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

Translation copyright © 2001 by Panem Könyvkiadó Kft.

ISBN 963 545 280 2

A kiadásért felel a Panem Könyvkiadó Kft. ügyvezetője, 2001

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

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

Tartalomjegyzék

Előszó a magyar kiadáshoz

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

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

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

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

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

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

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

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

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

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

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

Dr. Benczúr András

Előszó

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

A könyv használata

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

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

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

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

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

Előismeretek

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

Feladatok

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

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

Támogatás a World Wide Weben

A könyv honlapja:

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

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

Köszönetnyilvánítások

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

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

1. fejezet

Bevezetés az adatbázis-kezelő rendszerek implementálásába

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

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

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

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

A legfontosabb szakkifejezések áttekintése

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

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

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

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

1.2. Egy adatbázis-kezelő rendszer áttekintése

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

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

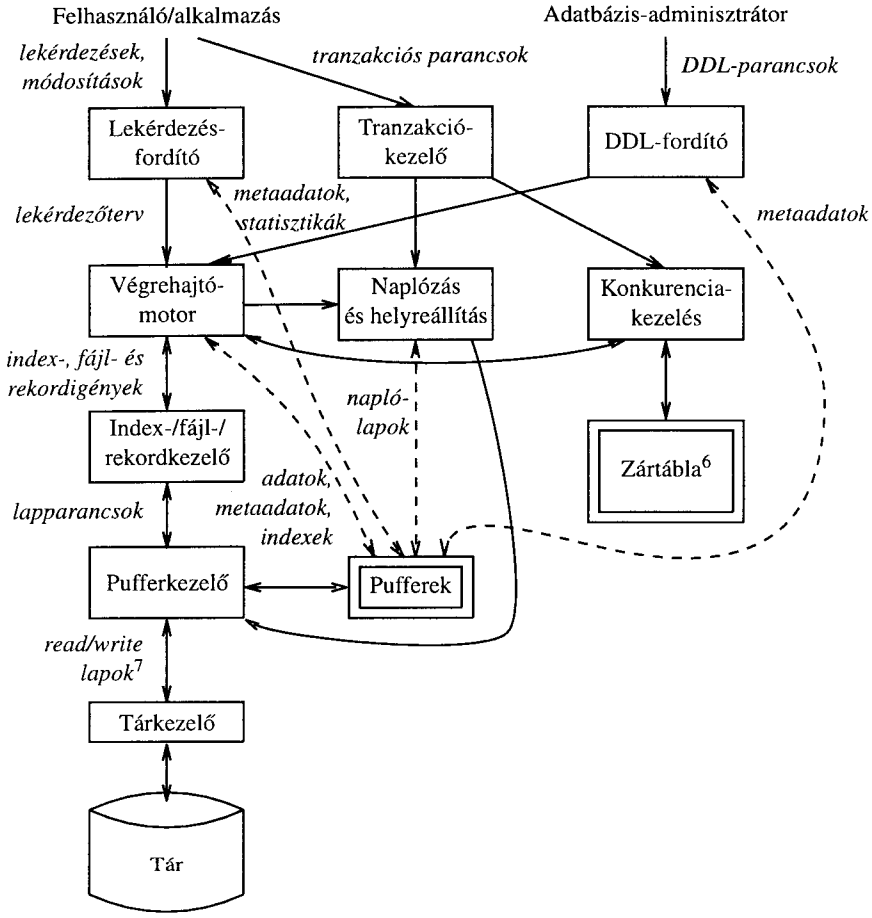
1.2.1. Az adatdefiníciós nyelv parancsai

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

1.2.2. A lekérdezés feldolgozásának áttekintése

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

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



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

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

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

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

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

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

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

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

   Ezeket az alkotórészeket az 1.2.4. részben fogjuk tovább vizsgálni.

1.2.3. A központi memória pufferei és a pufferkezelő

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

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

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

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

1.2.4. A tranzakció feldolgozása

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

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

A tranzakciók ACID-tulajdonságai

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

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

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

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

1.2.5. A lekérdezésfeldolgozó

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

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

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

   A lekérdezésfordító a metaadatok és az adatokra vonatkozó statisztikák alapján tudja nagy valószínűséggel eldönteni, hogy melyik műveletsorozat lesz a leggyorsabb. Például egy index létezése az egyik tervet sokkal gyorsabbá teheti egy másik tervhez képest.
2. A *végrehajtómotor* felelős a választott lekérdezőterv minden egyes lépésének a végrehajtásáért. A lekérdezőmotor az adatbázis-kezelő rendszer egyéb komponensének legtöbbjével is kapcsolatba lép. Ez történhet közvetlenül vagy a puffereken keresztül. Ahhoz, hogy kezelhessük az adatokat, előbb az adatbázisból be kell hozni őket a pufferekbe. Ehhez kapcsolatba kell lépni az ütemezővel is, nehogy zár alá helyezett adathoz akarjunk hozzáférni. Ezenkívül a naplókezelővel is kapcsolatba kell lépni, hogy az adatbázis-változások biztosan megfelelő módon legyenek naplózva.

1.3. A könyv vázlatos felépítése

Az adatbázisrendszerek implementálásának témaköre nagy vonalakban három részre osztható fel:

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

A fenti témák mindegyikét a könyv több fejezeten keresztül tárgyalja.

1.3.1. Előismeretek

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

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

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

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

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

1.3.2. A tárkezelés áttekintése

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

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

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

1.3.3. A lekérdezésfeldolgozás áttekintése

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

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

1.3.4. A tranzakciófeldolgozó áttekintése

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

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

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

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

1.3.5. Az információintegráció áttekintése

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

1.4. Az adatmodellek és nyelvek áttekintése

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

1.4.1. A relációs modell áttekintése

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

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

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

Ennek a relációnak a sémája az alábbi:

```
Film(filmcím, év, hossz)
```

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

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

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

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

1.4.2. Az SQL áttekintése

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

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

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

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

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

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

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

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

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

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


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

1.3. ábra. *A Paramount színészeinek megkeresése*

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

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

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

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

Alkérdések

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

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

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

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

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

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

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

Nézettáblák

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

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

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

1.6. ábra. *A Paramount filmjeit meghatározó nézettábla*

1.4.3. A relációs és objektumorientált adatok

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

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

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

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

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

Hasznos lehet, ha összegezzük a következő hasonlóságokat:

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

1.5. Összefoglalás

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

1.6. Irodalomjegyzék

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

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

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

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

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

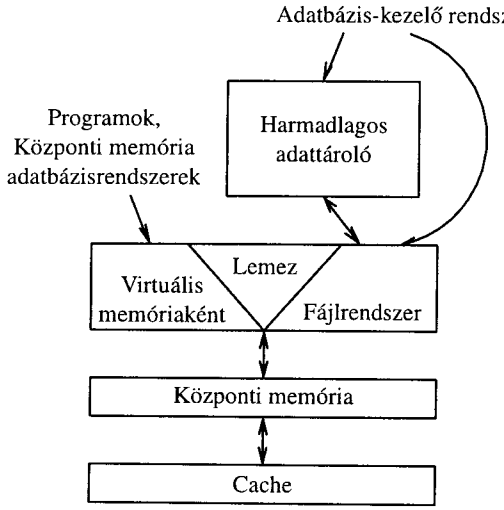
Az első kérdést ez a fejezet fedi le, míg a második a következő három fejezet témája lesz.

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 roboteszközök, melyek nagyszámú optikai lemezt vagy mágneses szalagot tároló kazetta (tape cartridge) tárolására és elérésére szolgálnak. A kétfázisú, többutas, összefésüléses rendezés nevű speciális algoritmust használjuk arra, hogy példát mutassunk olyan algoritmusra, amely hatékonyan használja a memóriahierarchiát.

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

2.1. A memóriahierarchia

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



2.1. ábra. *A memóriahierarchia*

2.1.1. Cache[1]

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

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

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

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

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

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

2.1.2. A központi memória

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

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

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

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

A számítógépes mennyiségek 2 hatványai

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

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

2.1.3. Virtuális memória

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

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

Moore törvénye

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

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

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

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

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

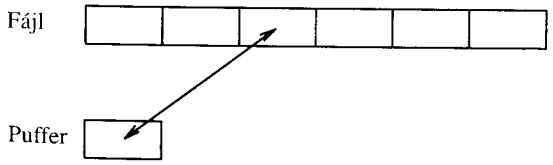
2.1.4. Másodlagos tárolás

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

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

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

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



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

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

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

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

2.1.5. Harmadlagos tárolás

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

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

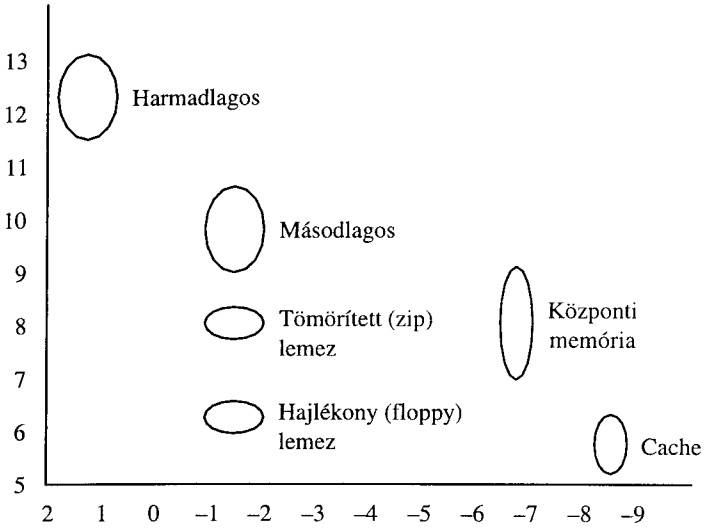
1. *Ad hoc szalagos tárolás.* A legegyszerűbb – és az elmúlt években sokáig az egyetlen – megvalósítása a harmadlagos tárolónak az, hogy az adatokat orsós vagy kazettás szalagokra mentjük ki, és ezeket utána tárolókba helyezzük. Amikor valamilyen információra van szükség a harmadlagos tárolóról, akkor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

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

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

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

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



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

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

2.1.6. Felejtő és nem felejtő tárolás

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

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

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

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

2.1.7. Feladatok

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

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

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

2.2. Lemezek

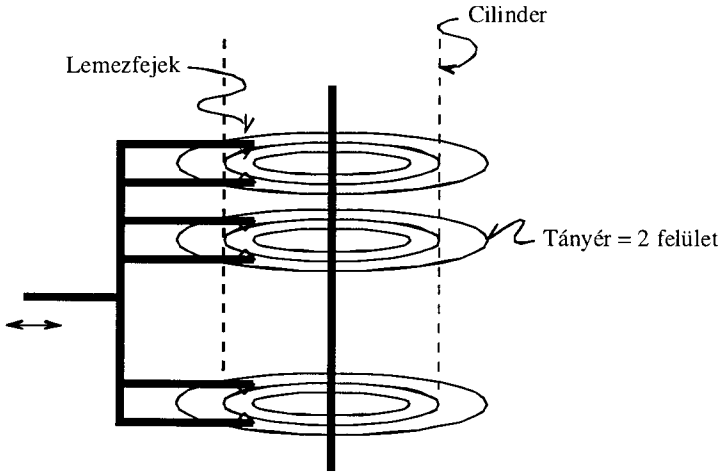
Az adatbázis-kezelő rendszerek egyik fontos jellemzője a másodlagos tárolók használata. A másodlagos tárolás szinte kizárólag mágneses lemezeken alapul. Így tehát ahhoz, hogy az adatbázis-kezelő rendszerek implementálásában használt ötleteket megindokolhassuk, először meg kell vizsgálnunk részletesen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

2.2.1. A lemezek mechanikája

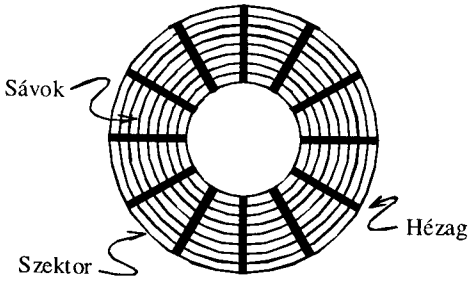
A 2.4. ábrán egy lemezmeghajtó két alapvető fontosságú mozgó részét láthatjuk, ezek a *lemezgyűjtemény* és a *fejszerelvény*. A lemezgyűjtemény egy vagy több kör alakú *tányérból* áll, melyek egy központi tengely körül forognak. A tányérok felső és alsó felülete is vékonyan be van vonva egy mágneses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

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

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



2.4. ábra. *Egy tipikus lemez*



2.5. ábra. *Egy lemez felszínének felülnézete*

Szektorok kontra blokkok

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

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

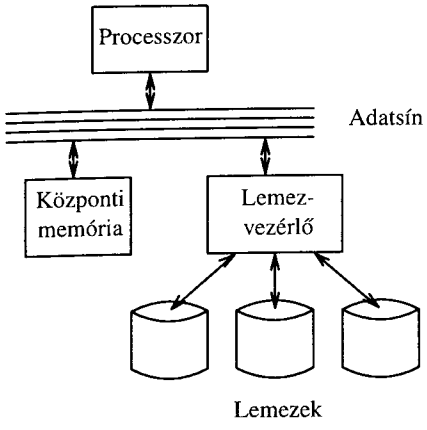
A 2.4. ábrán látható második mozgatható darab a fejszerelvény, amely a *lemezfejeket* tartja, áll. Mindegyik felülethez egy fej tartozik, mely igen-igen közel kerül a felülethez, de sohasem érintkezik vele (mert ha belezuhan a fej, akkor a lemez tönkremegy, és minden elvész, amit rajta tároltunk). A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

2.2.2. A lemezvezérlő

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

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

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



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

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

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

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

2.2.3. A lemeztárolók jellemzői

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

- *A lemezgyűjtemény forgási sebessége.* Általános az 5400 fordulat/perc (rpm), azaz egy körülfordulás 11 milliszekundum alatt történik, de találhatók ennél magasabb és alacsonyabb értékek is.
- *Az egységhez tartozó tányérok száma.* A tipikus meghajtónak körülbelül 5 tányérja és ennélfogva 10 felülete van, de 30 felülettel rendelkező lemezmeghajtókat is lehet találni. A szokásos hajlékony- és tömörített lemezeknek egyetlen tányérjuk van két felülettel. Az egyoldalas hajlékonylemez már elavult, de azért még ilyet is lehet találni. Ennek egy tányérja van, de csak egyetlen felületét lehet használni.
- *A felületen található sávok száma.* Egy felületen akár 10 000 sáv is lehet, bár a hajlékonylemezeknél ez a szám sokkal kisebb; lásd a 2.2. példát.
- *A sávokra jutó bájtok száma.* A szokásos lemezmeghajtóban 10^5 vagy több bájt jut egy sávra, de a hajlékonylemezek sávjai természetesen kevesebb bájtot tartalmaznak. Már említettük, hogy a sávokat szektorokra osztjuk. A 2.5. ábrán 12 szektor szerepel minden sávban, de a modern lemezek esetében 500 szektor is juthat egy sávba. A szektorok egyenként körülbelül 512 és 4096 közé eső számú bájtot tartalmazhatnak.

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

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

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

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

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

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

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

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

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

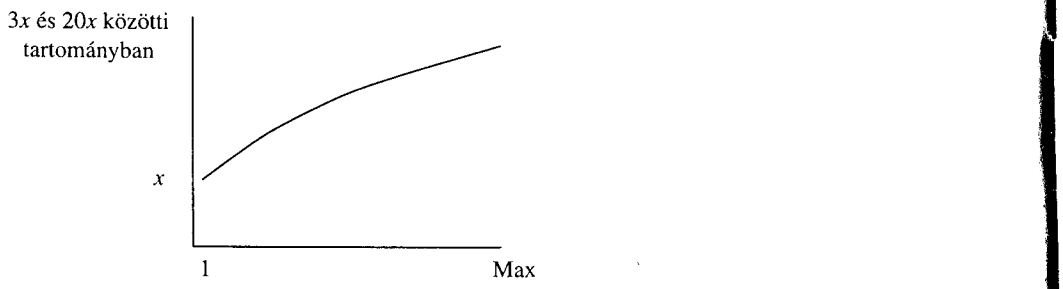
2.2.4. A lemezhozzáférés jellemzői

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

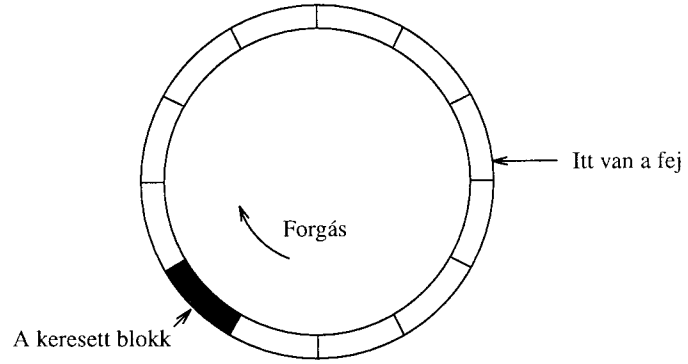
a) a fejek arra a cilinderre állnak, amelyik tartalmazza azt a sávot, melyen a blokk elhelyezkedik, és
b) a blokkot tartalmazó szektorok a lemezfej alá kerülnek a teljes lemezgyűjtemény forgása által.

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

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



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



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:

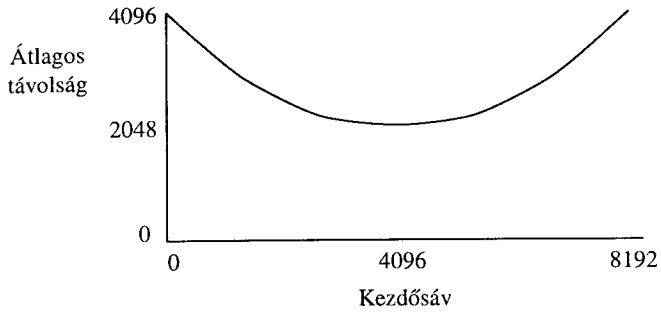
$$36 \times \frac{7}{256} + 324 \times \frac{8}{256} = 11{,}109.$$

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 áltagosan a sávok negyedét, azaz 2048 sávot fog megtenni. Egy kis számolással belátható, hogyha a kezdő pozíció az első cilindertől a 4096. cilinderig változik, akkor a fej által szükségszerűen megtett átlagos távolság négyzetesen csökken 4096-tól 2048-ig. Hasonlóan látható, hogy ha a kezdő pozíció 4096-tól 8192-ig változik, akkor a megtett átlagos távolság 4096-ig fog négyzetesen növekedni. Mindezt a 2.9. ábra szemlélteti.

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

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

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

2.2.5. Blokkok írása

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

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

2.2.6. Blokkok módosítása

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

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

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

---

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

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

2.2.7. Feladatok

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

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

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

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

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

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

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

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

2.3. A másodlagos tárolók hatékony használata

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

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

2.3.1. A számítás I/O-modellje

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

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

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

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

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

2.3.2. Adatok rendezése a másodlagos tárolóban

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

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

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

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

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

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

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

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

2.3.3. Az összefésülő rendezés (Merge-Sort)

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

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

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

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

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

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

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

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

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

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

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

2.3.4. Kétfázisú, többutas, összefésülő rendezés

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

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

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

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

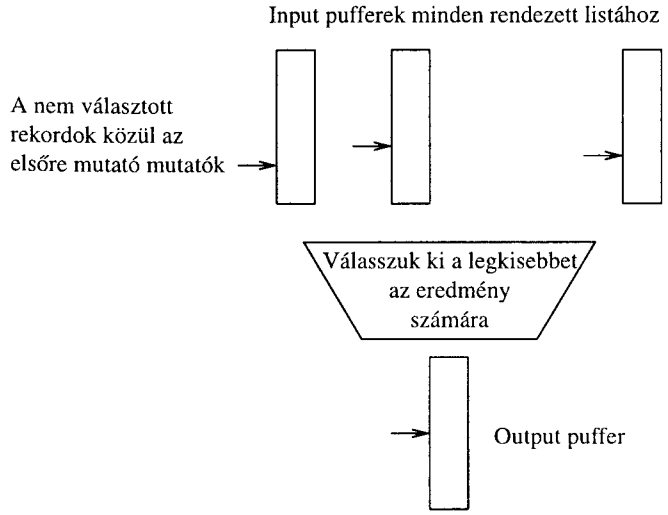
Az ily módon megadott *első fázis* végén az eredeti reláció összes rekordját egyszer olvastuk be a központi memóriába, minden rekord egy központi memória méretű rendezett részlistának lesz része, és ezeket a rendezett részlistákat kiírtuk a lemezre.

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

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

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

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



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

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

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

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

2.3.5. A többutas összefésülés kiterjesztése nagyobb relációkra

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

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

Milyen nagynak kell egy blokknak lennie?

A *Megatron 747* lemezt használó algoritmusaink elemzése során azt tételeztük fel, hogy 4 Kbájt méretű minden blokk. Ezzel szemben indokolható az is, hogy ennél nagyobb blokkméret előnyösebb lenne. Emlékezzünk vissza a 2.3. példára, ahol kiszámoltuk, hogy körülbelül fél milliszekundum a 4 K méretű blokk átviteli ideje, és körülbelül 14 milliszekundum az átlagos keresési idő és rotációs késés együttesen. Ha megduplázánk 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 megduplázánk 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, méghozzá úgy, hogy minden sávon a blokkok a rendezett lista egymás utáni blokkjai.
2. Ha egy adott listáról bizonyos rekordokra van szükségünk, akkor az egész sávot vagy cilindert beolvassuk.

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

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

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

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

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

2.4.6. A stratégiák előnyeinek és hátrányainak összegzése

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

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

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

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

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

Cilinder alapú szervezés

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

Több lemez használata

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

Tükrözés

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

Lift algoritmus

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

Korai beolvasás/Dupla pufferezés

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

2.4.7. Feladatok

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

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

2.15. ábra. *Négy blokkelérési igény beérkezési ideje*

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

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

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

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

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

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

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

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

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

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

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

2.5. Lemezhibák

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

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

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

2.5.1. Ideiglenes meghibásodás

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

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

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

2.5.2. Ellenőrző összegek

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

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

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

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

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

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

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

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

2.5.3. Stabil tárolás

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

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

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

A stabil tárolás olvasási elve a következő:

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

2.5.4. A stabil tárolás hibakezelő képessége

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

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

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

2.5.5. Feladatok

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

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

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

2.6. Lemezhiba helyreállítása

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

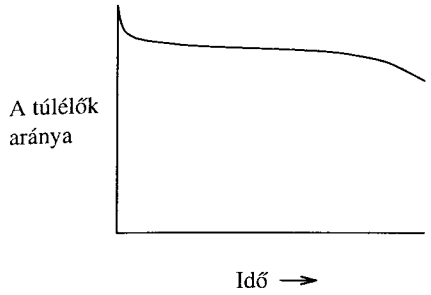
Sok különböző sémát fejlesztettek ki, hogy csökkentsék a lemezhibából származó adatveszteséget. Ezek általában redundanciát okoznak, például a 2.5.2. részben tárgyalt paritás-ellenőrzés ötletének kiterjesztésével, vagy a 2.5.3. részben leírtakhoz hasonló megduplázott szektorok használatával. Ezeknek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

2.6.1. A lemezek meghibásodási modelljei

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

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

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



2.16. ábra. *Lemezekre vonatkozó meghibásodási ráta görbéje*

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

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

2.6.2. A tükrözés mint redundanciatechnika

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

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

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

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

2.6.3. Paritásblokkok

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

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

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

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

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

4. lemez: 01100010

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

Olvasás

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

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

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

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

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

1. lemez: 11110000

ami pontosan megegyezik az első lemez első blokkjával. □

Írás

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

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

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

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

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

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

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

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

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

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

A modulo-2 összegzések algebrája

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

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

Hibajavítás

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

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

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

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

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

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

2.6.4. Egy továbbfejlesztés: az 5. szintű RAID

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Olvasás

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

Írás

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

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

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

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

2.18. ábra. *A hét lemez első blokkja*

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

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

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

Hibajavítás

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

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

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

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

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

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

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

2.6.6. Feladatok

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

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

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

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

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

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

További észrevételek a 6. szintű RAID-del kapcsolatban

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

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

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

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

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

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

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

a) Mik a redundáns lemezek megfelelő blokkjai?

Hibajavító kódok és a 6. szintű RAID

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

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

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

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

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

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

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

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

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

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

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

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

2.7. Összefoglalás

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

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

- *Lemezhibák típusai:* A rendszereknek kezelni kell tudniuk a hibákat azért, hogy elkerüljék az adatvesztést. Az alapvető lemezhibatípusok a következők: ideiglenes (ha elégszer megismételjük a műveletet, akkor egy idő után az írási vagy olvasási hiba nem fog előfordulni), állandó (adatok romlottak el a lemezen, és nem lehet helyesen olvasni őket), a lemez tönkremenése (a teljes lemez olvashatatlanná vált).
- *Ellenőrző összegek:* Ha egy plusz paritás-ellenőrző bittel egészítjük ki a bitsorozatainkat úgy, hogy a bitsorozatban szereplő egyesek számát az extra bit párossá teszi, akkor az ideiglenes és állandó hibákat felismerhetjük, bár nem tudjuk őket kijavítani.
- *Stabil tárolás:* Minden adatból két másolatot készítünk, és ügyelünk arra, hogy milyen sorrendben írjuk ezeket a másolatokat. Ezáltal egyetlen lemez használható arra, hogy kivédjünk majdnem minden, egy szektorra vonatkozó állandó hibát.
- *RAID:* Többféle séma létezik arra vonatkozóan, hogy lehet egy vagy több lemez hozzáadásával elérni, hogy az adatok túlélhessenek egy lemeztönkremenést. Az 1. szintű RAID a lemezek tükrözését jelenti. A 4. szintű RAID egy plusz lemez hozzáadását jelenti, amely az összes többi lemez megfelelő bitjeinek paritás-ellenőrzéseit 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.