizetés legalább 150 ezer dollár, tehát jobbra megyünk. Ott a csomópontnál összehasonlítjuk a 35 éves kort a 47-tel, ez balra irányít bennünket. A harmadik szinten ismét a fizetéseket hasonlítjuk össze, és a mi értékünk nagyobb, mint a 300 ezer dollár elválasztó érték. Így egy levélhez értünk, ami a (25, 400) és a (45, 350) pontokat tartalmazza, és ez lenne a helye az új, (35, 400) pontnak.

Nincs hely a blokkban három pont számára, tehát szét kell vágnunk. A negyedik szint az életkor szerinti vágás, tehát egy életkort kell választanunk, ami olyan egyenletesen osztja el a rekordokat, amennyire csak lehetséges. A középső érték a 35, egy jó választás, tehát a levelet helyettesítjük egy belső csomóponttal, amely az életkor = 35-



5.15. ábra. *A* kd-*fa a (35, 500) rekord beszúrása után*

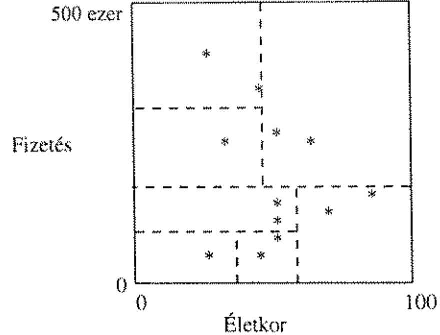
[1] Felmerülhet az a gond, hogy olyan sok pont van azonos értékkel az adott dimenzióban, hogy az adott kosárban csak egy érték van ahhoz a dimenzióhoz, és így nem tudjuk szétvágni. Ekkor megpróbálhatjuk szétvágni egy másik dimenzió mentén, vagy használhatunk túlcsordulásblokkot.



5.13. ábra. *Egy* kd-*fa*

5.15. példa: Az 5.13. ábrán egy *kd*-fa látható a már megszokott „arany ékszer" példánk tizenkét pontjával. Az egyszerűség kedvéért olyan blokkokat használunk, amelyekbe csak két rekord fér el, ezeket a blokkokat és a tartalmukat a négyzet alakú levelek mutatják. A belső csomópontok ovális alakúak egy attribútummal – vagy életkor, vagy fizetés – és egy értékkel. Például a gyökér a fizetés szerint vág ketté, a bal oldali részfában minden rekordban a fizetés kisebb, mint 150 ezeR dollár, míg a jobb oldali részfában miden rekord legalább 150 ezer dollár fizetés értékű.

A második szinten a vágás életkor szerinti. A gyökér bal oldali gyereke a 60 éves életkornál választ el, tehát minden rekordra ennek a bal oldali részfájában az életkor kisebb mint 60, és a fizetés 150 ezer dollárnál kevesebb. A jobb oldali részfájában az életkor legalább 60, és a fizetés 150 ezer dollárnál kevesebb. Az 5.14. ábra mutatja, hogy a különböző belső csomópontok hogyan osztják fel a pontok terét levélblokkokra. Például, a vízszintes vonal a fizetés = 150 értéknél a gyökérnél lévő vágást mutatja. A vonal alatti rész függőlegesen a 60 éves életkornál válik ketté, míg a felette lévő rész – a gyökér jobb oldali gyerekében lévő határoló értéknek megfelelően – a 47 éves életkornál. □



5.14. ábra. *Az 5.13. ábrán látható fának megfelelő felosztások (partíciók)*

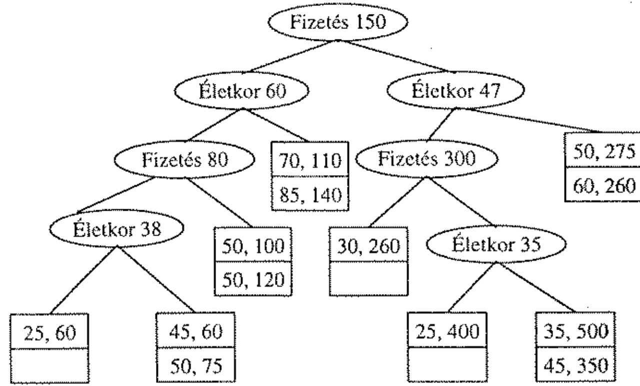
5.3.4. Műveletek a *kd*-fákon

Egy sor keresése, ha adott minden dimenzió értéke, ugyanúgy történik, mint a bináris keresőfákban. Minden belső csomópontnál eldöntjük, hogy merre kell továbbmenni, és így eljutunk egy levélhez, amelynek a blokkját keresstük.

Egy beszúrás végrehajtásához úgy járunk el, mint a keresésnél. Végül egy levélhez jutunk, és ha annak a blokkjában van hely, az új adatpontot betesszük oda. Ha nincs hely, kettévágjuk a blokkot, és megosztjuk a tartalmát, aszerint, hogy azon a szinten, amin a szétvágandó levél van, melyik a megfelelő attribútum. Létrehozunk egy új belső csomópontot, amelynek a gyerekei a két új blokk, és úgy állítjuk be az elhatároló értéket a belső csomópontban, hogy megfeleljen a vágásnak, amit épp most készítettünk.[1]

5.16. példa: Tegyük fel, hogy valaki, aki 35 éves, és 500 ezer dollár a fizetése, arany ékszert vásárol. A gyökérnél kezdünk, tudjuk, hogy a fizetés legalább 150 ezer dollár, tehát jobbra megyünk. Ott a csomópontnál összehasonlítjuk a 35 éves kort a 47-tel, ez balra irányít bennünket. A harmadik szinten ismét a fizetéseket hasonlítjuk össze, és a mi értékünk nagyobb, mint a 300 ezer dollár elválasztó érték. Így egy levélhez értünk, ami a (25, 400) és a (45, 350) pontokat tartalmazza, és ez lenne a helye az új, (35, 400) pontnak.

Nincs hely a blokkban három pont számára, tehát szét kell vágnunk. A negyedik szint az életkor szerinti vágás, tehát egy életkort kell választanunk, ami olyan egyenletesen osztja el a rekordokat, amennyire csak lehetséges. A középső érték a 35, egy jó választás, tehát a levelet helyettesítjük egy belső csomóponttal, amely az életkor = 35-



5.15. ábra. *A* kd-*fa a (35, 500) rekord beszúrása után*

[1] Felmerülhet az a gond, hogy olyan sok pont van azonos értékkel az adott dimenzióban, hogy az adott kosárban csak egy érték van ahhoz a dimenzióhoz, és így nem tudjuk szétvágni. Ekkor megpróbálhatjuk szétvágni egy másik dimenzió mentén, vagy használhatunk túlcsordulásblokkot.

nél vág. A belső csomóponttól balra egy levélblokk lesz a (25, 400) rekorddal, míg a jobbra lévő levél blokkban lesz a másik két rekord, ahogy az 5.15. ábrán látható. □

Az ebben a fejezetben tárgyalt bonyolultabb lekérdezéseket szintén támogatják a *kd*-fák. Íme az alapvető ötletek és az algoritmusok áttekintése.

Lekérdezések részleges egyezéssel

Amennyiben adottak bizonyos attribútumok értékei, akkor ha olyan szinten vagyunk, amelyhez tartozó attribútum értéke ismert, akkor mehetünk valamelyik konkrét irányba. Ha nem tudjuk az aktuális csomóponthoz tartozó attribútum értékét, akkor meg kell vizsgálni mindkét gyereket. Például, ha az 5.13. ábrán, azokat a pontokat keressük, amelyekre az életkor = 50, akkor meg kell néznünk a gyökér mindkét gyerekét, mivel a gyökér a fizetés szerint vág. Azonban, a gyökér bal oldali gyerekénél csak balra kell továbbmenni, míg a gyökér jobb oldali gyerekénél csak a jobb oldali részfát kell felderítenünk. Tegyük fel a példa kedvéért, hogy van egy tökéletesen kiegyensúlyozott, kétdimenziós fánk, amelynek sok szintje van, és a kereséshez az egyik dimenzió meg van adva. Akkor a másikhoz tartozó szinteken mindig meg kell vizsgálnunk mindkét utat, végül is el kell érnünk körülbelül annyi levelet, amennyi a levelek teljes számának a négyzetgyöke.

Tartománylekérdezések

Néha a tartomány lehetővé teszi számunkra, hogy csak a csomópont egyik gyereke felé menjünk tovább, de ha a tartomány tartalmazza a csomópont elhatároló értékét, akkor mindkét gyereket meg kell néznünk. Például, ha az életkor tartománya 35 és 55 közötti, a fizetésé 100 ezer dollártól 200 ezer dollárig, akkor az 5.13. ábra fáját a következőképpen járhatjuk be: A fizetés tartománya tartalmazza a gyökérnél lévő 150 ezer dollár értéket, így mindkét gyereket meg kell néznünk. A bal oldali gyereknél a tartomány teljesen a bal oldalra esik, tehát ahhoz a csomóponthoz megyünk, ahol a fizetés 80 ezer dollár. Most a tartomány teljesen jobbra esik, így elértük a levelet, amiben az (50, 100) és az (50, 120) rekordok vannak, ezek mindketten megfelelnek a tartománylekérdezésnek. Visszatérve a gyökér jobb oldali gyerekéhez, az elválasztó érték az életkor = 47, ezért meg kell néznünk mindkét részfát. A 300 ezer dollár fizetésértékű csomópontnál csak balra mehetünk, és így megtaláljuk a (30, 260) pontot, ami kívül esik a tartományon. Az életkor = 47 csomópont jobb oldali gyerekénél találunk további két pontot, de ezek is kívül esnek a tartományon.

Legközelebbi szomszéd-lekérdezések

Használjuk ugyanazt a megközelítést, amit az 5.3.2. részben tárgyaltunk. Kezeljük úgy a feladatot, mint egy tartománylekérdezést a megfelelő tartománnyal, és ismételjük meg egy nagyobb tartománnyal, ha szükséges.

5.3.5. A *kd*-fák alkalmazása másodlagos tárolók esetén

Tegyük fel, hogy egy n levelű *kd*-fát egy állományban tárolunk. Ebben az esetben a gyökértől a levélig tartó út átlagos hossza $\log_2 n$, mint minden bináris fánál. Ha minden egyes csomópontot egy blokkban tárolunk, amikor bejárunk egy utat csomópontonként egy lemez I/O-műveletet kell végrehajtanunk. Például, ha $n = 1000$, akkor körülbelül 10 lemez I/O-műveletre lesz szükségünk, ami sokkal több, mint egy tipikus B-fa esetén szükséges – még ennél sokkal nagyobb állományok esetén is – 2 vagy három lemez I/O-művelet. Ráadásul, mivel a *kd*-fák belső csomópontjaiban viszonylag kevés információ van, a blokk nagy része elpazarolt hely.

Nem tudjuk teljesen megoldani a hosszú út és kihasználatlan terület kettős problémáját. Viszont itt van két megközelítés, ami némi javulást fog hozni a hatékonyságban.

Többutas elágazások a belső csomópontokban

A *kd*-fák belső csomópontjai jobban hasonlítanának a B-fa-csomópontokra, ha több kulcs-mutató párjuk lenne. Ha n kulcsunk lenne egy csomópontban, akkor az a attribútum értékeit $n + 1$ tartományra oszthatnánk. Ha $n + 1$ mutató lenne, követhetnénk az egy megfelelőt ahhoz a részfához, amely csak olyan pontokat tartalmaz, amelyekre az a attribútum értéke abba a tartományba esik. Problémák akkor lépnek fel, amikor megpróbáljuk újraszervezni a csomópontokat azzal a céllal, hogy megtartsuk az eloszlást és az egyensúlyt, ahogyan a B-fáknál tettük. Például, tegyük fel van egy csomópontunk, ami az életkor szerint választ szét, és össze kell olvasztanunk a két gyere-

Semmi sem tart örökké

Az ebben a fejezetben tárgyalt adatstruktúrák lehetővé teszik, hogy beszúráskor és törléskor helyi döntéseket hozzunk, hogyan kell átszervezni a struktúrát. Sok adatbázis-módosítás után, ezen helyi döntések hatása valamiféle kiegyensúlyozatlanságot vihet a struktúrába. Például túl sok üres kosara lehet egy rácsos állománynak, vagy egy *kd*-fa erősen kiegyensúlyozatlan lehet.

Minden adatbázisnál teljesen szokásos, hogy időnként újraszervezik. Az adatbázis visszatöltésekor megvan a lehetősége annak, hogy úgy hozzuk létre az indexstruktúrákat, hogy – legalábbis abban a pillanatban – olyan kiegyensúlyozottak és hatékonyak legyenek, amennyire csak lehetséges az adott típusú indexeknél. Az ilyen újraszervezés költségét a kiegyensúlyozatlansághoz vezető sok módosítás számlájára lehet írni, így az egy módosításra eső költség kicsi. Azonban ehhez az kell, hogy „le tudjuk kapcsolni" az adatbázist, azaz elérhetetlenné tudjuk tenni a visszatöltés idejére. Az ilyen helyzet vagy okoz gondot, vagy nem az alkalmazástól függően. Például sok adatbázist leállítanak éjszakára, amikor senki sem használja őket.

két, amelyek a fizetés szerint vágnak. Nem készíthetünk egyszerűen egy csomópontot, amely tartalmazza a két gyerek fizetés tartományait, mivel ezek a tartományok általában átfedik egymást. Vegyük észre, mennyivel könnyebb lenne, ha (mint a B-fáknál) a két gyerek egyaránt tovább finomítaná az életkori felosztást.

A belső csomópontok blokkokba csoportosítása

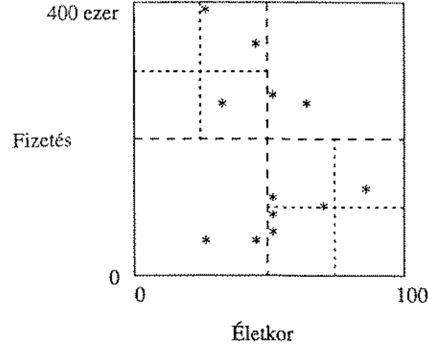
Az előzőek helyett megtarthatjuk azt az elvet, hogy a fa csomópontjainak csak két gyereke van. Több belső csomópontot tehetünk egyetlen blokkba. Abból a megfontolásból, hogy minimalizáljuk a lemezről beolvasandó blokkok számát, miközben lefelé haladunk egy útvonalon, az a legjobb, ha egy csomópont és – bizonyos szint mélységig – annak az összes leszármazottja egyetlen blokkban van. Így, ha egyszer lekérdezünk egy adott csomópontot tartalmazó blokkot, biztosak lehetünk abban, hogy fel tudunk használni néhány további csomópontot is ugyanebből a blokkból, ezzel lemez I/O-műveleteket takarítunk meg. Például tegyük fel, hogy három belső csomópontot tudunk tárolni egy blokkban. Akkor az 5.13. ábrán lévő fa esetén berakhatjuk a gyökeret és két gyerekét egy blokkba. Aztán a fizetés = 80-hoz tartozó csomópontot, és a bal oldali gyerekét betehetjük egy másik blokkba, így csak a fizetés = 300 csomópont maradt, ami egy újabb blokkba kerülhet; ezt például elhelyezhetnénk az előző két csomópont blokkjába is, viszont az osztozkodás jelentős munkát igényel, amikor a fa nő vagy csökken. Így, ha a (20, 60) pontot akarjuk megtalálni, csak két blokkot kell bejárnunk, bár négy csomóponton haladunk át.

5.3.6. Quad-fák

Egy quad-fában minden egyes belső csomópont megfelel egy négyzet alakú területnek kétdimenziós esetben, illetve egy k dimenziós kockának k dimenzió esetén. Mint a fejezet más adatstruktúrái esetén is, elsősorban a kétdimenziós esetet tekintjük át[2]. Ha egy négyzetbe eső pontok száma nem több, mint ami elfér egy blokkban, akkor úgy gondolhatunk erre a négyzetre, mint egy fa levelére, és egy blokkal ábrázoljuk, ami a pontjait tartalmazza. Ha túl sok a pont ahhoz, hogy beleférjen egy blokkba, akkor úgy kezelhetjük a négyzetet, mint egy belső csomópontot, amelynek a gyerekei felelnek meg a négyzet négy negyedének.

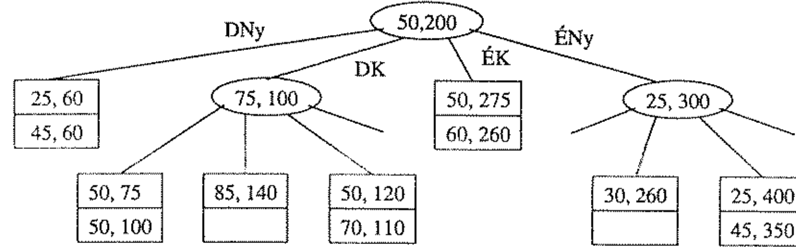
5.17. példa: Az 5.16. ábra az arany ékszer adatpontjait mutatja területekbe szervezve, amelyek megfelelnek egy quad-fa csomópontjainak. Az egyszerű számolás kedvéért, a szokásos teret leszűkítettük úgy, hogy a fizetés 0 és 400 ezer dollár közé essen, és nem 500 ezer dollárig, mint a fejezet többi példájában. Továbbra is feltesszük, hogy két rekord fér egy blokkba.

[2] Ilyenkor használatos a négyágú fa elnevezés is. *A fordító megjegyzése.*



5.16. ábra. *Quad-faként szervezett adatok*

A fát pontosan mutatja az 5.17. ábra. Az iránytű jelöléseit használjuk a negyedekhez, és a csomópontok gyerekeihez (azaz DNy a délnyugati negyedet jelöli, azaz a középponttól balra és lefelé lévő pontokat). A gyerekek sorrendje is mindig olyan, mint a gyökérnél feltüntetett. A belső csomópontok a terület középpontjának a koordinátáit jelölik.



5.17. ábra. *Egy quad-fa*

Mivel az egész tér 12 pontból áll, és csak két pont fér egy blokkba, szét kell vágnunk a teret részekre, amit a szaggatott vonal mutat az 5.16. ábrán. A kapott negyedek közül kettőnek – a délnyugatinak és az északkeletinek –, csak két pontja van. Ezek ábrázolhatók levélként, és nem kell tovább darabolni őket.

A maradék két negyednek kettőnél több pontja van. Mindkettőt tovább negyedeljük, ahogy azt az 5.16. ábrán a szaggatott vonal mutatja. Az eredményül kapott részeknek már csak kettő vagy kevesebb pontja van, így nem szükséges a további darabolás. □

Mivel k dimenzió esetén egy quad-fában egy belső csomópontnak 2^k gyereke van, található k-nak egy olyan tartománya, amelyre a csomópontok kényelmesen elhelyezhetők egy blokkban. Például, ha 128, azaz 2^7 mutató fér el egy blokkban, akkor $k = 7$ megfelelő dimenziószám. A kétdimenziós esetben azonban, a helyzet nem sokkal

jobb, mint a *kd*-fáknál; egy belső csomópontnak négy gyereke van. Továbbá, míg a *kd*-fa esetén a csomópont számára megválaszthatjuk az elhatároló pontot, addig itt kénytelenek vagyunk a quad-fa terület közepét választani, ami vagy egyenletesen osztja szét a terület pontjait, vagy nem. Várhatóan – különösen akkor, ha a dimenziószám nagy – sok üres mutató lesz a belső csomópontokban (ami az üres részeknek felel meg). Természetesen valamivel okosabban is ábrázolhatjuk a sokdimenziós csomópontokat úgy, hogy csak a nem üres mutatókat tároljuk, és egy leírót készítünk, mely megadja, hogy melyik részre vonatkozik, így jelentős helyet takaríthatunk meg.

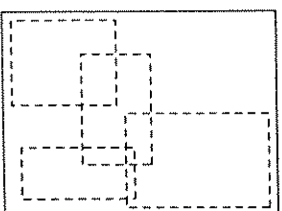
Nem megyünk bele a részletekbe az alapvető műveleteket illetően, amiket a *kd*-fák esetén az 5.3.4. részben tárgyaltunk. Az algoritmusok quad-fa esetén hasonlóak a *kd*-fákéhoz.

5.3.7. R-fák

Egy *R-fa* (régiófa) egy olyan adatstruktúra, amely a B-fa néhány alapötletét általánosítja többdimenziós adatokra. Emlékezzünk arra, hogy a B-fa-csomóponthoz egy kulcshalmaz tartozik, amely egy egyenest szakaszokra oszt. Az egyenes pontjai pontosan egy szakaszhoz tartoznak, ahogy az 5.18. ábra mutatja. Tehát a B-fa megkönnyíti a pontok elérését; ha feltételezzük, hogy a pont a B-fa-csomópont által ábrázolt egyenesen van, akkor egyértelműen meg tudjuk határozni a csomópontnak azt a gyerekét, ahol a pont megtalálható.

5.18. ábra. *A B-fa csomópontjai az egy egyenes mentén levő kulcsokat diszjunkt darabokra osztják*

Egy R-fa viszont olyan adatokat ábrázol, amelyek két- vagy többdimenziós területek lehetnek, ezeket *adatrégiónak* nevezzük. Egy R-fa belső csomópontja egy *belső régiónak* felel meg, vagy egyszerűen csak „régió"-nak, ami általában nem adatrégió. Elméletileg a régió lehet bármilyen alakú, de a gyakorlatban általában téglalap vagy valami más egyszerű alakzat. A R-fa-csomópontnak – kulcsok helyett – alrégiói vannak, amelyek gyerekeinek a tartalmát ábrázolják. Az 5.19. ábra bemutat egy R-fa-csomópontot, amely a nagy vastag vonalas téglalaphoz van rendelve. A pontozott vonalas téglalapok ábrázolják az alrégiókat, amelyek a négy gyerekéhez vannak rendelve. Megjegyezzük, hogy az alrégiók nem fedik le az egész régiót, ami elfogadható, amíg a nagy régión fekvő adatrégiók teljesen benne vannak valamelyik kis területben. Továbbá, a kis régiók átfedhetik egymást, bár kívánatos az átfedéseket minimalizálni.



5.19. ábra. *Egy R-fa-csomópont régió és a gyerekeinek az alrégiói*

5.3.8. Műveletek az R-fákon

Egy tipikus lekérdezés, amihez egy R-fa jól használható, az a „hol-vagyok-én" lekérdezés, amely megad egy P pontot, és azt az adatrégiót vagy régiókat kérdezi, amelyikben a pont benne van. A gyökértől indulunk, amihez az egész régió hozzá van rendelve. Megvizsgáljuk a gyökérben található alrégiókat, és meghatározzuk azokat a gyerekeit, amelyek olyan belső régióknak felelnek meg, amik tartalmazzák a P pontot. Ilyen régió lehet: 0, 1 vagy több.

Ha nincs ilyen régió, akkor készen vagyunk, a P nincs benne egyik adatrégióban sem. Ha van legalább egy belső régió, amely tartalmazza a P pontot, akkor minden ilyen régiónak megfelelő gyereknél rekurzívan tovább kell keresnünk. Így eljutva egy vagy több levélhez, találjuk meg a tényleges adatrégiókat, és abban vagy a teljes adatrekordot, vagy csak egy mutatót a keresett adatrekordra.

Amikor egy új R régiót szúrunk be egy R-fába, a gyökértől elindulva próbálunk meg egy olyan alrégiót találni, amibe az R beleillik. Ha egynél több ilyen régió van, kiválasztunk egyet, elmegyünk a neki megfelelő gyerekhez, és ott megismételjük az eljárást. Ha nincs ilyen R-et tartalmazó alrégió, akkor meg kell növelnünk az alrégiók valamelyikét, ennek a kiválasztása azonban általában nem egyszerű. Intuitíven: a régiókat csak a feltétlen szükséges mértékben akarjuk növelni, vagyis a gyerek alrégiói közül azt kell választani, amelyik a legkevésbé fog növekedni; ennek a régiónak változtassuk meg a határait úgy, hogy R-et tartalmazza, majd rekurzívan szúrjuk be R-et a megfelelő gyereknél.

Végül eljutunk egy levélhez, ahová be kell szúrnunk az R régiót. Ha nincs hely a levélben R számára, akkor a levelet szét kell vágnunk. Most is több lehetőség közül választhatunk. Általában azt akarjuk, hogy a két részterület a lehető legkisebb legyen, de még – többek közt – tartalmazzák az eredeti levél összes adatrégióját is. Amikor a levelet szétvágjuk, a felette lévő csomópontban az eredeti levél régió-mutató párját kicseréljük két régió-mutató értékkel, amelyek a két új levélnek felelnek meg. Ha van hely a szülőben, akkor készen vagyunk. Különben – mint a B-fák esetén – rekurzívan szétvágjuk a csomópontokat a fán felfelé haladva.

5.18. példa: Vizsgáljuk meg azt az esetet, amikor az 5.1. ábrán látható térképhez egy új régiót akarunk hozzáadni. Tegyük fel, hogy a levelekben hat régiónak van hely. További feltevés, hogy az 5.1. ábra mind a hat régiója egyetlen levélben van, ennek a régióját a külső (folyamatos vonallal rajzolt) téglalap ábrázolja az 5.20. ábrán.

Tegyük fel, hogy a helyi rádiótelefon-társaság egy antennát akar telepíteni az 5.20. ábrán látható helyre. Mivel a hét adatrégió már nem fér el egy levélben, szétvágjuk a levelet, négy régió lesz az egyikben és három a másikban. Több lehetőségünk is van, ezek közül azt a felosztást választottuk, amit a 5.20. ábra mutat (a belső, szaggatott