5. fejezet

Többdimenziós indexek

Az eddig tárgyalt indexstruktúrák *egydimenziósak* voltak; azaz mindegyik egyetlen keresési kulcsot tartalmazott, és olyan rekordokat adott vissza, amelyek megfeleltek az adott keresésikulcs-értéknek. Azt gondolhatjuk, hogy a keresési kulcs mindig egyetlen attribútum vagy mező. Azonban olyan index is lehet egydimenziós, amelynek keresési kulcsa több mező egyesítése. Ha egy egydimenziós indexet akarunk, amelynek a keresési kulcsa az $(F_1, F_2, ..., F_k)$ mezőkből áll, akkor képezhetjük a keresésikulcs-értéket az egyes mezőértékek egymás után helyezésével, elsőnek az F_1 értékét, másodiknak az F_2 értékét és így tovább. Ezeket az értékeket valami speciális jellel elválaszthatjuk, hogy a képzett érték és az F_1, F_2, ... F_k értékek listája közötti megfeleltetést egyértelművé tegyük.

5.1. példa: Ha az F_1 és F_2 mező értéke szöveg, illetőleg egész szám, és a # karakter nem fordulhat elő a szöveg mezőben, akkor az F_1 = `'abcd'` és az F_2 = 123 értékek egyesítését az `'abcd#123'` karaktersorozattal ábrázolhatjuk. □

A 4. fejezetben az egydimenziós kulcstér előnyeit többféle módon is kihasználtuk:

- A szekvenciális állományok és a B-fák indexeinél feltételeztük, hogy a kulcsok rendezett sorrendben vannak.
- A tördelőtáblák használata megköveteli, hogy a keresési kulcs teljes egészében ismert legyen bármely kereséskor. Ha egy kulcs több mezőből áll, és azokból akár csak egy is ismeretlen, már nem tudjuk a tördelőfüggvényt alkalmazni, hanem helyette végig kell keresni az összes kosarat.

Sok alkalmazás azt igényli tőlünk, hogy az adatokat kétdimenziós vagy néha magasabb dimenziós térben ábrázolhatónak tekintsük. Némelyeket ezek közül ki tud szolgálni egy szokásos adatbázis-kezelő rendszer, de vannak speciális rendszerek, amelyeket eleve többdimenziós alkalmazásokhoz terveztek. Egy fontos dolog, ami megkülönbözteti ezeket a szakosított rendszereket a többitől az, hogy olyan adatstruktúrákat használnak, amelyek támogatnak bizonyosfajta lekérdezéseket, amelyek nem általánosak az SQL-alkalmazásokban. Az 5.1. rész bemutat tipikus lekérdezéseket, amelyek olyan indexeket használnak, amiket arra terveztek, hogy többdimenziós

adatokat és többdimenziós lekérdezéseket támogassanak. Azután az 5.2. és az 5.3. részben a következő adatstruktúrákat tárgyaljuk:

1. *Rácsos állományok,* amelyek az egydimenziós tördelőtáblák egyfajta többdimenziós kiterjesztései.
2. *Particionált tördelőfüggvények,* ez egy másik módszer, amellyel többdimenziós adatokra alkalmazzuk a tördelőtáblák ötletét.
3. *Többkulcsos indexek,* ahol egy *A* attribútum indexe elvezet egy másik *B* attribútum indexeihez, az *A* minden lehetséges értékére.
4. *kd-fák,* amelyek a B-fák általánosításai ponthalmazokra.
5. *Quad-fák,* olyan fák, amelyben egy csomópont minden gyereke egy többdimenziós kockának felel meg.
6. *R-fák,* a B-fák olyan általánosítása, amely alkalmas területgyűjtemények kezelésére.

Végül, az 5.4. részben a *bittérképindexnek* nevezett struktúrát tárgyaljuk. Ezekben az indexek tömör kódok, amelyek adott mezőben adott értéket tartalmazó rekordok elérésére használhatók. Ezek a megoldások napjainkban kezdenek megjelenni a nagy kereskedelmi adatbázis-kezelő rendszerekben, és időnként kiváló választásnak bizonyulnak egydimenziós indexek kezelésére. Azonban bizonyosfajta többdimenziós lekérdezések megválaszolására is hatékony eszközök lehetnek.

5.1. Többdimenziós alkalmazások

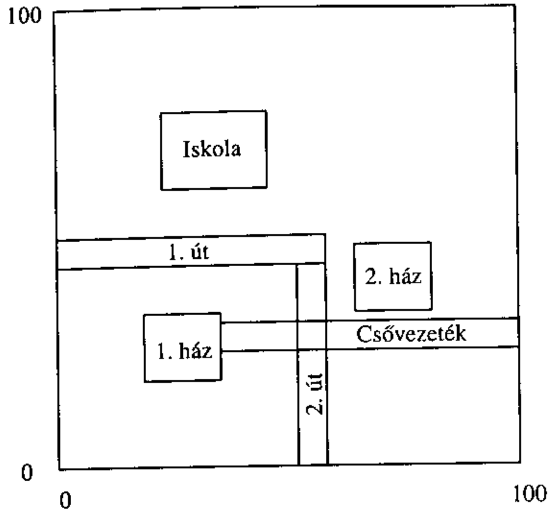
A többdimenziós alkalmazások két általános osztályát fogjuk áttekinteni. Az egyik *földrajzi* természetű, ahol az adatok a két- vagy néha a háromdimenziós világ elemei. A másik a dimenziók egy elvontabb fogalmát foglalja magában. Kissé pongyolán fogalmazva, tekinthetjük egy reláció minden attribútumát egy-egy dimenziónak, és minden sorát egy pontnak abban a térben, amelyet ezek a dimenziók meghatároznak.

Ebben a részben elemezzük azt is, hogyan használhatók a hagyományos indexek, például a B-fák, a többdimenziós lekérdezések támogatására. Bár bizonyos esetekben megfelelőek, arra is vannak példák, ahol a speciális struktúrák messze felülmúlják őket.

5.1.1. Térinformatikai rendszerek

Egy *térinformatikai rendszer* az objektumokat (tipikusan) egy kétdimenziós térben tárolja. Az objektumok lehetnek pontok vagy alakzatok. Ezek az adatbázisok gyakran térképek, ahol a tárolt objektumok házakat, utakat, hidakat, csővezetékeket és sok más fizikai tárgyat ábrázolhatnak. Egy ilyen térképre látunk mintát az 5.1. ábrán.

Azonban számos másfajta felhasználás is lehetséges. Például egy integrált áramkör terve is területek kétdimenziós térképe, ezek gyakran meghatározott anyagból készült téglalapok, úgynevezett rétegek. Hasonlóan, az ablakokat és ikonokat egy képernyőn szintén tekinthetjük kétdimenziós objektumok gyűjteményének.



5.1. ábra. *Néhány objektum egy kétdimenziós térben*

A térinformatikai rendszerek lekérdezései nem tipikus SQL-lekérdezések, de némi erőfeszítéssel sok kifejezhető SQL-ben. Ilyen típusú lekérdezésekre példák az alábbiak:

1. *Lekérdezések részleges egyezéssel.* Értékeket adunk meg egy vagy több dimenzióra, és az olyan pontokat keressük, amelyek megfelelnek a megadott értékeknek, a megadott dimenziókban.
2. *Tartománylekérdezések.* Tartományokat adunk meg egy vagy több dimenzióra, és az olyan pontok halmazát kérjük, amelyek ezeken a tartományokon belül vannak, vagy ha alakzatokat ábrázolunk, az olyan alakzatok halmazát, amelyek részben vagy teljesen a tartományon belül vannak. Ezek a lekérdezések a 4.3.4. részben tárgyalt egydimenziós tartománylekérdezések általánosításai.
3. *Legközelebbi szomszéd-lekérdezések.* Egy adott ponthoz legközelebbi pontot keressük. Például, ha egy pont egy várost jelent, és mi egy adott kisvároshoz legközelebbi 100 000-nél nagyobb lélekszámú várost akarjuk megtalálni.
4. *Hol-vagyok-én-lekérdezések.* Egy adott pontban vagyunk, és tudni akarjuk, hogy melyik alakzatban van benne ez a pont, ha van egyáltalán olyan alakzat, amiben benne van. Egy ismert példa erre, ami akkor történik, amikor az egérrel kattintunk, és a rendszer meghatározza, hogy mely látható elemre kattintottunk.

5.1.2. Adatkockák

Az utóbbi időkben megjelent az adatbázis-kezelő rendszerek azon családja, amit időnként *adatkocka*rendszereknek neveznek, amelyek az adatot többdimenziós térben létezőnek tekintik. Ezeket a 11.4. részben tárgyaljuk majd részletesebben, de a következő példa rávilágít az alapötletükre.

Többdimenziós adatokat sok vállalat gyűjt a *döntéstámogató* alkalmazások számára, amelyekkel elemzik az információkat, olyanokat mint az eladások, hogy jobban megértsék a vállalat működését. Például egy áruházlánc feljegyezheti minden eladásáról az alábbiakat:

1. a dátumot és időt,
2. az áruházat, amelyben az eladás történt,
3. a vásárolt árucikket,
4. az árucikk színét,
5. az árucikk méretét,

és esetleg az eladás egyéb tulajdonságait.

Szokásos az adatokat egy relációnak tekinteni, ahol minden tulajdonság a reláció egy attribútuma. Ezeket az attribútumokat egy többdimenziós tér, az „adatkocka" dimenzióinak tekinthetjük. Minden sor egy pont ebben a térben. Az elemzők aztán olyan kérdéseket tesznek fel, amelyek rendszerint csoportosítják az adatokat néhány dimenzió mentén, és a csoportokat összegzik valamilyen összesítő függvénnyel (aggregátorral). Egy jellemző példa lehet: „add meg a rózsaszínű ingek 1998-as eladásait áruházanként, havi bontásban".

5.1.3. Többdimenziós lekérdezések SQL-ben

Lehetséges a fent említett alkalmazások mindegyikét mint hagyományos, relációs adatbázist megvalósítani, és a felmerült lekérdezéseket SQL-ben megfogalmazni. Nézzünk néhány példát.

5.2. példa: Tegyük fel, hogy a legközelebbi szomszéd-lekérdezést akarjuk megválaszolni egy kétdimenziós térben lévő ponthalmazon. A pontokat ábrázolhatjuk valós számpárokból álló relációként.

```
Pontok(x, y)
```

Ennek két attribútuma van, x és y, amelyek a pont x és y koordinátái. A `Pontok` reláció további – itt nem mutatott – attribútumai esetleg a pontok egyéb jellemzőit ábrázolják.

Tegyük fel, hogy a (10.0, 20.0) ponthoz legközelebbi pontot keressük. Az 5.2. ábrán szereplő lekérdezés megtalálja a legközelebbi pontot, illetve pontokat, ha több ilyen is van. Minden egyes p pontra megnézi, létezik-e olyan másik q pont, amelyik közelebb van a (10.0, 20.0) ponthoz. A távolságok összehasonlítása úgy történik, hogy a (10.0, 20.0) pont és a lekérdezésben szereplő pont x, illetve y koordinátáinak különbségét négyzetre emeli és összeadja. Megjegyezzük, hogy nem kell gyököt vonni az összegből, hogy megkapjuk a tényleges távolságot, mert a távolság négyzetének összehasonlítása ugyanazt az eredményt adja, mintha a távolságértékeket magukat hasonlítanánk össze. □

```
SELECT *
FROM PONTOK p
WHERE NOT EXISTS(
    SELECT *
    FROM PONTOK q
    WHERE (q.x-10.0)*(q.x-10.0)+(q.y-20.0)*(q.y-20.0)<
          (p.x-10.0)*(p.x-10.0)+(p.y-20.0)*(p.y-20.0)
);
```

5.2. ábra. *Azon pontok keresése, amelyeknél nincs közelebbi a (10.0, 20.0) ponthoz*

5.3. példa: A téglalapok szokásos alakzatok a térinformatikai rendszerekben. Egy téglalapot többféleképpen ábrázolhatunk. Egyik népszerű forma a bal alsó sarok és a jobb felső sarok koordinátáinak megadása. Ezután a `Téglalapok` relációval ábrázolhatjuk a téglalapok gyűjteményét, ennek attribútumai a téglalap azonosítója, a négy koordináta, ami leírja a téglalapot, és esetleg bármi egyéb jellemzője a téglalapnak, amit rögzíteni szeretnénk. A következő relációt fogjuk használni ebben a példában:

```
Téglalapok(az, xba, yba, xjf, yjf)
```

Az attribútumok sorrendben a téglalap azonosítója, a bal alsó sarok x koordinátája, a bal alsó sarok y koordinátája, illetve a jobb felső sarok két koordinátája.

Az 5.3. ábrán szereplő lekérdezés azokat a téglalapokat keresi, amelyek tartalmazzák a (10.0, 20.0) pontot. A `WHERE` feltétel egyszerű. A (10.0, 20.0) pontot akkor tartalmazza a téglalap, ha a bal alsó sarkának x koordinátája 10.0 vagy attól balra van, és az y koordinátája 20.0 vagy az alatti. A jobb felső sarokra egyúttal az $x = 10.0$ vagy attól jobbra, és $y = 20.0$ vagy a feletti érték kell legyen. □

```
SELECT az
FROM Téglalapok
WHERE xba   <= 10.0  AND  yba <=    20.0    AND
      xjf   >= 10.0  AND  yjf >=    20.0;
```

5.3. ábra. *Téglalap(ok) keresése, amely(ek) tartalmaz(nak) egy adott pontot*

5.4. példa: Az adatkockarendszereknek megfelelő adatok általában *ténytáblába* – ezekben az alapadatok vannak rögzítve (pl. minden eladás) – és *dimenziótáblákba* – ezek tartalmazzák az egyes értékekhez tartozó jellemzőket az egyes dimenziókban – vannak szervezve. Például, ha az áruház, amely eladott valamit, az egy dimenzió, akkor az áruházhoz tartozó dimenziótábla megadhatja az áruház vezetőjének a címét, a telefonszámát és a nevét.

Ebben a példában csak a ténytáblával foglalkozunk, amelynek feltevésünk szerint az 5.1.2. részben javasolt dimenziói vannak. Tehát a ténytábla a következő reláció:

```
Eladások(nap, áruház, cikk, szín, méret)
```

Az „összegezd a rózsaszínű ingek eladását áruházanként, napi bontásban" lekérdezés az 5.4. ábrán látható. A lekérdezés csoportosítja az eladásokat a `nap` és az `áruház` dimenzió értékei szerint, miközben összefogja a többi dimenziót a `COUNT` összesítő függvénnyel. Az adatkocka csak azon részeire figyelünk, ami bennünket érdekel, így a `WHERE` feltétel csak a rózsaszín ingek sorait választja ki. □

```
SELECT nap, áruház, COUNT(*) AS összEladás
FROM Eladások
WHERE cikk  =  'ing' AND
      szín  =  'rózsaszín'
GROUP BY nap, áruház;
```

5.4. ábra. *A rózsaszínű ingek eladásának összegzése*

5.1.4. Tartománylekérdezések végrehajtása hagyományos indexekkel

Most tekintsük át, hogy a 4. fejezetben leírt indexek milyen mértékben segíthetik a tartománylekérdezések megválaszolását. Az egyszerűség kedvéért tegyük fel, hogy két dimenziónk van. Az x és az y dimenziók mindegyikére tehetünk egy másodlagos indexet. Mindkettőhöz B+-fát használva különösen könnyen lehet értékek egy tartományát visszanyerni bármelyik dimenzió esetén.

Ha mindkét dimenzióra adott egy tartomány, akkor kezdhetjük az x-hez tartozó B-fával, hogy visszanyerjük az összes olyan rekord mutatóját, amely az x-hez megadott tartományba esik. Azután az y B-fáját használva megkapjuk az összes olyan ponthoz tartozó rekord mutatóját, amelynek az y koordinátája az y-hoz megadott tartományba esik. Végül vesszük a mutatók metszetét, a 4.2.3. rész ötletét használva. Ha a mutatók elférnek a központi memóriában, akkor a szükséges lemez I/O-műveletek száma megegyezik a két B-fában a megvizsgálandó levélcsomópontok számával, plusz még néhány lemez I/O-művelet, amíg megtaláljuk a B-fákban lefelé az utat (lásd a 4.3.7. részt). Ehhez kell még hozzáadni a megfelelő rekordok eléréséhez szükséges lemez I/O-műveletek számát.

5.5. példa: Tekintsük 1 000 000 pontnak egy feltételezett halmazát, amely pontok véletlenszerűen oszlanak el a kétdimenziós térben, és az x, illetve y koordinátáik egyaránt 0 és 1000 közé esnek. Tegyük fel, hogy 100 pont rekordja fér el egy blokkban, és egy átlagos B-fa-levél körülbelül 200 kulcs-mutató párt tartalmaz (emlékeztetünk rá, hogy egy B-fa-blokk nem feltétlen minden bejegyzése foglalt minden időpontban). Feltételezzük továbbá, hogy van B-fa-index x-hez és y-hoz is.

Képzeljünk el egy olyan tartománylekérdezést, amely egy a tér közepén levő 100 egység oldalú négyzetbe eső pontok számát kérdezi le: $450 \le x \le 550$ és $450 \le y \le 550$. Az x-hez tartozó B-fát használva megkaphatjuk az összes rekord mutatóját, amely az x szerinti tartományba esik, ez körülbelül 100 000 mutató lehet, és ez elférhet a központi memóriában. Hasonlóan, az y-hoz tartozó B-fát használva megkaphatjuk az összes olyan rekord mutatóját is, amely y szerint a kívánt tartományba esik, ezekből ismét körülbelül 100 000 lesz. E két halmaz metszete körülbelül 10 000 mutató, és ezzel

a metszetben szereplő 10 000 mutatóval elérhető rekordok alkotják a választ a lekérdezésünkre.

Most becsüljük meg a tartománylekérdezés megválaszolásához szükséges lemez I/O-műveletek számát. Először is, ahogy a 4.3.7. részben rámutattunk, általában megvalósítható, hogy mindegyik B-fa gyökerét a központi memóriában tartsuk. Mivel keresésikulcs-értékek egy tartományát keressük a B-fákban, és a mutatók a levelekben e szerint a keresési kulcs szerint rendezettek, így mindössze annyit kell tennünk ahhoz, hogy dimenziónként hozzáférjünk a kb. 100 000 mutatóhoz, hogy átvizsgálunk egy köztes szintű csomópontot, valamint az összes levelet, amely a kívánt mutatókat tartalmazza. Mivel feltételeztük, hogy egy levél körülbelül 200 kulcs-mutató párt tartalmaz, így mindkét B-fa esetén körülbelül 500 levélblokkot kell megnéznünk. Ha ehhez hozzáadjuk B-fánként az egy köztes szintű csomópontot, akkor összesen 1002 lemez I/O-műveletet kapunk.

Végül vissza kell nyerni a blokkokat, amelyek a 10 000 kívánt rekordot tartalmazzák. Ha ezek véletlenszerűen vannak tárolva, azt várhatjuk, hogy ezek közel 10 000 különböző blokkban helyezkednek el. Mivel az egymilliós teljes állomány – a feltevés szerint 100 rekord tölt meg egy blokkot – 10 000 blokkban tárolódik, ezért nagyjából az adatállomány minden blokkját végig kell néznünk. Így, legalábbis ebben a példában, a hagyományos indexek nem, vagy csak alig segítették a tartománylekérdezés megválaszolását. Természetesen, ha a tartomány kisebb lett volna, a mutatóhalmazok metszetének létrehozása lehetővé tehette volna számunkra, hogy a keresés az adatállomány blokkjainak töredékére korlátozódjon. □

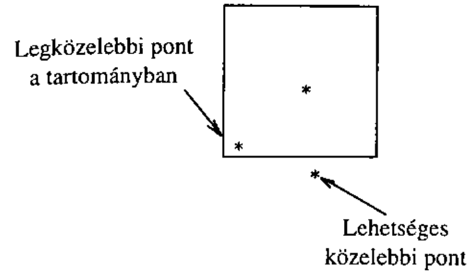
5.1.5. Legközelebbi szomszéd-lekérdezések végrehajtása hagyományos indexekkel

Majdnem minden általunk használt adatstruktúra lehetővé teszi a legközelebbi szomszéd-lekérdezések megválaszolását azáltal, hogy kiválasztunk egy tartományt minden dimenzióhoz, megválaszoljuk a tartománylekérdezést, majd kiválasztjuk a célhoz legközelebbi pontot a tartományon belül. Két probléma merülhet fel:

1. Nincs pont a kiválasztott tartományban.
2. A tartományon belüli legközelebbi pont lehet, hogy nem a legközelebbi.

Vizsgáljuk meg mindkét esetet az 5.2. példa legközelebbi szomszéd-lekérdezésével kapcsolatban, használva az 5.5. példában bemutatott x és y dimenzióra feltételezett indexeket. Ha jó okunk van feltételezni, hogy a (10.0, 20.0)-hoz d távolságon belül létezik pont, akkor használhatjuk az x-hez tartozó B-fát, hogy visszanyerjük azon pontok mutatóit, amelyek x koordinátája $10 - d$ és $10 + d$ között van. Azután használhatjuk az y-hoz tartozó B-fát, hogy visszanyerjük azon pontok mutatóit, amelyek y koordinátája $20 - d$ és $20 + d$ között van.

Ha van egy vagy több pont a metszetben, és minden mutatóhoz feljegyeztük az x, illetve az y koordinátát is (ami a keresési kulcs volt az indexhez), akkor a metszetben



5.5. ábra. *Legközelebbi pont a tartományban, ám lehet közelebbi pont a tartományon kívül*

rendelkezésünkre áll minden egyes pont koordinátája. Ezért meghatározhatjuk, hogy melyik a (10.0, 20.0)-hoz legközelebbi pont, és visszanyerhetjük annak a rekordját. Sajnos nem lehetünk biztosak benne, hogy van pont az adott ponthoz d távolságon belül, ezért lehet, hogy meg kell ismételni az eljárást egy nagyobb d értékkel.

Azonban, ha van is pont az átnézett tartományban, bizonyos esetekben az adott ponthoz – mely példánkban a (10.0, 20.0) – legközelebbi pont a tartományon belül, lehet, hogy a d távolságnál messzebb van. Egy ilyen helyzetet mutat az 5.5. ábra. Ha ez a helyzet, akkor meg kell növelnünk a tartományt, és újra kell keresnünk, hogy meggyőződjünk arról, hogy nincs közelebbi pont. Ha az eddig talált legközelebbi pont és az adott pont távolsága d', és $d' > d$, akkor meg kell ismételnünk a keresést d'-t használva d helyett.

5.6. példa: Tekintsük az 5.5. példában szereplő adatokat és indexeket. Ha a legközelebbi szomszédját keressük az adott $P = (10.0, 20.0)$ pontnak, választhatjuk a $d = 1$ értéket. Átlagosan egy pont jut a terület egy egységére, és $d = 1$ esetén megtalálunk minden olyan pontot, amely a P pont körüli 2.0 oldalú négyzetbe esik, ezeknek a várható száma 4.

Ha megvizsgáljuk az x koordinátához tartozó B-fát a $9.0 \le x \le 11.0$ tartománylekérdezés esetén, akkor a lekérdezés körülbelül 2000 pontot fog találni, így legalább 10 levelet be kell járnunk, de valószínűbb, hogy 11-et (mivel valószínűtlen, hogy az $x = 9.0$ értékű pontok éppen egy levél elején kezdődnek). Mint az 5.5 példában, várhatóan a B-fák gyökereit a központi memóriában tudjuk tartani, így csak egy lemez I/O-műveletre van szükségünk egy köztes szintű csomópont eléréséhez, és 11 lemez I/O-műveletre a levelekhez. További 12 lemez I/O-művelet kell, hogy az y koordináta B-fájában megkeressük azokat a pontokat, amelyeknek az y koordinátája 19.0 és 21.0 között van.

Ha a megközelítőleg 4000 mutató metszetét képezzük a központi memóriában, várhatóan négy rekordunk lesz, amelyek közül kikerülhet a (10.0, 20.0) pont legközelebbi szomszédja. Feltéve, hogy van legalább egy ilyen rekord, a mutatókhoz tartozó x és y koordinátákból meghatározhatjuk melyik a legközelebbi szomszéd. Még kell egy lemez I/O-művelet, hogy visszanyerjük a kívánt rekordot, így összesen 25 lemez I/O-művelet kellett, hogy befejezzük a lekérdezést. Azonban, ha $d = 1$ értékhez tartozó négyzetben nincs pont, vagy a legközelebbi pont távolsága az adott ponttól nagyobb, mint 1, meg kell ismételnünk a lekérdezést egy nagyobb d értékkel. □

Azt a következtetést vonhatjuk le az 5.6. példából, hogy a hagyományos indexek nem feltétlenül rosszak a legközelebbi szomszéd-lekérdezésekhez, de lényegesen több lemez I/O-műveletet igényelnek, mint amennyit használnánk, ha mondjuk egy rekordot adott kulcs és a kulcshoz tartozó B-fa alapján (amely valószínűleg csak két vagy három lemez I/O-művelettel járna) keresnénk meg. Az ebben a fejezetben ajánlott módszerek általában jobb teljesítményt nyújtanak, és ezeket használják a többdimenziós adatokat támogató szakosított adatbázis-kezelő rendszerek is.

5.1.6. A hagyományos indexek további korlátjai

A fent említett struktúrák költsége nem jobb tartománylekérdezésekre sem, mint a legközelebbi szomszéd-lekérdezések esetén. Gyakorlatilag az 5.6. példában úgy közelítettünk a legközelebbi szomszéd-lekérdezés megoldásához, hogy tulajdonképpen – egy kisméretű tartománnyal minden dimenzióra – tartománylekérdezéssé alakítottuk át azt, és reméltük, hogy a tartomány mérete elégséges ahhoz, hogy legalább egy pont beleessen. Következésképpen, ha egy tartománylekérdezéshez nagyobb tartományokat választanánk, és az adatstruktúrák indexek lennének minden dimenzióra, akkor a megfelelő rekordok mutatóinak eléréséhez szükséges lemez I/O-műveletek száma minden dimenzióban több lenne, mint amennyit az 5.6. példában találtunk.

Az 5.4. ábrán látható lekérdezés többdimenziós összegzése szintén nem szerencsés. Ha van indexünk a cikk és a szín attribútumra, akkor megtalálhatjuk a rózsaszínű ingek eladásához tartozó összes rekordot, a metszetüket véve, ahogyan az 5.6. példában tettük. Azonban az olyan lekérdezések esetében, amelyeknél a szín és a cikk attribútumok mellett más attribútumok is adottak, inkább az azokra az attribútumokra megadott indexekre lenne igény.

Sőt amíg az adatállományt rendezetten tudjuk tartani az öt attribútum valamelyike szerint, már nem tudjuk két attribútum szerint sorrendben tartani, nem is szólva az ötről. Így az 5.4. példában mutatott alakú lekérdezések többségénél szükség lenne az adatállomány minden vagy majdnem minden blokkjának az elérésére. Az ilyen típusú lekérdezések végrehajtása rendkívül költséges lenne, különösen ha az adatok háttértárolón vannak.

5.1.7. A többdimenziós indexstruktúrák áttekintése

A többdimenziós adatok lekérdezését támogató legtöbb adatstruktúra a következő két kategória egyikébe tartozik:

1. tördelőtábla alapú,
2. fastruktúrájú.

Mindkét fenti struktúránál feladunk valamit abból, amivel rendelkeztünk a 4. fejezet egydimenziós struktúráinál.

- A tördelőtábla alapú elgondolásoknál – rácsos állományok és particionált tördelőfüggvények az 5.2. részben – a továbbiakban nem lesz meg az az előnyünk, hogy a lekérdezésünkre adott válasz pontosan egy kosárban van. Azonban minden ilyen elgondolásnál a keresés a kosarak egy részhalmazára korlátozódik.
- A fastruktúrájú elgondolásoknál feladunk legalább egyet a következő B-fa-tulajdonságok közül:

1. A fa kiegyensúlyozottságát, ahol az összes levél ugyanazon a szinten van.
2. A facsomópontok és a lemezblokkok közötti megfeleltetést.
3. A sebességet, amellyel az adatok módosítását végre lehet hajtani.

Amint látni fogjuk az 5.3. részben, a fák gyakran mélyebbek lesznek bizonyos részeken, mint máshol, és gyakran a mélyebb részek sok pontot tartalmazó területeknek felelnek meg. Szintén látni fogjuk, hogy gyakran egy fa csomópontjához tartozó információ lényegesen kisebb méretű annál, mint ami elfér egy blokkban. Így valamilyen hasznos módon blokkokba célszerű csoportosítani a csomópontokat.

5.1.8. Feladatok

5.1.1. feladat: Írjunk SQL-lekérdezéseket az 5.3. példából vett

```
Téglalapok(az, xba, yba, xjf, yjf)
```

relációt használva, amelyek megválaszolják a következő kérdéseket:

* a) Keressük meg azon téglalapok halmazát, amelyeknek van közös része azzal a téglalappal, amelynek bal alsó sarka a (10.0, 20.0), a jobb felső sarka pedig a (40.0, 30.0) pont.
b) Keressük meg azon téglalappárokat, melyek átfedik egymást.
c) Keressük meg azokat a téglalapokat, amelyek teljesen tartalmazzák az a)-ban említett téglalapot.
d) Keressük meg azokat a téglalapokat, melyek teljesen benne vannak az a)-ban említett téglalapban.
! e) Keressük meg azokat a „téglalapokat" a `Téglalapok` relációban, amelyek valójában nem téglalapok, azaz fizikailag nem létezhetnek.

Mindegyik lekérdezésnél mondjuk meg, mely indexek – ha vannak ilyenek – segíthetnek a kívánt sorok elérésében.

5.1.2. feladat: Az 5.4. példából vett

```
Eladások(nap, áruház, cikk, szín, méret)
```

relációt használva adjuk meg a következő lekérdezéseket SQL-ben:

* a) Listázzuk ki az ingek színeit, és az eladások összesített számát az olyan színekre, amelyekből 1000-nél többet adtak el.
b) Listázzuk ki az ingek eladásait áruházanként és színenként.
c) Listázzuk ki az összes árucikk eladásait áruházanként és színenként.
! d) Listázzuk ki minden árucikkre és színre azt az áruházat, amely a legtöbbet adta el, és listázzuk ki ezeknek az eladásoknak a számát is.

5.1.3. feladat: Oldjuk meg újra az 5.5. példát azzal a feltételezéssel, hogy a tartománylekérdezés egy középen lévő n egység oldalú négyzetre vonatkozik, ahol n egy tetszőleges 1 és 1000 közötti érték. Hány lemez I/O-művelet szükséges? Milyen n értékeknél segítenek az indexek ténylegesen?

* **5.1.4. feladat:** Ismételjük meg az 5.1.3. feladatot úgy, hogy a rekordok adatállománya rendezett x-re.

!! **5.1.5. feladat:** Tegyük fel, hogy egy négyzetben véletlenszerűen elosztott pontjaink vannak (ahogy az 5.6. példában), és egy legközelebbi szomszéd-lekérdezést akarunk végrehajtani. Választunk egy d távolságot, és megkeressük az összes olyan pontot, ami abba a $2d$ oldalú négyzetbe esik, amelynek a középpontja az adott pont. A keresésünk akkor sikeres, ha találunk a négyzetben legalább egy pontot, amelynek a távolsága az adott ponttól d vagy annál kisebb.

* a) Ha egységnyi területen átlagosan egy pont van, adjuk meg annak a valószínűségét d függvényében, hogy sikeresek leszünk.
b) Ha sikertelenek vagyunk, meg kell ismételnünk a keresést egy nagyobb d-vel. Tegyük fel az egyszerűség kedvéért, hogy minden esetben, amikor sikertelenek vagyunk, akkor megduplázzuk a d-t, és kétszer annyi a költségünk, mint amennyi az előző kereséskor volt. Ismét csak feltesszük, hogy egységnyi területen átlagosan egy pont van. Melyik az a d kezdőérték, ami a minimális várható keresési költséget adja?

5.2. Tördelésen alapuló struktúrák többdimenziós adatokhoz

Ebben a részben áttekintünk két olyan adatstruktúrát, amely általánosítja az egyetlen kulcs használatára épülő tördelőtáblákat. Mindkét esetben a ponthoz rendelt kosár az összes attribútum, illetve dimenzió függvénye. Az egyik szerkezet az ún. „rácsos állomány", ez általában nem tördeli az értékeket a dimenzió mentén, hanem inkább felosztja a dimenziót, rendezve az értékeket a dimenzió mentén. A másik az ún. particionált tördelés, ami tényleg tördeli a különböző dimenziókat, és minden egyes dimenzió részt vesz a kosársorszám kialakításában.

5.2.1. Rácsos állományok

Az egyik legegyszerűbb adatstruktúra, amely gyakran felülmúlja teljesítményben az egydimenziós indexeket a többdimenziós adatokat magában foglaló lekérdezéseknél, a *rácsos állomány* (grid file). Gondoljunk egy pontokból álló térre, amelyet rácsok osztanak fel. Minden egyes dimenzióban *rácsvonalak* (grid lines) osztják fel a teret *sávokra* (stripes). Azokat a pontokat, amelyek egy rácsvonalra esnek, ahhoz a sávhoz tartozónak tekintjük, amelynek a rácsvonal az alsó határa. A különböző dimenziókhoz különböző számú rácsvonal tartozhat, és a szomszédos rácsvonalak között különböző lehet a távolság, még azonos dimenzión belül is.

5.7. példa: Bevezetjük e fejezet állandó példáját: a lekérdezésünk legyen „ki vásárol arany ékszert?". Képzeljük el az arany ékszerek vásárlóinak adatbázisát, amely számos dolgot mond nekünk minden egyes vásárlóról – a nevét, címét stb. Azonban, hogy a dolgokat egyszerűbbé tegyük, feltételezzük, hogy csak a vásárló életkora és fizetése a lényeges attribútum. A példa adatbázisunkban 12 vásárló van, akiket a következő életkor-fizetés párokkal ábrázolhatunk.

(25, 60)	(45, 60)	(50, 75)	(50, 100)
(50, 120)	(70, 110)	(85, 140)	(30, 260)
(25, 400)	(45, 350)	(50, 275)	(60, 260)

Az 5.6. ábrán látható a 12 pont elhelyezkedése egy kétdimenziós térben. Néhány rácsvonalat is választottunk mindegyik dimenzióban. Ehhez az egyszerű példához két rácsvonalat választottunk mindegyik dimenzióban, ezzel a teret 9 téglalap alakú területre osztottuk, de persze semmi nem indokolja, hogy ugyanannyi vonalat használjunk minden dimenzióban. Azt szintén megengedtük, hogy a vonalak között különböző távolságok legyenek. Például az életkor dimenziónál, a három terület – amire a két függőleges vonal osztja a teret – szélessége 40, 15 és 45.

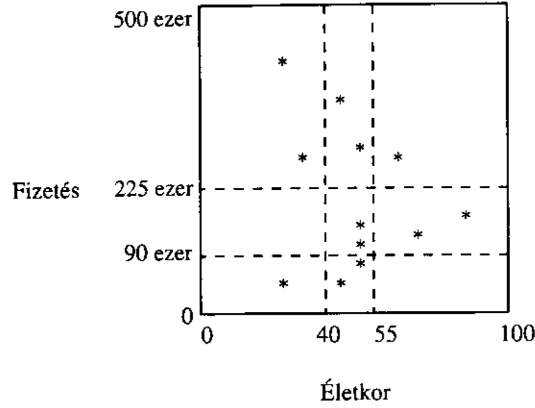
Ebben a példában nem esik pont a rácsvonalakra. De általánosságban a pont a téglalaphoz tartozik, ha az alsó vagy a bal oldali élére esik, és nem tartozik hozzá, ha a felső vagy a jobb oldali élén van. Például az 5.6. ábrán lévő középső téglalap azokat a pontokat tartalmazza, amelyekre $40 \le \textit{életkor} < 55$ és $90 \le \textit{fizetés} < 225$. □

5.2.2. Keresés rácsos állományban

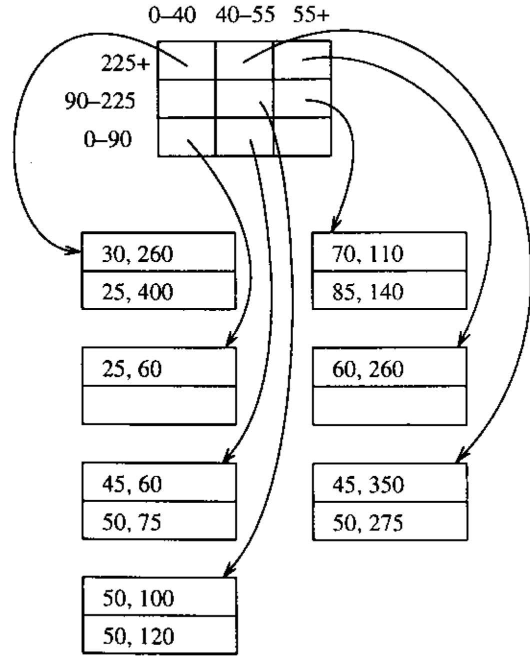
Mindegyik terület – amire a teret felosztottuk – úgy tekinthető, mint egy kosár egy tördelőtáblában, és minden ezen a területen lévő pontnak megvan a rekordja az ahhoz a kosárhoz tartozó valamely blokkban. Túlcsordulásblokkok használhatók – ha szükséges – a kosár méretének növelésére.

A kosarak egydimenziós tömbje helyett – ahogyan azt a hagyományos tördelőtábláknál láttuk – a rácsos állomány olyan tömböt használ, amelynek a dimenziószáma megegyezik az adatállományéval. Ahhoz, hogy egy ponthoz a megfelelő kosarat meg-

határozzuk, tudnunk kell minden dimenzióra az értékek listáját, ahol a rácsvonalak vannak. Egy pont hasítása így némileg különbözik attól, mint amikor egy tördelőfüggvényt alkalmazunk a változóinak értékeire. Vesszük a pont komponenseit, és meghatározzuk a pont helyzetét az ahhoz a dimenzióhoz tartozó rácson. A pont dimenziónként meghatározott helyzetei együtt határozzák meg magát a kosarat.



5.6. ábra. *Egy rácsos állomány*



5.7. ábra. *Egy rácsos állomány, amely az 5.6. ábra pontjait ábrázolja*

5.8. példa: Az 5.7. ábrán láthatjuk az 5.6. ábra adatainak elhelyezését a kosarakban. Mivel a rácsok a teret mindkét dimenziónál három területre osztják, a kosarak tömbje egy 3×3 -as mátrix. Kettő a kosarak közül üres:

1. A fizetés 90 ezer és 225 ezer dollár között, és az életkor 0 és 40 év között,

valamint

2. A fizetés 90 ezer dollár alatt, és az életkor 55 év fölött.

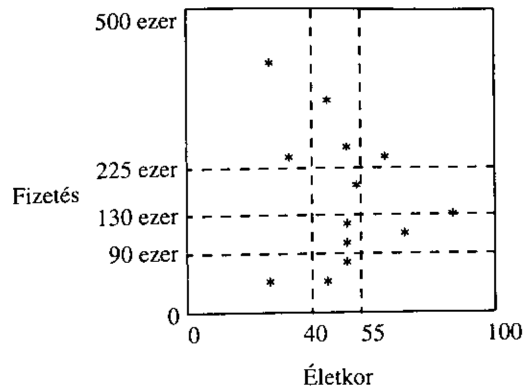
Ezért nem is jelölünk blokkot ehhez a két kosárhoz. A többi kosárhoz tartozó blokkban maximum két pont szerepelhet, ami egy mesterségesen alacsonyan tartott érték. Ebben az egyszerű példában nincs olyan kosár, amelynek kettőnél több tagja lenne, így túlcsordulásblokkokra sincs szükség. □

5.2.3. Beszúrás rácsos állományba

Amikor rekordot szúrunk be egy rácsos állományba, először a rekordkeresés eljárását követjük, és az új rekordot az így kapott kosárba tesszük. Ha van hely a kosár blokkjában, nincs is más teendőnk. Akkor van gond, ha nincs hely a kosárban. Két általános megközelítés lehetséges:

1. Adjunk túlcsordulásblokkokat a kosárhoz, ha szükséges. Ez a megoldás jól működik, egészen addig, amíg a kosár blokkjainak lánca nem válik túl naggyá. Ez utóbbi esetben a kereséshez, beszúráshoz vagy a törléshez szükséges lemez I/O-műveletek száma végül elfogadhatatlanul nagyra nőhet.
2. Szervezzük újra a struktúrát rácsvonalak hozzáadásával vagy elmozgatásával. Ez a megoldás hasonló a 4.4. részben tárgyalt dinamikus tördelési technikához, de vannak további problémái, mert a kosarak tartalma függ a dimenziótól. Azaz egy rácsvonal hozzáadása szétvágja az adott vonal mentén lévő összes kosarat. Ennek következtében elképzelhető, hogy nem lehetséges olyan rácsvonalat választani, amely mindegyik kosár számára a legjobb lenne. Például, ha egy kosár túl nagy, lehet hogy nem tudunk egy szétvágandó dimenziót vagy egy vágási pontot választani anélkül, hogy sok üres kosarat ne képeznénk vagy számos telit ne hagynánk meg.

5.9. példa: Tegyük fel, hogy valaki, aki 52 éves és 200 ezer dollár jövedelme van, arany ékszert vásárol. Ez a vásárló az 5.6. ábrán a középső téglalapba tartozik. Azonban, most már három rekord van a kosárban. Egyszerűen hozzáadhatunk egy túlcsordulásblokkot. Ha szét akarjuk vágni a kosarat, ki kell választanunk vagy az életkor, vagy a fizetés dimenziót, és választanunk kell egy új rácsvonalat, ami létrehozza a felosztást. Csak három lehetőség van, hogy egy olyan rácsvonalat vezessünk be, amely úgy vágja ketté a középső kosarat, hogy két pont kerüljön az egyik oldalra, és egy a másikra, amely esetünkben a legegyenletesebb lehetséges szétvágás.



5.8. ábra. *Az (52, 200) pont beszúrása a kosarak szétvágásával*

1. Egy függőleges vonal, mint például az életkor = 51, ami elválasztja a két 50 évest az 52-től. Ez a vonal nem vág szét semmit az alatta és felette lévő kosarakból, mivel mindkét pontja ennek a két további – 40–55 életkorhoz tartozó – kosárnak az életkor = 51 vonal bal oldalára esik.
2. Egy vízszintes vonal, ami elválasztja a középső kosárban a fizetés = 200 pontot a másik két ponttól. Választhatunk olyan számot, mondjuk a 130-at, amely szétvágja a jobbra lévő kosarat is (azaz az életkor 55–100 és fizetés 90–225 kosarat).
3. Egy vízszintes vonal, ami elválasztja a fizetés = 100 pontot a másik két ponttól. Ismét javasolhatunk olyan számot, mondjuk a 115-öt, amely szétvágja a tőle jobbra lévő kosarat is.

Az 1. lehetőség valószínűleg nem javasolt, mert nem vág szét egyetlen további kosarat sem; több üres kosarunk lesz, és egyetlen foglalt kosárnak sem csökkenti a méretét. A 2. és a 3. lehetőség egyformán jó, de mi a 2.-at választanánk, mivel a hozzáadott vízszintes fizetés = 130 rácsvonal közelebb van a 90 és 225 alsó, illetve felső határ közepéhez, mint a 3. választásakor. A kosarak eredményül kapott felosztását az 5.8. ábra mutatja. □

5.2.4. A rácsos állományok hatékonysága

Tekintsük át mennyi lemez I/O-műveletet igényel a rácsos állomány a különféle típusú lekérdezések esetén. Eddig a rácsos állomány kétdimenziós változatára koncentráltunk, bár használható tetszőleges számú dimenzió esetén. Egyik fő probléma a sokdimenziós eseteknél, hogy a kosarak száma exponenciálisan nő a dimenziók számával. Ha a tér nagy darabjai üresek, akkor sok üres kosár lesz. Megvilágíthatjuk a problémát akár két dimenzióban is. Tegyük fel, hogy szoros az összefüggés az életkor és a fizetés között, akkor az 5.6. ábra összes pontja az átló mentén fekszik, így mindegy hová helyezzük a rácsvonalakat, az átlótól távolabb lévő kosarak üresek lesznek.

Rácsos állomány kosarainak elérése

Míg egy háromszor hármas rácsban – amilyen az 5.7. ábrán látható – könnyű megtalálni egy pont megfelelő koordinátáit, ne feledjük: egy rácsos állománynak nagyon sok sávja lehet mindegyik dimenziójában. Ha így van, akkor indexet kell létrehoznunk minden egyes dimenzióhoz. Egy ilyen index keresési kulcsa az adott dimenzió felosztási értékeinek halmaza.

Adott egy v érték valamely koordinátára, és keressük azt a legnagyobb w kulcsértéket, amely kisebb vagy egyenlő mint v. Az indexben a w-hez rendelt érték a mátrix azon sora, vagy oszlopa lesz, amelyikbe v esik. Megadva az értékeket mindegyik dimenzióhoz, megtalálhatjuk azt a helyet, ahova a mátrixban a kosár mutatója esik. Ezzel a mutatóval aztán közvetlenül elérhetjük a blokkot.

Szélsőséges esetekben a mátrix olyan nagy, hogy a kosarak nagy része üres, és nem áll módunkban tárolni az üres kosarakat. Ekkor úgy kell kezelnünk a mátrixot, mint egy relációt, amelynek az attribútumai a nem üres kosarak sarkai, és egy záró attribútum ábrázolja a mutatót a kosárra. Az ilyen relációban való keresés is többdimenziós, de a mérete kisebb, mint magának az adatállománynak.

Azonban, ha az adatok eloszlása jó, és az adatállomány maga nem túl nagy, akkor ki tudjuk választani a rácsvonalakat úgy, hogy:

1. Kellően kisszámú kosár van ahhoz, hogy a kosármátrixot a központi memóriában tartsuk, így nem kell külön lemez I/O-művelet ahhoz, hogy megnézzük a mátrixot, vagy új sorokat, illetve oszlopokat adjunk hozzá, amikor új rácsvonalat veszünk fel.
2. A rácsvonalak értékeinek indexét is a memóriában tudjuk tartani mindegyik dimenzióban (lásd a „Rácsos állomány kosarainak elérése" című bekeretezett részt), vagy el tudjuk kerülni az indexeket, és a dimenziók rácsvonalait meghatározó értékeken bináris keresést tudunk alkalmazni a központi memóriában.
3. Egy tipikus kosárnak legfeljebb néhány túlcsordulásblokkja lehet, tehát ez nem okoz túl sok további lemez I/O-műveletet, amikor végignézzük a kosarat.

Ezen előfeltevések mellett, bemutatjuk a rácsos állományok viselkedését a lekérdezések néhány fontos osztályára.

Adott pont keresése

A megfelelő kosárhoz vezet bennünket, így a szükséges lemez I/O-művelet csak ennek a kosárnak a beolvasása. Beszúrás vagy törlés esetén egy további lemezírás szükséges. Ha a beszúrás túlcsordulásblokk létrehozásához vezet, az egy további írási műveletet jelent.

Lekérdezések részleges egyezéssel

Az ilyen lekérdezésre példák: „keresd meg az összes 50 éves vásárlót", vagy „keresd meg az összes vásárlót, akinek a fizetése 200 ezer dollár". Most meg kell néznünk az összes kosarat a kosármátrix egy sorában vagy oszlopában. A lemez I/O-műveletek száma igen magas lehet, ha sok kosár van abban a sorban, illetve oszlopban.

Tartománylekérdezések

Egy tartománylekérdezés a rács egy téglalap alakú területét határozza meg, és az összes pont, amit a területen belüli kosarakban találunk a lekérdezés eredményéhez tartozik, néhány olyan pont kivételével, amelyek a kijelölt terület határára eső kosarakban találhatók. Például, ha meg akarjuk találni a 35–45 éves vásárlókat, akiknek a fizetése 50 ezer–100 ezer dollár, akkor négy kosarat kell megnéznünk az 5.6. ábra bal alsó részén. Ebben az esetben minden kosár a határokon van, így jó sok pontot meg kell néznünk, amelyek esetleg nem tartoznak a kérdésre adott válaszhoz. Azonban ha olyan területet vizsgálunk, amely sok kosarat tartalmaz, akkor ezek nagy része szükségképpen belső, így minden pontjuk a válaszhoz tartozik. Tartománylekérdezéseknél a lemez I/O-műveletek száma nagy is lehet, mivel esetleg sok kosarat kényszerülünk megvizsgálni. Bár a tartománylekérdezések hajlamosak nagy eredményhalmazt produkálni, általában nem kell sokkal több blokkot megvizsgálnunk, mint a minimális blokkszám, amennyibe az eredmény egyáltalán elhelyezhető tetszőleges szervezés esetén.

Legközelebbi szomszéd-lekérdezések

Adott egy P pont, a keresést azzal a kosárral kezdjük, amelyikhez ez a pont tartozik. Ha találunk legalább egy pontot abban, akkor van egy Q jelöltünk a legközelebbi szomszédra. Azonban lehetnek a szomszédos kosarakban olyan pontok, amelyek közelebb vannak P-hez mint a Q; a helyzet hasonló, mint amit az 5.5. ábra mutat. Meg kell vizsgálnunk, vajon a P és az őt tartalmazó kosár határai közötti távolság kisebb-e, mint P és Q távolsága. Ha vannak ilyen határok, akkor minden ilyen határ túloldalán lévő szomszédos kosarat szintén át kell néznünk. Valójában, ha a kosarak olyan téglalapok, amelyek sokkal hosszabbak az egyik dimenzió mentén, mint a másikban, akkor szükséges lehet megvizsgálni még olyan kosarakat is, amelyek nem is szomszédosak a P pontot tartalmazóval.

5.10. példa: Tegyük fel, hogy az 5.6. ábrán, a $P = (45, 200)$ ponthoz legközelebbi pontot keressük. A kosárban az (50, 120) pont van hozzá a legközelebb, a távolsága 80.2. Az alsó három kosárban nem lehet egy pont sem közelebb a (45, 200)-hoz, mivel a fizetés részük legfeljebb 90, így ezeket kihagyhatjuk a keresésből. Azonban a másik öt kosarat át kell néznünk, és azt találjuk, hogy valójában két egyformán közeli pont van: (30, 260) és (60, 260), P-től 61.8 távolságra. Általában, a legközelebbi szomszéd keresését néhány kosárra, és így néhány lemez I/O-műveletre lehet korlátozni. Azonban, mivel a P ponthoz legközelebbi kosarak lehetnek üresek, nem tudunk könnyen felső korlátot adni a keresés költségére. □

5.2.5. Particionált tördelőfüggvények

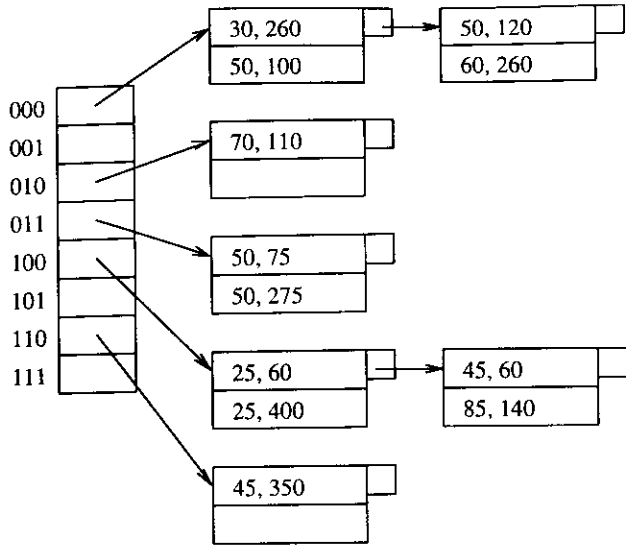
A tördelőfüggvényeknek az argumentuma lehet attribútumértékek egy listája, bár a tipikus az, hogy csak egyetlen attribútumból képzik a tördelőértékeket. Például, ha a egy egész értékű attribútum, b pedig egy szöveg értékű attribútum, akkor hozzáadhatjuk a értékéhez b minden egyes karakterének ASCII kód értékét, majd az így kapott értéket elosztjuk a kosarak számával, és vesszük ennek maradékát. Az eredményt használhatjuk egy tördelőtábla kosárszámaként, mint egy indexet az (a, b) attribútumpáron.

Azonban az ilyen tördelőtábla csak olyan lekérdezésekhez használható, amelyekben az a és a b értéke egyaránt adott. Célszerűbb választás úgy tervezni a tördelőfüggvényt, hogy állítson elő, mondjuk k számú bitet. Ezt a k bitet felosztjuk az n darab attribútum között úgy, hogy a tördelő érték k_i darab bitjét készítjük el az i-edik attribútumból, $i = 1, 2, \ldots, n$ értékekre, ahol $\sum_{i=1}^{n} k_i = k$. Pontosabban, a h tördelőfüggvény valójában a $(h_1, h_2, \ldots, h_n)$ tördelőfüggvények olyan listája, ahol a h_i-t az i-edik attribútum értékére alkalmazzuk, és az k_i hosszú bitsorozatot állít elő. Azt a kosarat, amelybe a tördelőérték képzésében részt vevő n attribútumra a $(v_1, v_2, \ldots, v_n)$ értéket felvevő sort kell helyezni, úgy kapjuk meg, hogy egymás után összekapcsoljuk a $h_1(v_1)h_2(v_2)\ldots h_n(v_n)$ bitsorozatokat.

5.11. példa: Ha van egy tördelőtáblánk 10 bites kosárszámokkal (1024 kosár), akkor felhasználhatunk ebből négy bitet az a, és a maradék hat bitet a b attribútumhoz. Tegyük fel, hogy van egy sorunk, amelyben az a értéke A és a b értéke B, esetleg még vannak más attribútumok is, amelyek nem vesznek részt a tördelésben. Az A értéket leképezzük – az a attribútumhoz tartozó – h_a tördelőfüggvénnyel, és így kapunk négy bitet, mondjuk: 0101. Azután a B-t képezzük le a h_b tördelőfüggvénnyel, legyen a kapott hat bit: 111000. Ehhez a sorhoz rendelt kosárszám ezért 0101111000, azaz a két bitsorozat egymás után helyezésével kapott érték.

A tördelőfüggvényt ilyen módon felosztva hasznos számunkra, ha bármely – egy vagy több – olyan attribútumértékét ismerjük, amelyik szerepel a tördelőfüggvényben. Például, ha adott az a attribútum egy A értéke, és erre $h_a(A) = 0101$, akkor tudjuk, hogy azok a sorok, amelyeknél az a értéke A, csak abban a 64 kosárban lehetnek, amelyeknek a száma 0101... alakú, ahol a ... tetszőleges hat bitet jelölhet. Hasonlóan, ha a b attribútum B értéke adott, kiszűrhetjük azt a 16 kosarat, amely az ilyen sorokat tartalmazhatja, mert a kosárszámuk a $h_b(B)$ hat hosszú bitsorozattal végződik. □

5.12. példa: Vegyük az 5.7. példa „arany ékszer" adatait, amelyeket egy particionált tördelőtáblában akarunk tárolni, amelynek nyolc kosara van (azaz hárombitesek a ko-



5.9. ábra. *Egy particionált tördelőtábla*

sárszámok). Feltesszük, mint korábban is, hogy mindössze két rekord fér egy blokkba. Egy bitet szánunk az életkor attribútumnak, és a maradék két bitet a fizetés attribútumnak.

Az életkor tördelőfüggvényének az életkort 2-vel osztva keletkező maradékot választjuk, azaz a páros életkor olyan kosárba kerül, aminek a száma $0xy$ alakú, ahol x és y tetszőleges bit lehet. A páratlan életkort tartalmazó rekord olyan kosárba kerül, amelynek száma $1xy$ alakú. A fizetés tördelőfüggvénye legyen a fizetés (ezresekben) maradéka 4-gyel osztva. Például az 57 ezer dollár fizetés maradéka 1, ha 4-gyel osztjuk, ez olyan kosárba kerül, amelynek a száma $z01$, ahol z tetszőleges bitérték.

Az 5.9. ábrán az 5.7. példa adatait látjuk, ebben a tördelőtáblában elhelyezve. Figyeljük meg, mivel leginkább a 10-zel osztható életkorokat és fizetéseket használtunk, a tördelőfüggvény nem osztja el a pontokat túl jól. A nyolc kosárból kettőben négy-négy rekord van, így ezekhez túlcsordulásblokk szükséges, miközben három másik kosár üres. □

5.2.6. A rácsos állományok és a particionált tördelés összehasonlítása

Az ebben a részben tárgyalt két adatstruktúra hatékonysága teljesen eltérő. Íme az összehasonlítás főbb pontjai.

- A particionált tördelőtáblák gyakorlatilag teljesen használhatatlanok legközelebbi szomszéd- vagy tartománylekérdezések esetén. A gond az, hogy a pontok fizikai távolságát a kosárszámok közelsége nem tükrözi. Természetesen tervezhetünk egy tördelőfüggvényt úgy, hogy egy a attribútum legkisebb értékéhez az első bitsorozatot rendelje (csupa 0), a sorrendben következő értékhez a következő bitsorozatot (00...01) és így tovább. Ha így tennénk, akkor újra feltalálnánk a rácsos állományt.
- Egy jól megválasztott tördelőfüggvény véletlenszerűen osztja el a pontokat a kosarak között, így a kosarak foglaltsága nagyjából egyenletes lesz. Azonban a rácsos állományok – különösen, ha a dimenziók száma nagy – várhatóan sok kosarat hagynak üresen vagy majdnem üresen. A sejthető oka ennek az, hogy ha sok attribútum van, valószínű, hogy bizonyos összefüggés van legalább néhányuk között. Például, ahogy az 5.2.4. részben említettük, az életkor és a fizetés közötti összefüggés azt okozhatja, hogy az 5.6. ábra legtöbb pontja az átlóhoz közel fekszik, miközben a téglalap nagy része üres. Következésképpen kevesebb kosarat használhatunk, és/vagy kevesebb túlcsordulásblokkra van szükség egy particionált tördelőtáblában, mint egy rácsos állományban.

Ebből következően, ha csak a lekérdezések részleges egyezéssel típusú lekérdezést kell támogatnunk, amelyeknél néhány attribútumértéke adott, a többi teljesen meghatározatlan, akkor a particionált tördelőfüggvény valószínűleg jobb teljesítményt nyújt, mint a rácsos állomány. Megfordítva, ha legközelebbi szomszéd- vagy tartománylekérdezéseket kell gyakran végrehajtanunk, akkor előnyben részesíthetjük a rácsos állományok használatát.

5.2.7. Feladatok

5.2.1. feladat: Az 5.10. ábrán 12 PC leírása látható. Tegyük fel, hogy csak a sebességre és a merevlemez méretére akarunk indexet tervezni.

* a) Válasszunk öt rácsvonalat (összesen a két dimenzióhoz) úgy, hogy ne kerüljön kettőnél több pont egyik kosárba sem.

Modell	*Sebesség*	*Memória*	*Merevlemez*
A	300	32	6,0
B	333	64	4,0
C	400	64	12,7
D	350	32	10,8
E	450	96	14,0
F	400	128	12,7
G	450	128	18,1
H	233	32	4,0
I	266	64	6,0
J	300	64	6,0
K	350	64	12,0
L	400	128	6,0

5.10. ábra. *Néhány PC és a tulajdonságai*

Kis kosarak kezelése

Általában úgy gondolunk a kosárra, mint ami körülbelül egyblokknyi adatot tartalmaz. Azonban vannak helyzetek, amikor annyi kosarat kellene létrehoznunk, hogy egy átlagos kosár csak töredékét tárolja annak a rekordszámnak, ami elfér egy blokkban. Például, sokdimenziós adatok esetén sok kosárra lesz szükségünk, ha sok részre készülünk osztani mindegyik dimenzió mentén. Így, e rész struktúráinál és az 5.3. rész fa alapú szerkezeteinél is, lehet, hogy azt választjuk, hogy több kosarat (vagy facsomópontot) rakunk egy blokkba. Ha ezt tesszük, néhány fontos dologra emlékeznünk kell.:

- A blokkfej információjának tartalmaznia kell, hogy melyik rekord hol van, és hogy melyik kosárhoz tartozik.
- Ha egy rekordot szúrunk be a kosárba, lehet, hogy nincs hely abban a blokkban, amely a kosarat tartalmazza. Ha így van, szét kell vágnunk a blokkot valamilyen módon. El kell döntenünk, melyik kosár melyik blokkba kerüljön, meg kell találni a kosarak rekordjait, és a megfelelő blokkba tenni, valamint beállítani a kosártáblát, hogy a megfelelő blokkra mutasson.

! b) Szét tudjuk-e választani a pontokat úgy, hogy legfeljebb kettő lehet egy kosárban, ha csak négy rácsvonalat használunk? Mutassuk meg hogyan, vagy bizonyítsuk be, hogy ez nem lehetséges.

! c) Javasoljunk egy particionált tördelőfüggvényt, amely szétosztja ezeket a pontokat négy kosárba úgy, hogy legfeljebb négy pont lehet egy kosárban.

! **5.2.2. feladat:** Tegyük fel, hogy az 5.10. ábra adatait egy háromdimenziós rácsos állományban akarjuk elhelyezni, amelyik a sebesség, a memória és a merevlemez attribútumokon alapul. Ajánljunk egy felosztást minden egyes dimenzióra, ami jól osztja el az adatokat.

5.2.3. feladat: Válasszunk egy particionált tördelőfüggvényt, amelyik egy-egy bitet használ a sebesség, a memória és a merevlemez attribútumokhoz, és jól osztja el az 5.10. ábra adatait.

5.2.4. feladat: Tegyük fel, hogy az 5.10. ábra adatait egy rácsos állományba tesszük, amelynek csak a sebesség és a memória a dimenziói. A felosztások a 310, 375 és 425 sebességeknél, és a 40 és 75 memóriaértékeknél vannak. Tegyük fel azt is, hogy csak két pont fér egy kosárba. Javasoljunk jó vágást arra az esetre, ha az alábbi pontot szúrjuk be:

* a) Sebesség = 250 és memória = 48.

b) Sebesség = 333 és memória = 48.

5.2.5. feladat: Tegyük fel, hogy az $R(x, y)$ relációt egy rácsos állományban tároljuk. Mindkét attribútum 0 és 1000 közötti értékeket vehet fel. Ennek a rácsos állománynak a felosztása történetesen egyenletes, x-re minden 20 egységnél van egy felosztás, azaz az osztópontok 20, 40, 60 és így tovább, y-ra pedig 50 egységenként, azaz 50, 100, 150 és így tovább.

a) Hány kosarat kell megvizsgálnunk, hogy megválaszoljuk a következő tartománylekérdezést?

```
SELECT *
FROM R
WHERE 310 < x AND x < 400 AND 520 < y AND y < 730;
```

*! b) Legközelebbi szomszéd-lekérdezést akarunk végrehajtani a (110, 205) pontra. Azzal a kosárral kezdjük a keresést, amelyiknek a bal alsó sarka (100, 200) és a jobb felső sarka (120, 250), és azt találjuk, hogy a legközelebbi pont ebben a kosárban, a (115, 220). Mely kosarakat kell még átnéznünk ahhoz, hogy megbizonyosodjunk róla, hogy ez a legközelebbi pont?

! **5.2.6. feladat:** Tegyük fel, hogy van egy rácsos állományunk három rácsvonallal (azaz négy sávval) minden dimenziójában. Azonban az (x, y) pontoknak történetesen van valami különleges tulajdonságuk. Mondjuk meg a nem üres kosarak lehetséges legnagyobb számát, ha:

* a) A pontok egy vonalon vannak, azaz létezik két konstans, a és b, úgy, hogy $y = ax + b$ minden egyes (x, y) pontra.

b) A pontok között másodfokú összefüggés van, azaz létezik három konstans, a, b és c, úgy hogy $y = ax^2 + bx + c$ minden egyes (x, y) pontra.

5.2.7. feladat: Tegyük fel, hogy az $R(x, y, z)$ relációt egy particionált tördelőtáblában tároljuk, aminek 1024 kosara van (azaz 10 bites a kosárcím). Az R-re vonatkozó lekérdezések mindegyike pontosan egyet ad meg az attribútumok közül, és bármelyiket a három attribútum közül egyforma valószínűséggel adhatják meg. Ha a tördelőfüggvény 5 bitet készít az x alapján, 3 bitet az y alapján és 2 bitet a z alapján, akkor mennyi a kosarak átlagos száma, amelyeket meg kell vizsgálni egy lekérdezés megválaszolásához?

!! **5.2.8. feladat:** Tegyük fel, hogy van egy tördelőtáblánk, amelynek a kosarai 0-tól $2^n - 1$-ig vannak számozva, azaz a kosárcím n bit hosszú. A táblában egy olyan relációt akarunk tárolni, amelynek az attribútumai x és y. Egy lekérdezés vagy az x-et, vagy az y-t határozza meg, de sohasem mind a kettőt egyszerre. Legyen p annak a valószínűsége, hogy az x értéke a megadott.

a) Tegyük fel, hogy a tördelőfüggvényt úgy osztjuk fel, hogy m bitet használunk x-re, és a maradék $n - m$ bitet y-ra. Mi a megvizsgálandó kosarak várható száma m, n és

p függvényében, ami szükséges ahhoz, hogy megválaszoljunk egy véletlenszerű lekérdezést?

b) Mely m értékre (mint n és p függvénye) lesz minimális a kosarak várható száma? Ne aggódjunk, hogy ez az m valószínűleg nem egész szám lesz.

*! **5.2.9. feladat:** Tegyük fel, hogy van egy $R(x, y)$ relációnk, 1 000 000 véletlenszerűen elosztott ponttal. x és y egyaránt 0 és 1000 közötti értékeket vehet fel. Az R relációnak 100 sora fér egy blokkba. Úgy döntünk, hogy egy rácsos állományt használunk, mindegyik dimenzióban egyforma lépésközzel elhelyezett rácsvonalakkal, ahol a sávok szélessége m. Olyan m-et akarunk választani, amelyre minimális a lemez I/O-műveletek száma, ami ahhoz kell, hogy beolvassuk az összes olyan kosarat, amely egy 50 egység oldalú négyzetre vonatkozó tartománylekérdezés megválaszolásához kell. Feltehető, hogy a négyzet oldalai sosem esnek egybe a rácsvonalakkal. Ha az m-et túl nagyra választjuk, akkor sok túlcsordulásblokkunk lesz minden kosárhoz, és sok pontja a kosárnak a lekérdezés tartományán kívül lesz. Ha az m-et túl kicsire választjuk, akkor túl sok kosár lesz, és valószínűleg nem lesznek tele adattal a blokkok. Mi az m legjobb értéke?

5.3. Faszerű struktúrák többdimenziós adatokhoz

Most további négy struktúrát fogunk áttekinteni, amelyek hasznosak többdimenziós adatokra vonatkozó tartomány- vagy legközelebbi szomszéd-lekérdezések esetén. E célból tárgyaljuk a:

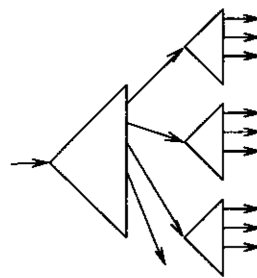
1. többkulcsos indexeket,
2. *kd*-fákat,
3. Quad-fákat,
4. R-fákat.

Az első hármat ponthalmazokhoz szánták. Az R-fa általában területek halmazának ábrázolására használatos, de pontokhoz is hasznos lehet.

5.3.1. Többkulcsos indexek

Tegyük fel, hogy van valahány attribútumunk, amik az adatpontjaink dimenzióit képviselik, és támogatni akarjuk a tartománylekérdezéseket vagy a legközelebbi szomszéd-lekérdezéseket ezeken a pontokon. Egy egyszerű faszerű szerkezet ezen pontok eléréséhez az indexek indexe, vagy általánosabban egy fa, amelyben a csomópontok, minden egyes szinten valamelyik attribútum indexei.

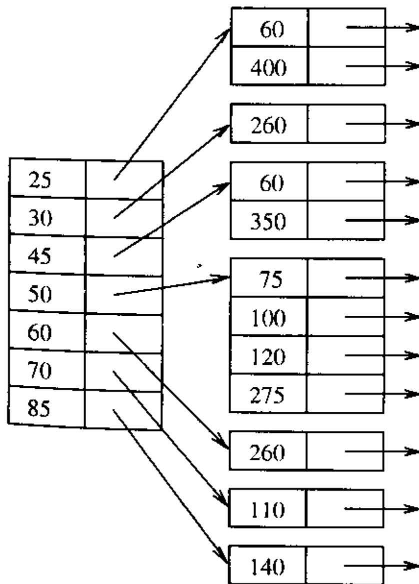
Az ötletet az 5.11. ábra mutatja két attribútum esetén. A „fa gyökere” egy index, a



5.11. ábra. *Egymásba ágyazott indexek használata különböző kulcsokon*

két attribútum közül az elsőre. Ez az index lehet bármelyik fajta szokásos index, mint a B-fa vagy a tördelőtábla. Az index hozzárendel minden egyes keresésikulcs-értékhez – azaz, az első attribútum értékeihez – egy mutatót a másik indexre. Ha V az első attribútum egy értéke, akkor az index, amit elérünk, a V értéket és annak mutatóját követve, az egy olyan pontok halmazának indexe, amelyeknél az első attribútum értéke V, a második attribútum értéke pedig tetszőleges.

5.13. példa: Az 5.12. ábra egy többkulcsos indexet mutat a megszokott „arany ékszer” példánkhoz, ahol az életkor az első attribútum, a második pedig a fizetés. A



5.12. ábra. *Többszintű indexek az életkor/fizetés adatokhoz*

gyökérindex az életkoron, az 5.12. ábra bal oldalán látható. Még nem mutattuk meg azt, hogy hogyan működik az index. Például, a kulcs-mutató párok, amelyek az index hét sorát alkotják, a B-fa levelein szétterjedhetnek. Azonban lényeges, hogy csak azok a kulcsok szerepelnek az életkorok között, amelyekhez van egy vagy több adatpont, és az index egyszerűvé teszi egy adott értékhez rendelt mutató megtalálását.

Az 5.12. ábra jobb oldalán hét index van, amelyek magukhoz a pontokhoz való hozzáférést teszik lehetővé. Például, ha követjük azt a mutatót, amely a gyökérindexben az 50 életkor értékhez van rendelve, elérünk egy kisebb rekordszámú indexhez, amelyben a fizetés a kulcs, és a négy kulcsérték az a négy fizetésérték, ami az életkor = 50 értékhez tartozó pontokban van. Megint csak nem jelöltük az ábrán, hogyan van megvalósítva az index, csak a kulcs-mutató párosítást adtuk meg. Amikor követjük a mutatókat, amelyek az egyes értékekhez (75, 100, 120 és 275) vannak rendelve, megkapjuk az ábrázolt egyedi rekordokat. Például, ha a 100 értékhez rendelt mutatót követjük, megtaláljuk azt a személyt, akinek az életkora 50, és a fizetése 100 ezer dollár. □

Többkulcsos index esetén a második vagy további szintű indexek nagyon kicsik lehetnek. Például az 5.12. ábrán található négy második szintű index, amiben csak egyetlen pár van. Ezért célszerű úgy megvalósítani ezeket, mint egyszerű táblákat, amelyekből többet berakhatunk egy blokkba, ahogyan azt az 5.2.5. részben a „Kis kosarak kezelése" című keretes rész javasolta.

5.3.2. A többkulcsos indexek hatékonysága

Nézzük meg milyen hatékony egy többkulcsos index a különféle többdimenziós lekérdezések esetén. Két attribútum esetére koncentrálunk, bár az általánosítás kettőnél több attribútum esetére magától értetődő.

Lekérdezések részleges egyezéssel

Ha az első attribútum van megadva, az elérés igen hatékony. A gyökérindexet használjuk annak az egy alindexnek a megtalálására, amely az elérni kívánt pontokhoz vezet. Például, ha a gyökér egy B-fa-index, akkor két vagy három lemezműveletet végzünk, amíg megkapjuk a megfelelő alindexet, és aztán a többi lemez I/O-művelet ahhoz szükséges, hogy elérjük teljes egészében az indexet, és magukat a pontokat az adatállományban. Másrészt azonban, ha az első attribútum értéke nem adott, akkor végig kell nézni az összes alindexet, ami időigényes eljárás lehet.

Tartománylekérdezések

A többkulcsos index nagyon jó a tartománylekérdezéshez, amennyiben az egyes indexek maguk támogatják a tartománylekérdezést a saját attribútumukon (azaz, ha B-fa-indexek). Egy tartománylekérdezés megválaszolásához használjuk a gyökérindexet és az első attribútum tartományát, így megtalálhatjuk az összes olyan alindexet, amely

tartalmazhat megfelelő pontokat. Azután ezeket az alindexeket vizsgáljuk meg a második attribútumhoz adott tartományt használva.

5.14. példa: Tegyük fel, hogy az 5.12. ábra szerinti többkulcsos indexünk van, és a tartománylekérdezésünk a $35 \le$ életkor ≤ 55, és a $100 \le$ fizetés ≤ 200. Megvizsgálva a gyökérindexet, a 45 és az 50 kulcsértékeket találjuk az életkori határok között. Kövessük a kapcsolódó mutatókat a két alindexhez a fizetésen. A 45 éves életkorhoz nincs fizetés a 100–200 tartományban, míg az 50 éves korhoz tartozó indexben van két ilyen fizetés: a 100 és a 120. Tehát csak két pont van az adott tartományban: az (50, 100) és az (50, 120). □

Legközelebbi szomszéd-lekérdezések

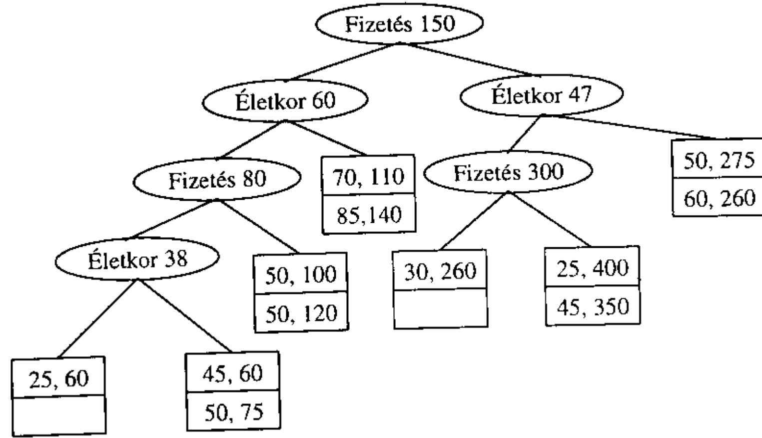
A legközelebbi szomszéd-lekérdezés megválaszolása többkulcsos index segítségével ugyanazt az eljárást követi, mint a legtöbb adatstruktúra esetén ebben a fejezetben. Az (x_0, y_0) pont legközelebbi szomszédjának megkereséséhez választunk egy d távolságot, amelyre várható, hogy találunk néhány pontot az (x_0, y_0) ponttól d távolságon belül. Azután megválaszoljuk az $x_0 - d \le x \le x_0 + d$ és az $y_0 - d \le y \le y_0 + d$ tartománylekérdezést. Ha az jön ki, hogy nincs pont ebben a tartományban, vagy ha van is pont, de a legközelebbi pont távolsága is nagyobb (x_0, y_0)-tól, mint d (azért még lehet közelebbi pont a tartományon kívül, ahogyan megtárgyaltuk az 5.1.5. részben), akkor meg kell növelnünk a tartományt, és újra kell keresnünk. Viszont tudjuk olyan sorrendben végezni a keresést, hogy a közelebbi helyeket nézzük át először.

5.3.3. *kd*-fák

Egy *kd*-fa (*k* dimenziós keresőfa) egy központi memóriabeli adatstruktúra, amely a bináris keresőfa általánosítása többdimenziós adatokra. Bemutatjuk az ötletet, majd megtárgyaljuk, hogyan lehet alkalmazni blokk módú tárolókra. A *kd*-fa egy bináris fa, amelyben minden belső csomóponthoz hozzá van rendelve egy a attribútum, és egy V érték, ami szétválasztja az adatpontokat két részre: az egyik rész, azon adatpontokból áll, amelyekre az a érték kisebb V-nél, és a másik rész, amelyben az a érték nagyobb vagy egyenlő mint V. A fa egymás alatti szintjein az attribútumok különböznek, miközben a szintek váltogatják az attribútumokat, körbe járva az összes dimenzión.

A klasszikus *kd*-fában, az adatpontok a csomópontokban vannak elhelyezve csakúgy, mint a bináris keresőfában. Azonban az alapötleten két módosítást fogunk végrehajtani annak érdekében, hogy a blokk módú tárolók bizonyos, korlátozott előnyeit kihasználhassuk.

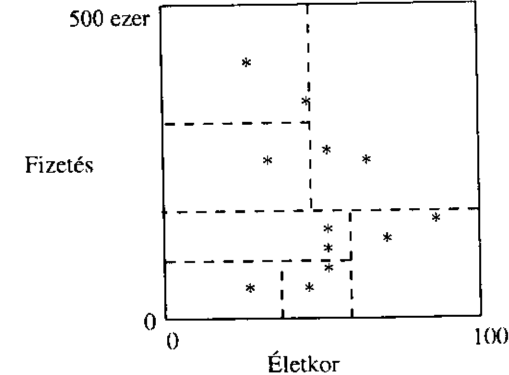
1. A belső csomópontokban csak egy attribútum lesz, egy elhatároló érték ehhez az attribútumhoz, és a mutatók a bal és a jobb gyerekre.
2. A levelek blokkok lesznek, annyi rekordnak biztosítva helyet, amennyi a blokkban elfér.



5.13. ábra. *Egy* kd-*fa*

5.15. példa: Az 5.13. ábrán egy *kd*-fa látható a már megszokott „arany ékszer" példánk tizenkét pontjával. Az egyszerűség kedvéért olyan blokkokat használunk, amelyekbe csak két rekord fér el, ezeket a blokkokat és a tartalmukat a négyzet alakú levelek mutatják. A belső csomópontok ovális alakúak egy attribútummal – vagy életkor, vagy fizetés – és egy értékkel. Például a gyökér a fizetés szerint vág ketté, a bal oldali részfában minden rekordban a fizetés kisebb, mint 150 ezeR dollár, míg a jobb oldali részfában miden rekord legalább 150 ezer dollár fizetés értékű.

A második szinten a vágás életkor szerinti. A gyökér bal oldali gyereke a 60 éves életkornál választ el, tehát minden rekordra ennek a bal oldali részfájában az életkor kisebb mint 60, és a fizetés 150 ezer dollárnál kevesebb. A jobb oldali részfájában az életkor legalább 60, és a fizetés 150 ezer dollárnál kevesebb. Az 5.14. ábra mutatja, hogy a különböző belső csomópontok hogyan osztják fel a pontok terét levélblokkokra. Például, a vízszintes vonal a fizetés = 150 értéknél a gyökérnél lévő vágást mutatja. A vonal alatti rész függőlegesen a 60 éves életkornál válik ketté, míg a felette lévő rész – a gyökér jobb oldali gyerekében lévő határoló értéknek megfelelően – a 47 éves életkornál. □



5.14. ábra. *Az 5.13. ábrán látható fának megfelelő felosztások (partíciók)*

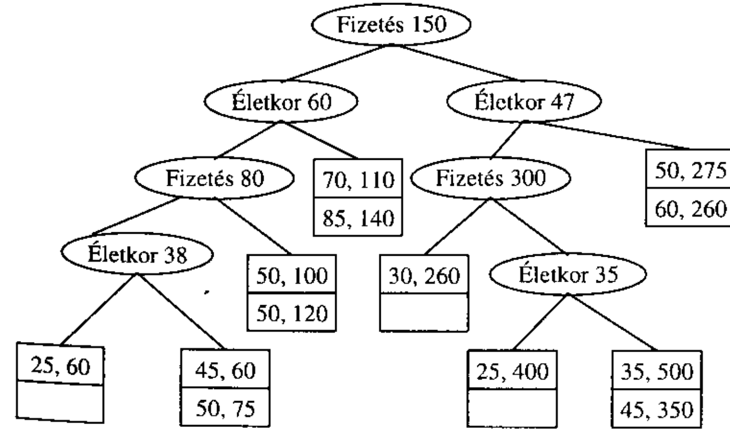
5.3.4. Műveletek a *kd*-fákon

Egy sor keresése, ha adott minden dimenzió értéke, ugyanúgy történik, mint a bináris keresőfákban. Minden belső csomópontnál eldöntjük, hogy merre kell továbbmenni, és így eljutunk egy levélhez, amelynek a blokkját kerestük.

Egy beszúrás végrehajtásához úgy járunk el, mint a keresésnél. Végül egy levélhez jutunk, és ha annak a blokkjában van hely, az új adatpontot betesszük oda. Ha nincs hely, kettévágjuk a blokkot, és megosztjuk a tartalmát, aszerint, hogy azon a szinten, amin a szétvágandó levél van, melyik a megfelelő attribútum. Létrehozunk egy új belső csomópontot, amelynek a gyerekei a két új blokk, és úgy állítjuk be az elhatároló értéket a belső csomópontban, hogy megfeleljen a vágásnak, amit épp most készítettünk.[1]

5.16. példa: Tegyük fel, hogy valaki, aki 35 éves, és 500 ezer dollár a fizetése, arany ékszert vásárol. A gyökérnél kezdünk, tudjuk, hogy a fizetés legalább 150 ezer dollár, tehát jobbra megyünk. Ott a csomópontnál összehasonlítjuk a 35 éves kort a 47-tel, ez balra irányít bennünket. A harmadik szinten ismét a fizetéseket hasonlítjuk össze, és a mi értékünk nagyobb, mint a 300 ezer dollár elválasztó érték. Így egy levélhez értünk, ami a (25, 400) és a (45, 350) pontokat tartalmazza, és ez lenne a helye az új, (35, 400) pontnak.

Nincs hely a blokkban három pont számára, tehát szét kell vágnunk. A negyedik szint az életkor szerinti vágás, tehát egy életkort kell választanunk, ami olyan egyenletesen osztja el a rekordokat, amennyire csak lehetséges. A középső érték a 35, egy jó választás, tehát a levelet helyettesítjük egy belső csomóponttal, amely az életkor = 35-



5.15. ábra. *A* kd-*fa a (35, 500) rekord beszúrása után*

[1] Felmerülhet az a gond, hogy olyan sok pont van azonos értékkel az adott dimenzióban, hogy az adott kosárban csak egy érték van ahhoz a dimenzióhoz, és így nem tudjuk szétvágni. Ekkor megpróbálhatjuk szétvágni egy másik dimenzió mentén, vagy használhatunk túlcsordulásblokkot.

nél vág. A belső csomóponttól balra egy levélblokk lesz a (25, 400) rekorddal, míg a jobbra lévő levél blokkban lesz a másik két rekord, ahogy az 5.15. ábrán látható. □

Az ebben a fejezetben tárgyalt bonyolultabb lekérdezéseket szintén támogatják a *kd*-fák. Íme az alapvető ötletek és az algoritmusok áttekintése.

Lekérdezések részleges egyezéssel

Amennyiben adottak bizonyos attribútumok értékei, akkor ha olyan szinten vagyunk, amelyhez tartozó attribútum értéke ismert, akkor mehetünk valamelyik konkrét irányba. Ha nem tudjuk az aktuális csomóponthoz tartozó attribútum értékét, akkor meg kell vizsgálni mindkét gyereket. Például, ha az 5.13. ábrán, azokat a pontokat keressük, amelyekre az életkor = 50, akkor meg kell néznünk a gyökér mindkét gyerekét, mivel a gyökér a fizetés szerint vág. Azonban, a gyökér bal oldali gyerekénél csak balra kell továbbmenni, míg a gyökér jobb oldali gyerekénél csak a jobb oldali részfát kell felderítenünk. Tegyük fel a példa kedvéért, hogy van egy tökéletesen kiegyensúlyozott, kétdimenziós fánk, amelynek sok szintje van, és a kereséshez az egyik dimenzió meg van adva. Akkor a másikhoz tartozó szinteken mindig meg kell vizsgálnunk mindkét utat, végül is el kell érnünk körülbelül annyi levelet, amennyi a levelek teljes számának a négyzetgyöke.

Tartománylekérdezések

Néha a tartomány lehetővé teszi számunkra, hogy csak a csomópont egyik gyereke felé menjünk tovább, de ha a tartomány tartalmazza a csomópont elhatároló értékét, akkor mindkét gyereket meg kell néznünk. Például, ha az életkor tartománya 35 és 55 közötti, a fizetésé 100 ezer dollártól 200 ezer dollárig, akkor az 5.13. ábra fáját a következőképpen járhatjuk be: A fizetés tartománya tartalmazza a gyökérnél lévő 150 ezer dollár értéket, így mindkét gyereket meg kell néznünk. A bal oldali gyereknél a tartomány teljesen a bal oldalra esik, tehát ahhoz a csomóponthoz megyünk, ahol a fizetés 80 ezer dollár. Most a tartomány teljesen jobbra esik, így elértük a levelet, amiben az (50, 100) és az (50, 120) rekordok vannak, ezek mindketten megfelelnek a tartománylekérdezésnek. Visszatérve a gyökér jobb oldali gyerekéhez, az elválasztó érték az életkor = 47, ezért meg kell néznünk mindkét részfát. A 300 ezer dollár fizetésértékű csomópontnál csak balra mehetünk, és így megtaláljuk a (30, 260) pontot, ami kívül esik a tartományon. Az életkor = 47 csomópont jobb oldali gyerekénél találunk további két pontot, de ezek is kívül esnek a tartományon.

Legközelebbi szomszéd-lekérdezések

Használjuk ugyanazt a megközelítést, amit az 5.3.2. részben tárgyaltunk. Kezeljük úgy a feladatot, mint egy tartománylekérdezést a megfelelő tartománnyal, és ismételjük meg egy nagyobb tartománnyal, ha szükséges.

5.3.5. A *kd*-fák alkalmazása másodlagos tárolók esetén

Tegyük fel, hogy egy n levelű *kd*-fát egy állományban tárolunk. Ebben az esetben a gyökértől a levélig tartó út átlagos hossza $\log_2 n$, mint minden bináris fánál. Ha minden egyes csomópontot egy blokkban tárolunk, amikor bejárunk egy utat csomópontonként egy lemez I/O-műveletet kell végrehajtanunk. Például, ha $n = 1000$, akkor körülbelül 10 lemez I/O-műveletre lesz szükségünk, ami sokkal több, mint egy tipikus B-fa esetén szükséges – még ennél sokkal nagyobb állományok esetén is – 2 vagy három lemez I/O-művelet. Ráadásul, mivel a *kd*-fák belső csomópontjaiban viszonylag kevés információ van, a blokk nagy része elpazarolt hely.

Nem tudjuk teljesen megoldani a hosszú út és kihasználatlan terület kettős problémáját. Viszont itt van két megközelítés, ami némi javulást fog hozni a hatékonyságban.

Többutas elágazások a belső csomópontokban

A *kd*-fák belső csomópontjai jobban hasonlítanának a B-fa-csomópontokra, ha több kulcs-mutató párjuk lenne. Ha n kulcsunk lenne egy csomópontban, akkor az a attribútum értékeit $n + 1$ tartományra oszthatnánk. Ha $n + 1$ mutató lenne, követhetnénk az egy megfelelőt ahhoz a részfához, amely csak olyan pontokat tartalmaz, amelyekre az a attribútum értéke abba a tartományba esik. Problémák akkor lépnek fel, amikor megpróbáljuk újraszervezni a csomópontokat azzal a céllal, hogy megtartsuk az eloszlást és az egyensúlyt, ahogyan a B-fáknál tettük. Például, tegyük fel van egy csomópontunk, ami az életkor szerint választ szét, és össze kell olvasztanunk a két gyere-

Semmi sem tart örökké

Az ebben a fejezetben tárgyalt adatstruktúrák lehetővé teszik, hogy beszúráskor és törléskor helyi döntéseket hozzunk, hogyan kell átszervezni a struktúrát. Sok adatbázis-módosítás után, ezen helyi döntések hatása valamiféle kiegyensúlyozatlanságot vihet a struktúrába. Például túl sok üres kosara lehet egy rácsos állománynak, vagy egy *kd*-fa erősen kiegyensúlyozatlan lehet.

Minden adatbázisnál teljesen szokásos, hogy időnként újraszervezik. Az adatbázis visszatöltésekor megvan a lehetősége annak, hogy úgy hozzuk létre az indexstruktúrákat, hogy – legalábbis abban a pillanatban – olyan kiegyensúlyozottak és hatékonyak legyenek, amennyire csak lehetséges az adott típusú indexeknél. Az ilyen újraszervezés költségét a kiegyensúlyozatlansághoz vezető sok módosítás számlájára lehet írni, így az egy módosításra eső költség kicsi. Azonban ehhez az kell, hogy „le tudjuk kapcsolni" az adatbázist, azaz elérhetetlenné tudjuk tenni a visszatöltés idejére. Az ilyen helyzet vagy okoz gondot, vagy nem az alkalmazástól függően. Például sok adatbázist leállítanak éjszakára, amikor senki sem használja őket.

két, amelyek a fizetés szerint vágnak. Nem készíthetünk egyszerűen egy csomópontot, amely tartalmazza a két gyerek fizetés tartományait, mivel ezek a tartományok általában átfedik egymást. Vegyük észre, mennyivel könnyebb lenne, ha (mint a B-fáknál) a két gyerek egyaránt tovább finomítaná az életkori felosztást.

A belső csomópontok blokkokba csoportosítása

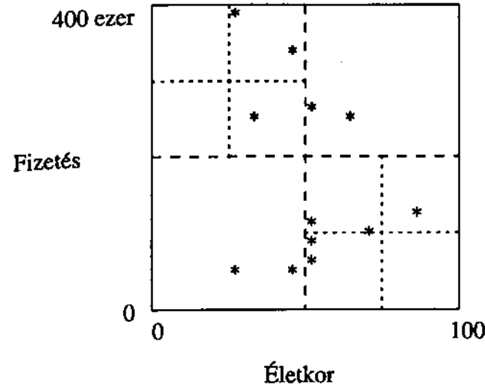
Az előzőek helyett megtarthatjuk azt az elvet, hogy a fa csomópontjainak csak két gyereke van. Több belső csomópontot tehetünk egyetlen blokkba. Abból a megfontolásból, hogy minimalizáljuk a lemezről beolvasandó blokkok számát, miközben lefelé haladunk egy útvonalon, az a legjobb, ha egy csomópont és – bizonyos szint mélységig – annak az összes leszármazottja egyetlen blokkban van. Így, ha egyszer lekérdezünk egy adott csomópontot tartalmazó blokkot, biztosak lehetünk abban, hogy fel tudunk használni néhány további csomópontot is ugyanebből a blokkból, ezzel lemez I/O-műveleteket takarítunk meg. Például tegyük fel, hogy három belső csomópontot tudunk tárolni egy blokkban. Akkor az 5.13. ábrán lévő fa esetén berakhatjuk a gyökeret és két gyerekét egy blokkba. Aztán a fizetés = 80-hoz tartozó csomópontot, és a bal oldali gyerekét betehetjük egy másik blokkba, így csak a fizetés = 300 csomópont maradt, ami egy újabb blokkba kerülhet; ezt például elhelyezhetnénk az előző két csomópont blokkjába is, viszont az osztozkodás jelentős munkát igényel, amikor a fa nő vagy csökken. Így, ha a (20, 60) pontot akarjuk megtalálni, csak két blokkot kell bejárnunk, bár négy csomóponton haladunk át.

5.3.6. Quad-fák

Egy quad-fában minden egyes belső csomópont megfelel egy négyzet alakú területnek kétdimenziós esetben, illetve egy k dimenziós kockának k dimenzió esetén. Mint a fejezet más adatstruktúrái esetén is, elsősorban a kétdimenziós esetet tekintjük át[2]. Ha egy négyzetbe eső pontok száma nem több, mint ami elfér egy blokkban, akkor úgy gondolhatunk erre a négyzetre, mint egy fa levelére, és egy blokkal ábrázoljuk, ami a pontjait tartalmazza. Ha túl sok a pont ahhoz, hogy beleférjen egy blokkba, akkor úgy kezelhetjük a négyzetet, mint egy belső csomópontot, amelynek a gyerekei felelnek meg a négyzet négy negyedének.

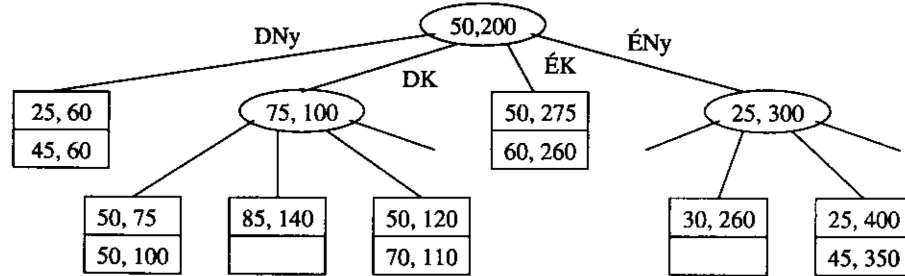
5.17. példa: Az 5.16. ábra az arany ékszer adatpontjait mutatja területekbe szervezve, amelyek megfelelnek egy quad-fa csomópontjainak. Az egyszerű számolás kedvéért, a szokásos teret leszűkítettük úgy, hogy a fizetés 0 és 400 ezer dollár közé essen, és nem 500 ezer dollárig, mint a fejezet többi példájában. Továbbra is feltesszük, hogy két rekord fér egy blokkba.

[2] Ilyenkor használatos a négyágú fa elnevezés is. *A fordító megjegyzése.*



5.16. ábra. *Quad-faként szervezett adatok*

A fát pontosan mutatja az 5.17. ábra. Az iránytű jelöléseit használjuk a negyedekhez, és a csomópontok gyerekeihez (azaz DNy a délnyugati negyedet jelöli, azaz a középponttól balra és lefelé lévő pontokat). A gyerekek sorrendje is mindig olyan, mint a gyökérnél feltüntetett. A belső csomópontok a terület középpontjának a koordinátáit jelölik.



5.17. ábra. *Egy quad-fa*

Mivel az egész tér 12 pontból áll, és csak két pont fér egy blokkba, szét kell vágnunk a teret részekre, amit a szaggatott vonal mutat az 5.16. ábrán. A kapott negyedek közül kettőnek – a délnyugatinak és az északkeletinek –, csak két pontja van. Ezek ábrázolhatók levélként, és nem kell tovább darabolni őket.

A maradék két negyednek kettőnél több pontja van. Mindkettőt tovább negyedeljük, ahogy azt az 5.16. ábrán a szaggatott vonal mutatja. Az eredményül kapott részeknek már csak kettő vagy kevesebb pontja van, így nem szükséges a további darabolás. □

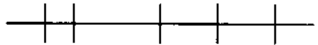
Mivel k dimenzió esetén egy quad-fában egy belső csomópontnak 2^k gyereke van, található k-nak egy olyan tartománya, amelyre a csomópontok kényelmesen elhelyezhetők egy blokkban. Például, ha 128, azaz 2^7 mutató fér el egy blokkban, akkor $k = 7$ megfelelő dimenziószám. A kétdimenziós esetben azonban, a helyzet nem sokkal

jobb, mint a *kd*-fáknál; egy belső csomópontnak négy gyereke van. Továbbá, míg a *kd*-fa esetén a csomópont számára megválaszthatjuk az elhatároló pontot, addig itt kénytelenek vagyunk a quad-fa terület közepét választani, ami vagy egyenletesen osztja szét a terület pontjait, vagy nem. Várhatóan – különösen akkor, ha a dimenziószám nagy – sok üres mutató lesz a belső csomópontokban (ami az üres részeknek felel meg). Természetesen valamivel okosabban is ábrázolhatjuk a sokdimenziós csomópontokat úgy, hogy csak a nem üres mutatókat tároljuk, és egy leírót készítünk, mely megadja, hogy melyik részre vonatkozik, így jelentős helyet takaríthatunk meg.

Nem megyünk bele a részletekbe az alapvető műveleteket illetően, amiket a *kd*-fák esetén az 5.3.4. részben tárgyaltunk. Az algoritmusok quad-fa esetén hasonlóak a *kd*-fákéhoz.

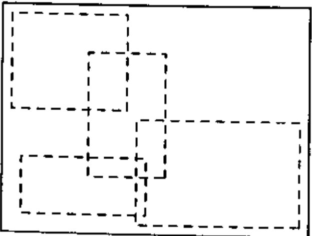
5.3.7. R-fák

Egy *R-fa* (régiófa) egy olyan adatstruktúra, amely a B-fa néhány alapötletét általánosítja többdimenziós adatokra. Emlékezzünk arra, hogy a B-fa-csomóponthoz egy kulcshalmaz tartozik, amely egy egyenest szakaszokra oszt. Az egyenes pontjai pontosan egy szakaszhoz tartoznak, ahogy az 5.18. ábra mutatja. Tehát a B-fa megkönnyíti a pontok elérését; ha feltételezzük, hogy a pont a B-fa-csomópont által ábrázolt egyenesen van, akkor egyértelműen meg tudjuk határozni a csomópontnak azt a gyerekét, ahol a pont megtalálható.



5.18. ábra. *A B-fa csomópontjai az egy egyenes mentén levő kulcsokat diszjunkt darabokra osztják*

Egy R-fa viszont olyan adatokat ábrázol, amelyek két- vagy többdimenziós területek lehetnek, ezeket *adatrégiónak* nevezzük. Egy R-fa belső csomópontja egy *belső régiónak* felel meg, vagy egyszerűen csak „régió"-nak, ami általában nem adatrégió. Elméletileg a régió lehet bármilyen alakú, de a gyakorlatban általában téglalap vagy valami más egyszerű alakzat. A R-fa-csomópontnak – kulcsok helyett – alrégiói vannak, amelyek gyerekeinek a tartalmát ábrázolják. Az 5.19. ábra bemutat egy R-fa-csomópontot, amely a nagy vastag vonalas téglalaphoz van rendelve. A pontozott vonalas téglalapok ábrázolják az alrégiókat, amelyek a négy gyerekéhez vannak rendelve. Megjegyezzük, hogy az alrégiók nem fedik le az egész régiót, ami elfogadható, amíg a nagy régión fekvő adatrégiók teljesen benne vannak valamelyik kis területben. Továbbá, a kis régiók átfedhetik egymást, bár kívánatos az átfedéseket minimalizálni.



5.19. ábra. *Egy R-fa-csomópont régió és a gyerekeinek az alrégiói*

5.3.8. Műveletek az R-fákon

Egy tipikus lekérdezés, amihez egy R-fa jól használható, az a „hol-vagyok-én" lekérdezés, amely megad egy P pontot, és azt az adatrégiót vagy régiókat kérdezi, amelyikben a pont benne van. A gyökértől indulunk, amihez az egész régió hozzá van rendelve. Megvizsgáljuk a gyökérben található alrégiókat, és meghatározzuk azokat a gyerekeit, amelyek olyan belső régióknak felelnek meg, amik tartalmazzák a P pontot. Ilyen régió lehet: 0, 1 vagy több.

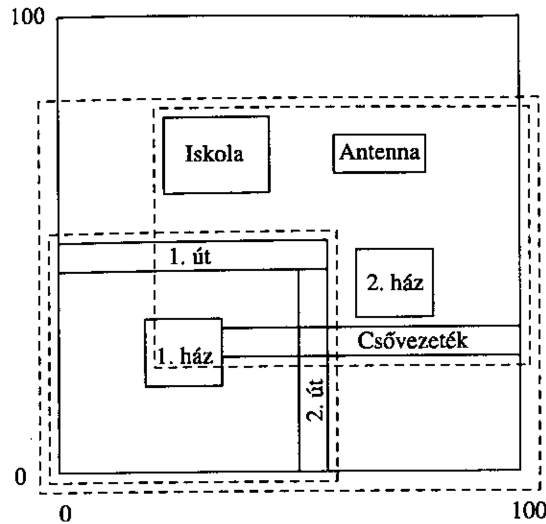
Ha nincs ilyen régió, akkor készen vagyunk, a P nincs benne egyik adatrégióban sem. Ha van legalább egy belső régió, amely tartalmazza a P pontot, akkor minden ilyen régiónak megfelelő gyereknél rekurzívan tovább kell keresnünk. Így eljutva egy vagy több levélhez, találjuk meg a tényleges adatrégiókat, és abban vagy a teljes adatrekordot, vagy csak egy mutatót a keresett adatrekordra.

Amikor egy új R régiót szúrunk be egy R-fába, a gyökértől elindulva próbálunk meg egy olyan alrégiót találni, amibe az R beleillik. Ha egynél több ilyen régió van, kiválasztunk egyet, elmegyünk a neki megfelelő gyerekhez, és ott megismételjük az eljárást. Ha nincs ilyen R-et tartalmazó alrégió, akkor meg kell növelnünk az alrégiók valamelyikét, ennek a kiválasztása azonban általában nem egyszerű. Intuitíven: a régiókat csak a feltétlen szükséges mértékben akarjuk növelni, vagyis a gyerek alrégiói közül azt kell választani, amelyik a legkevésbé fog növekedni; ennek a régiónak változtassuk meg a határait úgy, hogy R-et tartalmazza, majd rekurzívan szúrjuk be R-et a megfelelő gyereknél.

Végül eljutunk egy levélhez, ahová be kell szúrnunk az R régiót. Ha nincs hely a levélben R számára, akkor a levelet szét kell vágnunk. Most is több lehetőség közül választhatunk. Általában azt akarjuk, hogy a két részterület a lehető legkisebb legyen, de még – többek közt – tartalmazzák az eredeti levél összes adatrégióját is. Amikor a levelet szétvágjuk, a felette lévő csomópontban az eredeti levél régió-mutató párját kicseréljük két régió-mutató értékkel, amelyek a két új levélnek felelnek meg. Ha van hely a szülőben, akkor készen vagyunk. Különben – mint a B-fák esetén – rekurzívan szétvágjuk a csomópontokat a fán felfelé haladva.

5.18. példa: Vizsgáljuk meg azt az esetet, amikor az 5.1. ábrán látható térképhez egy új régiót akarunk hozzáadni. Tegyük fel, hogy a levelekben hat régiónak van hely. További feltevés, hogy az 5.1. ábra mind a hat régiója egyetlen levélben van, ennek a régióját a külső (folyamatos vonallal rajzolt) téglalap ábrázolja az 5.20. ábrán.

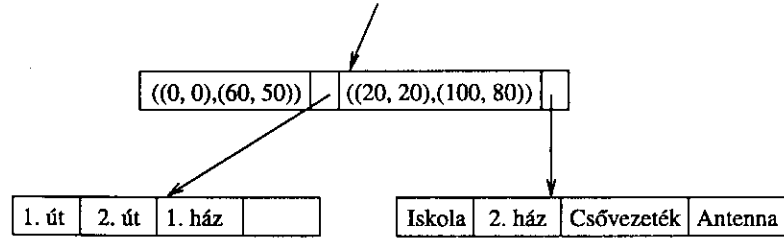
Tegyük fel, hogy a helyi rádiótelefon-társaság egy antennát akar telepíteni az 5.20. ábrán látható helyre. Mivel a hét adatrégió már nem fér el egy levélben, szétvágjuk a levelet, négy régió lesz az egyikben és három a másikban. Több lehetőségünk is van, ezek közül azt a felosztást választottuk, amit a 5.20. ábra mutat (a belső, szaggatott



5.20. ábra. *Objektumhalmaz szétvágása*

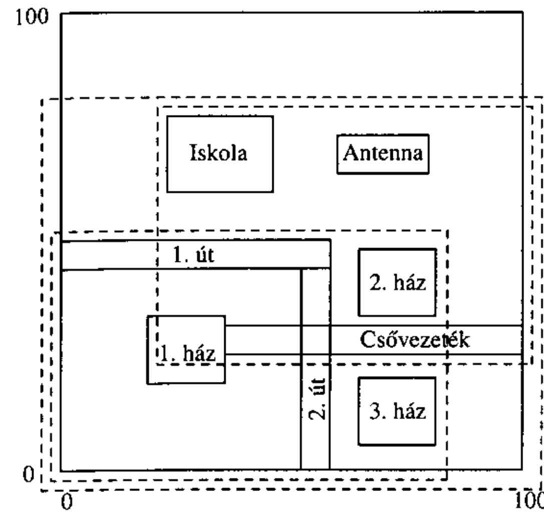
vonallal jelzett téglalapok), amely minimálissá teszi az átfedést, miközben olyan egyenletesen vágja szét a levelet, amennyire csak lehetséges.

Az 5.21. ábrán azt mutatjuk meg, hogy a két új levél hogyan illeszkedik be az R-fába. E csomópontok szülője tartalmazza a mutatókat mindkét levélre, és emellett a mutatókhoz kapcsoltan az adott levelet befedő téglalap alakú terület bal alsó és jobb felső sarkának koordinátáit is. □



5.21. ábra. *Egy R-fa*

5.19. példa: Tegyük fel, hogy beszúrunk egy újabb házat a 2. ház alá, a bal alsó sarok koordinátái (70, 5), a jobb felsőé pedig (80, 15). Mivel a ház nincs benne teljesen egyik levél régiójában sem, választanunk kell egy régiót, amit megnövelünk. Ha az alsó alrégiót növeljük meg, ami az 5.21. ábrán az első levélnek felel meg, akkor 1000 négyzetegységgel növeljük meg a területet, mivel 20 egységnyit növeljük jobb felé. Ha a másik alrégiót növeljük meg, az alját lejjebb húzva 15 egységgel, akkor a hozzáadott terület 1200 négyzetegység. Az elsőt részesítjük előnyben, és az új, megváltozott régiók az 5.22. ábrán láthatók. Az 5.21. ábra felső csomópontjában lévő régió leírását is meg kell változtatnunk ((0, 0), (60, 50))-ről ((0, 0), (80, 50))-re. □



5.22. ábra. *Terület kiterjesztése az új adat elhelyezéséhez*

5.3.9. Feladatok

5.3.1. feladat: Mutassunk többszintű indexet az 5.10. ábra adataihoz, ha az indexek (az adott sorrendben):

a) a sebesség és a memória,
b) a memória és a merevlemez,
c) a sebesség, a memória és a merevlemez.

5.3.2. feladat: Helyezzük el az 5.10. ábra adatait egy *kd*-fában. Tegyük fel, hogy két rekord fér egy blokkba. Minden egyes szinten válasszunk egy határoló értéket, amely olyan egyenletesen osztja szét a rekordokat, amennyire csak lehetséges. A vágandó attribútumok sorrendje legyen a következő:

a) a sebesség és a memória (felváltva),
b) a sebesség, a memória és a merevlemez (felváltva),
c) amelyik attribútum a legegyenletesebb vágást eredményezi az egyes csomópontokra.

5.3.3. feladat: Tegyük fel, hogy van egy $R(x, y, z)$ relációnk, ahol az x és y attribútumpár együtt alkotja a kulcsot. Az x attribútum 1 és 100, az y pedig 1 és 1000 közötti értéket vehet fel. Minden egyes x-hez 100 különböző értékű y rekord, és minden egyes y-hoz 10 különböző értékű x rekord tartozik. Vegyük észre, hogy így 10 000 rekord van az R-ben. Többkulcsos indexet akarunk használni, ami segít nekünk megválaszolni a következő alakú lekérdezéseket:

```
SELECT z
FROM R
WHERE x = C AND y = D;
```

ahol C és D konstans. Tegyük fel, hogy a blokkok tíz kulcs-mutató párt tudnak tárolni, és sűrű indexeket akarunk létrehozni minden szinten, esetleg ritka magasabb szintű indexeket felettük, emiatt minden index egyetlen blokkal indul. Szintén feltesszük, hogy kezdetben minden index- és adatblokk a lemezen van.

* a) Hány lemez I/O-művelet szükséges a fenti formájú lekérdezés megválaszolásához, ha az x szerinti az első index?
 b) Hány lemez I/O-művelet szükséges a fenti formájú lekérdezés megválaszolásához, ha az y szerinti az első index?
! c) Tegyük fel, hogy 11 blokkot tudunk folyamatosan a memóriában tartani. Mely blokkokat választanánk, és az x-et vagy az y-t tennénk az első indexnek, ha a továbbiakban szükséges lemez I/O-műveletek számát minimalizálni szeretnénk?

5.3.4. feladat: Az 5.3.3.a) feladat struktúrája esetén, hány lemez I/O-művelet szükséges a $20 \leq x \leq 35$ és $200 \leq y \leq 350$ tartománylekérdezés megválaszolásához? Tegyük fel, hogy az adatok egyenletesen oszlanak el, azaz a várható számú pontot találjuk bármilyen tartományon belül.

5.3.5. feladat: Az 5.13. ábrán látható fa esetén, mely új pontok kerülnének:

* a) a (30, 260) pontot tartalmazó blokkba?
 b) az (50, 100) és az (50, 120) pontokat tartalmazó blokkba?

5.3.6. feladat: Mutassuk meg az 5.15. ábrán látható fa lehetséges kibővítéseit, ha beszúrjuk előbb a (20, 110), majd azután a (40, 400) pontokat.

! **5.3.7. feladat:** Említettük, hogy ha egy *kd*-fa teljesen kiegyensúlyozott, és végrehajtunk egy „lekérdezés részleges egyezéssel" típusú kérést, ahol a két attribútum egyikének az értéke adott, akkor kereséskor körülbelül négyzetgyök n számú levelet kell bejárnunk az n levélből.

a) Magyarázzuk meg, hogy miért?
b) Ha a fa felváltva vág a d dimenzióban, és ezek közül a dimenziók közül m-nek az értékét megadtuk, akkor a levelek mekkora részét kell várhatóan végignézni?
c) Milyen a b) teljesítménye a particionált tördelőtáblához viszonyítva?

5.3.8. feladat: Helyezzük el az 5.10. ábra adatait egy quad-fába, amelynek a dimenziói: a sebesség és a memória. Tegyük fel, hogy a sebesség 100-tól 500-ig mehet, és a memória 0-tól 256-ig.

5.3.9. feladat: Ismételjük meg az 5.3.8. feladatot, hozzáadva harmadik dimenzióként a merevlemezt, ami 0 és 32 közötti lehet.

*! **5.3.10. feladat:** Tételezzük fel, hogy egy quad-fa középpontját szabadon megválaszthatjuk, ekkor vajon feloszthatók-e mindig olyan résznegyedekre a negyedek, amelyekben azonos számú pont van (illetve amilyen egyenletesesen csak lehet, ha a negyedben lévő pontok száma nem osztható 4-gyel)? Igazoljuk a választ.

! **5.3.11. feladat:** Tegyük fel, hogy van egy adatbázisunk, amiben 1 000 000 régió van, amelyek átfedhetik egymást. Egy R-fa csomópontjaiban (blokkjaiban) 100 régió-mutató pár fér el. Bármely csomópontban ábrázolt régiónak 100 alrégiója van, és az átfedések ezen régiók között olyanok, hogy a 100 alrégió összege 150%-a az eredeti régió méretének. Ha egy „hol-vagyok-én" lekérdezést hajtunk végre egy adott pontra, mennyi a visszakeresendő blokkok várható száma?

! **5.3.12. feladat:** Az 5.22. ábrán szereplő R-fában egy új régió vagy abba az alrégióba kerülhet, amelyik az iskolát tartalmazza, vagy abba, amelyik a 3. házat. Adjuk meg azokat a téglalap alakú régiókat, amelyekre az iskolát tartalmazó alrégiót részesítenénk előnyben (azaz, ez a választás minimalizálja az alrégió méretnövekedését).

5.4. Bittérképindexek

Nézzünk most egy olyan indexet, amely sokban különbözik az eddig tárgyalt típusúaktól. Kezdjük azzal, hogy egy állandó számú rekordból álló állományt tekintünk, amelyben a rekordok sorszámai 1, 2, ..., n. Továbbá az állomány adatstruktúrája olyan, amely lehetővé teszi, hogy könnyen megtaláljuk az i-edik rekordot, tetszőleges i-re.

Egy F mezőhöz tartozó *bittérképindex* tulajdonképpen n hosszú bitvektorok olyan gyűjteménye, amelyben minden, az F mezőben előfordulható értékhez tartozik egy bitvektor. A v értékhez tartozó vektor az i-edik helyen 1-et tartalmaz, ha az i-edik rekordban az F mező értéke v, és 0-t, ha nem.

5.20. példa: Vegyünk egy állományt, amelyben a rekordoknak két mezője van: F és G, amelyek egész, illetve szöveg típusúak. Az aktuális állomány 6 rekordot tartalmaz, amelyek 1-től 6-ig vannak számozva, és a következők az értékei sorban: (30, `foo`), (30, `bar`), (40, `baz`), (50, `foo`), (40, `bar`), (30, `baz`).

Az első mezőhöz, F-hez tartozó bittérképnek három bitvektora lesz, mindegyik 6 hosszú. Az első, amely a 30 értékhez tartozik: 110001, mivel az első, a második és a hatodik rekordban $F = 30$. A másik kettő, a 40 és 50 értéhez tartozó rendre: 001010 és 000100.

A G-hez tartozó bittérképindexnek szintén három bitvektora lesz, mivel három különböző érték fordul elő benne. A három bitvektor:

Érték	*Vektor*
`foo`	100100
`bar`	010010
`baz`	001001

Mindegyik esetben az 1-esek mutatják, hogy melyik rekordban fordul elő a megfelelő szöveg. □

5.4.1. Indítékok a bittérképindexekhez

Először úgy tűnik, hogy a bittérképindexek túl sok helyet igényelnek, különösen, ha sok különböző érték tartozik egy mezőhöz, mivel a bitek száma összesen a rekordszám és az értékek számának szorzata. Például, ha a mező egy kulcs, és n rekordunk van, akkor n^2 bit szükséges az összes bitvektorhoz az adott mező esetén. Azonban tömörítést használva a bitek számát közelíthetjük n-hez, függetlenül a különböző értékek számától, ahogy azt az 5.4.2. részben majd fogjuk látni.

Szintén gyanítható, hogy gondok lesznek a bittérképindexek kezelésével is. Például kihasználják, hogy a rekordok száma ugyanaz marad egész idő alatt. Hogyan fogjuk megtalálni az i-edik rekordot, amikor az állományhoz hozzáadódnak illetve törlődnek rekordok? Hasonlóan, egy mezőérték feltűnhet vagy eltűnhet. Hogyan találjuk meg hatékonyan egy érték bittérképét? Ezeket és a kapcsolódó kérdéseket az 5.4.4. részben tárgyaljuk.

Viszont a bittérképindexek előnye az, hogy a lekérdezések részleges egyezéssel sok esetben nagyon hatékonyan válaszolhatók meg a segítségével. Bizonyos értelemben a kosarak előnyeit kínálják, amit a 4.16. példában tárgyaltunk, ahol megkaptuk a `Film` azon sorait, amelyekben néhány attribútum értéke adott volt anélkül, hogy először vissza kellett volna nyernünk minden egyes attribútumra az összes megfelelő rekordot. Egy példával illusztráljuk a lényeget.

5.21. példa: Visszautalunk a 4.16. példára, ahol a

```
Film(cím, év, hossz, gyártó)
```

relációt kérdeztük le a következő lekérdezéssel:

```
SELECT cím
FROM Film
WHERE gyártó = 'Disney' AND
      év = 1995;
```

Tegyük fel, hogy a `gyártó` és az `év` attribútumon is van bittérképindex. Akkor vehetjük az `év = 1995` és `gyártó = 'Disney'` vektorok metszetét, azaz a vektorok bitenkénti AND művelettel vett eredményét, ami azt a vektort eredményezi, amelyben az i-edik pozíción akkor és csak akkor van 1-es, ha az i-edik `Film` sor egy olyan filmhez tartozik, amit a Disney gyártott 1995-ben.

Ha vissza tudjuk nyerni a `Film` sorait a sorszámuk alapján, akkor csak azokat a blokkokat kell beolvasnunk, amelyek egy vagy több ilyen sort tartalmaznak, pontosan úgy, ahogy a 4.16. példában is tettük. Ahhoz, hogy a két bitvektor metszetét vegyük, be kell őket olvasni a memóriába, ami egy lemezműveletet igényel minden egyes blokkra, amelyet a két vektor egyike elfoglal. Mint említettük, később érintjük mindkét témát: az 5.4.4. részben a rekordok elérését a sorszámuk alapján, és az 5.4.2. részben annak biztosítását, hogy a bitvektorok ne foglaljanak el túl sok helyet. □

A bittérképindexek a tartománylekérdezések megválaszolását is segíthetik. A következőkben megnézünk egy példát, amely egyaránt mutatja azt, hogy hogyan használjuk őket tartománylekérdezéshez, valamint azt is részleteiben bemutatjuk rövid bitvektorokkal, hogy hogyan használhatók a bitvektorok bitenkénti AND és OR műveletei arra, hogy megtaláljuk a választ a lekérdezésünkre anélkül, hogy meg kellene néznünk egyetlen rekordot is azokon kívül, amelyekre szükségünk van.

5.22. példa: Tekintsük az aranyékszer-adatokat, amelyeket az 5.7. példában vezettünk be. Tegyük fel, hogy a példa tizenkét pontja az alábbi, 1-től 12-ig számozott rekord:

1: (25, 60)	2: (45, 60)	3: (50, 75)	4: (50, 100)
5: (50, 120)	6: (70, 110)	7: (85, 140)	8: (30, 260)
9: (25, 400)	10: (45, 350)	11: (50, 275)	12: (60, 260)

Az első elemre, az életkorra hét különböző érték van, így az életkor bittérképindexe a következő hét vektorból áll:

25: 100000001000	30: 000000010000	45: 010000000100
50: 001110000010	60: 000000000001	70: 000001000000
85: 000000100000		

A fizetés komponensre tíz különböző érték van, így a fizetés bittérképindexnek a következő tíz bitvektora van:

60: 110000000000	75: 001000000000	100: 000100000000
110: 000001000000	120: 000010000000	140: 000000100000
260: 000000010001	275: 000000000010	350: 000000000100
400: 000000001000		

Tegyük fel, hogy meg akarjuk találni azokat az ékszervásárlókat, akiknek életkora a 45–55, fizetése pedig a 100–200 tartományba esik. Először a tartományba eső életkor értékekhez tartozó bitvektorokat keressük meg, ebben a példában csak két ilyen van: 010000000100 és 001110000010 a 45, illetve az 55 értékhez tartozó. Ha vesszük a bitenkénti OR-művelet eredményét, akkor egy új bitvektorunk lesz, amelyben az i-edik helyen akkor és csak akkor van 1, ha az életkor a rekordban a kívánt tartományba esik. Ez a bitvektor a 011110000110.

Azután megkeressük azokat a bitvektorokat, amelyek a 100 és 200 ezer közé eső fizetésértékekhez tartoznak. Négy ilyen van, a megfelelő fizetésértékek: 100, 110, 120 és 140; a bitenkénti OR eredménye pedig: 000111100000.

Az utolsó lépés a bitenkénti AND értékét venni ennek a két vektornak, amiket az OR-művelettel számoltunk ki. Azaz:

011110000110 AND 000111100000 = 000110000000

Így arra jutottunk, hogy csak a negyedik és az ötödik rekord van a kívánt tartományban, amelyek az (50, 100) és az (50, 120) pontok. □

5.4.2. Tömörített bittérképek

Tegyük fel, hogy egy n rekordot tartalmazó állomány F mezőjén van egy bittérképindexünk, és m különböző érték fordul elő az állományban az F mezőben. Ekkor az index összes bitvektorában a bitek száma mn. Ha a blokkok mondjuk 4096 bájt hosszúak, akkor 32 768 bit fér egy blokkba, tehát a szükséges blokkok száma: $mn/32768$. Ez a szám lehet kicsi az egész állomány tárolásához szükséges blokkok számához viszonyítva, de minél nagyobb az m értéke, annál több helyet foglal le a bittérképindex.

Viszont, ha az m nagy, az 1-es a bitvektorokban nagyon ritka lesz, pontosabban annak valószínűsége, hogy egy tetszőleges bit 1-es: $1/m$. Ha az 1-es ritka, akkor lehetőségünk van úgy kódolni a bitvektorokat, hogy átlagosan sokkal kevesebb, mint n bitet tartalmazzanak. Egy szokásos módszer a *szakaszhosszkódolásnak* vagy sorkifejtő kódolásnak nevezett, ahol egy szakaszt – ami i darab egymás utáni 0 bitből, majd egy ezeket követő 1-esből áll – az i egész szám valamilyen megfelelő bináris kódjával ábrázolunk. Majd egymás után rakjuk a kódokat az összes szakaszra, és az így kapott bitsorozat a bitvektor kódolt változata.

Elképzelhető, hogy az i egészet egyszerűen úgy ábrázoljuk, hogy bináris számként írjuk fel. Azonban ez az egyszerű szerkezet nem mindig megfelelő, mert nem lehet a kódok sorozatát a benne foglalt szakaszok hosszának egyértelmű meghatározásával szétszedni (lásd a „A bináris számok nem megfelelőek a szakaszhosszkódoláshoz" című bekeretezett részt). Így az i egész szám kódja, ami a szakasz hosszát mutatja, bonyolultabb kell legyen, mint az egyszerű bináris ábrázolás.

A sok lehetséges kódolási szerkezet közül egyet fogunk használni. Létezik jobb, bonyolultabb szerkezet, ami az itt elért tömörítés mértékét majdnem kétszeresére tudja javítani, de csak akkor, ha a jellemző szakaszhossz nagyon nagy. A szerkezetünknél először meg kell határoznunk, hogy az i binárisan ábrázolva hány bitből áll. Ez a j szám, ami közelítőleg $\log_2 i$, unárisan ábrázolva $j - 1$ darab 1-esből és egy 0-sból áll. Aztán folytathatjuk az i bináris értékével.[3]

5.23. példa: Ha $i = 13$, akkor $j = 4$, azaz 4 bit kell az i bináris ábrázolásához. Így az i kódoltan 1110-val kezdődik. Ezt követi az i binárisan, vagyis 1101. Tehát a 13 kódolva 11101101.

Az $i = 1$ kódolva 01, és az $i = 0$ kódolva 00. Mindkét esetben $j = 1$, tehát egyetlen 0 a kezdet, és ezt a 0-t követi az i-t ábrázoló egy bit. □

Ha egymás mögé rakjuk a kódolt egészek sorozatait, mindig vissza tudjuk állítani a szakaszhosszak sorozatát, és ezért a eredeti bitvektor visszaállítható. Tegyük fel, hogy átnéztünk már valahány kódolt bitet, és most egy bitsorozat elején vagyunk, amely egy bizonyos i egész szám kódja. Továbbmegyünk az első 0-ig, így meghatározzuk a j értékét. Azaz, j egyenlő azon bitek számával, amennyit le kell olvasnunk, amíg elérünk az első 0-hoz (beleértve ezt a 0-t is a bitek számába). Ha már ismerjük a j érté-

[3] Ténylegesen, a $j = 1$ esetet leszámítva (azaz, ha $i = 0$, vagy $i = 1$), biztosak lehetünk abban, hogy az i kettes számrendszerben felírva 1-gyel kezdődik. Így számonként megspórolhatunk 1 bitet, ha ezt az 1-est elhagyjuk, és csak a maradék $j - 1$ bitet használjuk.

> **A bináris számok nem megfelelőek a szakaszhosszkódoláshoz**
>
> Tegyük fel, hogy az i darab 0-ból, és utána egy 1-esből álló szakaszhoz az i egész szám bináris értékét rendeljük. Akkor a 000101 bitvektor két szakaszból áll, amelyeknek a hossza 3, illetve 1. Ezek az egészek binárisan ábrázolva 11 és 1, tehát a 000101 szakaszhosszkódolásának eredménye 111. Azonban, hasonló számítás mutatja, hogy a 010001 bitvektor kódja szintén 111, és a 010101 a harmadik olyan, amelynek a kódja szintén 111. Így a 111 nem dekódolható egyértelműen bitvektorrá.

két, akkor vegyük a következő j bitet, ez adja kettes számrendszerben ábrázolva az i egész számot. Továbbá, ha végignéztük az i-t ábrázoló biteket, akkor tudjuk, hol van a következő egész kódjának a kezdete, így meg tudjuk ismételni az eljárást.

5.24. példa: Fejtsük vissza a 11101101001011 sorozatot. Az elején kezdve, a 4-edik biten találjuk az első 0-t, tehát $j = 4$. A következő 4 bit: 1101, tehát megállapíthatjuk, hogy az első egész a 13. A 001011 maradt, amit vissza kell fejtenünk.

Mivel az első bit 0, tudjuk, hogy a következő bit magát az egészet ábrázolja, ez a szám a 0. Így eddig a 13, 0 sorozatot fejtettük vissza, és a maradék visszafejtendő sorozat a 1011.

Az első 0-t a második pozíción találjuk, amiből következik, hogy az utolsó két bit ábrázolja az utolsó egészet, ami 3. A szakaszhosszak teljes sorozata tehát 13, 0, 3. Ezekből a számokból felépíthetjük a tényleges bitvektort: 0000000000000110001. □

Gyakorlatilag minden bitvektor, amit így dekódolunk, 1-essel végződik, és a záró 0-kat nem állítjuk vissza. Mivel feltételezhetően ismerjük az állományban lévő rekordok számát, a további 0-kat hozzá tudjuk adni. Azonban, mivel a 0 egy bitvektorban azt jelenti, hogy a megfelelő rekord nincs benne a kívánt halmazban, nem is kell tudnunk a rekordok teljes számát, és figyelmen kívül hagyhatjuk a záró 0-kat.

5.25. példa: Alakítsunk át néhány, az 5.22. példában szereplő bitvektort a mi szakaszhosszkódunkra. Az első három (25, 30, 45) életkorhoz tartozó vektorok: 100000001000, 000000010000, illetve 010000000100. Ezek közül az elsőhöz a (0, 7) szakaszhosszsorozat tartozik. A 0 kódja 00, a 7 kódja 110111. Így a 25 éves életkor bitvektora a 00110111 sorozattá alakul.

Hasonlóan, a 30 éves életkornak csak egy szakasza van, hét 0-val. Tehát ennek a kódja: 110111. A 45 éves kor bitvektorának két szakasza van (1, 7). Mivel az 1 kódja 01, és mint meghatároztuk, a 7 kódja 110111, a harmadik bitvektor kódja: 01110111. □

A tömörítés az 5.25. példában nem nagy. Azonban nem láthatjuk az igazi előnyöket, ha n, a rekordok száma kicsi. Hogy méltányolni tudjuk a kódolás értékét, tegyük

fel, hogy $m = n$, vagyis a mezőnek, amelyen a bittérképindexet létrehozzuk, minden értéke egyedi. Megjegyezzük, hogy egy i hosszú szakasz kódja körülbelül $2 \log_2 i$ bit. Mindegyik bitvektorban egyetlen 1-es van, tehát egyetlen szakaszból áll, és annak a szakasznak a hossza nem nagyobb, mint n. Így a $2 \log_2 n$ bit egy felső korlát ebben az esetben a bitvektor kódjának hosszára.

Mivel n bitvektor van az indexben (mert $m = n$), az indexet alkotó bitek teljes száma legfeljebb $2n \log_2 n$. Megjegyezzük, hogy kódolás nélkül n^2 bit kellett volna. Ha $n > 4$, akkor $2n \log_2 n < n^2$, és ahogy n nő a $2n \log_2 n$ tetszőlegesen kisebb lesz, mint n^2.

5.4.3. Műveletek szakaszhosszkódolt bitvektorokon

Ha bitenkénti AND vagy OR műveleteket kell végrehajtanunk kódolt bitvektorokon, nemigen van más választásunk, mint visszafejteni őket, és a műveletet az eredeti bitvektorokon hajtani végre. Azonban nem kell az egész dekódolást egyszerre végrehajtani. A tömörítési szerkezet, amit leírtunk, lehetővé teszi, hogy csak egy szakaszt fejtsünk vissza egyszerre, és így meg tudjuk határozni, hogy hol a következő 1-es mindegyik, a műveletben résztvevő bitvektorban. Ha OR a művelet, az eredménybe is 1-est teszünk azon a pozíción, ha AND a művelet, akkor és csak akkor teszünk 1-est, ha mindkét operandusban a következő 1-es ugyanazon a pozíción van. A leírt algoritmus bonyolult, de egy példa kellően világossá teheti.

5.26. példa: Tekintsük az 5.25. példában a 25 és 30 életkorra kapott kódolt bitvektorokat: 00110111, illetve 110111. Az első szakaszukat könnyen dekódolhatjuk; azt kapjuk, hogy 0, illetve 7. Azaz az első 1-es a 25-höz tartozó bitvektorban az első pozíción fordul elő, míg a 30 esetén a bitvektorban az első 1-es a nyolcadik helyen van. Így egy 1-est képezünk az első pozícióra.

Aztán dekódolnunk kell a 25 életkorhoz tartozó következő szakaszt, mivel az a bitvektor adhat még egyest a 8-as pozíció előtt, ahol a 30-hoz tartozó bitvektorban egyes van. Azonban a 25-ös életkor következő szakasza 7, ami azt jelenti, hogy a bitvektorban a következő 1-es a 9-dik helyen van. Tehát hat 0-t helyezünk el és egy 1-est a 8-dik pozícióra, ami a 30-hoz tartozó bitvektorból jön. Ez a bitvektor nem járul hozzá több 1-essel az eredményhez. Az egyes a 9-dik pozícióra a 25-höz tartozó bitvektor alapján kerül, és ez a bitvektor sem ad további 1-eseket.

Arra jutottunk, hogy a két bitvektor OR műveletének eredménye 100000011. Az eredeti bitvektorok hosszát nézve, ami 12, láthatjuk, hogy ez nincs teljesen rendben, ugyanis három záró 0 bit lemaradt. Ha tudjuk, hogy az állományban a rekordok száma 12, hozzáadhatjuk azokat a 0-kat a végéhez. Azonban érdektelen, hogy hozzáragasztjuk-e azokat a 0-kat, mert csak 1-es bit esetén kell beolvasnunk rekordot. Ebben a példában nem fogjuk beolvasni a 10 és 12 közötti rekordokat semmiképpen. □

5.4.4. Bittérképindexek kezelése

Leírtuk a bittérképindexek műveleteit anélkül, hogy három fontos kérdést említettünk volna:

1. Amikor keresünk egy bitvektort egy adott értékhez, vagy bitvektorokat, amik megfelelnek egy tartományba eső értékeknek, hogyan tehetjük ezt hatékonyan?
2. Ha kiválasztottunk egy rekordhalmazt, ami válasz a kérdésünkre, hogyan nyerhetjük vissza azokat a rekordokat hatékonyan?
3. Ha rekordok beszúrása vagy törlése megváltoztatja az adatállományt, hogyan igazítjuk hozzá a változásokhoz egy adott mező bittérképindexét?

Bitvektorok keresése

Az első kérdés megválaszolható olyan technikák alapján, amiket már tanultunk. Gondoljunk úgy a bitvektorokra, mint rekordokra, amelyeknek a kulcsa a bitvektornak megfelelő érték (bár maga az érték nincs benne a „rekordban"). Ekkor bármilyen másodlagos indexelési technika hatékonyan támogatja az értékekhez tartozó bitvektorok elérését. Például, ha B-fát használunk, amelynek levelei kulcs-mutató párokat tartalmaznak; a mutató a kulcsértékhez tartozó bitvektorhoz vezet. A B-fa gyakran jó választás, mert könnyen támogatja a tartománylekérdezéseket, de a tördelőtáblák vagy az indexszekvenciális állományok szintén választhatók.

A bitvektorokat szintén tárolni kell valahol. Legjobb úgy elképzelni őket, mint változó hosszú rekordokat, mivel általában nőni fognak, ahogy egyre több rekordot adunk az adatállományhoz. Ha a bitvektorok – esetleg tömörített formában – jellemzően rövidebbek, mint egy blokk, akkor megfontolandó, hogy többet rakjunk egy blokkba, és átrendezzük őket, ha szükséges. Ha a bitvektorok tipikusan hosszabbak egy blokknál, akkor meggondolandó, hogy blokkok láncában tároljuk egyesével őket. A 3.4. rész technikái hasznosak lehetnek.

Rekordok keresése

Most gondoljuk át a második kérdést: ha egyszer meghatároztuk, hogy szükségünk van az adatállomány k-adik rekordjára, hogyan találjuk meg? Ismét csak alkalmazhatók a már ismert technikák. Képzeljük el a k-adik rekordot úgy, mint aminek k a keresési kulcsa (bár ez a kulcs nincsen benne a rekordban). Ekkor létrehozhatunk egy másodlagos indexet az adatállományhoz, aminek a keresési kulcsa a rekordszám.

Ha nincs rá ok, hogy az állományt valami más módon szervezzük, a rekordsorszámot akár az elsődleges index keresési kulcsaként is használhatjuk, ahogy a 4.1. részben tárgyaltuk. Ekkor az állomány szervezése különösen egyszerű, mivel a rekordsorszámok sosem változnak (még törléskor sem), és az új rekordokat csak az adatállomány végéhez kell adnunk. Így az adatállomány blokkjait teljesen tele lehet rakni

ahelyett, hogy az állomány közepén a beszúrások számára külön helyet kellene hagyni, mint ahogy szükséges volt a 4.6. részben, az indexszekvenciális állományok általános esetén él.

Adatállományok módosításának kezelése

Két szempontból jelentenek problémát az adatállomány-módosítások a bittérképindexre:

1. A rekordsorszámok nem változhatnak, ha egyszer kiosztották azokat.
2. Az adatállomány változásai szükségessé teszik a bittérképindex változtatását is.

Az 1. pont következménye, hogy ha töröljük az i rekordot, a legegyszerűbb „nyugdíjazni" a számát, a helyére pedig egy „sírkövet" tenni az adatállományban. A bittérképindexet szintén változtatni kell, mivel a bitvektorban, amelyben 1-es van az i-edik helyen, az 1-est 0-ra kell cserélni. Megjegyezzük, hogy meg tudjuk találni a megfelelő bitvektort, mert tudjuk, hogy milyen érték volt az i-edik rekordban a törlés előtt.

Következőként tekintsük át az új rekord beszúrását. Tárolhatjuk a következő lehetséges rekordszámot, és ezt rendeljük hozzá az új rekordhoz. Azután minden bittérképindexhez meg kell határoznunk az új rekord hozzá tartozó mezőjében lévő értéket, és az ahhoz az értékhez tartozó bitvektort módosítani kell úgy, hogy a végéhez hozzátegyünk egy 1-est. Gyakorlatilag ennek az indexnek az összes többi bitvektora kap egy 0-t a végére, de ha olyan tömörítési eljárást használunk, mint az 5.4.2. részben, akkor nem szükséges a tömörített értékeket változtatni.

Speciális eset, ha az új rekord olyan értéket tartalmaz az indexelt mezőben, ami még nem fordult elő. Ez esetben ehhez az értékhez szükségünk van egy új bitvektorra, és ezt a bitvektort és a megfelelő értéket be kell szúrni abba a másodlagos indexstruktúrába, amit egy adott értékhez tartozó bitvektor visszakereséséhez használunk.

Végül nézzük az adatállomány i-edik rekordjának a módosítását, amikor egy olyan mező változik mondjuk v értékről w-re, amelyhez van bittérképindex. Meg kell találnunk a v érték bitvektorát, és az i-edik pozíción az 1-est 0-ra kell váltani. Ha van bitvektor a w-hez, akkor a 0-t az i-edik helyen 1-re kell változtatni. Ha még nincs bitvektor a w-hez, akkor létrehozunk egyet, ahogy az előző bekezdésben leírtuk azt az esetet, amikor a beszúrás egy új értéket eredményez.

5.4.5. Feladatok

5.4.1. feladat: Az 5.10. ábra adataival mutassuk meg a bittérképindexeket a következő attribútumokra:

* a) sebesség,
b) memória és
c) merevlemez,

mind i) nem tömörített alakban, mind ii) tömörített alakban, az 5.4.2. rész szerkezetét használva.

5.4.2. feladat: Az 5.22. példa bittérképeit használva keressük meg azokat az ékszervásárlókat, akiknek az életkora a 25–40 tartományban van, és a fizetésük 0 és 100 közötti.

5.4.3. feladat: Tekintsünk egy 1 000 000 rekordot tartalmazó állományt, amely az F mezőjében m különböző értéket tartalmaz.

a) Hány bájtos az F mező bittérképindexe az m függvényében?
! b) Tegyük fel, hogy az 1-től 1 000 000-ig számozott rekordokban az F mező értékei round-robin módon adottak, így minden egyes érték előfordul bármely m egymás utáni rekordban. Hány bájtot használna fel egy tömörített index?

!! **5.4.4. feladat:** Az 5.4.2. részben említettük, hogy csökkenteni lehetne a bitek számát $2 \log_2 i$-ről – amennyit abban a részben használtunk az i szám kódolásához –, közel $\log_2 i$-ig. Mutassuk meg, hogyan lehet tetszőlegesen megközelíteni ezt a határt, amennyiben az i elég nagy. *Tanács:* Az i bináris kódjának hosszát unárisan kódoltuk. Tudnánk-e a kód hosszát binárisan kódolni?

5.4.5. feladat: Kódoljuk a következő bittérképeket az 5.4.2. részben használt szerkezetet használva:

* a) 0110000000100000100.
b) 1000001000000100110 1.
c) 0001000000000001000001000 0.

*! **5.4.6. feladat:** Rámutattunk, hogy a tömörített bittérképindexek közelítőleg $2n \log_2 n$ bitet használnak fel egy n rekordos állomány esetén. Hasonlítsuk össze ezt a bitmennyiséget egy B-fa-index által felhasznált bitek számával. Emlékezzünk rá, hogy a B-fa-index mérete függ a kulcs és a mutató hosszától, és (bizonyos mértékig) a blokk méretétől is. Használhatunk azonban ésszerű becsléseket ezekre a paraméterekre a számításainkban. Miért részesítjük esetleg előnyben a B-fákat, még akkor is, ha több helyet foglalnak le, mint a tömörített bittérképek?

5.5. Összefoglalás

- *Többdimenziós adatok:* Sok alkalmazás, mint például a térképészeti adatbázisok vagy az eladási és készletnyilvántartó adatok, felfoghatók úgy, mint pontok egy két- vagy többdimenziós térben.
- *Többdimenziós indexeket igénylő lekérdezések:* Azok a lekérdezéstípusok, amelyeket a többdimenziós adatokon támogatni kell, többek között: a lekérdezések részle-

ges egyezéssel (a dimenziók egy részhalmazára megadott értékeknek megfelelő pontok), a tartománylekérdezések (az egyes dimenziókra megadott tartományokon belüli pontok), a legközelebbi szomszéd (egy adott ponthoz legközelebbi pont), és a „hol-vagyok-én" (régió vagy régiók, amelyek egy adott pontot tartalmaznak).

- *Legközelebbi szomszéd-lekérdezések végrehajtása:* Sok adatstruktúra lehetővé teszi a legközelebbi szomszéd-lekérdezések végrehajtását úgy, hogy végrehajtunk egy tartománylekérdezést a kiszemelt pont körül, és megnöveljük a tartományt, ha nincs pont abban a tartományban. Óvatosnak kell lenni, mert hiába találunk pontot egy téglalap alakú tartományban, az nem feltétlenül zárja ki annak a lehetőségét, hogy van közelebbi pont a téglalapon kívül.
- *Rácsos állományok:* A rácsos állomány felszeleteli a pontok terét minden egyes dimenzió mentén. A rácsvonalak távolsága lehet különböző, és dimenziónként lehet különböző számú rácsvonal. A rácsos állományok jól támogatják a tartománylekérdezéseket, a lekérdezéseket részleges egyezéssel, és a legközelebbi szomszéd-lekérdezéseket, legalábbis, ha az adatok viszonylag egyenletes eloszlásúak.
- *Particionált tördelőtáblák:* Egy particionált tördelőfüggvény minden egyes dimenzióból a kosár számának néhány bitjét képezi. A részleges egyezéses lekérdezéseket jól támogatják, és hatékonyságuk nem függ az adatok egyenletes eloszlásától.
- *Többkulcsos indexek:* Egy egyszerű többdimenziós struktúra, amelynek van egy gyökere, ami index az egyik attribútumon, és ez elvezet egy második attribútumon lévő indexek gyűjteményéhez, amik egy harmadik attribútum indexeihez vezethetnek és így tovább. A tartomány és a legközelebbi szomszéd-lekérdezésekhez hasznosak.
- *kd-fák:* Ezek a fák olyanok, mint a bináris keresőfák, de a különböző szinteken különböző attribútumok szerint ágaznak el. A részleges egyezéses, a tartomány és a legközelebbi szomszéd-lekérdezéseket egyaránt jól támogatják. A facsomópontok meggondolt blokkokba csomagolása szükséges ahhoz, hogy alkalmassá tegyük a másodlagos tárolókon végzett műveletekhez.
- *Quad-fák:* A quad-fa a többdimenziós kockát „negyedekre" osztja, és rekurzívan tovább osztja a „negyedeket" ugyanolyan módon, ha túl sok pont van bennük. A részleges egyezéses, a tartomány és a legközelebbi szomszéd-lekérdezéseket támogatják.
- *R-fák:* Az ilyen fajta fa általában régiók gyűjteményét ábrázolja, nagyobb régiók hierarchiájába csoportosítva azokat. Segíti a „hol-vagyok-én" lekérdezéseket, és ha az elemi régiók ténylegesen pontok, más – ebben a fejezetben tanulmányozott – lekérdezéseket is támogat.
- *Bittérképindexek:* A többdimenziós lekérdezéseket egy olyan index támogatja, amely a pontokat vagy a rekordokat rendezi, és bitvektorokkal jelöli azoknak a rekordoknak a helyét, amelyeknek egy adott értéke van a megfelelő attribútumon. Ezek az indexek a tartomány-, legközelebbi szomszéd- és a részleges egyezéses lekérdezéseket támogatják.
- *Tömörített bittérképek:* Azért, hogy helyet takarítsunk meg, a bittérképindexeket – amelyek gyakran nagyon kevés 1-est tartalmazó vektorokból állnak – tömörítjük a szakaszhosszkódolást használva.

5.6. Irodalomjegyzék

A legtöbb ebben a fejezetben tárgyalt adatstruktúra az 1970-es években és az 1980-as évek elején végzett kutatások terméke. A *kd*-fákat [2] vezeti be. A másodlagos tárolókra megfelelő módosítások [3]-ban és [13]-ban jelennek meg. A particionált tördelést és annak használatát részleges egyezéses lekérdezésekre [12] és [5] tárgyalja. Viszont az 5.2.8. feladat tervezési elképzelése [14]-ből származik.

A rácsos állományok [9]-ben jelennek meg. A quad-fák [6]-ban találhatók meg. Az R-fákat [8] vezeti be, és két jól ismert kiterjesztése [15] és [1].

A bittérképindexnek érdekes a története. Egy Nucleus nevű cég, amelyet Ted Glaser alapított, szabadalmaztatta az ötletét és kifejlesztett egy adatbázis-kezelő rendszert, amelyikben az indexstruktúra és az adatábrázolás is bittérképindex volt. A vállalkozás az 1980-as évek végén csődbe ment, de az ötletet napjainkban is beépítették több nagy kereskedelmi adatbázisrendszerbe. Az első publikáció ebben a témában [10]. Az ötlet legújabb kiterjesztése [11].

A többdimenziós tárolási struktúráknak számos összegzése van. Az egyik legkorábbi [4]. A legújabb áttekintések [16]-ban és [7]-ben találhatók. Az előbbi több, egyéb jelentős adatbázis témakörről szóló tanulmányt is tartalmaz.

1. N. Beekmann, H.–P. Kriegel, R. Schneider, and B. Seeger, „The R*-tree: an efficient and robust access method for points and rectangles," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1990), pp. 322–331.
2. J. L. Bentley, „Multidimensional binary search trees used for associative searching," *Comm. ACM* **18**:9 (1975), pp. 509–517.
3. J. L. Bentley, „Multidimensional binary search trees in database applications," *IEEE Trans. on Software Engineering* **SE-5**:4 (1979), pp. 333–340.
4. J. L. Bentley and J. H. Friedman, „Data structures for range searching," *Computing Surveys* **13**:3 (1979), pp. 397–409.
5. W. A. Burkhard, „Hashing and trie algorithms for partial match retrieval," *ACM Trans. on Database Systems* **1**:2 (1976), pp. 175–187.
6. R. A. Finkel and J. L. Bentley, „Quad trees, a data structure for retrieval on composite keys," *Acta Informatica* **4**:1 (1974), pp. 1–9.
7. V. Gaede and O. Gunther, „Multidimensional access methods," *Computing Surveys* **30**:2 (1998), pp. 170–231.
8. A. Guttman, „R-trees: a dynamic index structure for spatial searching," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1984), pp. 47–57.
9. J. Nievergelt, H. Hinterberger, and K. Sevcik, „The grid file: an adaptable, symmetric, multikey file structure," *ACM Trans. on Database Systems* **9**:1 (1984), pp. 38–71.
10. P. O'Neil, „Model 204 architecture and performance," *Proc. Second Intl. Workshop on High Performance Transaction Systems*, Springer-Verlag, Berlin, 1987.
11. P. O'Neil and D. Quass, „Improved query performance with variant indexes," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1997), pp. 38–49.