```
1) WITH
2)   RECURSIVE P(x) AS
3)       (SELECT * FROM R)
4)     UNION
5)       (SELECT * FROM Q)

6)   RECURSIVE Q(x) AS
7)     (SELECT SUM(x) FROM P)

8) SELECT * FROM P;
```

5.26. ábra. *Olyan nem rétegzett lekérdezésösszesítéssel, mely szabálytalan az SQL3-ban*

1. P a Q és egy R EDB reláció egyesítése.

2. Q-nak egy sora van, mely a P elemeinek összesítését tartalmazza.

Ezeket a feltételeket kifejezhetjük egy `WITH` utasítással, azonban ez az utasítás megszegi az SQL3 monotonitásra vonatkozó feltételeit. Az 5.26. ábra lekérdezése P értékét keresi.

Tételezzük fel, hogy eredetileg R a 12) és a 34) sorokból áll, P és Q üres, ahogy azt a fixpontszámításban is feltételeztük. Az 5.27. ábra bemutatja az első hat lépésben kiszámított értékeket. Ne feledjük, hogy a relációk új értékeit az összes reláció előző lépésbeli értékei alapján számítjuk ki. Így az első lépés után P egyezni fog az R-rel, míg Q üres lesz, mivel a kiszámításához P régi értékét használtuk.

A második lépésben a 3)–5) sorok közötti rész eredménye az $R = \{(12), (34)\}$ halmaz lesz, tehát ez lesz P új értéke, mely megegyezik a régivel. Q új értéke a $\{(46)\}$ halmaz lesz, mivel $12 + 34 = 46$.

A harmadik lépésben $P = \{(12), (34), (46)\}$ és $Q = \{(46)\}$.

A negyedik lépésben P értéke ugyanaz, de Q értéke megváltozik $\{(92)\}$-re, mert $12 + 34 + 46 = 92$. Figyeljük meg, hogy Q elvesztette a 46) sort, és egy új sort kapott helyette. Tehát a 46) sor beszúrása P-be egy sor kitörlését eredményezte Q-ból (véletlenül ugyanannak a sornak a kitörlését). Ez ellenkezik a monotonitási szabállyal, amit az SQL3 előír, tehát az 5.26. ábra lekérdezése szabálytalan. Általában a $2i$-edik lépésben, $P = \{(12), (34), (46i - 46)\}$, míg Q értéke $(46i)$. □

Lépés	P	Q
1)	{(12), (34)}	∅
2)	{(12), (34)}	{(46)}
3)	{(12), (34), (46)}	{(46)}
4)	{(12), (34), (46)}	{(92)}
5)	{(12), (34), (92)}	{(92)}
6)	{(12), (34), (92)}	{(138)}

5.27. ábra. *Iteratív fixpontszámítás nem monoton összesítés esetén*

Új értékek használata fixpont számításában

Elgondolkozhatnánk azon, vajon miért használtuk P régi értékeit Q új értékeinek kiszámításában az 5.57. és 5.58. példában. Ha P új értékeit használtuk volna, akkor az eredmény függene a relációk `WITH` záradékbeli felsorolási sorrendjétől. Az 5.57. példában P és Q az egyik lehetséges fixponthoz konvergálna, a kiértékelési sorrendtől függően. Az 5.58. példában P és Q még mindig nem konvergálna, és tulajdonképpen minden lépésben megváltoznának.

5.10.6. Feladatok

5.10.1. feladat: A 4.36. példában használtuk a következő relációt:

```
Folytatások(film, folytatás)
```

mely egy film azonnali folytatásait adja meg. Definiáltuk még a `Sorozat` IDB relációt is, melynek elemei olyan (x, y) sorok, hogy y vagy x azonnali folytatása, vagy egy folytatás folytatása stb.

a) Adjuk meg `Sorozat` definícióját egy rekurzív SQL3 kifejezés segítségével.

b) Adjunk meg egy olyan rekurzív SQL3 lekérdezést, mely azokat az (x, y) filmpárokat eredményezi, melyekre y egy nem direkt folytatása x-nek.

c) Adjunk meg egy olyan rekurzív SQL3 lekérdezést, mely azokat az (x, y) filmpárokat eredményezi, melyekre y folytatása x-nek, de nem direkt folytatása és nem egy direkt folytatás folytatása.

! d) Adjunk meg egy olyan rekurzív SQL lekérdezést, mely azokat az x filmeket eredményezi, melyeknek legalább két folytatása volt. Mindkét folytatás lehet direkt.

! e) Adjunk meg egy rekurzív SQL3 lekérdezést, mely azokat az (x, y) filmpárokat eredményezi, melyekre y folytatása x-nek, és y-nak legfeljebb egy folytatása van.

5.10.2. feladat: A 4.4.3. feladatban bevezettük a következő relációt:

```
Kapcsolatok(osztály, kosztály, mult)
```

amely leírja azt, hogy egy ODL osztály hogyan viszonyul más osztályokhoz. Általában a relációnak van egy (c, d, m) sora, ha létezik egy kapcsolat a c osztály és a d osztály között. Ez a kapcsolat többértékű, ha $m =$ `'multi'` és egyértékű, ha

m = `'single'`. A 4.4.3. feladatban azt is ismertettük, hogy a `Kapcsolatok` relációt tekinthetjük úgy, mint egy gráfot, melynek a csúcsai osztályok és a c osztálytól a d osztályig akkor és csakis akkor létezik egy m címkéjű él, ha (c, d, m) egy él a `Kapcsolatok`-ban.

a) Adjunk meg egy rekurzív SQL3 lekérdezést mely azokat a (c, d) párokat adja meg, melyekre létezik egy útvonal c-től d-ig.

* b) Adjunk meg egy rekurzív SQL3 lekérdezést, mely azokat a (c, d) párokat adja meg, melyekre létezik egy útvonal c-től d-ig, mely végig egyértékű kapcsolatokból áll.

*! c) Adjunk meg egy rekurzív SQL3 lekérdezést, mely azokat a (c, d) párokat adja meg, melyekre létezik egy útvonal c-től d-ig, melyben legalább az egyik él többértékű kapcsolatot jelképez.

d) Adjunk meg egy rekurzív SQL3 lekérdezést, mely azokat a (c, d) párokat adja meg, melyekre létezik egy útvonal c-től d-ig, de nem létezik köztük olyan útvonal, mely végig egyértékű kapcsolatokból áll.

! e) Adjunk meg egy rekurzív SQL3 lekérdezést, mely azokat a (c, d) párokat adja meg, melyekre létezik egy útvonal c-től d-ig, mely váltakozóan egyértékű és többértékű kapcsolatokat jelképező élekből áll.

f) Adjunk meg egy rekurzív SQL3 lekérdezést, mely azokat a (c, d) párokat adja meg, melyekre létezik egy útvonal c-től d-ig és d-től c-ig, mely végig egyértékű kapcsolatokból áll.

*! **5.10.3. feladat:** Tételezzük fel, hogy az 5.23. ábra `Eljut` relációt meghatározó lekérdezését módosítanánk úgy, hogy a rekurzió legyen nemlineáris. Pontosabban, a 6)–8) sorokat kicserélnénk a következőkre:

```
6) (SELECT Első.honnan, Második.hova
7)  FROM Eljut AS Első, Eljut AS Második
8)  WHERE Első.hova = Második.honnan)
```

amelyben az `Eljut` reláció önmagával van összekapcsolva. Az i-edik lépésben milyen hosszúak az `Eljut` relációba beszúrt új utak?

5.11. Összefoglalás

- *SQL:* Az SQL a legfontosabb lekérdezőnyelve az adatbázis-kezelő rendszereknek. 1997-ben a legnagyobb hatással a kereskedelmi rendszerekre az SQL2 szabvány volt. Jelenleg az SQL3 van előkészítés alatt.
- *Select-From-Where lekérdezések:* Az SQL lekérdezések legáltalánosabb alakja select-from-where alakú. Segítségével több reláció szorzatára (a `FROM` záradékkal) alkalmazhatunk egy feltételt (`WHERE` záradék), és levetíthetjük a kívánt komponensekre (`SELECT` záradék).
- *Alkérdések:* A select-from-where lekérdezéseket alkérdésként is használhatjuk egy másik lekérdezés `WHERE` záradékában. Az `EXISTS`, `IN`, `ALL` és `ANY` operátorok logikai értékű feltételeket fejezhetnek ki az alkérdések eredményeként keletkezett relációkkal kapcsolatban.
- *Halmazműveletek relációkon:* Elvégezhetjük relációknak vagy relációkat definiáló lekérdezéseknek egyesítését, metszetét és különbségét a `UNION`, `INTERSECT` és `EXCEPT` kulcsszavak használatával.
- *A relációk multihalmaz modellje:* Az SQL a relációkat multihalmazoknak és nem halmazoknak tekinti. Az ismétlődéseket megszüntethetjük a `DISTINCT` kulcsszó segítségével, míg az `ALL` kulcsszó biztosítja, hogy az ismétlődések ne legyenek kitörölve olyan esetben, amikor az lenne az alapértelmezett.
- *Összesítések:* A reláció egy oszlopában megjelenő értékeket összesíthetjük a `SUM`, `AVG`, `MIN`, `MAX` és `COUNT` kulcsszavak segítségével. A sorokat az összesítés előtt csoportosíthatjuk a `GROUP BY` kulcsszavakkal. Bizonyos csoportokat kitörölhetünk az eredményből a `HAVING` kulcsszó segítségével.
- *Változtatási utasítások:* Az SQL biztosítja a lehetőséget a sorok módosítására egy relációban. Beszúrhatunk sorokat az `INSERT`, törölhetünk a `DELETE` és módosíthatunk az `UPDATE` kulcsszavak segítségével.
- *Adatdefiníció:* Az SQL biztosít utasításokat az adatbázis sémájának definíciójára. A `CREATE TABLE` utasítás segítségével tárolt relációkat (táblákat) definiálhatunk, megadva az attribútumait és az attribútumok típusát. A `CREATE DOMAIN` utasítást használhatjuk értéktartományok definiálására, melyeket aztán relációsémák definiálásában használhatunk. A `CREATE` utasítások segítségével definiálhatjuk az attribútumok alapértelmezett értékét is.
- *Sémák módosítása:* Egy adatbázissémát módosíthatunk az `ALTER` utasítással. A módosítás tartalmazhat attribútumtörlést vagy új attribútum definiálását, és az

alapértelmezett érték módosítását. A `DROP` utasítással megszüntethetünk relációkat, értelmezési tartományokat vagy más sémaelemeket.

- *Index:* Az SQL szabvány ugyan nem tartalmazza, de a kereskedelmi rendszerek általában biztosítják az indexek definiálásának lehetőségét, mely meggyorsít egyes lekérdezéseket és módosításokat, melyek keresési feltételében egy adott mezőre adott érték szerepel.

- *Nézettáblák:* A nézettáblán keresztül felépíthető egy reláció (a nézettábla) az adatbázisban tárolt táblákból. A nézettáblák ugyanúgy lekérdezhetők, mint a tárolt táblák, de az SQL átalakítja a lekérdezést úgy, hogy az a nézettábla definiálásánál használt tárolt táblákon fog lefutni.

- *Nullértékek:* Az SQL biztosít egy speciális értéket, a `NULL`-t, mely olyan komponensek helyén szerepel a relációkban, melyeknek nem tudjuk a konkrét értékét. A `NULL`-ra vonatkozó aritmetikai és logikai műveletek különböznek a szokásostól. Egy `NULL` érték összehasonlítása bármely értékkel, legyen az akár maga a `NULL` is, `ISMERETLEN` logikai értéket eredményez. Ez a logikai érték úgy viselkedik a logikai kifejezésekben, mintha az `IGAZ` és a `HAMIS` között helyezkedne el.

- *Összekapcsolási kifejezések:* Az SQL-ben megtalálhatóak olyan kifejezések mint a `NATURAL JOIN`, melyeket relációkra vagy lekérdezésekben vagy relációdefiniálásokban egy `FROM` záradékban lehet alkalmazni.

- *Külső összekapcsolások:* Az `OUTER JOIN` operátor összekapcsolja a relációkat de az eredménybe beszúrja azokat a sorokat is, melyek nem kapcsolódnak össze a másik relációból egy sorral sem; ezeknek a soroknak az ismeretlen komponensei `NULL` értékekkel töltődnek ki.

- *SQL3 rekurzió:* Az SQL3 szabvány leírja az ideiglenes relációk rekurzív definiálásának módját és azt, hogyan lehet ezeket a relációkat később lekérdezésekben használni. A szabvány azt javasolja, hogy a negáció és az összesítések, melyek rekurzióban jelennek meg legyenek rétegzettek; azaz egy rekurzívan definiált relációt ne lehessen definiálni saját maga negáltja vagy összesítése segítségével.

5.12. Irodalomjegyzék

Az SQL2 és SQL3 szabványokhoz az National Institute of Science and Technology (Amerikai Tudományos és Műszaki Intézet) segítségével lehet hozzájutni. A dokumentumokat névtelen FTP vagy HTTP kapcsolaton keresztül lehet megszerezni. A host neve `speckle.ncsl.nist.gov`. Az SQL2 szabványt és az SQL3 szabvány aktuális verzióját a következő könyvtárban lehet megtalálni:


```
isowg3/dbl/BASEdocs
```

Külön felhívnánk a figyelmet az SQL2 formális leírását tartalmazó fájlra:

```
isowg3/BASEdocs/sql-92.bnf
```

Az `isowg3/x3h2` könyvtár az SQL2 és SQL3 szabványokra vonatkozó aktuális és régebbi dokumentumokat tartalmaz. Ezen dokumentumok riportszáma X3H2-vel kezdődik.

A HTTP kapcsolatot a következő URL-en keresztül lehet elérni:

```
http://speckle.ncsl.nist.gov/~ftp/
```

mely után a feljebb említett valamelyik könyvtár útvonala következik.

Számos könyv mutatja be részletesen sz SQL programozást, főleg a [2], [3] és [6] kiadványt ajánljuk elolvasásra.

Az SQL-t először a [4]-ben definiálták. Az R rendszer részeként lett először megvalósítva [1]. Az [5] hivatkozás az SQL3 rekurzióját is bemutatja.

1. Astrahan, M. M. és mások, „System R: a relational approach to data management", *ACM Transactions on Database Systems* **1**:2, 97–137. oldal, 1976.

2. Celko, J., *SQL for Smarties*, Morgan-Kaufmann, San Francisco, 1995.

3. Date, C. J. és Darwen, H., *A Guide to the SQL Standard*, Addison-Wesley, Reading, MA, 1993.

4. Chamberlin, D. D. és mások, „SEQUEL 2: a unified approach to data definition, manipulation, and control", *IBM Journal of Research and Development* **20**:6, 560–575. oldal, 1976.

5. Finkelstein, S. J., Mattos, N., Mumick, I. S. és Pirahesh, H. „Expressing recursive queries in SQL", ISO WG3 report X3H2–96–075, 1996. március.

6. Melton, J. és Simon, A. R. *Understanding the New SQL: A Complete Guide*, Morgan-Kaufmann, San Francisco, 1993.

6. fejezet

Megszorítások és triggerek az SQL-ben

Ebben a fejezetben az SQL-nek azokat a sajátosságait tekintjük át, amelyek az aktív elemek létrehozásával kapcsolatosak. Egy aktív elem olyan kifejezés vagy utasítás, amit egyszer megírunk, eltárolunk az adatbázisban, és azt várjuk el tőle, hogy a megfelelő időpillanatokban lefusson. Ez az időpont lehet egy esemény bekövetkezése, mint például egy adott relációba való beszúrás, vagy lehet az adatbázisnak olyan megváltozása, amikor egy logikai értékű feltétel igazzá válik.

Az egyik komoly nehézség, amivel az alkalmazásfejlesztők szembe találják magukat, hogy az adatbázis módosításakor az új információ nagyon sokféleképpen lehet hibás. A legegyszerűbb módon úgy lehetne biztosítani, hogy az adatbázis-módosítások ne eredményezhessenek helytelen sorokat a relációkban, hogy minden beszúrás, törlés és módosítás utasításhoz megírnánk azokat az ellenőrzéseket, amelyek biztosítják az adatok helyességét. Sajnos ezek a helyességet biztosító feltételek gyakran elég bonyolultak és mindig ismétlődnek, hiszen a programoknak ugyanazokat az ellenőrzéseket kellene végrehajtaniuk minden módosítás után.

Szerencsére az SQL2 számos lehetőséget kínál arra, hogy az *épségi megszorításokat* az adatbázisséma részeként adjuk meg. Ebben a fejezetben a legfontosabb módszereket fogjuk áttekinteni. Az első ilyen a kulcsfeltételek megadása, amikor egy attribútum vagy egy attribútumhalmaz a reláció kulcsaként van megadva. A következő, amit tárgyalni fogunk a *hivatkozási épség*, ami azt a követelményt fejezi ki, hogy egy reláció egy vagy több attribútumának az értéke (pl. `elnökAzon` a `Stúdió`-ban) elő kell hogy forduljon egy másik reláció adott attribútumának vagy attribútumainak értékeként (pl. `Azonosító` a `GyártásIrányító`-ban). Ezután az értéktartományokra megadható megszorításokat fogjuk megvizsgálni. Ilyen megszorítás lehet az egyediség elvárása, az értéktartomány korlátozása bizonyos értékekre és a `NULL` értékek megtiltása. A fejezet ezután következő részében a sorokra vonatkozó megszorításokat, a reláció egészére vonatkozó megszorításokat és a több relációt érintő megszorításokat fogjuk áttekinteni. Ez utóbbiakat, amelyeket önálló megszorításoknak fogunk nevezni, minden alkalommal ellenőrzi a rendszer, amikor az érintett relációk bármelyike módosul.

Végül pedig a fejezet végén a triggereket fogjuk bemutatni. Ezek olyan aktív elemek, amelyek bizonyos események hatására jönnek működésbe. Ilyen esemény lehet például egy adott relációba való beszúrás. A triggerek nincsenek benne az SQL2

szabványban, hanem csak az azt követő SQL3-ban szerepelnek. Az SQL3-at e könyv írásakor még nem fejezték be, ennek ellenére már számos adatbázis-kezelő rendszer kínál a felhasználóknak valamilyen fajta triggert.

6.1. Kulcsok az SQL-ben

Talán a legfontosabb fajtamegszorítás egy adatbázisban annak deklarálása, hogy egy attribútum vagy egy attribútumhalmaz egy relációnak a kulcsát alkotja. Ez más szavakkal azt jelenti, hogy nem lehet két olyan sora a relációnak, amelyek a kulcsként megadott attribútumon – vagy a kulcsként megadott attribútumhalmazon – megegyeznek. A kulcsfeltétel, hasonlóan sok más megszorításhoz, az SQL nyelv `CREATE TABLE` utasításán belül adható meg. Kulcsokat két módon deklarálhatunk, az egyik a `PRIMARY KEY`, a másik a `UNIQUE` kulcsszó segítségével történik. Az előbbi módon csak egy kulcsot adhatunk meg egy táblához, a „unique" deklaráció azonban többször is szerepelhet.

6.1.1. Kulcsok megadása

Az elsődleges kulcs (primary key) a reláció egy vagy több attribútumából állhat. Az elsődleges kulcsot a `CREATE TABLE` utasításon belül kétfajta módon adhatjuk meg.

1. Az attribútumot deklarálhatjuk elsődleges kulcsként akkor, amikor az attribútumot felsoroljuk a relációsémában.
2. Hozzáadhatunk a sémában deklarált elemek listájához (amelyek eddig csak attribútumokat tartalmaztak) egy további deklarációt, amelyik azt adja meg, hogy egy attribútum vagy egy attribútumhalmaz kulcsot alkot.

Az első módszer esetén a `PRIMARY KEY` kulcsszót kell az attribútum neve és típusa után írnunk. A második módszer esetén egy új elemet kell az attribútumok listája után írnunk, amelyik a `PRIMARY KEY` kulcsszóból és a kulcsot alkotó attribútum vagy attribútumok zárójelezett listájából áll. Ha a kulcs több attribútumból áll, akkor mindenképpen a második módszert kell használnunk.

6.1. példa: Vegyük ismét a `FilmSzínész` reláció sémáját az 5.32. példából. Ennek a relációnak az elsődleges kulcsa a `név`. Ezt megadhatjuk a `név` attribútumot deklaráló sorban. A 6.1. ábra az 5.13. ábrának egy módosítása, amelyik már ezt a változást tükrözi.

A másik módszer szerint egy különálló definícióban adhatjuk meg az elsődleges kulcsot. Az 5.13. ábra 5. sora után adjuk meg az elsődleges kulcs deklarációját, és ebben az esetben nem kell azt a 2. sorban megadnunk. Az eredményül kapott sémadeklarációt a 6.2. ábrán láthatjuk. □

```
1) CREATE TABLE FilmSzínész (
2)    név CHAR(30) PRIMARY KEY,
3)    cím VARCHAR(255),
4)    nem CHAR(1),
5)    születésnap DATE
   );
```

6.1. ábra. *A név attribútum megadása elsődleges kulcsként*

```
1) CREATE TABLE FilmSzínész (
2)    név CHAR(30),
3)    cím VARCHAR(255),
4)    nem CHAR(1),
5)    születésnap DATE,
6)    PRIMARY KEY (név)
   );
```

6.2. ábra. *Az elsődleges kulcs különálló deklarálása*

Figyeljük meg, hogy a 6.1. példa esetében a 6.1. ábrán látható megadási módot és a 6.2. ábrán látható megadási módot is használhatjuk, mert az elsődleges kulcs egyetlen attribútumból áll. Abban az esetben, ha a kulcs több attribútumból áll, akkor a 6.2. ábra módszerét kell alkalmaznunk. Például a `Film` nevű reláció sémájának megadásakor, ahol a `cím` és `év` attribútumok együttese alkot kulcsot, a következő sort kell az attribútumok listája után írnunk:

```
PRIMARY KEY (cím, év)
```

A másik lehetőség a kulcsok deklarálására a `UNIQUE` kulcsszó használatával történik. Ez a kulcsszó pontosan ott fordulhat elő, ahol a `PRIMARY KEY` kulcsszó: vagy az attribútum neve és típusa után, vagy egy különálló elemként, mindkét esetben a `CREATE TABLE` utasítás részeként. Ennek a deklarációnak is ugyanaz az értelme mint az elsődleges kulcs deklarációnak, de amíg az elsődleges kulcsból csak egyet adhatunk meg egy táblához, addig a `UNIQUE` deklarációból akármennyit.

6.2. példa: A 6.1. ábra 2. sora helyett a következőt is írhattuk volna:

```
2) név CHAR(30) UNIQUE,
```

Vagy a 3. sort a következőre módosíthatnánk, ha úgy gondoljuk, hogy két filmszínésznek nem lehet ugyanaz a címe (ez persze eléggé kétes feltételezés):

```
3) cím VARCHAR(255) UNIQUE,
```

Hasonlóképpen módosítható a 6.2. ábra 6. sora a következőre:

```
6) UNIQUE (név)
```

□

A *kulcsfeltétel* egy olyan megszorítás, amit a `PRIMARY KEY` vagy a `UNIQUE` kulcsszóval deklarálunk. A 6.2.2. alfejezetben az „Elsődleges kulcsok és egyedi attribútumok" című bekeretezett részben majd összefoglaljuk a két deklaráció közötti különbséget.

6.1.2. Kulcsfeltételek teljesülésének biztosítása

Emlékeztetünk az 5.7.7. alfejezetre, az indexek tárgyalására, ahol láttuk, hogy habár az SQL szabványnak nem része, mégis minden SQL megvalósítás lehetőséget kínál valamilyen módon arra, hogy indexeket hozzunk létre az adatbázisséma definíciójának részeként. Az teljesen természetesnek tűnik, hogy az elsődleges kulcsra indexet hozzunk létre azért, hogy az olyan lekérdezéseket gyorsabban megválaszolhassuk, amelyek az elsődleges kulcs bizonyos értéke alapján keresnek. Ezenkívül létrehozhatunk további indexeket is más attribútumokra, amelyeket a `UNIQUE` kulcsszóval kulcsnak deklaráltunk.

Ezáltal ha a lekérdezés olyan feltételt tartalmaz a `WHERE` kulcsszó után, ami a kulcsot egy konkrét értékkel teszi egyenlővé – például a 6.1. példa `FilmSzínész` relációjában `név` = 'Audrey Hepburn' –, akkor a keresett sort nagyon gyorsan megtalálhatjuk, hiszen nem kell a reláció összes sorát végignézni.

Nagyon sok SQL megvalósításban az indexet létrehozó utasításban megadhatjuk a `UNIQUE` kulcsszót, ami az attribútumot kulcsként deklarálja, amellett, hogy létrehozza az indexet is az attribútumra. Például a következő utasítás ugyanazt eredményezi mint az 5.7.7. alfejezetben példaként szereplő – indexet létrehozó – utasítás, de ez utóbbi utasítás még egy kulcsfeltételt is deklarál a `Film` reláció `év` attribútumára.

```
CREATE UNIQUE INDEX ÉvIndex ON Film(év);
```

Nézzük meg röviden, hogy egy SQL rendszer hogyan tudja biztosítani a kulcsfeltételek teljesülését. Alapvetően a feltételt minden olyan esetben ellenőrizni kell, amikor az adatbázist módosítjuk. Nyilvánvaló azonban, hogy egy R relációra vonatkozó kulcsfeltétel csak akkor sérülhet meg, amikor R-et módosítjuk. Valójában az R-ből való törlés sem okozhat problémát, csak a beszúrás és módosítás. Ezért a gyakorlatban az SQL rendszerek csak akkor ellenőrzik a kulcsfeltételeket, amikor a relációra vonatkozóan beszúrás vagy módosítás történik.

A kulcsként deklarált attribútumokra létrehozott index alapvető fontosságú ahhoz, hogy a rendszer hatékonyan tudja ellenőrizni a kulcsfeltételeket. Ha van ilyen index, akkor a relációba való beszúrás vagy a kulcsérték módosítása esetén az indexben tudjuk ellenőrizni, hogy van-e már a relációnak olyan sora, amelyik a kulcs attribútumokban az éppen most megadott értéket tartalmazza. Ha már van ilyen sor, akkor a rendszer nem engedi, hogy a módosítás megtörténjen.

Ha nincs index a kulcs attribútumokra, attól még ellenőrizhetők a kulcsfeltételek, de ilyenkor a rendszernek a teljes relációt végig kell néznie, hogy megtudja van-e a relációnak olyan sora, amelyik az adott értékeket tartalmazza kulcsként. Ez a folyamat nagyon időigényes, és ez a nagyméretű relációk módosítását szinte csaknem lehetetlenné teszi.

6.1.3. Feladatok

* **6.1.1. feladat:** A 3.9. alfejezetben bevezetett `Film` adatbázisban minden relációhoz adtunk meg kulcsot. Módosítsuk az 5.7.1. feladatban szereplő sémadeklarációkat úgy, hogy azok tartalmazzák a relációk kulcsait is. Ne feledjük, hogy a `SzerepelBenne` relációban a három attribútum együttesen alkot kulcsot.

6.1.2. feladat: Javasoljunk megfelelő kulcsokat a 4.1.1. feladatban szereplő PC adatbázis relációihoz. Módosítsuk az 5.7.2. feladatban szereplő SQL sémát úgy, hogy az tartalmazza ezeknek a kulcsoknak a deklarációját.

6.1.3. feladat: Javasoljunk megfelelő kulcsokat a 4.1.3. feladatban szereplő `Csatahajók` adatbázis relációihoz. Módosítsuk az 5.7.3. feladatban szereplő SQL sémát úgy, hogy az tartalmazza ezeknek a kulcsoknak a deklarációját.

6.2. Hivatkozási épség és idegen kulcsok

Egy másik fajta fontos megszorítás az adatbázissémára vonatkozóan, hogy bizonyos attribútumok értékeinek megfelelő értelme legyen. Vagyis például a `Stúdió` reláció `elnökAzon` attribútumának egy bizonyos gyártásirányítóra kell utalnia. Az erre vonatkozó „hivatkozási épség" megszorítás azt jelenti, hogy ha a `Stúdió` reláció egy sorában az `elnökAzon` attribútum értéke *c*, akkor ez nem lehet akármi, hanem egy létező gyártásirányító azonosítója kell hogy legyen. Az adatbázis terminológiáját használva a létező gyártásirányító azt jelenti, hogy szerepel a `GyártásIrányító` relációban. Vagyis kell lennie a `GyártásIrányító` relációban egy olyan sornak, amelyben az azonosító *c*.

6.2.1. Idegen kulcsok megadása

Az SQL-ben egy reláció azon attribútumát vagy attribútumait *idegen kulcsnak* deklarálhatjuk, amelyek egy másik reláció (ez lehet akár ugyanaz a reláció is) bizonyos attribútumaira hivatkoznak. Ez a deklaráció két dolgot jelent egyszerre.

1. A másik reláció azon attribútumait, amelyekre hivatkozunk, elsődleges kulcsként kell deklarálni abban a relációban.
2. Minden értéknek, ami az első relációban az idegen kulcs egy attribútumában szerepel, szerepelnie kell a második reláció megfelelő attribútumában is. Ez azt jelenti, hogy van egyfajta hivatkozásiépség-megszorítás, ami a két attribútumot vagy attribútumhalmazt összekapcsolja.

Elsődleges kulcsok és egyedi attribútumok

A `PRIMARY KEY` deklaráció majdhogynem szinonimájaként tekinthető a `UNIQUE` deklarációnak. A legfontosabb különbség, hogy egy táblának csak egy `PRIMARY KEY`-vel megadott elsődleges, ugyanakkor bármennyi `UNIQUE`-kal deklarált kulcsa lehet. Van azonban ezenkívül még néhány további finom különbség is.

1. Egy idegen kulcs csak az elsődleges kulcsára hivatkozhat egy relációnak.
2. Egy adatbázis-kezelő rendszer készítője az elsődleges kulcs fogalmához néhány további speciális dolgot is hozzákapcsolhat, amelyek nem részei az SQL2 szabványnak. Például lehetséges, hogy egy konkrét adatbázis-kezelő mindig létrehoz egy indexet (ahogyan azt a 6.1.2. alfejezetben láttuk) az elsődleges kulcsként megadott attribútumra – esetleg még akkor is ha a kulcs több attribútumból áll –, de más egyéb esetekben a felhasználónak magának kell az indexet létrehoznia. Az is elképzelhető, hogy egy táblát mindig az elsődleges kulcsa szerint rendezve tárol egy rendszer, ha a táblának van elsődleges kulcsa.

Az idegen kulcsot is kétféleképpen deklarálhatjuk, ugyanúgy ahogyan azt az elsődleges kulcsok esetében láttuk.

a) Ha az idegen kulcs egyetlen attribútum, akkor az attribútum neve és típusa után adhatjuk meg, hogy az egy másik tábla egy attribútumára hivatkozik. (Annak az attribútumnak ott elsődleges kulcsnak kell lennie.) A deklaráció formája a következő:

```
REFERENCES <tábla> (<attribútum>)
```

b) A másik lehetőség, hogy a `CREATE TABLE` utasításban az attribútumok listája után egy külön deklarációban adjuk meg, hogy bizonyos attribútumok idegen kulcsot alkotnak. Itt kell megadnunk azt a táblát, és azokat az attribútumokat, amelyekre az idegen kulcs hivatkozik. (Ezeknek az attribútumoknak ez esetben is elsődleges kulcsnak kell lenniük.) A deklaráció formája ekkor a következő:

```
FOREIGN KEY <attribútumok> REFERENCES <tábla> (<attribútumok>)
```

6.3. példa: Tegyük fel, hogy a `Stúdió` relációt szeretnénk deklarálni, amelynek attribútumai `Stúdió(név, cím, elnökAzon)`, elsődleges kulcsa a `név`, és az `elnökAzon` attribútuma idegen kulcs, amelyik a `GyártásIrányító(név, cím, azonosító, nettóBevétel)` reláció `azonosító` attribútumára hivatkozik.

A deklarációt megadhatjuk a következő formában:

```
CREATE TABLE Stúdió (
   név CHAR(30) PRIMARY KEY,
   cím VARCHAR(255),
   elnökAzon INT REFERENCES GyártásIrányító(azonosító)
);
```

Vagy megadhatjuk úgy is, hogy az idegen kulcsot külön deklaráljuk:

```
CREATE TABLE Stúdió (
   név CHAR(30) PRIMARY KEY,
   cím VARCHAR(255),
   elnökAzon INT,
   FOREIGN KEY elnökAzon REFERENCES
                     GyártásIrányító(azonosító)
);
```

Vegyük észre, hogy a hivatkozott `azonosító` attribútum a `GyártásIrányító` táblának valóban elsődleges kulcsa. Ahhoz, hogy az `elnökAzon` attribútumot a `Stúdió` táblában jogosan adhassuk meg idegen kulcsként, amelyik a `GyártásIrányító` tábla `azonosító` attribútumára hivatkozik, szükséges is, hogy ez utóbbi attribútumot elsődleges kulcsként deklaráljuk.

Mindkét fenti deklarációnak az a jelentése, hogy ha egy érték szerepel a `Stúdió` tábla egy sorának `elnökAzon` oszlopában, akkor ennek az értéknek szerepelnie kell a `GyártásIrányító` tábla valamely sorának azonosító oszlopában is. Kivétel az az eset, ha a `Stúdió` tábla egy sorában az `elnökAzon` oszlopban `NULL` érték szerepel. Ilyenkor nem követelmény, hogy a `NULL` érték szerepeljen a másik tábla azonosító oszlopában. Általában nem is szokás megengedni, hogy az elsődleges kulcs attribútumban `NULL` érték szerepelhessen (lásd. a 6.3.1. alfejezetnél). □

6.2.2. Hivatkozási épség fenntartása

Láttuk, hogy hogyan kell deklarálni egy idegen kulcsot, és azt is láttuk, hogy ez a deklaráció azt jelenti, hogy az idegen kulcsnak bármely nem-`NULL` értéke elő kell hogy forduljon a hivatkozott reláció megfelelő attribútumában. Vajon hogyan lehet ezt a megszorítást fenntartani az adatbázis módosításai közben? Az adatbázis készítője három lehetséges megoldás közül választhat.

Alapértelmezés szerinti eljárás: Ha a feltétel megsérülne, a módosítást visszautasítjuk

Az SQL-ben az az alapértelmezés szerinti eljárás, hogy minden módosítást, ami megsérti a hivatkozásiépség-megszorítást, visszautasít a rendszer. Vegyük például a 6.3. példát, ahol a `Stúdió` tábla `elnökAzon` értékeinek szerepelniük kell a `GyártásIrányító` tábla `azonosító` értékei között. A következő műveleteket a rendszer visszautasítja, vagyis egy futási hibát generál.

1. Ha megpróbálunk egy olyan sort beszúrni a `Stúdió` táblába, amelyben az `elnökAzon` értéke nem `NULL`, és nem egyezik meg a `GyártásIrányító` táblabeli egyik `azonosító`-val sem. A beszúrást a rendszer visszautasítja, és a sor nem kerül be a táblába.
2. Ha megpróbáljuk módosítani a `Stúdió` tábla egy sorát, és az `elnökAzon` attribútumot egy olyan nem-`NULL` értékre változtatjuk, amelyik nem egyezik meg a `GyártásIrányító` táblabeli egyik `azonosító`-val sem. A módosítást a rendszer visszautasítja, és a sor változatlan marad.
3. Ha megpróbálunk kitörölni egy sort a `GyártásIrányító` táblából, amelyben az `azonosító` értéke szerepel a `Stúdió` tábla `elnökAzon` oszlopában. A törlést a rendszer visszautasítja, és a sor benne marad a táblában.
4. Ha megpróbáljuk módosítani a `GyártásIrányító` tábla egy sorát oly módon, hogy az `azonosító` értékét is megváltoztatjuk és a régi `azonosító` értéke szerepel a `Stúdió` tábla `elnökAzon` oszlopában. A rendszer ezt is visszautasítja, és nem módosítja a táblát.

Továbbgyűrűző eljárás

Egy másik módszer a hivatkozott táblára vonatkozó törlések és módosítások kezelésére, mint például a `GyártásIrányító` tábla esetére, az úgynevezett továbbgyűrűző eljárás. A fenti példák közül a harmadik és a negyedik típusú módosítás kezelhető ily módon. Eszerint a módszer szerint, ha kitörlünk a `GyártásIrányító` táblából egy olyan sort, ami egy stúdió vezetőjére vonatkozik, akkor a hivatkozási épség megőrzése érdekében a rendszer kitörli a `Stúdió` tábla azon sorait is, amelyek erre az imént törölt sorra hivatkoztak. A módosításokat a rendszer hasonlóan kezeli. Ha c_1-ről c_2-re változtatjuk egy gyártásirányító azonosítóját, és van olyan sora a `Stúdió` táblának, amelyben az `elnökAzon` értéke c_1, akkor a rendszer ezt az értéket is c_2-re módosítja.

A NULL értékre állítás módszere

Egy harmadik lehetőség a fenti probléma kezelésére, ha a törölt vagy módosított stúdióelnökökhöz tartozó `elnökAzon` értéket `NULL`-ra változtatja a rendszer. Ezt hívjuk a *NULL értékre állítás* módszerének.

A fenti módszerek közül az alkalmazni kívántat a törlés és módosítás esetére külön-külön, egymástól függetlenül megadhatjuk, amikor az idegen kulcsot deklaráljuk.

```
1) CREATE TABLE Stúdió (
2)   név CHAR(30) PRIMARY KEY,
3)   cím VARCHAR(255),
4)   elnökAzon INT REFERENCES GyártásIrányító(azonosító)
5)   ON DELETE SET NULL
6)   ON UPDATE CASCADE
   );
```

6.3. ábra. *Módszer megadása a hivatkozási épség megőrzésére*

A megadásuk úgy történik, hogy a törlés esetén használandó módszert az `ON DELETE`, a módosítás esetén alkalmazandót pedig az `ON UPDATE` kulcsszó után adjuk meg. A módszerekre vonatkozó kulcsszavak pedig NULL értékre állítás esetén `SET NULL`, továbbgyűrűző módszer esetén pedig `CASCADE`.

6.4. példa: Nézzük meg, hogy hogyan kell módosítanunk a `Stúdió(név, cím, elnökAzon)` reláció 6.3. példában szereplő deklarációját, hogy a `GyártásIrányító(név, cím, azonosító, nettóBevétel)` relációra vonatkozó törlések és módosítások esetén az általunk kívánt módszert alkalmazza a rendszer.

A 6.3. ábrán az említett példában szereplő első `CREATE TABLE` utasítást láthatjuk az `ON DELETE` és `ON UPDATE` záradékkal kiegészítve. Az 5. sor azt adja meg, hogy ha a `GyártásIrányító` táblából kitörlünk egy sort, akkor az összes `Stúdió`-beli sorban, amelyekben az éppen törölt gyártásirányító volt az elnök, az `elnökAzon` értékét `NULL`-ra kell változtatni. A 6. sor azt mondja meg, hogy ha a `GyártásIrányító` tábla egy sorában megváltoztatjuk az azonosítót, akkor a `Stúdió` tábla megfelelő soraiban ugyanarra az értékre kell változtatni az `elnökAzon` értékét.

Figyeljük meg, hogy a példában a törlés esetén a NULL értékre állítás tűnik értelmes megoldásnak, míg a módosítás esetén a továbbgyűrűző módszer a logikus. Ha ugyanis egy stúdió elnöke visszavonul, attól még a stúdió tovább működik, egy darabig esetleg elnök nélkül. Ha azonban egy stúdió elnökének megváltozik az azonosítója, az valószínűleg csak valamilyen adminisztratív változást jelent. Valószínűleg az illető továbbra is elnöke marad a stúdiónak, és ezért ezt a változást abban a relációban is követni kellene. □

6.2.3. Feladatok

6.2.1. feladat: Adjuk meg a következő hivatkozásiépség-megszorítások deklarációit a film adatbázisra:

```
Film(cím, év, hossz, színes, stúdióNév, producerAzon)
SzerepelBenne(filmCím, év, színészNév)
FilmSzínész(név, cím, nem, születésnap)
GyártásIrányító(név, cím, azonosító, nettóBevétel)
Stúdió(név, cím, elnökAzon)
```


Lógó sorok és módosítási eljárások

Azokat a sorokat, amelyekben olyan idegen kulcs értéke szerepel, ami nincs benne a hivatkozott táblában, *lógó soroknak* nevezzük. Emlékezzünk rá, hogy azokat a sorokat is lógó soroknak neveztük, amelyek egy összekapcsolás esetén kimaradtak az eredményből. A fenti két dolog szoros kapcsolatban áll egymással. Ha egy sorbeli idegen kulcs értéke hiányzik a hivatkozott táblából, akkor az adott sor nem lesz benne a relációnak a hivatkozott relációval vett összekapcsolásában.

A lógó sorok pontosan azok a sorok lesznek, amelyek megsértik az adott idegen kulcs megszorítására vonatkozó hivatkozási épséget.

- Az alapértelmezés szerinti eljárás a törlésekre és módosításokra vonatkozóan azt jelenti, hogy azok és csak azok a műveletek tiltottak, amelyek lógó sorokat hoznak létre abban a relációban, amelyik a másikra hivatkozik.
- A továbbgyűrűző eljárás törli vagy módosítja az összes létrejött lógó sort, attól függően, hogy az aktuális művelet a hivatkozott relációra vonatkozó törlés vagy módosítás.
- A NULL értékre állítás módszere az idegen kulcsot `NULL`-ra változtatja az összes lógó sorban.

a) Egy film producerének szerepelnie kell a gyártásirányítók között. A `GyártásIrányító` tábla olyan módosításait, amelyek ezt a megszorítást megsértenék, utasítsa vissza a rendszer.

b) A feltétel legyen ugyanaz mint az előbb, de a megszorítást megsértő módosítások esetén a `Film` reláció `producerAzon` oszlopát változtassa a rendszer `NULL`-ra.

c) A feltétel legyen ugyanaz mint a)-ban, de a megszorítást megsértő törlések esetén törölje a rendszer a `Film` reláció megfelelő sorait is.

d) A `SzerepelBenne` relációban levő filmeknek benne kell lenniük a `Film` táblában is. A megszorítást megsértő utasításokat a rendszer utasítsa vissza.

e) A `SzerepelBenne` relációban levő színészeknek benne kell lenniük a `FilmSzínész` táblában is. A megszorítást megsértő utasításokat a rendszer kezelje le a megfelelő sorok törlésével.

*! **6.2.2. feladat:** Azt a megszorítást szeretnénk megadni, hogy minden filmnek, ami a `Film` relációban szerepel, legalább egy színésszel benne kell lennie a `Szerepel-`

`Benne` relációban is. Megadhatjuk-e ezt egy idegen kulcs segítségével? Indokoljuk meg a választ!

6.2.3. feladat: Adjuk meg a következő hivatkozásiépség-megszorításokat a 4.1.3. példában szereplő adatbázissémára vonatkozóan. A kulcsokat értelemszerűen adjuk meg, és a megszorításokat megsértő utasításokat kezeljük le úgy, hogy a megfelelő értékeket `NULL`-ra változtatjuk.

```
Hajóosztályok(hajóosztály, típus, ország,
              ágyúkSzáma, kaliber, vízkiszorítás)
Hajók(név, hajóosztály, felavatva)
Csaták(név, dátum)
Kimenetelek(hajó, csata, eredmény)
```

* a) Minden `Hajók` táblabeli hajóosztálynak szerepelnie kell a `Hajóosztályok` táblában is.

b) Minden `Kimenetelek` táblabeli csatának szerepelnie kell a `Csaták` táblában is.

c) Minden `Kimenetelek` táblabeli hajónak szerepelnie kell a `Hajók` táblában is.

6.3. Attribútumértékekre vonatkozó megszorítások

Az eddigiekben áttekintettük a kulcsfeltételeket, amelyek azt követelik meg, hogy bizonyos attribútumok értéke különböző legyen a reláció összes sorában, és megvizsgáltuk az idegenkulcs-megszorításokat, amelyek két reláció attribútumai között teremtenek kapcsolatot. A következőkben egy harmadik fajta fontos megszorítást tárgyalunk: azokat, amelyek azt korlátozzák, hogy bizonyos attribútumok milyen értékeket vehetnek fel. Ezek a megszorítások két különböző módon fejezhetők ki:

1. Attribútumra vonatkozó megszorítás, mely a relációséma definiálásakor adható meg.

2. Értéktartományra vonatkozó megszorítás, majd az attribútum hozzárendelése az értéktartományhoz.

A 6.3.1. alfejezetben bevezetünk egy egyszerű megszorítástípust az attribútum értékére vonatkozóan, ami azt mondja ki, hogy az attribútum nem veheti fel a `NULL` értéket. A 6.3.2. alfejezetben áttekintjük a fenti 1. módon kifejezett megszorítások alapvető formáját. Ezeket *attribútumra vonatkozó* `CHECK` *feltételeknek* fogjuk hívni. A 2. módon kifejezett megszorításokat, az értéktartományokra vonatkozó megszorításokat, a 6.3.3. alfejezetben tárgyaljuk. Vannak további, még általánosabb fajta megszorítások, ezekről a 6.4. alfejezetben lesz szó. Ez utóbbiak segítségével korlátozhatjuk a teljes sorokra vonatkozó módosításokat, vagy akár az egy vagy több teljes relációra vonatkozó módosításokat, de használhatjuk ezeket egyetlen attribútum értékének korlátozására is.

6.3.1. NOT NULL feltételek

Egy egyszerű megszorítás, ami egy attribútumhoz hozzárendelhető a `NOT NULL` feltétel. Ez nem engedi meg olyan sorok előfordulását, amelyben az adott attribútum értéke `NULL`. A megszorítást úgy deklarálhatjuk, hogy a `CREATE TABLE` utasításban az attribútum megadása után a `NOT NULL` kulcsszót szerepeltetjük.

6.5. példa: Tegyük fel, hogy a `Stúdió` relációban az `elnökAzon` értéke nem lehet `NULL`. Ezt megadhatjuk úgy, hogy a 6.3. ábra 4. sorát a következőre változtatjuk:

```
4)elnökAzon INT REFERENCES GyártásIrányító(azonosító)
  NOT NULL
```

Ennek a változtatásnak a következő következményei lesznek:

- Nem módosíthatjuk egy sor `elnökAzon` attribútumát `NULL`-ra.
- Nem szúrhatunk be egy olyan sort a `Stúdió` relációba, amelyre csak a nevet és a címet adtuk meg, mert ekkor az új sorban az `elnökAzon` értéke `NULL` lenne.
- Nem alkalmazhatjuk a NULL értékre állítás módszerét a 6.3. ábra 5. sorához hasonló módon, ahol úgy oldaná meg a rendszer az idegenkulcs-megszorítások megsértését, hogy az `elnökAzon` értékét `NULL`-ra változtatná.

□

6.3.2. Attribútumra vonatkozó CHECK feltételek

Bonyolultabb megszorítások rendelhetők hozzá egy attribútumhoz, ha a deklarációban a `CHECK` kulcsszót használjuk. A kulcsszót egy zárójelbe tett feltétel követi, amelyet az attribútum minden értékének ki kell elégítenie. A gyakorlatban egy attribútumra vonatkozó `CHECK` feltétel olyan, mint egy egyszerű korlátozás az attribútum értékeire vonatkozóan. Ilyen lehet például a megengedett értékek felsorolása, vagy egy aritmetikai egyenlőtlenség. Elvben azonban a feltétel bármi lehet, ami egy SQL lekérdezésben a `WHERE` után állhat. Hivatkozhat például az éppen korlátozni kívánt attribútumra. Ha azonban a feltétel másik relációra, vagy más attribútumokra hivatkozik, akkor a

megfelelő relációkat egy alkérdés `FROM` záradékában kell megadnunk. Ez magára a korlátozni kívánt attribútum relációjára is vonatkozik.

Egy attribútumra vonatkozó `CHECK` feltételt akkor ellenőriz a rendszer, amikor valamelyik sorban az adott attribútum új értéket kap. Ez történhet egy módosítás útján vagy egy beszúrás alkalmával. Ha az új érték megsérti a feltételt, akkor a rendszer az adatmódosítást visszautasítja. Amint azt a 6.7. példában látni fogjuk, az attribútumra vonatkozó `CHECK` feltételt a rendszer nem biztos, hogy ellenőrzi, ha egy adatbázis-módosítás nem változtatja meg azt az attribútumot, amelyikhez a feltételt hozzárendeltük. Ez oda vezethet, hogy a feltétel megsérül. Először nézzünk meg egy egyszerű attribútumra vonatkozó `CHECK` feltételt.

6.6. példa: Tegyük fel, hogy azt követeljük meg, hogy az azonosító számok legalább hatjegyűek legyenek. Ehhez a 6.3. ábra 4. sorát, amelyik a `Stúdió(név, cím, elnökAzon)` reláció sémájának egy részét deklarálja, a következőre módosíthatjuk:

```
4) elnökAzon INT REFERENCES GyártásIrányító(azonosító)
             CHECK (elnökAzon >= 100000)
```

Vegyünk egy másik példát. A `FilmSzínész(név, cím, nem, születésnap)` reláció `nem` attribútumát az 5.13. ábrán `CHAR(1)` típusúnak deklaráltuk, ami egyetlen karaktert jelent. Valójában azonban azt szeretnénk, ha ez az egy karakter csak 'F' vagy 'N' lehetne. Ezt elérhetjük, ha az 5.13. ábra 4. sorát a következővel helyettesítjük:

```
4) nem CHAR(1) CHECK (nem IN ('F', 'N')),
```

A fenti feltétel egy konkrét, két értéket tartalmazó konstans halmazt használ, és azt mondja ki, hogy a nem attribútum értékeinek ebben a halmazban kell lennie. □

Az ellenőrizendő feltétel más attribútumokra vagy a reláció további soraira is hivatkozhat, sőt akár más relációkra is. Ilyenkor a feltételnek egy alkérdést kell tartalmaznia. Ahogy korábban említettük a feltétel bármi lehet, ami egy SQL select utasítás `WHERE` utáni részében megengedett. Tudatában kell azonban lennünk, hogy a feltétel ellenőrzése csak az adott attribútumhoz van hozzárendelve és nem az összes, a feltételben szereplő relációhoz vagy attribútumhoz. Ennek eredményeként a feltétel hamissá válhat, ha valamelyik másik, a feltételben szereplő érték megváltozik.

6.7. példa: Azt gondolhatnánk, hogy egy idegenkulcs-megszorítást szimulálni tudunk egy attribútumra vonatkozó `CHECK` feltétellel, ami megköveteli a hivatkozott érték létezését. A következő egy *hibás* kísérlet annak megkövetelésére, hogy a `Stúdió(név, cím, elnökAzon)` reláció soraiban szereplő `elnökAzon` értékeknek elő kell fordulniuk a `GyártásIrányító(név, cím, azonosító, nettóBevétel)` reláció azonosító attribútumában.

Tegyük fel, hogy a 6.3. ábra 4. sorát a következővel helyettesítjük:

```
4) elnökAzon INT CHECK
        (elnökAzon IN (SELECT azonosító
                       FROM GyártásIrányító))
```

Ez egy szabályos, attribútumra vonatkozó `CHECK` feltétel, de vizsgáljuk meg, hogy mit is eredményez pontosan.

- Ha megpróbálunk egy új sort beszúrni a `Stúdió` táblába, és a sor olyan `elnökAzon` értéket tartalmaz, ami nem egy gyártásirányítónak az azonosítója, akkor a rendszer a beszúrást visszautasítja.
- Ha megpróbáljuk módosítani a `Stúdió` tábla egy sorát és az új `elnökAzon` érték nem egy gyártásirányítónak az azonosítója, akkor a rendszer a módosítást visszautasítja.
- Ha azonban a `GyártásIrányító` relációt változtatjuk meg, mondjuk törlünk egy stúdiónak az elnökére vonatkozó sort, akkor ez a változás már láthatatlan lesz a `CHECK` feltétel számára. Így a rendszer a törlést megengedi, pedig az `elnökAzon` attribútumra vonatkozó `CHECK` feltétel megsérült.

A 6.4.2. alfejezetben majd látni fogjuk, hogy ezt a fenti feltételt hogyan tudjuk helyesen kifejezni kifinomultabb megszorítások segítségével. □

6.3.3. Értéktartományokra vonatkozó megszorítások

Egy attribútum értékeit úgy is korlátozhatjuk, hogy egy értéktartományt deklarálunk (lásd az 5.7.6. alfejezetet) egy hasonló megszorítással, majd ezt az értéktartományt adjuk meg az attribútum típusaként. Az egyetlen fontos különbség a két megszorítás között, hogy amikor az értéktartomány egy értékére szeretnénk valamilyen feltételt megadni, akkor arra nem tudunk semmilyen módon hivatkozni. Az attribútumra vonatkozó megszorítás esetén az attribútum nevével hivatkozhattunk az értékére. Az SQL2 úgy oldja meg ezt a problémát, hogy egy speciális kulcsszót, a `VALUE`-t kínálja arra, hogy az értéktartomány egy értékére hivatkozhassunk.

6.8. példa: Egy `NemÉrtékek` nevű értéktartományt, amelyik csak az 'F' és az 'N' karakterek előfordulását engedi meg, a következőképpen deklarálhatunk:

```
CREATE DOMAIN NemÉrtékek CHAR(1)
  CHECK (VALUE IN ('F', 'N'));
```

Mikor kell egy megszorítást ellenőrizni?

Rendes körülmények között egy SQL rendszer nem enged meg olyan adatbázis-módosítást, amelyik megsért egy megszorítást. Néha azonban néhány összetartozó módosítást kell elvégeznünk, amelyek közül az egyik megsért egy feltételt, de egy másik helyreállítja azt. Például a 6.3. példában a `Stúdió` reláció `elnökAzon` attribútumát idegen kulcsként adtuk meg, amelyik a `GyártásIrányító` reláció `azonosító` attribútumára hivatkozik. Ha fel akarunk vinni az adatbázisba egy új stúdiót és annak elnökét, akkor, ha a stúdiót visszük fel elsőként, megsértjük az idegen kulcs feltételt.

Úgy tűnik, hogy a problémát megoldhatjuk azzal, hogy először az elnököt visszük fel a `GyártásIrányító` relációba. Tegyük fel azonban, hogy van egy további megszorításunk, amelyik szerint a `GyártásIrányító`-beli azonosítóknak szerepelniük kell vagy a `Stúdió`-beli elnök azonosítók között vagy a `Film`-beli producerazonosítók között. Ebben az esetben már egyik utasítássorrend sem lesz megfelelő.

Szerencsére az SQL2 lehetővé teszi, hogy egy megszorítást DEFERRED-ként deklaráljunk. Ilyen esetben a megszorítás ellenőrzését a rendszer addig nem végzi el, amíg vége nincs a tranzakciónak. (A tranzakció az adatbázison végzett műveletek egy egysége. Lásd a 7.2. alfejezetben.) A stúdiónak és a hozzá tartozó elnöknek a felvitelét egy tranzakcióba tehetjük, és így elkerülhetjük a megszorításnak ezt a látszólagos megsértését.

Majd az 5.13. ábra 4. sora helyett a következőt írhatjuk:

```
4) nem NemÉrtékek,
```

A 6.6. példában szereplő elvárásunkat, hogy az `elnökAzon` azonosító legalább hatjegyű legyen, a következő értéktartomány-deklarációval érhetjük el:

```
CREATE DOMAIN AzonÉrtékek INT
   CHECK (VALUE>=100000);
```

Majd az `elnökAzon` attribútumot a következőképpen kell deklarálnunk:

```
4) elnökAzon AzonÉrtékek REFERENCES
      GyártásIrányító(azonosító)
```

□

6.3.4. Feladatok

6.3.1. feladat: Adjuk meg a következő megszorításokat a

```
Film(cím, év, hossz, színes, stúdióNév, producerAzon)
```

reláció attribútumaira vonatkozóan:

* a) Az év nem lehet korábbi 1895-nél.

b) A hossz nem lehet 60-nál kisebb és 250-nél nagyobb.

* c) A stúdió neve csak Disney, Fox, MGM vagy Paramount lehet.

6.3.2. feladat: Adjuk meg a következő megszorításokat a 4.1.1. példában szereplő séma attribútumaira vonatkozóan:

```
Termék(gyártó, modell, típus)
PC(modell, sebesség, memória, merevlemez, CD, ár)
Laptop(modell, sebesség, memória, merevlemez,
        képernyő, ár)
Nyomtató(modell, szín, típus, ár)
```

a) Egy laptop sebessége nem lehet kisebb 100-nál.

b) Egy CD sebessége csak 4×, 6×, 8× vagy 12× lehet.

c) A nyomtatók típusa csak lézer, tintasugaras vagy mátrix lehet.

d) A termékek típusa csak PC, laptop vagy nyomtató lehet.

e) Egy PC-ben levő memória legalább a merevlemez 1%-a kell hogy legyen.

6.4. Globális megszorítások

A következőkben még bonyolultabb megszorítások deklarációjával fogunk foglalkozni, amelyek több attribútumra vonatkozó feltételeket, sőt akár több relációt érintő feltételeket is tartalmazhatnak. A témát két különálló részre osztjuk majd:

1. *Sorra vonatkozó* CHECK *feltételek*, amelyek egyetlen reláció soraira tesznek különféle megszorításokat.

2. *Önálló megszorítások*, amelyek teljes relációkat vagy ugyanazt a relációt befutó több sorváltozót is tartalmazhatnak a feltételükben.

6.4.1. Sorra vonatkozó CHECK feltételek

Egy *R* tábla soraira vonatkozó megszorítást megadhatunk úgy, hogy a CREATE TABLE utasításban az attribútumok, a kulcsok és az idegen kulcsok deklarációja után megadjuk a CHECK kulcsszót, majd zárójelek között egy feltételt. Ez a feltétel bármi lehet, ami a WHERE után szerepelhet. A rendszer a megadott feltételt az *R* egy sorára vonatkozó feltételként értelmezi. Hasonlóan azonban az attribútumra vonatkozó CHECK feltételek esetéhez, a feltétel itt is hivatkozhat más relációkra vagy az *R* reláció más soraira. Ezek a hivatkozások most is alkérdésben szerepelhetnek.

A sorra vonatkozó CHECK feltételeket a rendszer akkor ellenőrzi, amikor egy új sort szúrunk be *R*-be, vagy amikor *R* egy sorát módosítjuk. A feltétel minden új, illetve módosított sorra külön kerül kiértékelésre. Ha az eredmény hamis lesz, akkor a feltételt megsértő sorra vonatkozó beszúrás vagy módosítás utasítást a rendszer visszautasítja. Ha azonban a feltétel egy másik relációra hivatkozik egy alkérdésben (ez lehet akár maga az *R* is) és ennek a másik relációnak a módosítása okozza azt, hogy *R* egy sorára a feltétel hamis lesz, akkor a rendszer emiatt nem utasítja vissza azt az utasítást. Vagyis ugyanúgy ahogy a 6.7. példában szereplő attribútumra vonatkozó CHECK feltétel esetén említettük, a sorra vonatkozó CHECK feltételek is láthatatlanok más relációk számára.

Noha a sorra vonatkozó CHECK feltételek meglehetősen bonyolult feltételeket is tartalmazhatnak, mégis célszerűbb a bonyolult ellenőrzéseket az SQL nyelv önálló megszorításaiban megadni, amelyeket a 6.4.2. alfejezetben tárgyalunk majd. Ennek oka – ahogy korábban már említettük –, hogy a sorra vonatkozó CHECK feltételeket az utasítások bizonyos körülmények között megsérthetik. Ha azonban egy sorra vonatkozó CHECK feltétel csak az ellenőrizendő sor attribútumait tartalmazza és nincs benne alkérdés, akkor a feltétel mindig igaz marad. Nézzünk egy egyszerű, sorra vonatkozó CHECK feltételt, amelyik egy sor több attribútumát is tartalmazza.

6.9. példa: Idézzük fel az 5.32. példában szereplő FilmSzínész tábla sémadeklarációját. A 6.4. ábra megismétli a CREATE TABLE utasítást és kiegészíti azt egy kulcs deklarációval – a UNIQUE kulcsszó használatával –, valamint egy további megszorítással, amelyik egy lehetséges „következetességi feltétel", amelyet szeretnénk ellenőrizni. Ez a feltétel azt mondja ki, hogy ha egy színész neme férfi akkor a neve nem kezdődhet 'Ms.'-sel.

```
1) CREATE TABLE FilmSzínész (
2)    név CHAR(30) UNIQUE,
3)    cím VARCHAR(255),
4)    nem CHAR(1),
5)    születésnap DATE,
6)    CHECK (nem = 'N' OR név NOT LIKE 'Ms.%')
   );
```

6.4. ábra. *Egy megszorítás a FilmSzínész táblára*

A 2. sorban a név attribútumot a reláció kulcsaként adtuk meg, a 6. sor pedig a megszorítást deklarálja. A megszorítás feltétele igaz minden nőnemű filmszínészre és minden olyan színészre, akinek a neve nem 'Ms.'-sel kezdődik. A feltétel azokra a sorokra lesz hamis, amelyekben a nem oszlop értéke férfi és a név 'Ms.'-sel kezdődik. Éppen ezeket a sorokat szerettük volna kizárni a relációból. □

> ## A megszorítások helyes megfogalmazása
>
> Sokszor van szükségünk olyan megszorításra, amelyik hasonlít a 6.9. példában szereplőhöz, vagyis ahol olyan sorok előfordulását szeretnénk kizárni, amelyek két vagy több feltételnek is eleget tesznek. Ilyenkor a CHECK kulcsszót a feltételek tagadásainak kell követnie, közöttük az OR kulcsszót megadva. A 6.9. példában az első feltétel az lenne, hogy a színész férfi legyen. Ezért használtuk a nem = 'N' feltételt, mint ennek tagadását. (Használhattuk volna a nem <> 'F' feltételt is.) A második feltétel, hogy a név 'Ms.'-sel kezdődjön, aminek a tagadására a NOT LIKE összehasonlítást használtuk. Ez az összehasonlítás magát a feltételt tagadja, ami eredetileg az SQL-ben név LIKE 'Ms.%' lett volna.

6.4.2. Önálló megszorítások

Az attribútumra vonatkozó megszorításoktól eljutottunk a sorra vonatkozó megszorításokig. Néha azonban ezek sem elegendőek számunkra. Előfordulhat, hogy olyan megszorításra lenne szükségünk, ami egy reláció egészére ad meg feltételt, például egy oszlopbeli értékek összegére vagy más aggregáltjára vonatkozóan. Szükség lehet olyan megszorításra is, amelyik több relációra vonatkozik. Valójában az idegenkulcs-megszorítások már ilyenek, amelyek két relációt kapcsolnak össze, de ezek csak meglehetősen korlátozott formában adhatók meg.

Az SQL2-beli önálló megszorítások azok, amelyek bármilyen feltétel ellenőrzését – ami a WHERE után megengedett – lehetővé teszik a számunkra. Szokás ezeket általános megszorításoknak is nevezni. A korábban említett megszorítás típusok mindig valamilyen séma elemhez voltak hozzárendelve – általában táblához vagy értéktartományhoz – az önálló megszorítások azonban maguk is sémaelemek.

Hasonlóan a többi sémaelemhez, ezeket is egy CREATE utasítással hozzuk létre. Az utasítás formája a következő:

1. A CREATE ASSERTION kulcsszó,
2. Az önálló megszorítás neve,
3. A CHECK kulcsszó,
4. Egy zárójelben megadott feltétel.

A nem teljes körű ellenőrzés hiba vagy lehetőség?

Talán csodálkozunk azon, hogy az attribútumra és a sorra vonatkozó CHECK feltételek esetén miért engedi meg a rendszer azok megsértését, ha azok másik relációra vagy az adott reláció másik soraira hivatkoznak. Ennek az az oka, hogy ezek a megszorítások hatékonyabban valósíthatók meg, mint az önálló megszorítások. Az attribútumra vagy a sorra vonatkozó CHECK feltételek esetén a megszorítást csak a beszúrandó vagy módosítandó sorra kell kiértékelnie a rendszernek, míg az önálló megszorításokat minden esetben ki kell értékelni, ha a bennük szereplő bármelyik reláció megváltozik. Azt, hogy ezek a további kiértékelések megérik-e az adatbázis módosításainak hosszabb futási idejét, azt az adatbázis tervezőjének kell eldöntenie. Mi mindenesetre azt tanácsoljuk a tervezőknek, hogy ne használjanak olyan attribútumra vagy sorra vonatkozó CHECK feltételt, ami megsérülhet. Ezzel hosszú távon garantálható a program biztonsága.

Vagyis az utasítás formája:

```
CREATE ASSERTION <név> CHECK (<feltétel>)
```

Egy önálló megszorításban megadott feltételnek mindig igaznak kell lennie, és bármilyen adatbázis-módosítást, ami a feltétel megsértését eredményezné, a rendszer visszautasít. Emlékeztetünk rá, hogy az eddig tárgyalt CHECK feltételek bizonyos körülmények között megsérülhetnek, ha azok alkérdéseket is tartalmaznak.

Máshogyan kell megadnunk a sorra vonatkozó CHECK feltételeket és az önálló megszorításokat. A sorra vonatkozó ellenőrzések hivatkozhatnak annak a relációnak az attribútumaira, amelynek deklarációjában szerepelnek. Például a 6.4. ábra 6. sorában a nem és név attribútumokat anélkül használtuk, hogy megadtuk volna, hogy melyik relációban szerepelnek. Azok a FilmSzínész tábla sorainak elemeire utalnak, hiszen ez az a tábla, amelyiket a CREATE utasításban éppen deklaráltunk.

Egy önálló megszorítás feltételében nincs ilyen alapértelmezés. Ott minden, a feltételben szereplő attribútumra magában a megszorításban kell megadnunk, hogy melyik táblában van. Ez általában a táblának egy select-from-where kifejezésben való szerepeltetésével történik. Mivel a feltételnek egy logikai értéket kell szolgáltatnia, ezért szokásos a feltétel eredményeit valamilyen módon aggregálni, hogy egyetlen igaz vagy hamis értéket kapjunk. Például megadhatjuk a feltételt úgy, hogy az egy relációt adó kifejezés legyen és erre alkalmazzuk a NOT EXISTS műveletet. Vagyis a megszorítás azt jelenti, hogy ez a reláció mindig üres. Egy másik lehetőség, hogy valamilyen aggregátor műveletet – pl. a SUM műveletet – alkalmazzuk egy reláció egy oszlopára, és az eredményt összehasonlítjuk egy konstanssal. Ilyen módon előírhatjuk például, hogy az összeg mindig kisebb legyen valamilyen korlátnál.


```
CREATE ASSERTION GazdagElnök CHECK
   (NOT EXISTS
      (SELECT *
      FROM Stúdió, GyártásIrányító
      WHERE elnökAzon = azonosító
      AND nettóBevétel < 10000000
      )
);
```

6.5. ábra. *Önálló megszorítás, amelyik a stúdióelnökök gazdagságát írja elő*

6.10. példa: Tegyük fel, hogy azt szeretnénk kikötni, hogy senki ne lehessen egy stúdió elnöke, ha a nettó bevétele nem éri el a 10 000 000 dollárt. Létrehozunk egy önálló megszorítást, amelyik azt mondja ki, hogy azon filmstúdiók halmaza, amelyekre az elnök nettó bevétele kisebb, mint 10 000 000 dollár, üres. Ez az önálló megszorítás az alábbi két relációt foglalja magába:

```
GyártásIrányító(név, cím, azonosító, nettóBevétel)
Stúdió(név, cím, elnökAzon)
```

Az önálló megszorítást a 6.5. ábrán láthatjuk.

Mellékesen megjegyezzük, hogy habár ez a megszorítás két relációra mond ki feltételt egyszerre, mégis megadhatnánk azt két, sorra vonatkozó CHECK feltétellel is az egyetlen önálló megszorítás helyett. Például a 6.3. példa CREATE TABLE utasításában egy további megszorítást adhatunk meg a Stúdió táblára, ahogyan azt a 6.6. ábrán láthatjuk.

Vegyük észre azonban, hogy a 6.6. ábrán szereplő feltételt csak akkor ellenőrzi a rendszer, amikor a Stúdió reláció változik meg. Azt az esetet, ha valamelyik stúdió elnökének a nettó bevétele a GyártásIrányító táblában 10 000 000 dollár alá csökken, a fenti megszorítás nem figyeli. Ahhoz, hogy az önálló megszorítás teljes eredményét elérjük, a GyártásIrányító tábla deklarációját is ki kell egészítenünk egy további megszorítással, amelyik azt írja elő, hogy a nettó bevétel legalább 10 000 000 dollár legyen, ha az adott személy egyben egy stúdió elnöke is. □

```
1) CREATE TABLE Stúdió (
2)    név CHAR(30) PRIMARY KEY,
3)    cím VARCHAR(255),
4)    elnökAzon INT REFERENCES GyártásIrányító(azonosító),
5)    CHECK (elnökAzon NOT IN
6)       (SELECT azonosító FROM GyártásIrányító
7)       WHERE nettóBevétel < 10000000)
      )
   );
```

6.6. ábra. *Egy önálló megszorításnak megfelelő, sorra vonatkozó CHECK feltétel*

6.11. példa: Vegyük a következő önálló megszorítást, amelyik csak egy relációt, a

```
Film(cím, év, hossz, színes, stúdióNév, producerAzon)
```

relációt érint. A megszorítás azt mondja ki, hogy egy stúdió által készített filmek összhosszúsága nem haladhatja meg a 10 000 percet.

```
CREATE ASSERTION ÖsszHossz CHECK (10000>=ALL
   (SELECT SUM(hossz) FROM Film GROUP BY stúdióNév)
);
```

A fenti megszorítás csak a `Film` relációra mond ki feltételt. Az önálló megszorítás használata helyett ezt kifejezhettük volna egy sorra vonatkozó CHECK feltétellel is. Ehhez a következő sorokat kellett volna a `Film` reláció sémadeklarációjához hozzáadnunk:

```
CHECK (10000>=ALL
   (SELECT SUM(hossz) FROM Film GROUP BY stúdióNév));
```

Figyeljük meg, hogy ez utóbbi feltétel tulajdonképpen a `Film` tábla minden egyes sorára vonatkozik. A fenti megfogalmazásban egyetlen attribútum neve sem szerepel konkrétan, hanem magát a feltételt az alkérdés biztosítja.

Azt is fontos megjegyezni, hogy ha a fenti ellenőrzést sorra vonatkozó CHECK feltétel alkalmazásával valósítanánk meg, akkor a feltételt a rendszer nem ellenőrizné

Megszorítások összehasonlítása

Az alábbi táblázatban összefoglaltuk a legfontosabb különbségeket az attribútumra vonatkozó CHECK feltételek, a sorra vonatkozó CHECK feltételek és az önálló megszorítások között.

Megszorítás típusa	*Hol deklaráljuk?*	*Mikor ellenőrzi a rendszer?*	*Minden esetben igaz marad?*
Attribútumra vonatkozó CHECK feltétel	Az attribútum megadásakor	A relációba való beszúráskor, vagy az attribútum módosításakor	Alkérdés esetén nem
Sorra vonatkozó CHECK feltétel	A relációséma megadásakor	A relációba való beszúráskor, vagy egy sor módosításakor	Alkérdés esetén nem
Önálló megszorítás	Az adatbázisséma megadásakor	Bármelyik, a feltételben szereplő reláció megváltozása esetén	Igen

a `Film` táblából való törlés esetén. Ebben a konkrét esetben ez nem is okoz problémát, hiszen ha a megszorítás feltétele a törlés előtt teljesült, akkor biztosan teljesülni fog a törlés után is. Ha azonban a megszorítás a filmek összhosszúságára vonatkozóan nem felső korlátot adna meg, ahogyan az most a példában szerepelt, hanem alsó korlátot, akkor a feltétel megsérülhetne, ha azt sorra vonatkozó CHECK feltétel segítségével valósítanánk meg. Önálló megszorítás használata esetén ez nem fordulhat elő. □

6.4.3. Feladatok

6.4.1. feladat: Említettük a 6.10. példában, hogy a 6.6. ábrán lévő sorra vonatkozó CHECK feltétel csak egy részét biztosítja annak, amit a 6.5. ábra önálló megszorítása garantál. Írjunk olyan megszorítást a `GyártásIrányító` táblára, amelyik a még szükséges ellenőrzéseket elvégzi.

6.4.2. feladat: Írjuk meg a következő megszorításokat sorra vonatkozó CHECK feltételek formájában az alábbi relációkra vonatkozóan:

```
Film(cím, év, hossz, színes, stúdióNév, producerAzon)
SzerepelBenne(filmCím, év, színészNév)
FilmSzínész(név, cím, nem, születésnap)
GyártásIrányító(név, cím, azonosító, nettóBevétel)
Stúdió(név, cím, elnökAzon)
```

Ha a megszorítás két relációt is érint, akkor írjunk mindkét relációhoz megfelelő megszorítást, hogy bármelyik relációra vonatkozó beszúrás vagy módosítás esetén a rendszer ellenőrizze a feltételt. Tegyük fel, hogy törlések nem fordulhatnak elő. A sorra vonatkozó CHECK feltételekkel megadott megszorítások teljesülését törlések esetén nem lehet biztosítani.

* a) 1939 előtt készült film nem lehet színes.

b) Egy filmszínész nem szerepelhet olyan filmben, amit a születése előtt készítettek.

! c) Két stúdiónak nem lehet azonos a címe.

*! d) Egy név, amelyik benne van a `FilmSzínész` táblában, nem szerepelhet a `GyártásIrányító` táblában is.

! e) A `Stúdió` táblában szereplő stúdióneveknek a `Film` tábla legalább egy sorában szerepelniük kell.

!! f) Ha egy film producere egyben egy stúdió elnöke is, akkor neki kell lennie a filmet készítő stúdió elnökének is.

6.4.3. feladat: Fejezzük ki az alábbi megszorításokat az SQL nyelv önálló megszorításaival. A megszorítások a 4.1.1. feladat relációira vonatkoznak:

```
Termék(gyártó, modell, típus)
PC(modell, sebesség, memória, merevlemez, CD, ár)
Laptop(modell, sebesség, memória, merevlemez,
       képernyő, ár)
Nyomtató(modell, szín, típus, ár)
```

* a) PC-gyártó nem készíthet laptopot.

*! b) Egy PC-gyártónak legalább ugyanolyan sebességű laptopot is kell gyártania, mint a PC sebessége.

! c) Ha egy laptopnak nagyobb memóriája van, mint egy PC-nek, akkor a laptop árának is magasabbnak kell lennie a PC áránál.

!! d) A modellszámnak egyedinek kell lennie a `PC`, a `Laptop` és a `Nyomtató` relációk együttesére nézve is.

!! e) Ha a `Termék` relációban szerepel egy modell és típusa, akkor a modellnek szerepelnie kell a típusnak megfelelő relációban is.

6.4.4. feladat: Adjuk meg a következő megszorításokat sorra vonatkozó `CHECK` feltételek formájában:

a) Egy 150 MHz-nél kisebb sebességű PC ára nem lehet több 1500 dollárnál.

b) Egy olyan laptopnak, amelynek képernyője kisebb, mint 11 hüvelyk, vagy legalább 1 gigabájtos merevlemezzel kell rendelkeznie, vagy 2000 dollárnál kevesebbe kell kerülnie.

6.4.5. feladat: Fejezzük ki az alábbi megszorításokat az SQL nyelv önálló megszorításaival. A megszorítások a 4.1.3. feladat relációira vonatkoznak:

```
Hajóosztályok(hajóosztály, típus, ország,
              ágyúkSzáma, kaliber, vízkiszorítás)
Hajók(név, hajóosztály, felavatva)
Csaták(név, dátum)
Kimenetelek(hajó, csata, eredmény)
```

a) Egyetlen hajóosztályban sem szerepelhet kettőnél több hajó.

! b) Egyetlen országnak sem lehet egyszerre csatahajója is és páncélos cirkálója is.

! c) Egy több mint kilenc ágyúval rendelkező hajó nem csatázhat egy kilencnél kevesebb ágyúval rendelkező hajóval úgy, hogy az előbbi elsüllyedjen.

! d) Egy hajó nem lehet felavatva mindaddig, amíg a hajóosztályának a nevét viselő hajó nincs felavatva.

! e) Minden hajóosztályhoz tartozik egy hajó, amelyik a hajóosztály nevét viseli.

6.4.6. feladat: Adjuk meg a következő megszorításokat sorra vonatkozó `CHECK` feltételek formájában:

a) Egyik hajóosztálynak sem lehetnek 16 hüvelyknél nagyobb kaliberű ágyúi.

b) Ha egy hajóosztályban az ágyúk száma több mint 9, akkor azok kalibere nem lehet nagyobb 14 hüvelyknél.

! c) Egyik hajó sem csatázhat a felavatása előtt.

6.5. Megszorítások módosítása

A megszorításokat bármikor módosíthatjuk, törölhetjük vagy újakat hozhatunk létre belőlük. Hogy ezeket a módosításokat pontosan hogyan kell megadnunk az attól függ, hogy a megszorítás értéktartományhoz, attribútumhoz, táblához vagy adatbázissémához van-e hozzárendelve.

6.5.1. Megszorítások elnevezése

Ahhoz, hogy módosítani vagy törölni tudjunk egy létező megszorítást, az szükséges, hogy a megszorításnak neve legyen. Az önálló megszorításoknak, amelyek az adatbázisséma részei, mindig van nevük, amit a `CREATE ASSERTION` utasítás részeként adunk meg. A többi fajta megszorításnak szintén adhatunk nevet. Ezt úgy tehetjük meg, hogy megszorítás elé beírjuk a `CONSTRAINT` kulcsszót és a megszorítás nevét.

6.12. példa: Még az elsődleges kulcs vagy az idegen kulcs deklarációhoz is megadhatunk nevet. Például a 6.1. ábra 2. sora helyett, ami azt mondja ki, hogy a `név` attribútum elsődleges kulcs, a következőt írhatjuk:

```
2) név CHAR(30) CONSTRAINT NévKulcs PRIMARY KEY,
```

Hasonlóképpen nevezhetjük el a 6.6. példában szereplő attribútumra vonatkozó `CHECK` feltételt:

```
4) nem CHAR(1) CONSTRAINT FérfiVagyNő
                    CHECK (nem IN ('F', 'N')),
```

A 6.8. példában szereplő, értéktartományra vonatkozó megszorításnak a következőképpen adhatunk nevet:

```
CREATE DOMAIN AzonÉrtékek INT
       CONSTRAINT Hatjegyű CHECK (VALUE>=100000);
```

Végül pedig a következő módon nevezhetjük el a 6.4. ábra 6. sorában szereplő, sorra vonatkozó CHECK feltételt:

```
6) CONSTRAINT Titulus
          CHECK (nem='N' OR név NOT LIKE 'Ms.\%')
```

□

6.5.2. Táblákra vonatkozó megszorítások megváltoztatása

Egy értéktartományhoz, attribútumhoz vagy táblához hozzárendelt megszorítások halmazát megváltoztathatjuk az ALTER utasítással. Az 5.7.4. alfejezetben már tárgyaltuk az ALTER TABLE utasítás néhány esetét, amellyel attribútumokat adhattunk hozzá egy táblához, illetve attribútumokat törölhettünk belőle. Az 5.7.6. alfejezetben megvizsgáltuk az ALTER DOMAIN utasítást, amellyel ott az alapértelmezés szerinti értéket változtattuk meg.

Ezekkel a fenti utasításokkal a megszorításokat is megváltoztathatjuk. Az attribútumra és a sorra vonatkozó CHECK feltételeket az ALTER TABLE utasítás segítségével változtathatjuk meg. Egy megszorítás törlése úgy történik, hogy az utasításon belül megadjuk a DROP kulcsszót és a megszorítás nevét. Egy újabb megszorítás megadásához az ADD kulcsszót kell megadnunk, majd magát a megszorítást.

6.13. példa: Nézzük meg, hogyan törölhetnénk majd vihetnénk fel újra a 6.12. példában szereplő megszorításokat. Elsőként töröljük azt a megszorítást, amelyik azt mondja ki, hogy a FilmSzínész táblában a név attribútum elsődleges kulcs:

```
ALTER TABLE FilmSzínész DROP CONSTRAINT NévKulcs;
```

Ugyanezen relációnak azt az attribútumra vonatkozó CHECK feltételét, ami a nem attribútum értékeit korlátozza a következő utasítással töröljük:

```
ALTER TABLE FilmSzínész DROP CONSTRAINT FérfiVagyNő;
```

Végül azt a megszorítást, hogy csak a nőnemű színészek neve kezdődhet 'Ms.'-sel a következőképpen törölhetjük:

Nevezzük el a megszorításokat

Jegyezzük meg, hogy célszerű a megszorításoknak nevet adni még akkor is, ha úgy gondoljuk, hogy soha nem fogunk azokra hivatkozni. Ha egyszer létrehoztuk a megszorítást név nélkül, később már nem fogunk tudni nevet adni neki akkor sem, ha bármilyen módon változtatni szeretnénk rajta.

```
ALTER TABLE FilmSzínész DROP CONSTRAINT Titulus;
```

Ha szeretnénk ismét érvényre juttatni a fenti megszorításokat, ezt a FilmSzínész reláció sémájának megváltoztatásával a következőképpen tehetjük meg:

```
ALTER TABLE FilmSzínész ADD CONSTRAINT NévKulcs
   PRIMARY KEY (név);
ALTER TABLE FilmSzínész ADD CONSTRAINT FérfiVagyNő
   CHECK (nem IN ('F', 'N'));
ALTER TABLE FilmSzínész ADD CONSTRAINT Titulus
   CHECK (nem='N' OR név NOT LIKE 'Ms.\%');
```

Ezek a most újból létrehozott megszorítások mind sorra vonatkozó CHECK feltételek. Attribútumra vonatkozó CHECK feltételek formájában nem is tudnánk őket újból létrehozni, legfeljebb azt tehetnénk meg, hogy ha az attribútumok típusa értéktartomány lenne, akkor az azokra vonatkozó megszorításokat változtatnánk meg a FilmSzínész tábla megváltoztatása helyett.

Az újból létrehozott megszorításokra a megszorítás nevének megadása nem kötelező. Az SQL azonban nem jegyzi meg, hogy korábban melyik megszorítás melyik névhez tartozott. Ezért amikor ismét létrehozunk egy korábbi megszorítást, akkor ismét meg kell adnunk a megszorítás teljes leírását, és nem hivatkozhatunk rá pusztán csak a korábbi nevével. □

6.5.3. Értéktartományokra vonatkozó megszorítások megváltoztatása

Az értéktartományokra vonatkozó megszorításokat ugyanúgy törölhetjük, illetve ugyanúgy vihetünk fel újat belőlük, mint ahogyan azt a sorra vonatkozó CHECK feltételek esetében tettük. Egy értéktartományra vonatkozó megszorítás törléséhez az ALTER utasításban a DROP kulcsszót majd a megszorítás nevét kell megadnunk. Egy új megszorítás felviteléhez az ALTER utasításban az ADD kulcsszót, a megszorítás nevét és a megszorítást definiáló CHECK feltételt kell megadnunk.

6.14. példa: Azt az értéktartományokra vonatkozó megszorítást, hogy az azonosítók legalább hatjegyűek legyenek, a következő utasítással törölhetjük:

```
ALTER DOMAIN AzonÉrtékek DROP CONSTRAINT HatJegyű;
```

Ugyanezt a megszorítást a következőképpen hozhatjuk ismét létre:

```
ALTER DOMAIN AzonÉrtékek ADD CONSTRAINT HatJegyű
  CHECK (VALUE >= 100000);
```

□

6.5.4. Önálló megszorítások megváltoztatása

Egy önálló megszorítást a `DROP ASSERTION` utasítással törölhetünk úgy, hogy a kulcsszavak után megadjuk a megszorítás nevét.

6.15. példa: A 6.10. példában szereplő önálló megszorítást a következőképpen törölhetjük:

```
DROP ASSERTION GazdagElnök;
```

A megszorítást úgy hozhatjuk ismét létre, hogy újból deklaráljuk azt, ahogyan a 6.10. példában tettük. □

6.5.5. Feladatok

6.5.1. feladat: Mutassuk meg, hogyan kell a film adatbázis relációsémáit módosítani a következő célok eléréséhez:

```
Film(cím, év, hossz, színes, stúdióNév, producerAzon)
SzerepelBenne(filmCím, év, színészNév)
FilmSzínész(név, cím, nem, születésnap)
GyártásIrányító(név, cím, azonosító, nettóBevétel)
Stúdió(név, cím, elnökAzon)
```

* a) Adjuk meg a `cím` és `év` attribútumokat a `Film` tábla kulcsaként.

b) Követeljük meg azt a hivatkozásiépség-megszorítást, hogy a filmek producereinek szerepelniük kell a `GyártásIrányító` táblában.

c) Írjuk elő, hogy egy film ne lehessen rövidebb 60 percnél és ne lehessen hosszabb 250 percnél.

*! d) Zárjuk ki, hogy egy név egyidejűleg előfordulhasson a filmszínészek és a gyártásirányítók között.

! e) Ne engedjük meg, hogy két stúdiónak azonos legyen a címe.

6.5.2. feladat: Mutassuk meg, hogy hogyan kell a csatahajók adatbázissémáit megváltoztatni, hogy abban szerepeljenek a következő sorra vonatkozó `CHECK` feltételek:

```
Hajóosztályok(hajóosztály, típus, ország,
                ágyúkSzáma, kaliber, vízkiszorítás)
Hajók(név, hajóosztály, felavatva)
Csaták(név, dátum)
Kimenetelek(hajó, csata, eredmény)
```

a) A `hajóosztály` és `ország` legyen kulcsa a `Hajóosztályok` relációnak.

b) Követeljük meg azt a hivatkozásiépség-megszorítást, hogy a `Csaták` táblában szereplő hajók szerepeljenek a `Hajók` táblában is.

c) Írjuk elő azt a hivatkozásiépség-megszorítást, hogy a `Kimenetelek` táblában szereplő hajók szerepeljenek a `Hajók` táblában is.

d) Írjuk elő, hogy egy hajónak se lehessen 14-nél több ágyúja.

! e) Zárjuk ki, hogy egy hajó a felavatása előtt csatában vehessen részt.

6.6. Triggerek az SQL3-ban

Az e fejezetben tanulmányozott megszorítások különböző formái mind az SQL2 szabványt követték. Ezek olyan végrehajtási modellel rendelkeztek, amelyben a rendszer akkor hajtja végre az ellenőrzést, ha az az elem, amire a megszorítás korlátozást fogalmaz meg, megváltozik. Például egy attribútumra vonatkozó `CHECK` feltételt akkor ellenőriz a rendszer, ha az attribútum értéke valamelyik sorban megváltozik. Ez előfordulhat egy új sor beszúrásával is.

A megszorítások megvalósítása magában foglalja az ellenőrzések kiváltását, vagy más szóval „elsütését", megfelelő események hatására. Ezért természetes kérdésként merül fel, hogy az elsütő események kiválasztását nem lehetne-e a rendszer helyett az adatbázis-programozóra bízni. Ez a megközelítés további lehetőségeket adna a felhasználónak, hogy az adatbázis-műveletek elsütésével ne csak a megszorítások megsértését akadályozhassa meg, hanem más célokat is megvalósíthasson. Ezért a tervezett SQL3 szabványban már a triggerek is szerepelnek. (trigger = elsüt, kivált) Ezek nagyon hasonlítanak a megszorításokra, de konkrétan megadott elsütő eseményekkel, és a feltétel eredményétől függő, konkrétan megadott végrehajtandó utasításokkal rendelkeznek. Érdekes módon a jelenleg forgalomban lévő rendszerek az aktív elemek tekintetében általában közelebb állnak az SQL3-hoz, mint az SQL2-höz. Ennek egyik lehetséges magyarázata az, hogy a forgalmazó cégek számára a triggerek megvalósítása bizonyos értelemben könnyebb, mint az önálló megszorításoké.

6.6.1. Triggerek és megszorítások

A *triggerek*, amelyeket szokás *esemény-feltétel-művelet szabályoknak* is nevezni, a korábban tárgyalt megszorításoktól három dologban térnek el.

1. A triggereket a rendszer csak akkor ellenőrzi, ha bizonyos, az adatbázis-programozó által megadott *események* bekövetkeznek. A megengedett események általában egy adott relációra vonatkozó beszúrás, törlés és módosítás. Egy másik fajta esemény, amit sok SQL rendszer megenged, a tranzakció befejeződése. (A tranzakciókat, amelyek tulajdonképpen elemi munkaegységek, a 7.2. alfejezetben tárgyaljuk majd.)
2. A kiváltó esemény azonnali megakadályozása helyett a trigger először egy *feltételt* vizsgál meg. Ha a feltétel nem teljesül, akkor a kiváltó eseményre válaszul a triggerrel összefüggésben semmi nem fog történni.
3. Ha a trigger feltétele teljesül, akkor a rendszer végrehajtja a triggerhez tartozó *műveletet*. Ez a művelet ezután megakadályozhatja a kiváltó esemény megtörténtét, vagy meg nem történtté teheti azt (pl. kitörölheti az éppen felvitt sorokat). A megadott művelet adatbázis-műveleteknek egy tetszőleges sorozata lehet, akár olyan műveleteket is tartalmazhat, amelyeknek semmi köze nincs a kiváltó eseményhez.

A következőkben áttekintjük az SQL3 triggereit, majd röviden átnézzük azokat a kiterjesztéseket, amelyekkel az SQL3 bővítette az SQL2 önálló megszorításait. Ezek a kiterjesztések szintén a triggerek néhány tulajdonságát foglalják magukba.

6.6.2. Az SQL3 triggerei

Az SQL3 trigger utasítása számos különböző lehetőséget kínál a felhasználónak az eseményre, a feltételre és a műveletre vonatkozó részében. Az alábbiakban felsoroljuk a legfontosabbakat.

1. A műveletet végrehajthatjuk a kiváltó esemény előtt, után vagy helyette.
2. A művelet hivatkozhat a műveletet kiváltó esemény által törölt, beszúrt vagy módosított sorok régi és új értékeire is.
3. Ha az esemény módosítás, akkor megadhatunk egy bizonyos oszlopot vagy oszlopok egy halmazát, amelyre az esemény vonatkozik.
4. Egy `WHEN` záradékban megadhatunk egy feltételt, és a műveletet csak akkor hajtja végre a rendszer, ha a szabály kiváltódik és a kiváltó esemény bekövetkezésekor a feltétel igaz.


```
1) CREATE TRIGGER NetBevétTrigger
2) AFTER UPDATE OF nettóBevétel ON GyártásIrányító
3) REFERENCING
4) OLD AS RégiSor,
5) NEW AS ÚjSor
6) WHEN(RégiSor.nettóBevétel > ÚjSor.nettóBevétel)
7) UPDATE GyártásIrányító
8) SET nettóBevétel = RégiSor.nettóBevétel
9) WHERE azonosító = ÚjSor.azonosító
10)FOR EACH ROW
```

6.7. ábra. *Egy SQL3-beli trigger*

5. A programozó megadhatja, hogy a művelet végrehajtása a következő két lehetőség közül melyik módon történjen meg:

   a) Minden módosított sorra egyszer, vagy

   b) Egy adatbázis-művelet által módosított összes sorra vonatkozóan egyszer.

Mielőtt a triggerek szintaktikai részleteire rátérnénk, nézzünk meg egy példát, amelyik rávilágít a legfontosabb szintaktikai és szemantikai szempontokra. A példában a trigger minden módosított sorra külön végrehajtásra kerül.

6.16. példa: Megadunk egy SQL3 szerinti triggert, amelyik a

`GyártásIrányító(név, cím, azonosító, nettóBevétel)`

táblára vonatkozik. A kiváltó esemény a `nettóBevétel` attribútum módosítása. A trigger hatása az lesz, hogy ha valaki megpróbálja csökkenteni a stúdióelnökök nettó bevételét, akkor a művelet a módosítás előtti állapotot hozza ismét létre. A trigger deklarációját a 6.7. ábrán láthatjuk.

Az 1. sor a `CREATE TRIGGER` kulcsszót és a trigger nevét tartalmazza. A 2. sor adja meg a kiváltó eseményt, ami most a `GyártásIrányító` tábla `nettóBevétel` attribútumának módosítása. A 3. sortól az 5. sorig terjedő rész lehetővé teszi a triggerbeli feltétel és művelet rész számára, hogy hivatkozni tudjanak a módosítás előtti régi, és a módosítás utáni új értékekre is. A módosítás előtti sorokra `RégiSor` néven, a módosítás utániakra `ÚjSor` néven lehet majd hivatkozni, ahogy azt a 4. és 5. sorban láthatjuk. A feltételben és a műveletben ezekre ugyanúgy hivatkozhatunk, mintha azok egy szokásos SQL lekérdezés `FROM` záradékában megadott sorváltozók lennének.

A 6. sor a trigger feltétel része. Azt mondja ki, hogy a műveletet csak akkor fogjuk végrehajtani, ha a régi nettó bevétel nagyobb, mint az új, vagyis az illető személy nettó bevétele csökkent.

A 7.-től a 9.-ig tartó sorok alkotják a trigger művelet részét. Ezek a sorok egy szabályos SQL-beli `UPDATE` utasítást alkotnak, amely utasításnak az a hatása, hogy az adott személy nettó bevételét visszaállítja a módosítás előtti értékre. Figyeljük meg,

hogy az utasítás a GyártásIrányító tábla összes sorát érintené, de a 9. sorban szereplő WHERE záradék biztosítja, hogy csak az a korábbi módosításban szereplő sor legyen érintve, amelyik a megfelelő azonosítóval rendelkezik.

Végül a 10. sor fejezi ki azt a követelményt, hogy a szabály kiváltása minden egyes sorra külön-külön történjen meg. Ha ezt a sort elhagynánk, akkor a trigger csak egyszer aktivizálódna egy SQL utasítás hatására, függetlenül attól, hogy az adott utasítás hány triggerkiváltó módosítást végzett a sorokban. □

Természetesen a 6.16. példa csak néhány sajátosságát mutatja be az SQL3 triggereinek. A következőkben azokat a lehetőségeket fogjuk felvázolni, amelyeket a triggerek kínálnak a számunkra, és megmutatjuk azt is, hogy hogyan lehet ezeket a lehetőségeket kifejezni.

- A 6.7. ábra 2. sora azt mondja ki, hogy a szabályban szereplő műveletet a kiváltó esemény után kell végrehajtani. Ezt az AFTER kulcsszó jelzi. Az AFTER kulcsszó helyett a következőket használhatnánk még:

    1. BEFORE. Ebben az esetben a WHEN után megadott feltételt a kiváltó esemény előtt vizsgálja meg a rendszer. Ha a feltétel igaz, akkor a trigger művelete végrehajtásra kerül. Majd ezután hajtja végre a rendszer a triggert kiváltó eseményt, függetlenül a feltétel igaz vagy hamis voltától.

    2. INSTEAD OF. Ha a WHEN után megadott feltétel igaz, akkor a trigger művelet része végrehajtásra kerül, magát a kiváltó eseményt azonban semmiképpen nem hajtja végre a rendszer.

- A módosítás (UPDATE) mellett további lehetséges kiváltó események még a beszúrás (INSERT) és a törlés (DELETE). A 6.7. ábra 2. sorában szereplő „OF nettóBevétel" záradék opcionálisan megadható, és a megadása azt jelenti, hogy csak az OF kulcsszó után felsorolt attribútumok módosítása (UPDATE) számít kiváltó eseménynek. Az OF záradék beszúrás (INSERT) és törlés (DELETE) események esetén nem adható meg, hiszen ezek az események mindig teljes sorokra vonatkoznak.

- A példában csak egyetlen SQL utasítás szerepelt a művelet részben, de megadható ott több ilyen utasítás is pontosvesszőkkel elválasztva.

- Ha a kiváltó esemény módosítás, akkor a módosítás előtti sort régi sornak, a módosítás utánit pedig új sornak tekinthetjük. Ezeknek a soroknak nevet adhatunk az OLD AS, illetve a NEW AS záradékok segítségével, ahogyan az a 4. és 5. sorban látható. Ha a kiváltó esemény beszúrás, akkor a beszúrt sornak a NEW AS záradék segítségével adhatunk nevet, az OLD AS záradék viszont ilyenkor nem használható. Törlés esetén az OLD AS záradék segítségével nevezhetjük el a törölt sort, ilyenkor viszont a NEW AS záradék nem használható.

- Ha elhagyjuk a 10. sorban szereplő FOR EACH ROW megkötést, akkor egy *sor szintű triggerből* (mint amilyen például a 6.7. ábrán is szerepel) *utasítás szintű trigger* lesz. Egy utasítás szintű trigger egyszer kerül végrehajtásra minden olyan utasításra, amelyik egy vagy több kiváltó eseményt generál. Ha például egy egész táblát módosítunk egy SQL-beli UPDATE utasítással, akkor egy utasítás szintű trigger egyszer fog végrehajtódni, míg egy sor szintű trigger minden sorra végrehajtásra kerül. Egy utasítás szintű triggerben nem hivatkozhatunk közvetlenül a régi és az új sorokra, ahogy azt a 4. és 5. sorban láttuk. Ilyenkor a régi sorok halmazára – ezek a törölt sorok, illetve a módosított sorok régi verziói – és az új sorok halmazára – ezek a beszúrt sorok, illetve a módosított sorok új verziói – úgy hivatkozhatunk, mint két relációra. A 6.7. ábra 4. és 5. sora helyett ilyen deklarációkat használhatnánk, mint OLD_TABLE AS RégiAdat és NEW_TABLE AS ÚjAdat. A RégiAdat annak a relációnak a neve, amelyik az összes régi sort tartalmazza, az ÚjAdat pedig az új sorokat tartalmazó relációé.

6.17. példa: Tegyük fel, hogy azt szeretnénk megakadályozni, hogy a gyártásirányítók nettó bevételének átlaga 500 000 dollár alá csökkenjen. Ezt a

```
GyártásIrányító(név, cím, azonosító, nettóBevétel)
```

táblára vonatkozó megszorítást egy beszúrás, egy törlés, vagy a nettóBevétel oszlop módosítása sértheti meg. Mind a háromtípusú eseményre egy triggert kell írnunk. A 6.8. ábrán a módosításra vonatkozó triggert láthatjuk. A beszúrásra és a törlésre vonatkozó triggerek hasonlóak, sőt még egy kicsit egyszerűbbek is.

A 3.-tól az 5.-ig tartó sorok azt adják meg, hogy annak a relációnak a neve, amelyik a régi sorokat tartalmazza: RégiAdat, annak a neve pedig, amelyik az új sorokat tartalmazza: ÚjAdat. Itt most az, hogy régi, illetve új, azzal az adatbázisművelettel kapcsolatban értendők, amelyik kiváltotta a triggert. Egy adatbázisművelet egy relációnak több sorát is módosíthatja, és ezért egy ilyen művelet végrehajtása után a RégiAdat és az ÚjAdat tábláknak nagyon sok sora is lehet.

Ha a kiváltó művelet módosítás, akkor a RégiAdat tábla a sorok módosítás előtti régi verzióját, az ÚjAdat tábla pedig a sorok módosítás utáni új verzióját tartalmazza. Ha törlés műveletre írnánk egy hasonló triggert, abban az esetben a törölt sorok lennének a RégiAdat táblában, és az ÚjAdat táblára vonatkozó deklaráció nem szerepelne a trigger leírásában. Beszúrás műveletre megírt triggerben pedig a beszúrt sorok lennének az ÚjAdat táblában, és a RégiAdat táblára vonatkozó deklaráció maradna el.

A 6.-tól a 8.-ig tartó sorok írják le a feltételt. A megadott feltétel akkor lesz igaz, ha a módosítás után az átlagos nettóbevétel legalább 500 000 dollár. Figyeljük meg, hogy a 8. sorban szereplő kifejezés a GyártásIrányító táblának azt a változatát adja meg, ami a módosítás után jönne létre.

Mivel a 2. sorban az INSTEAD OF kulcsszót adtuk meg, ezért a rendszer a nettóBevétel oszlop módosításaira tett kísérleteket megakadályozza és ezeket a módosításokat nem hajtja végre. Helyette a triggerben megadott feltétel alapján dönti el, hogy mit kell tennie. A példánkban, ha a módosítás után a gyártásirányítók átlagos

```
1) CREATE TRIGGER ÁtlagNetBevétTrigger
2) INSTEAD OF UPDATE OF nettóBevétel ON GyártásIrányító
3) REFERENCING
4)    OLD_TABLE AS RégiAdat,
5)    NEW_TABLE AS ÚjAdat

6) WHEN(500000 <=
7)    (SELECT AVG(nettóBevétel)
8)    FROM ((GyártásIrányító EXCEPT RégiAdat) UNION ÚjAdat)
   )
9) DELETE FROM GyártásIrányító
10) WHERE (név, cím, azonosító, nettóBevétel) IN RégiAdat;

11) INSERT INTO GyártásIrányító
12) (SELECT * FROM ÚjAdat);
```

6.8. ábra. *Az átlagos nettó bevétel megszorítása*

nettó bevétele még mindig legalább 500 000 dollár lenne, akkor a művelet részben megadott utasítások biztosítják, hogy a végeredmény ugyanaz legyen, mintha a módosítás hajtódott volna végre. Nevezetesen a 9. és 10. sorokban megadott utasítás kitörli azokat a sorokat, amelyeket a módosítás módosított volna, a 11. és 12. sorokban megadott utasítás pedig beszúrja ezeknek a soroknak az új változatát. □

6.6.3. Önálló megszorítások az SQL3-ban

Az SQL3 két fontos további lehetőséggel bővíti ki az SQL2-beli önálló megszorításokat.

1. Az önálló megszorításokat a programozó által megadott események váltják ki, és nem azok az események, amelyekről a rendszer úgy gondolja, hogy megsérthetik a megszorítást.

2. Az önálló megszorítás egy tábla egyes soraira is hivatkozhat, ezáltal sor szintű ellenőrzéseket is végezhet, és nemcsak a tábla vagy táblák egészére vonatkozót.

6.18. példa: A 6.10. példában szereplő `GazdagElnök` nevű önálló megszorítást láthatjuk a 6.9. ábrán, az SQL3-beli jelölésekkel megadva. Az 1. sorban a szokásos módon kezdődik a deklaráció. A 2.-tól a 6.-ig tartó sorokban találjuk azokat az eseményeket, amelyek kiválthatják a megszorítás ellenőrzését.

Ahhoz, hogy minden lehetséges adatbázis-módosítást figyelembe vegyünk, ami megsértheti a 7.–9. sorokban szereplő megszorítást, ahhoz figyelnünk kell az új stúdióelnököket és a gyártásirányítók nettó bevételében bekövetkező változásokat. Ezért a 3. és 4. sor kimondja, hogy a megszorítást ellenőrizni kell, ha a `Stúdió` táblába egy új sort szúrunk, vagy ha egy stúdió elnökének az azonosítója módosul. Az 5. és 6. sor azt mondja ki, hogy akkor is ellenőrizni kell, ha egy gyártásirányító nettó bevétele


```
1) CREATE ASSERTION GazdagElnök
2) AFTER
3)    INSERT ON Stúdió,
4)    UPDATE OF elnökAzon ON Stúdió,
5)    UPDATE OF nettóBevétel ON GyártásIrányító,
6)    INSERT ON GyártásIrányító
7) CHECK(NOT EXISTS
8)    (SELECT * FROM Stúdió, GyártásIrányító
9)    WHERE elnökAzon = azonosító AND nettóBevétel < 10000000
      )
   )
```

6.9. ábra. *Egy SQL3-beli önálló megszorítás*

módosul, vagy ha egy új gyártásirányítót viszünk fel a táblába. Ez utóbbi két eset is a megszorítás megsértéséhez vezethet. Maga az ellenőrzendő feltétel a 7.–9. sorokban szerepel, és ez lényegében ugyanaz, mint ami a 6.10. példában szerepelt. □

A legfontosabb különbség az SQL3-beli megközelítés és az SQL2-beli megközelítés között, hogy a 6.9. ábrán szereplő önálló megszorítás konkrétan megadja, hogy melyek azok az esetek, amikor a megszorítást ellenőrizni kell. Ebből adódik, hogy az SQL3-beli megszorítást könnyebb megvalósítani a rendszer készítőinek, viszont nehezebb használni a felhasználónak. Most ugyanis a felhasználónak kell a következőkről gondoskodnia:

1. Neki kell összegyűjtenie az összes olyan eseményt, ami a megszorítás ellenőrzését kiválthatja.

2. Ő viseli annak kockázatát, hogy az adatbázis következetlen állapotba kerül, ha az eseményeket nem megfelelően választotta ki.

6.6.4. Feladatok

6.6.1. feladat: Írjunk a 6.8. ábra triggeréhez hasonló, de SQL3-beli triggert a `GyártásIrányító` tábla beszúrás és törlés műveletére.

6.6.2. feladat: Írjuk meg a következő SQL3-beli triggereket vagy önálló megszorításokat a 4.1.1. feladat PC adatbázisára vonatkozóan.

```
Termék(gyártó, modell, típus)
PC(modell, sebesség, memória, merevlemez, CD, ár)
Laptop(modell, sebesség, memória, merevlemez,
        képernyő, ár)
Nyomtató(modell, szín, típus, ár)
```

* a) Amikor egy PC árát módosítjuk, ellenőrizzük, hogy nincs ugyanolyan sebességű, de kisebb árú PC.

b) Amikor egy új nyomtatót viszünk fel, ellenőrizzük, hogy az adott modellszám benne van a `Termék` táblában.

! c) A `Laptop` reláció bármilyen módosítása esetén, ellenőrizzük, hogy a laptopok átlagos ára gyártónként legalább 2000 dollár legyen.

! d) Amikor egy PC memóriáját vagy merevlemezét módosítjuk, ellenőrizzük, hogy a módosított PC-nek legalább 100-szor akkora a merevlemeze, mint a memóriája.

! e) Amikor egy új PC-t, laptopot vagy nyomtatót viszünk fel, ellenőrizzük, hogy az adott modellszám még nem szerepel a `PC`, `Laptop` vagy `Nyomtató` táblában.

6.6.3. feladat: A 4.1.3. feladatban szereplő adatbázissémára vonatkozóan írjunk egy vagy több SQL3 triggert vagy önálló megszorítást a következő célok megvalósítására:

```
Hajóosztályok(hajóosztály, típus, ország,
              ágyúkSzáma, kaliber, vízkiszorítás)
Hajók(név, hajóosztály, felavatva)
Csaták(név, dátum)
Kimenetelek(hajó, csata, eredmény)
```

* a) Ha egy új hajóosztályt viszünk fel a `Hajóosztályok` táblába, vigyünk fel egy hajót is, amelyiknek ugyanaz a neve, mint a hajóosztálynak, és a felavatási dátuma legyen `NULL` érték.

b) Ha egy új hajóosztályt viszünk fel, amelynek a vízkiszorítása nagyobb 35 000 tonnánál, akkor engedjük meg a beszúrást, de a vízkiszorítást változtassuk 35 000 tonnára.

! c) Ha egy új sort viszünk fel a `Kimenetelek` táblába, ellenőrizzük, hogy a hajó és a csata benne van-e a `Hajók`, illetve a `Csaták` táblában, és ha nincs, akkor szúrjuk be a megfelelő sort egyik vagy mindkét táblába, a hiányzó oszlopokba `NULL` értéket téve.

! d) Ha új sorokat viszünk fel a `Hajók` táblába, vagy a tábla `hajóosztály` attribútumát módosítjuk, ellenőrizzük, hogy egy országnak se lehessen több mint 20 hajója. Ha a felvitel ezt a feltételt megsértené, akkor ne engedjük meg a végrehajtását.

!! e) Ellenőrizzük minden lehetséges esetben, ami a feltételt megsértheti, hogy egy hajó nem vehet részt egy csatában, ha egy korábbi dátumú csatában őt elsüllyesztették. Ha e feltételt egy módosítás megsértene, akkor utasítsuk vissza a módosítást.

! **6.6.4. feladat:** Írjuk meg a következőket SQL3 triggerek vagy SQL3 önálló megszorítások formájában. A feladatok a film adatbázis tábláira vonatkoznak.

```
Film(cím, év, hossz, színes, stúdióNév, producerAzon)
SzerepelBenne(filmCím, év, színészNév)
FilmSzínész(név, cím, nem, születésnap)
GyártásIrányító(név, cím, azonosító, nettóBevétel)
Stúdió(név, cím, elnökAzon)
```

Feltételezhetjük, hogy a kívánt feltétel minden esetben teljesül mielőtt megpróbáljuk módosítani az adatbázist. Ha lehetséges, akkor az utasítás visszautasítása helyett inkább módosítsuk az adatbázist, még akkor is, ha ez `NULL` értékek vagy alapértelmezés szerinti értékek felvitelével jár.

a) Biztosítsuk azt a feltételt, hogy a `SzerepelBenne` relációban lévő színészek benne vannak a `FilmSzínész` relációban is.

b) Biztosítsuk azt a feltételt, hogy minden gyártásirányító egyben egy stúdió elnöke is, egy film producere is, vagy mindkettő egyidejűleg.

c) Biztosítsuk azt a feltételt, hogy minden filmnek legalább egy férfi és egy női szereplője is van.

d) Biztosítsuk azt a feltételt, hogy egy stúdió egy évben nem készíthet száznál több filmet.

e) Biztosítsuk azt a feltételt, hogy az ugyanabban az évben készült filmek hosszának átlaga nem lehet több mint 120 perc.

! **6.6.5. feladat:** A 6.17. példában a nem megengedhető módosításokat úgy kezeltük le, hogy először elvégeztük az ellenőrzést, majd ha a módosítás nem sértette meg a feltételt, akkor azt végrehajtottuk. Egy másik megközelítés lehetne, hogy megengedjük a módosítás végrehajtását, és ha az megsérti a feltételt, akkor visszaállítjuk az eredményt. Írjuk meg így a triggert.

6.7. Összefoglalás

- *Kulcsfeltétel:* Egy attribútumot vagy attribútumok egy halmazát kulcsként deklarálhatjuk a `UNIQUE` vagy a `PRIMARY KEY` kulcsszóval a relációséma megadásakor.
- *Hivatkozásiépség-megszorítások:* Előírhatjuk, hogy egy attribútum vagy egy attribútumhalmaz értékeinek elő kell fordulnia egy másik reláció valamelyik sorának elsődleges kulcs attribútumában vagy attribútumaiban. Ezt a relációséma megadásakor a `REFERENCES` vagy a `FOREIGN KEY` kulcsszóval adhatjuk meg.

- *Attribútumra vonatkozó CHECK feltételek:* Egy attribútum értékeire vonatkozó megszorítást előírhatunk úgy, hogy a CHECK kulcsszót és az ellenőrizendő feltételt megadjuk a relációsémában az attribútum deklarációja után. Egy másik lehetőség, hogy az attribútum típusaként egy értéktartományt adunk meg, és az ellenőrizendő feltételt az értéktartomány deklarációjában adjuk meg.
- *Sorra vonatkozó CHECK feltételek:* Egy reláció soraiban szereplő értékekre adhatunk meg ellenőrizendő feltételeket úgy, hogy a reláció deklarációja után megadjuk a CHECK kulcsszót és a feltételt.
- *Önálló megszorítások:* Önálló megszorítást az adatbázisséma elemeként deklarálhatunk a CHECK kulcsszónak és az ellenőrizendő feltételnek a megadásával. A feltétel vonatkozhat az adatbázisséma egy vagy több relációjára, és hivatkozhat a reláció egészére – például aggregátor függvények esetén –, vagy az egyes sorokra.
- *Az ellenőrzések kezdeményezése:* Az önálló megszorításokat a rendszer akkor ellenőrzi, amikor a feltételben szereplő relációk valamelyikére vonatkozó változtatás előidézheti a feltétel megsérülését. Az attribútumra és a sorra vonatkozó CHECK feltételeket a rendszer csak akkor ellenőrzi, ha az az attribútum vagy az a reláció, amire a feltétel vonatkozik, beszúrás vagy módosítás utasítás hatására változik meg. Így ez utóbbi megszorítások megsérülhetnek, ha olyan alkérdéseket is tartalmaznak, amelyek másik relációra vagy az adott relációnak más soraira hivatkoznak.
- *SQL3 triggerek:* A tervezett SQL3 szabvány a triggereket is tartalmazza, amelyek megadhatnak bizonyos eseményeket, amelyek életre hívják őket. Ilyen esemény lehet a beszúrás, a módosítás vagy a törlés. Amikor a trigger kiváltódik, a rendszer egy feltétel teljesülését ellenőrzi, amelynek igaz volta esetén egy megadott, SQL utasításokból álló sorozat hajtódik végre.
- *SQL3-beli önálló megszorítások:* Az SQL3 szabvány önálló megszorítása az SQL2-belitől eltér kissé. Ezeket az önálló megszorításokat hasonlóan az SQL3-beli triggerekhez egy vagy több esemény kelti életre. Ilyen esemény lehet például egy relációba való beszúrás. Az aktivizálódása után az SQL3-beli önálló megszorítás egy feltételt ellenőriz, ami a relációkra vagy a sorokra vonatkozhat, és ha a feltétel nem teljesül, visszautasítja az őt kiváltó módosító utasítást.

6.8. Irodalomjegyzék

Az 5. fejezet Irodalomjegyzékében már adtunk arra vonatkozóan információt, hogy hol érhetők el az SQL2 és SQL3 szabványok leírásai. [4] információforrásul szolgálhat az adatbázisrendszerek aktív elemeivel kapcsolatosan szinte minden kérdésben. [1] a legújabb véleményeket tárgyalja az aktív elemek SQL3-ban és a jövőbeni szab-

ványokban elfoglalt helyéről. [2] és [3] egy HiPAC nevű korai prototípus rendszert mutat be, amelyik aktív elemek használatát tette lehetővé.

1. Cochrane, R. J., H. Pirahesh, N. Mattos, „Integrating triggers and declarative constraints in SQL database systems", *Intl. Conf. on Very Large Database Systems*, pp. 567–579, 1996.
2. Dayal, U., és mások „The HiPAC project: combining active databases and timing constraints", *SIGMOD Record* 17:1, pp. 51–70, 1988.
3. McCarthy, D. R., U. Dayal, „The architecture of an active database management system", *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 215–224, 1989.
4. Widom, J., S. Ceri, *Active Database Systems*, Morgan-Kaufmann, San Francisco, 1996.

7. fejezet

Rendszerelemek az SQL-ben

Ebben a fejezetben az SQL2 szabvány alapján megvizsgáljuk, hogy hogyan illeszthető be az SQL a hagyományos programfejlesztési környezetekbe. A 7.1. alfejezetben megismerjük, hogy az SQL-t leggyakrabban közönséges programozási nyelveken – például C programnyelven – elkészített programokba illesztve alkalmazzák[1]. Ennek támogatására az SQL számos adatcsere-lehetőséget biztosít az SQL parancsokat beágyazó program változói és az SQL parancsokban megjelenő adatelemek között.

A 7.2. pont bevezeti a tranzakció fogalmát munkafolyamatok oszthatatlan részeinek leírására. Számos adatbázis-kezelési alkalmazás elkészítése során szükséges lehet, hogy több részműveletből összerakott összetett műveletek végrehajtása oszthatatlan módon történjen még akkor is, ha a rendszerben egyidejűleg több párhuzamosan futó aktív folyamat lehet (a tankönyvekben kézenfekvő példaként szokták felhozni a banki szoftverek ilyen irányú igényeit). Az SQL nyelv lehetőséget biztosít számunkra adatbázist módosító műveletek sorozatából tranzakciók kialakítására, az SQL-t támogató adatbázisrendszerek biztosítják a szükséges mechanizmusokat ezeknek a tranzakcióknak az oszthatatlan végrehajtásához.

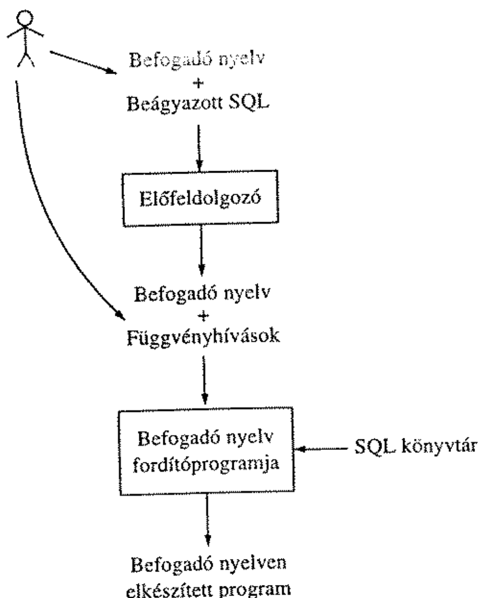
A 7.3. alfejezetben további rendszerelemeket ismerhetünk meg, például a kliens/szerver alkalmazáskészítési modell adatbázisokkal kapcsolatos részleteiről. Majd a 7.4. alfejezetben bemutatjuk, hogy az SQL milyen eszközökkel segít a jogosulatlan információlekérdezés megakadályozásában, és milyen módon rögzíthető az, hogy ki milyen információk elérésére jogosult.

[1] A fordító megjegyzése: ezeket a programozási nyelveket gyakran nevezik harmadik generációs nyelveknek.

7.1. Az SQL használata programozási környezetekben

Az eddigiekben csak az igen ritkán alkalmazott ún. *direkt* SQL-lel foglalkoztunk, amikor azt feltételeztük, hogy SQL parancsainkat egy azok bekérésére és végrehajtására felkészített SQL interpreterrel hajtatjuk végre. A gyakorlati életben előforduló alkalmazások nagy részénél az SQL parancsok valamilyen szoftverfejlesztő eszközzel elkészített eljárások vagy függvények részeként lesznek felhasználva: ezeket az alkalmazásokat általában valamilyen ún. befogadó nyelven (például C nyelven) készítik, de egyes függvények vagy eljárások törzsében SQL utasításokat helyeznek el az adatbázisban tárolt információk elérésére. Ennek a résznek a témája az ezzel kapcsolatos ismeretek bemutatása.

A 7.1. ábra egy SQL utasításokat is tartalmazó alkalmazói program elkészítésének jellemzőbb lépéseit szemlélteti. Láthatjuk, hogy a programozó feladata az adatfeldolgozó algoritmusnak a befogadó nyelven való megfogalmazása, és a tényleges adatmanipulációért felelős ún. beágyazott SQL utasítások programba szerkesztése. Ezután az egész programot egy ún. előfeldolgozó programmal alakítják át úgy, hogy a programba beágyazott SQL utasításokat a befogadó nyelv utasításaira alakítják. A folyamat során létrehozott befogadó nyelvi utasítások meghívhatnak például olyan függvénye-



7.1. ábra. *Beágyazott SQL utasításokat is tartalmazó program feldolgozása*

Az SQL2 szabvány által támogatott programozási nyelvek

Az SQL2 implementációkkal szemben támasztott alapvető követelmény, hogy biztosítsák a csatlakozást az ADA, C, Cobol, FORTRAN, M (régebbi nevén Mumps), Pascal és PL/I nyelven írt programokhoz. Az informatikával foglalkozó olvasó számára a felsorolt programnyelvek a Mumps kivételével már ismertek lehetnek. A Mumps programozási nyelvet elsősorban kórházi alkalmazások fejlesztésére használják. Mi a példáinkban a C programozási nyelvet fogjuk használni.

ket, amelyek paraméterükben egy SQL utasítást várnak karakterlánc formában,[2] és képesek a paraméterben kapott SQL utasítások végrehajtására. Az ábrán bemutatjuk azt az esetet is, amikor a programozó közvetlenül ezeknek az SQL utasítások végrehajtására képes függvényeknek a hívását ágyazza be a programba: ekkor nincs szükség a forrásprogramnak az előfeldolgozó programmal való átalakítására.

Ezután az előfeldolgozó program által elkészített tisztán befogadó nyelven írt forrásprogramok az illető befogadó nyelven elkészített más programoknál megismert módon fordíthatók le, majd az elkészült modulokat össze kell szerkeszteni az adatbázis-kezelő rendszer gyártója által szállított ilyen célú programkönyvtárakkal (ezek a könyvtárak tartalmazzák azoknak az előbb említett függvényeknek az implementációját, amelyek képesek karakterlánc formájában leírt SQL utasítások végrehajtására).

7.1.1. Az SQL és a programozási nyelvek különbözőségéből eredő problémák

Az SQL nyelv adatmodellje lényegesen különbözik a hagyományos – gyakran a befogadó – programozási nyelv adatmodelljétől, mivel az SQL magját a relációs adatmodell képezi (például a halmaz fogalomra épülve), míg a hagyományos programozási nyelvek az elemi adattípusaikon és az egyszerűbb típuskonstrukciós eszközökön kívül nem biztosítanak más eszközöket az adatmodell definiálására. Például az igen gyakran használt C programozási nyelv sem támogatja nyelvi eszközökkel a halmaz fogalom alkalmazását, míg az SQL nem támogatja a tömb, mutató és több más programnyelvi elem alkalmazását. Ezért szükségesek az elkövetkezőkben ismertetett mechanizmusok, amelyek támogatják az SQL-t és a befogadó nyelven írt kódrészeket egyaránt tartalmazó alkalmazások fejlesztését.

Az olvasó első ránézésre azt is gondolhatná, hogy a legjobb megoldás az, hogy egyáltalán ne keverjük a programozási nyelveket: vagy végezzünk minden számítást

[2] A fordító megjegyzése: az előfeldolgozó program ilyenkor a programba ágyazott SQL utasítást esetleg kisebb átalakítások után egyszerűen átadhatja az illető SQL utasítást végrehajtó függvénynek. Az esetleges átalakítások, például a programozó által kijelölt SQL utasítás és az azt beágyazó programrész közti kommunikáció megszervezése érdekében szükségesek.

SQL-ben, vagy egyáltalán ne is használjuk az SQL-t. Könnyen látható, hogy ez az út nem járható, mivel ha szükségünk van adatbázisok elérésére, akkor erre az SQL egy jól használható hatékony és magas szintű eszköz. Az SQL-t használva a programozónak nem kell törődnie azzal, hogy hogyan szervezheti meg az adatok hatékony tárolását a rendelkezésre álló háttértáron. Az SQL kizárólagos alkalmazása ellen is szólnak komoly érvek, mivel vannak dolgok, amiket egyáltalán nem lehet SQL nyelven kifejezni. Például nem írhatunk egy olyan SQL lekérdezést, amely kiszámítja egy tetszőleges szám faktoriálisát [n faktoriálisa, azaz $n! = n \times (n-1) \times (n-1) \times \ldots \times 2 \times 1$]. Ilyen számítások például C, Cobol, vagy az előbb említett nyelvek bármelyikén könnyen elvégezhetők. Hasonlóan nem lehetséges SQL nyelven például a lekérdezések eredményének grafikus megjelenítését specifikálni, amit könnyen megtehetünk például C programozási nyelven elkészített programokban (az ehhez szükséges grafikus megjelenítési könyvtárak alkalmazásával). Látható, hogy a gyakorlati életben használt adatfeldolgozó programok elkészítéséhez szükség van mind az SQL, mind pedig a hagyományos ún. *befogadó* programozási nyelvekre.

7.1.2. Az SQL és a befogadó nyelv közötti interfész

Az adatbázis eléréséért felelős SQL utasítások és a befogadó nyelven megírt alkalmazói program utasításai közti adatcsere a befogadó nyelven deklarált olyan változókon keresztül történhet, amelyek elérésére és módosítására az SQL utasításokból is lehetőség van. Ezekre az ún. *megosztott elérésű változókra*[3]) a befogadó programnyelven írt programokból egyszerűen nevüknek leírásával hivatkozhatunk, míg a beágyazott SQL utasításokban nevük elé egy kettőspont karaktert kell írni (az SQL utasítást átalakító előfeldolgozó ez alapján különböztetheti meg őket az SQL nyelvű utasítások által elérhető adatbáziselemek neveitől).

A beágyazott SQL utasításokat a legtöbb rendszerben az utasítás elejére kiírt `EXEC SQL` kulcsszavakkal kell megjelölni, ezzel jelölve az említett előfordító programnak, hogy egy SQL utasítás következik, amit valahogyan át kell alakítania egy megfelelő könyvtári függvény meghívására.

Az SQL2 szabvány definiál egy `SQLSTATE` nevű változót az SQL és a befogadó nyelvi környezetek összekapcsolására.[4] Ez a változó általában öt karaktert tartalmaz (általában egy ötelemű karakteres tömb). Az adatbázis elérését támogató könyvtárak úgy vannak elkészítve, hogy visszatéréskor ebben a változóban helyezik el a végrehajtott SQL művelet során fellépett esetleges problémákat leíró kódokat. Például a `'00000'` (öt darab nulla számjegy) azt jelzi, hogy a függvény végrehajtása során nem léptek fel problémák, míg a `'02000'` kód egy lekérdezési művelet végrehajtása után

[3] A fordító megjegyzése: a megosztott elérésű változókra a továbbiakban egyszerűen a változó elnevezést fogjuk használni. Amennyiben ettől a konvenciótól valahol eltérnénk, úgy azt világosan jelezni fogjuk.

[4] Azokban a rendszerekben is találhatunk egy hasonló célú változót, amelyek nem követik az SQL2 szabvány előírásait, de ezeknél a rendszereknél ennek a változónak a neve lehet, hogy nem `SQLSTATE`.

azt jelezheti, hogy nincs a lekérdezésben megadott kritériumoknak eleget tevő sor az adatbázisban. A befogadó nyelven elkészített program ennek a változónak a tartalmától függően eldöntheti, hogy mit is kell tennie egy adott esetben.

7.1.3. A deklarációs rész

A változók deklarációját két beágyazott SQL utasítás közé kell tenni az alábbi minta szerint:

```
EXEC SQL BEGIN DECLARE SECTION;
   ...
EXEC SQL END DECLARE SECTION;
```

A fenti két SQL utasítás közti részt (amit a példában kipontoztunk) *deklarációs résznek* nevezzük. A deklarációs részben a változódeklaráció szintaxisa megegyezik a befogadó nyelven megszokott deklarációs szintaxissal, és mivel a változókat a befogadó nyelven is és az SQL-ből is el akarjuk érni, ezért csak olyan adattípussal deklarálhatjuk őket, amelyek mindkét nyelven egyaránt elérhetők (például egészek, valósak, karakterláncok, tömbök).

7.1. példa: Az alábbi utasítások előfordulhatnak például a `Stúdió` relációt módosító C függvényekben.

```
EXEC SQL BEGIN DECLARE SECTION;
   char stúdióNév[15], stúdióCím[50];
   char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
```

Az első és az utolsó (negyedik) utasítás a deklarációs rész kezdetét és végét jelzik. A középső két sor közül az első két változót definiál: a `stúdióNév` és a `stúdióCím` változókat. Ezek mindketten karakterekből álló tömb típusú változók: a későbbiekben a `Stúdió` relációba beszúrni kívánt stúdiónév és stúdiócím párokat fogjuk bennük tárolni. A harmadik sorban pedig az `SQLSTATE` változót egy hat elem hosszú karaktertömbnek definiáljuk[5]. □

[5] Említettük, hogy az SQL2 szabvány az `SQLSTATE` értékét öt karakter hosszon kezeli, de a példában láthatjuk, hogy számára hat karaktert foglaltunk le. Ezt azért tettük így, hogy ezt a változót a C szabványos könyvtár karakterláncokat kezelő függvényeivel is manipulálhassuk, amelyek a karakterláncok végén egy azt lezáró ASCII NULL (azaz '\0') karaktert várnak. Például az `strcmp` függvényt fogjuk arra használni, hogy megbizonyosodjunk arról, hogy az `SQLSTATE` értéke egyenlő-e egy bizonyos értékkel (például '`00000`'-val), így elágazhatunk a programunkban aszerint, hogy sikeres volt-e egy SQL utasítás végrehajtása vagy sem. Az `SQLSTATE` tömb hatodik karakterének általában be kell állítani a karakterlánc-lezáró '\0' karakter értéket, amit a példáinkban nem fogunk feltüntetni (ez egy egyszerű C nyelvű értékadás: `SQLSTATE[5]='\0';` alakú – a C nyelven a tömbök indexelése nullától kezdődik).

7.1.4. Változók használata

Az SQL utasításokban bármely konkrét érték helyére írhatunk változót (mint már említettük, a nevét kettősponttal megelőzve). A következő példában a 7.1. példában definiált változók értékét fogjuk beszúrni a `Stúdió` relációba.

7.2. példa: A 7.2. ábrán egy olyan C nyelvű – `beolvasStúdió` nevű – függvény vázlatát láthatjuk, amely a felhasználótól bekér egy stúdiónevet és egy stúdiócímet, és beszúrja a beadott adatokat a `Stúdió` táblába. Az első négy sor megegyezik a 7.1. példában bemutatott változódeklarációval. A példából kihagytuk a képernyőre író és a billentyűzetről olvasó sorokat (azok helyét egy megjegyzésben megadtuk). Az 5. és a 6. sorban egy `INSERT` SQL utasítást láthatunk, amit az említett `EXEC SQL` kulcsszavak vezetnek be azt jelölve, hogy ez nem egy szokásos C nyelvű utasítás, hanem egy átalakítandó SQL utasítás. Az említett előfordító program majd elvégzi a szükséges átalakításokat az `EXEC SQL` kulcsszavakkal bevezetett utasításokon. Az 5. és 6. sorokban megadott beszúrandó értékek nem szövegkonstansok, mint azt az SQL-ben azon a helyen írhatnánk (lásd például az 5.27. példát), hanem (megosztott elérésű) változók, amelyeknek az aktuális értéke lesz behelyettesítve az SQL utasítás kiértékelésekor. A program a 6. sorban megadott változók aktuális értékéből állít össze egy sort, amit beszúr a táblába. □

Az `INSERT` utasítás mellett számos más SQL utasítás is van, amely megosztott elérésű változókat használva is beágyazható programokba. Minden beágyazott SQL utasítás az `EXEC SQL` kulcsszóval kell kezdődjön, és a konstans értékek helyett bárhol hivatkozhatnak változókra. Az összes olyan SQL utasítás beágyazható, amelynek nincs visszatérési értéke (azaz, amely nem lekérdezés: beágyazhatók például a beszúrást, törlést, módosítást leíró utasítások, valamint az adatbázissémát manipuláló utasítások, amelyek például táblákat vagy nézettáblákat törölnek vagy hoznak létre).

A `SELECT` SQL lekérdezések általában nem ágyazhatóak be, mivel egy halmazt

```
void beolvasStúdió() {

   1) EXEC SQL BEGIN DECLARE SECTION;
   2)    char stúdióNév[15], stúdióCím[50];
   3)    char SQLSTATE[6];
   4) EXEC SQL END DECLARE SECTION;

      /* Kiírja a képernyőre azt a felhasználónak szóló
         utasítást, hogy adjon be egy nevet és egy címet,
         és a beadott válaszokat eltárolja a stúdióNév és
         stúdióCím változókba */

   5) EXEC SQL INSERT INTO Stúdió(név, lakcím)
   6)    VALUES (:stúdióNév, :stúdióCím);
}
```

7.2. ábra. *Változók alkalmazása egy új stúdió beszúrásánál*

adnak vissza eredményül, és a legtöbb programozási nyelv ezt nem tudja hogyan kezelni. A SELECT utasítások alkalmazására két megoldás is kínálkozik:

1. Az *egyetlen sort eredményező lekérdezések* a lekérdezés eredményeként létrejött eredménysort eltárolhatják akár változókba is oly módon, hogy az eredménysor egyes komponensei külön-külön változókba lesznek elhelyezve.

2. Egy olyan lekérdezés, amely egynél több sort (azaz egy sorhalmazt) is visszaadhat eredményül, csak úgy hajtható végre, ha egy *sormutatót* definiálunk hozzá. A sormutató majd befutja az eredményreláció összes sorát, és egy-egy eredménysor egyes komponenseit külön-külön változókba elhelyezve juttathatjuk el az adatokat a befogadó programnak.

A következő alfejezetekben mindkét fenti lehetőséget megvizsgáljuk.

7.1.5. Egyetlen sort eredményező lekérdezések

Egy alkalmazásokba beágyazható egyetlen sort eredményező lekérdezés formája annyiban különbözik az SQL nyelv elemeinek bemutatásakor ismertetett közönséges SELECT utasítástól, hogy a SELECT záradék után egy INTO kulcsszót kell írni, e kulcsszó mögé pedig fel kell sorolni azokat a változókat, amelyekbe a lekérdezés eredményeként visszakapott adatokat el akarjuk tárolni. A már megismert szabályok alapján az itt felsorolt változók neve elé egy-egy kettőspont karaktert kell írni.

Amennyiben a lekérdezés egyetlen sort eredményez, úgy a megadott változók

```
void nettóFizetésKiírása() {

    1) EXEC SQL BEGIN DECLARE SECTION;
    2)     char stúdióNév[15];
    3)     int igazgatóNettóFizetése;
    4)     char SQLSTATE[6];
    5) EXEC SQL END DECLARE SECTION;

       /* Kérjünk be a felhasználótól egy stúdiónevet, és a
          választ tegyük a stúdióNév változóba */

    6) EXEC SQL SELECT nettóBevétel
    7) INTO :igazgatóNettóFizetése
    8) FROM Stúdió, GyártásIrányító
    9) WHERE elnökAzon = azonosító AND Stúdió.név = :stúdióNév;

       /* Itt kiírhatjuk az eredményt, miután megbizonyosodtunk
       arról, hogy az SQLSTATE változó öt darab nulla karaktert
       tartalmaz */
}
```

7.3. ábra. *Egy egyetlen sort eredményező lekérdezés beágyazása*

rendre felveszik az eredménysor komponenseiben képződött értékeket (az SQLSTATE változó pedig felveszi a sikeres végrehajtást jelző 5 darab nulla számjegyet tartalmazó értéket). Amennyiben a lekérdezés egyetlen sort sem eredményez, vagy éppen egynél több sort eredményez, akkor a felsorolt változók semmilyen értéket sem kapnak, és az adatbázisrendszer az SQLSTATE változóba beírja a megfelelő hibakódot.

7.3. példa: Most elkészítünk egy C nyelvű függvényt, amely bekéri egy stúdió nevét a felhasználótól, és kiírja a stúdió igazgatójának nettó jövedelmét. A program forráskódját a 7.3. ábra tartalmazza. Az 1–5. sorok tartalmazzák a szükséges változók deklarációját. Ezután – a megjegyzéssel jelölt helyen – következne a stúdió nevét bekérő programrész (az ezeket megvalósító C nyelvű sorokat a példából elhagytuk). Majd a 6–9. sorokban egy egyetlen sort eredményező lekérdezést láthatunk, melynek felépítése hasonlít a korábbi fejezetekben már megismert lekérdezések szerkezetéhez. A korábban bemutatott lekérdezésekhez képest itt két különbséget is megfigyelhetünk: egyrészt a hetedik sorban használjuk azt az INTO kulcsszóval bevezetett záradékot, amelyben megadjuk, hogy melyik változókban akarjuk visszakapni a lekérdezés eredményét; másrészt pedig a stúdióNév változó értékét helyezzük az SQL utasítás feltételei közé a kilencedik sorba (ahol régebben ehelyett például szöveges konstans értékeket írtunk). E lekérdezés végrehajtása után egyetlen soros válaszra számítunk, és a válaszsor láthatóan egyetlen oszlopot tartalmaz, amely a visszakapott nettóBevétel attribútum értékét tárolja. Ezen egyetlen sor egyetlen oszlopának tartalma az egész típusú, igazgatóNettóFizetése nevű változóban lesz eltárolva. □

7.1.6. Sormutatók

A több sort eredményező lekérdezések eredményén egy ún. sormutatóval lehet végigmenni soronként. Egy sormutató létrehozása és használata a következőképpen történhet:

1. Deklarálni kell a sormutatót. A sormutató deklarációjának legegyszerűbb módja a következő:

    a) A deklarációt tartalmazó utasítást az EXEC SQL kulcsszavakkal kell kezdeni.

    b) Ezután a DECLARE kulcsszót kell leírni.

    c) Ezután kell írni a sormutatót azonosító nevet (a sormutatóra a programban a továbbiakban az itt megadott néven hivatkozhatunk).

    d) Ezután a CURSOR FOR kulcsszavakat kell leírni.

    e) Ezután annak a relációnak a nevét kell leírni, amelynek a sorain végig akarunk menni, vagy azt a lekérdezést kell specifikálni, amelynek az eredményét soronként fel akarjuk dolgozni. Az így deklarált sormutató végigmehet a megadott reláció sorain, vagy a megadott lekérdezés által létrehozott eredménytábla sorain.

Egy sormutató deklarációjának általános alakja:

EXEC SQL DECLARE <sormutatónév> CURSOR FOR <lekérdezés>

2. Az EXEC SQL OPEN utasítással inicializálni kell a sormutatót. Amíg egy sormutatót nem inicializáltunk, addig nem férhetünk hozzá a tartalmához. Az EXEC SQL OPEN utasítás mögé le kell írni az inicializálni kívánt sormutató nevét.

3. Ezután soronként hozzáférhetünk a sormutató mögött levő relációhoz, vagy a sormutatón keresztül elérni kívánt lekérdezés eredményadataihoz. Ehhez egy ún. *fetch utasítást* használhatunk, amely a sormutatóval elérhető sorokból veszi a következőt, és tartalmát áthelyezi az utasításban megadott változókba. Ha egy sormutatóval már nincs több elérhető sor, akkor a fetch utasítás nem tesz semmit a benne megadott változókba, a SQLSTATE változóba pedig a '02000' konstans értéket teszi, ami azt jelenti, hogy nincs több beolvasható sor. Ebben az utasításban a következőket kell megadnunk:

   a) Az utasítást az EXEC SQL FETCH FROM kulcsszavakkal kell kezdeni.

   b) Ezután kell megadni az olvasni kívánt sormutató nevét.

   c) Ezután az INTO kulcsszót adjuk meg.

   d) Majd adjuk meg vesszővel elválasztva azoknak a változóknak a nevét, amelyekben az eredménysor egyes komponenseit akarjuk visszakapni (a komponensek tartalma a változók megadásának sorrendjében kerül a változókba).

Egy sormutató olvasó utasításának általános alakja a következő:

EXEC SQL FETCH FROM <sormutatónév> INTO <változók listája>

4. Végül a sormutatót le kell zárni, hogy a rendszer felszabadíthassa a sormutató számára lefoglalt erőforrásokat. Erre az EXEC SQL CLOSE utasítást használhatjuk, amely mögé a lezárni kívánt sormutató nevét kell megadni. Megjegyezzük, hogy egy sormutatót újra megnyithatunk az EXEC SQL OPEN utasítással, így újra elérhetjük az általa visszaadott adatokat.

7.4. példa: Meg szeretnénk tudni azoknak a filmgyártás-vezetőknek a nevét, akiknek a nettó jövedelmük egy előre megadott, exponenciálisan növekedően meghatározott sávok valamelyikébe esik; az egyes sávokat a jövedelem számjegyeinek a száma szerint definiáltuk. Egy olyan lekérdezést fogunk megírni, amely beolvassa a GyártásIrányító tábla összes sorának a nettóBevétel attribútumát egy jövedelem nevű változóba. A tábla sorain egy irányítókSormutató nevű sormutatóval haladunk végig, kiszámoljuk az aktuálisan beolvasott jövedelemérték számjegyeinek a számát, ez alapján határozzuk meg, hogy a beolvasott jövedelem mely sávba tartozik, és az erre a célra létrehozott számláló tömb megfelelő elemét eggyel megnöveljük.


```
1)  void jövedelemSávok() {

2)      int i, számjegyek, számláló[15];
3)      EXEC SQL BEGIN DECLARE SECTION;
4)          int jövedelem;
5)          char SQLSTATE[6];
6)      EXEC SQL END DECLARE SECTION;

7)      EXEC SQL DECLARE irányítókSormutató CURSOR FOR
8)          SELECT nettóBevétel FROM GyártásIrányító;

9)      EXEC SQL OPEN irányítókSormutató;

10)     for(i = 0; i < 15; i++) számláló[i] = 0;
11)     while (1) {
12)         EXEC SQL FETCH  FROM irányítókSormutató
                            INTO :jövedelem;
13)         if(NINCS_TÖBB_SOR) break;
14)         számjegyek = 1;
15)         while((jövedelem /= 10) > 0) számjegyek++;
16)         if(számjegyek <= 14) számláló[számjegyek]++;
        }
17)     EXEC SQL CLOSE irányítókSormutató;
18)     for(i = 0; i < 15; i++)
19)         printf("számjegyek = %d: előfordulások száma =
              %d\n", i, számláló[i]);
        }
```

7.4. ábra. *A jövedelmek exponenciálisan növekvő sávok szerinti osztályozása*

A jövedelemSávok nevű függvény 2. sorában definiáljuk azokat a változókat, amelyekre nem akarunk SQL utasításokban hivatkozni. Az itt deklarált számláló tömb 15 egész értéket tartalmaz, minden egyes jövedelemsávhoz egy-egy számlálót. Az i változót használjuk tömbindexként a számláló tömb elemein végighaladva. A számjegyek változóban számoljuk meg a jövedelem változóban levő érték leírásához szükséges számjegyek számát.

A 3.–6. sorokban deklaráljuk a jövedelem és az SQLSTATE változókat, amelyeket az SQL utasításokból is elérhetünk. Az irányítókSormutató nevű sormutatót a 7. és 8. sorokban deklaráljuk, amely a 8. sorban megadott lekérdezés eredményeként létrejött tábla sorain megy végig (maga a lekérdezés a GyártásIrányító tábla nettóBevétel oszlopának tartalmát adja vissza eredményül). A sormutatót a 9. sorban megnyitjuk, majd a 10. sorban inicializáljuk az előbb már említett számláló nevű tömb elemeit.

A függvény lényegi része a 11.–16. sorokban látható. A 12. sorban olvassuk be a következő rekordot, a beolvasott érték a jövedelem változóba kerül. Mivel a 8. sorban deklarált lekérdezés egyoszlopos, ezért itt elég csak egy változót megadni (általában az oszlopok számával megegyező számú változót kell megadni). A 13. sorban ellenőrizzük azt, hogy volt-e még beolvasható adat. A konstans "02000" érték meg-

adása helyett egy NINCS_TÖBB_SOR nevű makrót adunk meg, amelyet az alábbi módon definiálunk:

```
#define NINCS_TÖBB_SOR !(strcmp(SQLSTATE, "02000"))
```

Emlékezzünk rá, hogy a "02000" érték jelöli azt, ha nincs több adat a sormutató olvasása során. A 13. sorban azt ellenőrizzük, hogy sikerült-e adatot olvasni, és ha nem, akkor kiugrunk a ciklusból a 17. sorra.

Ha sikerült adatot olvasni, akkor a 14. sorban inicializáljuk a számjegyeket számláló változót 1-re. A 15. sorban megszámoljuk a beolvasott érték számjegyeinek a számát (minden egyes 10-zel való osztás után növeljük eggyel a számjegyeket számláló változót, amíg a 10-zel való osztások során nullát nem kapunk). Végül a 16. sorban megnöveljük eggyel a statisztikát tároló számláló tömb megfelelő elemét. Láthatjuk, hogy a programban a 15 vagy annál többjegyű számokkal nem foglalkozunk – a statisztikát nem befolyásoljuk az ilyen óriási számokkal (ilyenkor nem növelünk egyetlen tömbelemet sem).

A 17. sorban lezárjuk a sormutatót, majd a 18.–19. sorban kiírjuk a statisztikát tároló számláló tömb tartalmát. □

7.1.7. Sormutatóval történő módosítások

Amikor egy sormutatóval végigmegyünk egy adatbázistábla sorain, az egyes sorokat nemcsak olvashatjuk, hanem módosíthatjuk is azok tartalmát, vagy akár törölhetjük is az illető sort (lekérdezések eredményeként létrejött relációk sorainak olvasása során ezt általában nem tehetjük meg). Az ilyen módosító és törlő UPDATE és DELETE utasítások szintaxisa megegyezik az 5.6. alfejezetben megismertekkel attól eltekintve, hogy ott a WHERE záradékban csak a CURRENT OF kulcsszavakat adhatjuk meg, azután pedig annak a sormutatónak a nevét kell megadni, amelyikből a módosítandó vagy törlendő sort olvastuk. Természetesen egy ilyen módosító vagy törlő beágyazott SQL utasítás végrehajtása bekerülhet a befogadó nyelven írt elágazások egy-egy ágába, attól függően, hogy az alkalmazásnak milyen feladatot kell megoldania.

7.5. példa: A 7.5. ábrán egy olyan C függvényt láthatunk, amelynek a szerkezete hasonlít a 7.4. ábrán látottakhoz: mindkét példában egy irányítókSormutató nevű sormutatót deklarálunk, amely végigmegy a GyártásIrányító tábla sorain, de a 7.5. ábrán bemutatott program egy sor beolvasása után egy elágazást végez a jövedelem értéke alapján, és aszerint dönti el, hogy az illető sort módosítani vagy törölni fogja.

Ebben a példában is hivatkozunk a már megismert NINCS_TÖBB_SOR makróra (ez az érték jelzi, ha a sormutatóval beolvastuk az összes sort). A program 12. sorában megvizsgáljuk, hogy a jövedelem értéke 1000 $-nál kevesebb-e, és ha kevesebb, akkor töröljük az illető sort (a 13. sorban látható DELETE utasítással), egyébként pedig megduplázzuk a hozzá tartozó jövedelem értékét (a 15.–17. sorokban látható utasítással). □


```
1)  void jövedelemMódosítás() {

2)      EXEC SQL BEGIN DECLARE SECTION;
3)          int jövedelem;
4)          char SQLSTATE[6];
5)      EXEC SQL END DECLARE SECTION;

6)      EXEC SQL DECLARE irányítókSormutató CURSOR FOR
7)          SELECT nettóBevétel FROM GyártásIrányító;

8)      EXEC SQL OPEN irányítókSormutató;

9)      while(1) {
10)         EXEC SQL FETCH  FROM irányítókSormutató
                           INTO :jövedelem;
11)         if (NINCS_TÖBB_SOR) break;
12)         if (jövedelem < 1000)
13)             EXEC SQL DELETE FROM GyártásIrányító
                         WHERE CURRENT OF irányítókSormutató;
14)         else
15)             EXEC SQL UPDATE GyártásIrányító
                         SET nettóBevétel = nettóBevétel * 2
                         WHERE CURRENT OF irányítókSormutató;
        }
16)         EXEC SQL CLOSE irányítókSormutató;
    }
```

7.5. ábra. *Jövedelemadatok módosítása*

7.1.8. Sormutatókhoz megadható opciók

Az SQL2 szabvány számos további lehetőséget biztosít a sormutatók alkalmazására, melyeket itt röviden összefoglalunk, majd a 7.1.9.–7.1.11. alfejezetekben részletesebben megismerhetünk.

1. A feldolgozandó reláció sorainak feldolgozási sorrendje meghatározható.

2. Az elvégezhető változtatások hatása is befolyásolható.

3. A sormutatónak a feldolgozandó sorokon való mozgatási iránya is befolyásolható.

7.1.9. Behozandó sorok rendezése

Tekintsük először a sorok rendezését! Lehetőségünk van a sorokat bármely oszlop szerint rendezve megkapni. Ehhez a sormutató definícióját tartalmazó utasítást ki kell egészíteni az ORDER BY kulcsszavakkal, és ez után kell megadni a rendezési szem-

```
1) EXEC SQL DECLARE színészSormutató CURSOR FOR
2)     SELECT cím, év, stúdió, színészNév
3)     FROM Film, SzerepelBenne
4)     WHERE cím = filmCím AND év = filmÉv
5)     ORDER BY év, cím;
```

7.6. ábra. *Az* `ORDER BY` *záradék alkalmazása sormutatókban*

pontok listáját vesszőkkel elválasztva – hasonlóan, mint azt az 5.1.5. alfejezetben láthattuk. Rendezési szempontként megadhatjuk akár az eredményreláció valamelyik oszlopának a nevét, akár az eredményreláció valamely oszlopának a sorszámát (és ez utóbbi esetben a rendezés az eredményreláció annyiadik oszlopának a tartalma szerint történik, amely oszlopnak a sorszámát rendezési szempontként megadtuk). Amennyiben több rendezési szempontot is megadunk, úgy az adatok lexikografikusan lesznek rendezve: először az első megadott szempont szerint, majd az első szempont szerint megegyező rekordok egymáshoz képesti sorrendjét a második rendezési szempont szerinti sorrendjük határozza meg, és így tovább.

7.6. példa: Ebben a példában a `Film` és a `SzerepelBenne` táblák összekapcsolásával kapott relációnak a filmCím, év, színészNév és stúdió attribútumokra levetített sorait akarjuk megvizsgálni. A végeredményt rendezetten szeretnénk megkapni: év szerint, azon belül pedig cím szerint növekvő sorrendben. A következő (7.6. ábrán) az előbb említett adatokat szolgáltató sormutató definícióját láthatjuk.

Az első sor a sormutató definícióját vezeti be, majd a következő három sor szerepe megegyezik a korábban már megismert `SELECT` záradékok szerepével. Az 5. sor azt írja le, hogy a sorokat olyan sorrendben akarjuk visszakapni, hogy először a legrégebbit kapjuk vissza (azt, amelynek legkisebb az évet megadó komponense). Az ugyanahhoz az évhez tartozó filmekhez tartozó sorokat pedig a `cím` (filmcím) oszlop szerint rendezve akarjuk egymás után visszakapni. Az évek és a címek rendezésénél az adatbázis-kezelő rendszer rendre a numerikus, illetve ábécé szerinti rendezés szabályait alkalmazza. □

7.1.10. Egyidejű módosítások elleni védelem

Most azt az esetet fogjuk megvizsgálni, ami akkor állhat elő, ha egy alkalmazás a 7.6. példában említett sormutatóval olvassa be a sorokat, és eközben ez vagy egy másik alkalmazás módosítja a feldolgozás alatt álló `Film` vagy `SzerepelBenne` relációk tartalmát. Az adatbázisok egyidejű elérésével részletesebben is foglalkozunk a 7.2. alfejezetben. Itt most elégedjünk meg azzal, hogy van lehetőség az egyidejű elérésre, vagy akár módosításra is.

Ha célunk csak egy megadott szereplő kikeresése, és nem érdekel az, hogy látjuk-e a keresés közben beszúrt vagy törölt szereplők adatait, akkor a 7.6. példában megismert utasítás megfelel a céljainknak. Ha viszont a sormutató feldolgozása közben végzett konkurens módosítások hatását nem akarjuk látni a sormutatón beolvasott rekordokon, akkor a sormutatót az egyidejű módosításokkal szemben *érzéketlennek* kell de-

finiálni. Ez hasznos lehet például olyankor, ha a 7.6. példában említett feldolgozást végző függvény olyan új sorokat szúrna be a `SzerepelBenne` táblába, amelyeket a sormutatón keresztül később visszakapnánk, ezért a sormutatón keresztül végeláthatatlanul sok sort behozhatnánk.

7.7. példa: Most módosítsuk a 7.6. példában megadott beágyazott SQL utasítás első sorát a következőre:

```
1) EXEC SQL DECLARE színészSormutató
            INSENSITIVE CURSOR FOR
```

Ekkor az adatbázisrendszer a `színészSormutató` nevű sormutató megnyitása és lezárása közben a `Film` és a `SzerepelBenne` táblákon elvégzett módosításokat az alkalmazás elől teljesen eltakarja, vagyis ez nem fogja befolyásolni a visszakapott sorokat. □

Egy egyidejű módosításokkal szemben érzéketlen sormutató nagyon erőforrás-költséges abból a szempontból, hogy az SQL rendszernek sok idejébe kerül az adatelérési műveletek olyan módú megszervezése, hogy eközben a sormutató érzéketlenségét is biztosítsa (mint már említettük, ennek a megvalósításával a 7.2. alfejezetben foglalkozunk; addig elégedjünk meg azzal, hogy legegyszerűbb esetben ez megoldható úgy, hogy az adatbázisrendszer felfüggeszti a módosításokra érzéketlen sormutatóhoz tartozó lekérdezés alapjául szolgáló relációk egyidejű módosítását megkísérlő folyamatok futását – mint a `Film` vagy `SzerepelBenne`).

Egy-egy sormutatóról esetleg biztosan tudhatjuk, hogy végigolvasása során nem módosít egy *R* nevű relációt, amelyből adatokat olvas. Az ilyen sormutatók feldolgozása az ugyanezen táblán végzett módosításokra érzéketlen sormutatókkal egyidejűleg is történhet, ugyanis ilyenkor a két feldolgozó folyamat közül egyiknek a futását sem kell felfüggeszteni (hiszen ekkor nem fordulhat elő, hogy azt az *R* relációt módosítják, amelyet az említett erre érzéketlen sormutató is felhasznál). Ha egy sormutató definícióját kiegészítjük a `FOR READ ONLY` kulcsszavakkal, úgy az adatbázis-kezelő rendszer biztosan tudhatja róla, hogy végigolvasása során nem módosítja a benne felhasznált relációkat. Ha ezt az információt közöljük az adatbázis-kezelő rendszerrel, akkor az tudhatja, hogy nincs akadálya annak, hogy ezt a sormutatót más, valamelyik benne felhasznált reláció módosítására érzéketlen sormutatóval együtt futtassa.

7.8. példa: Ha tehát a 7.6. ábrán látott lekérdezést kiegészítenénk az alábbi (hatodik) sorral,

```
6) FOR READ ONLY;
```

akkor a `színészSormutató` sormutató által behozott rekordokra kiadott módosító vagy törlő (`UPDATE` vagy `DELETE`) utasítások nem futnának le, hibát okoznának. A hibáról az `SQLSTATE` változó értéke alapján szerezhetnénk tudomást. □

7.1.11. Sormutatók mozgatása

Végül megismerjük azt az opciót, amellyel befolyásolhatjuk a sormutatónak a feldolgozandó sorokon való mozgatásának az irányát. Alapértelmezés szerint a sorok feldolgozása a reláció elejénél kezdődik a reláció vége felé haladva. A programozó ezt a sorrendet módosíthatja: a sormutatóval akár többször is végig lehet menni a sorokon, mielőtt a sormutatót lezárnák. Ehhez a programozónak a következőket kell tennie:

1. A sormutató deklarációjánál a `CURSOR` kulcsszó elé illesszük be a `SCROLL` kulcsszót. Az SQL rendszer ez alapján tudja, hogy az illető sormutatót nem csak az előbb említett alapértelmezés szerinti sorrendet követve akarjuk használni.

2. A `FETCH` utasításban a `FETCH` kulcsszót követően az alábbi opciók megadására van lehetőségünk:

   a) `NEXT` vagy `PRIOR` kulcsszavak ahhoz, hogy a sorrendben következő vagy éppen megelőző rekordot akarjuk beolvasni (alapértelmezés a sorrendben következő rekord olvasását elvégző `NEXT`).

   b) `FIRST` vagy `LAST` kulcsszavak, hogy a sormutatóhoz hozzárendelt sorrendben a legelső vagy a legutolsó sort olvassuk be.

   c) `RELATIVE` kulcsszó, mögötte egy pozitív vagy negatív egész számmal: ezzel adhatjuk meg, hogy hány sorral akarunk előre/hátra menni a sormutató feldolgozásának alapértelmezés szerinti sorrendjéhez viszonyítva. A `RELATIVE 1` hatása megegyezik az előbb bemutatott `NEXT` kulcsszó hatásával, míg a `RELATIVE -1` hatása megegyezik a `PRIOR` kulcsszó hatásával.

   d) `ABSOLUTE` kulcsszó, mögötte egy pozitív vagy negatív egész számmal: ezzel a kulcsszóval a sormutatót egy megadott sorszámú sorra állíthatjuk, hogy a beolvasást attól a sortól kezdve folytathassuk (az előbb említett `RELATIVE` kulcsszó alkalmazásakor a sormutatót egy megadott számú sorral állíthatjuk előbbre/hátra; ott a pozicionálás az aktuális sorhoz viszonyítva történik). Ha az `ABSOLUTE` kulcsszó után pozitív számot adunk meg, akkor a sormutatóval elért reláció elejétől számítva a megadott sorra pozicionálhatunk. Negatív szám megadása esetén az utolsó sorhoz viszonyítva történik a pozicionálás. Az `ABSOLUTE 1` hatása megegyezik az előbb bemutatott `FIRST` kulcsszó hatásával, míg a `LAST` kulcsszó szinonimájaként az `ABSOLUTE -1`-et adhatunk meg.

7.9. példa: Most módosítsuk a 7.5. példában megírt programrészt úgy, hogy a sormutatóval az adatokat a reláció végéről visszafelé haladva olvassuk. Először módosítani kell a 7.5. példa 6. és 7. sorát, feltüntetve a `SCROLL` kulcsszót a sormutató deklarációjánál.

```
6) EXEC SQL DECLARE irányítókSormutató SCROLL CURSOR FOR
7)      SELECT nettóBevétel FROM GyártásIrányító;
```


A beolvasás inicializálását a `FETCH LAST` kulcsszóval kell bővíteni, míg a következő sor olvasását a `FETCH PRIOR` utasítással tehetjük meg. A 7.5. példa 9.–15. sorát a következőképpen módosítsuk:

```
EXEC SQL FETCH LAST FROM irányítókSormutató
                    INTO:jövedelem;
while(1) {
  /* ugyanaz, mint az eredeti kód 11-15. sora */
  EXEC SQL FETCH PRIOR FROM irányítókSormutató
                      INTO :jövedelem;
}
```

Az adatbázis-kezelő rendszer az alapértelmezés szerinti sorrendtől eltérő sorrendeket gyakran csak úgy tudja biztosítani, hogy előre felépíti az egész eredményrelációt, míg az alapértelmezés szerinti sorrendben történő olvasás során – mint azt például az alábbi lekérdezés előírja –

```
SELECT nettóBevétel FROM GyártásIrányító
```

az adatbázis-kezelő rendszer az első sort már visszaadhatja az alkalmazásnak még mielőtt további sorokat találna. ☐

Megjegyezzük, hogy a sormutató-olvasás többszöri irányváltoztatásának általában nincsenek hatékonysági alapjai.

7.1.12. Dinamikus SQL

Az eddigiekben mindvégig olyan beágyazott SQL utasításokkal foglalkoztunk, amelyeknek az alakja már a program lefordítása során pontosan ismert volt. Vannak azonban olyan esetek, amikor a kiértékelendő SQL utasítás csak a program futása közben állítható össze a befogadó nyelv utasításaival, a fordítási időben még nem. A befogadó nyelv utasításaival futásidőben összeállított SQL utasítások a fordítási időben még nem ismertek, ezért ezeknek a feldolgozását nem bízhatjuk egy SQL előfeldolgozó programra vagy a befogadó nyelv fordítóprogramjára.

Ilyen helyzet fordul elő például az SQL parancsértelmező programoknál: ezek bekérnek egy SQL utasítást a felhasználótól, végrehajtják azt, majd kiírják a végrehajtás eredményességét vagy éppen eredménytelenségét jelző üzeneteket. Ilyen például az 5. fejezetben vázolt, SQL lekérdezések végrehajtására képes felület; minden SQL alapú adatbázis-kezelő rendszer biztosít egy ilyen alkalmazói felületet.

A befogadó nyelven megírt program utasíthatja az SQL adatbázis-kezelő rendszert arra, hogy a felhasználótól beolvasott SQL utasítást elemezze, és állítsa elő egy olyan – belső ábrázolású – formáját, amelyet az adatbázis-kezelő rendszer végre tud hajtani. Ezeket a lépéseket két ún. *dinamikus SQL utasítás* segítségével végezhetjük el.

```
1) void kérdésBeolvasás() {
2)    EXEC SQL BEGIN DECLARE SECTION;
3)       char *kérdés;
4)    EXEC SQL END DECLARE SECTION;
5)    /* olvassuk be az SQL utasítást, a kérdésnek lefoglalt
      memóriaterületre állítsuk rá a kérdés változóban
      tárolt mutatót */
6)    EXEC SQL PREPARE SQLquery FROM :kérdés;
7)    EXEC SQL EXECUTE SQLquery;
  }
```

7.7. ábra. *Egy dinamikus SQL utasítás előkészítése és végrehajtása*

1. Az első ilyen utasítás az `EXEC SQL PREPARE` kulcsszavakkal kezdődik, azt követi egy megosztott elérésű *V* SQL változó neve, amit a `FROM` kulcsszó követ, majd ezt követi a befogadó nyelv valamely változója vagy kifejezése (általában karakterlánc típusú; itt adhatjuk meg azt az SQL utasítást, amelynek a végrehajtását elő akarjuk készíteni). Ennek az utasításnak a hatására a `FROM` kulcsszó után megadott karakterláncban tárolt SQL utasítást az adatbázis-kezelő rendszer elemzi, és olyan formára hozza, amelyet a későbbiekben könnyen végre lehet hajtani. Az utasítás elemzett, végrehajtható formája a *V* változóban lesz eltárolva.

2. Az elemzett formában meglevő SQL utasításokat az `EXEC SQL EXECUTE` kulcsszavakkal bevezetett utasításokkal hajthatjuk végre. Ezután azt a változót kell megadni, amely a korábban már elemzett SQL utasítás végrehajtható formáját tárolja.

E fenti két lépés egy utasításban is elvégezhető. Ennek az utasításnak az alakja

`EXEC SQL EXECUTE IMMEDIATE` <mit hajtson végre>

ahol az `IMMEDIATE` kulcsszót követően kell megadni azt a befogadó nyelven deklarált karakterlánc típusú változót vagy értéket, amely a végrehajtani kívánt SQL utasítást tartalmazza. A két lépés különválasztása akkor indokolt, ha az elemzett SQL utasítást többször is végre akarjuk hajtani, mivel ilyenkor a többszöri elemzéshez szükséges időt megtakaríthatjuk.

7.10. példa: A 7.7. ábrán vázolt C program beolvas egy SQL utasítást a szabványos bemenetről (alapértelmezés szerint a billentyűzetről) a `kérdés` változóba, előkészíti a végrehajtását, majd végrehajtja. Az `SQLquery` változó tartalmazza az utasítás előkészített formáját. Mivel a beolvasott SQL lekérdezést csak egyszer hajtjuk végre, ezért a 6. és 7. sorok helyett írhatnánk egyszerűen az alábbi sort is:

```
EXEC SQL EXECUTE IMMEDIATE :kérdés;
```

□

7.1.13. Feladatok

7.1.1. feladat: Készítsük el a beágyazott SQL lekérdezéseket az alábbi – 4.1.1. feladat leírásában megismert – adatbázissémához!

```
Termék(gyártó, modell, típus)
PC(modell, sebesség, memória, merevlemez, cd, ár)
Laptop(modell, sebesség, memória, merevlemez,
       képernyő, ár)
Nyomtató(modell, színes, típus, ár)
```

A feladat megoldásához nincs szükség működő program megírására; a nem SQL-specifikus befogadó nyelvi részeket – a mi korábbi példáinkhoz hasonlóan – elegendő világos „megjegyzésekkel" helyettesíteni (az elvégzendő feladat specifikációját megadva). A megoldás leírására bármely programozási nyelvet használhatunk.

A megvalósítandó lekérdezések a következők:

* a) Kérjünk be a felhasználótól egy árértéket, és keressük meg a PC-táblában a megadott árhoz legközelebb eső áron kapható számítógép adatait. A megtalált számítógépről írassuk ki a gyártóját, modellazonosítóját, sebességadatát.

b) Kérjük be a felhasználótól a számára elfogadható képernyőméretet, sebességet, RAM-méretet, winchesterméretet, majd keressük ki az összes olyan laptop számítógépet, amely megfelel az igényeknek. A megtalált laptopokról írassuk ki a technikai adataikat és gyártójuk nevét.

! c) Kérjük be a felhasználótól egy gyártó nevét, majd írjuk ki a képernyőre az illető gyártó összes termékét és az egyes termékek jellemzőit (a modellazonosítót, típusazonosítót és a további technikai adatokat).

!! d) Kérjünk be a felhasználótól egy összeget (egy PC és egy nyomtató vásárlására szánható maximális árat), valamint a megvásárolni kívánt PC-vel szemben támasztott minimális sebességkövetelményt. Keressük meg a legolcsóbb olyan PC és nyomtató kombinációt, amely megvásárolható a megadott összegből, és sebességét tekintve megfelel a felhasználó elvárásainak (vagyis nem lassabb a felhasználó által adott minimálisan elvárt sebességnél). Lehetőleg színes nyomtatót válasszunk. A program írja ki a kiválasztott PC és nyomtató kombináció modellazonosítóit.

e) Kérjük be a felhasználótól egy új PC törzsadatait: gyártójának nevét, modellazonosítóját, sebességét, RAM-méretét, winchesterméretét, CD-sebességét és árát. Ellenőrizzük az adatbázisban, hogy van-e benne már ilyen PC. Ha az adatbázisban már vannak ilyen adatok, akkor írjunk ki a képernyőre egy figyelmeztető üzenetet; ha pedig nincs, akkor vegyük fel az adatbázisba!

*! f) Csökkentsük le az összes „régi" PC árát 100 dollárral. Ügyeljünk arra, hogy a programunk futása közben felvett „új" PC-k árát ne módosítsuk!

7.1.2. feladat: Készítsük el a beágyazott SQL lekérdezéseket az alábbi adatbázissémához.

```
Hajóosztályok(hajóosztály, típus, ország,
              ágyúkSzáma, kaliber, vízkiszorítás)
Hajók(név, hajóosztály, felavatva)
Csaták(név, időpont)
Kimenetelek(hajó, csata, eredmény)
```

Az adatbázissémát a 4.1.3. feladatból vettük.
A megvalósítandó lekérdezések a következők:

a) Egy hajó tűzereje megközelítőleg arányos a rajta levő fegyverek számának és a fegyverek kalibere köbének a szorzatával. Keressük meg a legmagasabb tűzerejű hajóosztályt.

! b) Kérjük be a felhasználótól valamely csata nevét. Keressük meg a csatában részt vevő hajókhoz nyilvántartott országokat. Keressük meg annak az országnak a nevét, amelynek a legtöbb hajója odaveszett az illető csatában, és azt az országot, amelynek a legtöbb hajója megsebesült az illető csatában, és írassuk ki e két ország nevét.

c) Kérjük be a felhasználótól a `Hajóosztályok` tábla egy új sorának az adatait, majd kérjünk be további adatokat: az újonnan beadott hajóosztályba tartozó hajók neveit és felavatásuk időpontját. Vegyük fel ezeket az adatokat a fenti adatbázisba. Ne kelljen minden egyes hajónál beadni, hogy melyik osztályba tartozik, mivel mindegyik a feladat első lépéseként beadott osztályba tartozik.

! d) Vizsgáljuk meg a `Csaták`, `Kimenetelek`, `Hajók` táblákat. Keressünk olyan hajókat, amelyek részt vettek egy csatában még mielőtt vízre bocsátották volna. Ha ilyen hajót találunk, akkor írjuk ki ezt a minden bizonnyal hibás adatot a felhasználónak, és kérdezzük meg, hogy hogyan javítsuk az adatokat. Két javítási lehetőséget ajánljunk fel: vagy a csata időpontjának a módosítását, vagy pedig a hajó vízrebocsátásának az időpontját (ha szükséges, akkor a felhasználó végezhesse el mindkét módosítást).

*! **7.1.3. feladat:** Ebben a feladatban célunk a 7.1.1. feladatban is alkalmazott PC-reláció megfelelő sorainak a megkeresése.

```
PC(modell, sebesség, memória, merevlemez, cd, ár)
```

Keressük ki az olyan PC-ket, amelyeknél legalább két drágább, de ugyanolyan sebességű PC van az adatbázisban. Több megoldásmód is kínálkozik e probléma megoldására, de mi készítsünk olyan megoldást, amely egy sormutató mozgatásával dolgozik. Ehhez olvassuk be a PC-k adatait sebesség, illetve ár szerint rendezve. *Tanács:* Minden beolvasott sorhoz ugorjuk át a következő két sort, és nézzük meg, hogy változott-e a sebesség.

!! **7.1.4. feladat:** A 7.1.1. alfejezetben említettük, hogy SQL nyelven nem készíthető faktoriálist számító program. Ez az állítás az SQL2-re igaz, de – az 5.10. alfejezetben leírt – SQL3 rekurzió alkalmazása lehetővé teszi azt, hogy egy ehhez hasonló feladatot megoldó SQL programot készítsünk. Készítsünk egy SQL3 lekérdezést, amelyet egy olyan M relációra alkalmazunk, amely egyetlen sort és egyetlen oszlopot tartalmaz: (m)-et, ahol m egy egész szám. Az olyan $(n, n!)$ párok halmaza legyen a lekérdezés eredménye, ahol $1 \le n \le m$.

7.2. Tranzakciók az SQL-ben

Az eddigiekben feltételeztük, hogy az adatbázishoz egyidejűleg legfeljebb egy felhasználó fér hozzá, ezért a program által elvégzett adatbázis-műveletek egymás után hajtódnak végre: az egyik művelet eredményeként létrejött adatbázis-állapot szolgál a következő adatbázis-művelet kiindulási állapotaként. Továbbá feltételeztük, hogy az összes művelet teljes egészében végre lesz hajtva: sem szoftver-, sem pedig hardverhibák nem idézhetnek elő olyan helyzeteket, amelyekben az adatbázis állapota egy félbemaradt művelet eredménye.

A valós alkalmazásoknál sokkal bonyolultabb a helyzet. Ebben a részben először megvizsgáljuk, hogy milyen problémák származhatnak abból, hogy az adatbázis állapota nem a rajta végrehajtott műveletek eredményét tükrözi, majd megvizsgáljuk azt, hogy az SQL milyen eszközöket ad az ilyen és ehhez hasonló problémák elkerülésére.

7.2.1. Sorbarendezhetőség

Valós banki vagy például repülőgép-helyfoglalási alkalmazásoknál az adatbázist egyidejűleg akár több száz művelet is elérheti vagy módosíthatja. E műveleteket sok ezer felhasználó kezdeményezheti (például banki alkalmazottak, bankjegykiadó automaták, repülőjegy-irodák stb.), és nem kizárható, hogy két felhasználó egyidejűleg ugyanazt az adatelemet akarja elérni (például két felhasználó ugyanarra a számlára akar pénzt átutalni, vagy ugyanarra a repülőjáratra, vagy akár ugyanarra a meccsre akar jegyet váltani), és ilyenkor meglepő konfliktushelyzetek alakulhatnak ki. Egy példán keresztül mutatjuk be, hogy milyen hibák származhatnának abból, ha az adatbázis-kezelő rendszerek egyáltalán nem lennének korlátozva az egyes adatbázis-műveletek végrehajtási sorrendjében. Ilyen esetek sok problémát okozhatnának, ha nem figyelnénk rájuk, mint azt az alábbi példában majd láthatjuk. Hangsúlyozzuk, hogy a valós feladatokat megoldó adatbázis-kezelő rendszerekkel és alkalmazásaikkal általában nem fordulnak elő ilyen problémák, vagy legalábbis sok szándékos hibát el kell követni ahhoz, hogy a ma elterjedten használt adatbázis-kezelő rendszerek hasonló hibákat produkáljanak.

7.11. példa: Tegyük fel, hogy egy olyan – `helyfoglalás` nevű – függvényt akarunk készíteni, amely beolvassa az induló repülőgépjáratok adatait az egyes járatok helyfoglalási adataival együtt, majd megvizsgál egy előre megadott helyet, hogy szabad-e még, és ha szabad, akkor lefoglalja. A járatokat tároló adatbázistábla neve `Járatok` legyen, ennek attribútumai a `járatSzám`, `járatDátum`, `járatHely`, valamint `foglalt` (az egyes attribútumok szerepe nevük alapján látható). A helyfoglalást intéző függvény a 7.8. ábrán látható.

A 9.–11. sorban a `fogl` változóba lekérdezzük, hogy a kérdéses hely szabad-e (a változóba 0 vagy 1 érték kerül, annak megfelelően, hogy a hely szabad vagy foglalt). A 12. sorban elágazunk attól függően, hogy a kérdéses hely foglalt-e vagy sem. Ha a hely foglalt, akkor a 17. sorban folytatódik a program futása. Ha a kérdéses hely még szabad, akkor a 13.–15. sorokban foglaltra állítjuk, majd a 16. sorban eltároljuk azt is,

```
1)  EXEC SQL BEGIN DECLARE SECTION;
2)      int járat;
3)      char dátum[10];
4)      char hely[3]; /* Egy helyet (széket) két számjegy
                    és egy betű reprezentál */
5)      int fogl; /* logikai érték; IGAZ, ha a hely foglalt */
6)  EXEC SQL END DECLARE SECTION;

7)  void helyfoglalás() {
8)      /* itt jönne az a C kódrészlet, amely a felhasználótól bekéri egy járat azonosítóját, a járaton
        egy hely megnevezését, és eltárolja a függvény
        elején definiált három megfelelő változóba */
9)      EXEC SQL SELECT foglalt INTO :fogl
10)         FROM Járatok
11)         WHERE járatSzám = :járat AND járatDátum = :dátum
                AND járatHely = :hely;

12)     if (!fogl) {
13)         EXEC SQL UPDATE Járatok
14)             SET foglalt = 'B1'
15)             WHERE járatSzám = :járat
                    AND járatDátum = :dátum
                    AND járatHely = :hely;
16)         /* A helyfoglalási adatbázisba feljegyezzük a
            lefoglaló utas nevét, és értesítjük a fel-
        használót a lefoglalás sikerességéről */
        }
17)     else /* a kérdéses hely már foglalt, erről értesít-
            jük a felhasználót, majd új adatokat
            kérhetünk */
    }
```

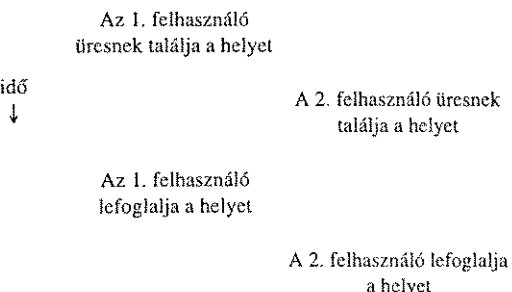
7.8. ábra. *Egy hely lefoglalása*

hogy kinek a részére foglaltuk le (igaz ezt a kódrészt nem mutattuk meg, gyakorlatban valószínűleg egy ezt tároló adatbázistáblába kell bejegyezni).

Most tegyük fel, hogy ezt a függvényt egyidejűleg – a 7.9. ábrán szemléltetett időzítéssel – két ügynök is meghívja, mindegyik ügynök ugyanazt a helyet szeretné eladni más-más vevőknek. Előfordulhat, hogy a két programpéldány egyidejűleg hajtja végre a 9. sorban látható lekérdezést, és mindketten szabadnak találják az illető járaton az illető helyet. Ekkor mindkettőjük lefoglalja az illető helyet a foglaltságot jelző bit értékének 1-re állításával, de az adatbázis tartalma azt a foglalási műveletet fogja tükrözni, amelynél a 13. sorban kezdődő módosítási művelet később futott le. Miután ezek a módosítások végrehajtódtak, mindkét ügynök megkapja a visszajelzést, hogy a helyfoglalásuk sikerült, és mindegyik azt gondolja, hogy az illető hely a saját ügyfele részére lett lefoglalva. □

A 7.11. példában egy olyan esetet láthattunk, amelyben két olyan művelet lett végrehajtva, amelyek önmagukban helyesek, de egyidejű végrehajtásuk hibás végeredményhez vezethet (esetünkben ahhoz, hogy csak egy hely lett lefoglalva, és mindegyik ügyfél azt gondolja, hogy az a hely részére lett lefoglalva). Ennek a problémának megoldására számos SQL mechanizmust használhatunk fel, melyeknek mindegyike a fenti két helyfoglaló program végrehajtásának *sorbarendezésén* alapul. Azt mondjuk, hogy egy azonos adatbázist módosító függvények (vagy programok) végrehajtása *sorban történik*, ha egy függvény (vagy program) végrehajtása teljesen befejeződik, mielőtt egy másik függvény (vagy program) végrehajtása elkezdődne. Akkor mondjuk, hogy függvények végrehajtása *sorbarendezhető*, ha végrehajtásuk eredménye megegyezik egy sorba rendezett végrehajtás eredményével (más magyar nyelvű terminológiák szerint nevezik ezt sorolható végrehajtásnak is).

Világos, hogy ha a bemutatott két `helyfoglalás ()` nevű függvényhívás sorban egymás után hajtódik végre, akkor az említett hiba nem fordulhat elő. Először az egyik felhasználó foglalása lesz könyvelve, majd a másiké (ha a második felhasználó ugyanazt a helyet akarja lefoglalni, amit az első már lefoglalt, akkor a program kiírja neki, hogy a hely már foglalt). Ez fontos lehet azoknak a felhasználóknak, akik lefoglalták a helyeket, de az adatbázis-kezelő számára csak az a fontos, hogy egy helyet csak egy felhasználó foglalhat le. Vegyük észre, hogy az adatbázis eddig is konzisz-



7.9. ábra. *Két felhasználó megpróbálja ugyanazt a helyet lefoglalni ugyanabban az időben*

A sorbarendezhetőség biztosítása

A gyakorlati életben általában nem lehet megkövetelni, hogy az elvégzendő műveletek egymás után legyenek végrehajtva: egyszerűen túl sok van belőlük, ezért ki kell használni a párhuzamosított végrehajtás nyújtotta hatékonyságnövelési lehetőségeket. Ezért az adatbázis-kezelő rendszereknek valamilyen mechanizmusokkal biztosítaniuk kell a sorbarendezhetőséget: a felhasználó az eredményt úgy látja, mintha a műveletek végrehajtása sorban történt volna még akkor is, ha a végrehajtás valójában nem sorban történik (az adatbázis-kezelők sorbarendezett végrehajtás eredményét biztosítják).

Mint azt az 1.2.4. alfejezetben már említettük, az adatbázis-kezelő programokban gyakori lépés relációk egyes sorainak, esetleg oszlopainak lefoglalása abból a célból, hogy ne érhesse el egyidejűleg két alkalmazás. Ha például a 7.11. példában a `helyfoglalás()` függvényt úgy készítettük volna el, hogy futása alatt mások számára nem engedélyezzük a `Járatok` reláció elérését, akkor a `helyfoglalás()` függvénnyel párhuzamosan minden olyan művelet végrehajtható, amely nem használja a járatokat nyilvántartó relációt (de az illető relációt használó műveletek ilyenkor nem hajthatók végre). Az 1.2.4. alfejezetben említettek szerint egy egész reláció lefoglalása túlságosan költséges művelet egy sor vagy egy egységnyi lemezterület lefoglalásához képest, mivel nagymértékben csökkenti a párhuzamosítás lehetőségét. Például ha a `helyfoglalás()` függvényben csak a módosítani kívánt sort foglaljuk le, akkor ilyenkor is végrehajtható lenne a `helyfoglalás()` függvény egy másik példánya – feltéve, hogy nem ugyanahhoz a sorhoz akar hozzáférni, mint amit valaki éppen foglal.

tens volt, hiszen benne egy helyhez csak egy felhasználót lehetett eltárolni. Az inkonzisztencia itt abban a hibás helyfoglalás eredményeként létrejött helyzetben van, hogy a repülőgépen mindkét felhasználónak ugyanarra a helyre kellene leülnie, ami nem megengedett.

7.2.2. Műveletek atomisága

Az előző alfejezetben szemléltetett, két vagy több egyszerre végrehajtott művelet sorba nem rendezhetőségből eredő problémák mellett előfordulhat az is, hogy egyetlen művelet végrehajtása közben sérül az adatbázis épsége, például egy hardver vagy szoftver rendszerösszeomlás miatt. Egy ilyen lehetséges helyzetet mutatunk be a 7.12. példában, bár itt is megjegyezzük, mint azt a 7.11. példa ismertetésénél tettük, hogy a valóságban használt adatbázis-kezelők és a jól elkészített alkalmazások az ilyen problémák kivédésére is képesek.


```
1)  EXEC SQL BEGIN DECLARE SECTION;
2)     int szla1, szla2; /* a két számlaszám helye */
3)     int egyenleg1; /* az első számla egyenlege */
4)     int összeg; /* az átutalni kívánt összeg */
5)  EXEC SQL END DECLARE SECTION;

6)  void átutal() {
7)     /* C programrész, mely bekéri a két számlaszámot és
       az átutalni kívánt összeget, és ezeket eltárolja
       szla1, szla2, összeg változókban */
8)     EXEC SQL SELECT egyenleg INTO :egyenleg1
9)           FROM Számlák
10)          WHERE számlaSzám = :szla1;
11)    if (egyenleg1 >= összeg) {
12)       EXEC SQL UPDATE Számlák
13)             SET egyenleg = egyenleg + :összeg
14)             WHERE számlaSzám = :szla2;
15)       EXEC SQL UPDATE Számlák
16)             SET egyenleg = egyenleg - :összeg
17)             WHERE számlaSzám = :szla1;
       }
18)    else /* C programrész, mely kiírja, hogy nincs elég
             pénz az átutalás teljesítéséhez */
    }
```

7.10. ábra. *Pénz átutalása egy számláról egy másikra*

7.12. példa: Most tekintsünk egy másik jellemző helyzetet, amely egy bank számlaadatbázisának feldolgozása során fordulhatna elő. A példában tegyük fel, hogy a bank a számlák adatait a `Számlák` nevű táblában tárolja, és ennek a táblának a sorai két oszlopot tartalmaznak: egy `számlaSzám`, illetve egy `egyenleg` nevű oszlopot.

Egy átutalási műveletet megvalósító `átutal()` nevű függvényt fogunk elkészíteni. E függvény egyszerűen bekér két számlaszámot és egy összeget, ellenőrzi, hogy az először megadott számú számlán van-e az átutalás teljesítéséhez szükséges pénz, és ha van, akkor a megadott összeget átutalja a másodszorra megadott számú számlára. A 7.10. ábra ennek a függvénynek a vázlatos felépítését szemlélteti.

A 7.10. ábrán vázolt programrész működése egyszerű. A 8.–10. sorokban megkeressük az első számla egyenlegét. A 11. sorban megvizsgáljuk, hogy az első számlán van-e elég pénz az átutalás teljesítéséhez. Ha van elég pénz, akkor a 12.–14. sorokban a másodszorra megadott számla egyenlegéhez hozzáadjuk az átutalandó összeget. A 15.–17. sorokban az először megadott számlát megterheljük az átutalni kívánt összeggel (az egyenlegből le lesz vonva az átutalandó összeg). Ha nincs elég pénz az átutalás teljesítéséhez, akkor a 18. sorban az `else` utáni ág lesz végrehajtva.

Most tekintsük azt az esetet, amikor például a programot futtató számítógép a 14. sor végrehajtása után összeomlik (akár szoftver-, akár hardverhiba miatt). Ekkor az adatbázis olyan állapotban van, hogy az átutalt összeg megjelenik az arra kijelölt

számlán, de az átutalt összeggel nem lesz megterhelve az a számla, amelyről a pénzt átutalták. Nyilvánvaló, hogy a bank ekkor elvesztett annyi pénzt, amennyit át akartak utalni. □

A 7.12. példában bemutatott probléma elkerülhető lenne, ha a számlát terhelő, illetve a számlán jóváíró műveletek együttesét atomian hajthatnánk végre. Ez azt jelenti, hogy vagy mindkét műveletet végre kell hajtani, vagy egyik műveletet sem szabad végrehajtani. Az ilyen jellegű atomi végrehajtás implementációja általában úgy történik, hogy az adatbázist módosító műveleteket egy segédterületen hajtják végre, és eredményük csak azután lesz *véglegesítve* magában az adatbázisban, miután a teljes munka befejeződött, és akkor az összes változás beépül az adatbázisba és láthatóvá válik más műveletek számára.

7.2.3. Tranzakciók

A 7.2.1. és 7.2.2. alfejezetekben megismert sorbarendezési és atomi műveletekkel kapcsolatos problémák megoldására vezették be az adatbázis-kezelőkben a tranzakciók fogalmát. Egy tranzakció alatt általában olyan adatbázis-elérési, illetve adatbázis-módosító műveletek csoportját értjük, melyeket atomian kell végrehajtani: vagy a csoportba tartozó minden műveletet végre kell hajtani, vagy – ha ez valamilyen oknál fogva nem lehetséges – egyet sem szabad közülük végrehajtani. Ezenkívül a korábbi SQL szabványok megkövetelték, hogy a tranzakciókat úgy kell végrehajtani, mintha sorba rendezve hajtanánk végre őket, vagyis a tranzakcióknak sorbarendezhetőeknek kellett lenniük. Az SQL2 szabványt implementáló rendszerekben – egy-két kivételtől eltekintve – a végrehajtással szemben alapértelmezés szerint támasztott követelmény a sorbarendezhetőség, de a programozó egyes tranzakciók párhuzamos végrehajtására ennél gyengébb végrehajtási követelményeket is rögzíthet.[6] Ezen gyengítési és módosítási lehetőségekkel egy későbbi alfejezetben fogunk foglalkozni.

Egy tranzakció egy adatbázist vagy annak sémáját lekérdező vagy módosító SQL utasítással kezdődik, vagyis nincs szükség arra, hogy ezt valamilyen tranzakciókezdést jelölő speciális utasítással közöljük az adatbázis-kezelővel. Ezzel szemben egy tranzakció befejeződésekor az adatbázis-kezelő rendszerrel közölnünk kell a befejeződés tényét az alábbi módok valamelyikén:

1. A COMMIT utasítással egy tranzakció sikeres befejeződését jelezzük. Egy sikeresen befejeződött tranzakció kezdete óta végrehajtott utasítások által az adatbázison végzett módosítások *véglegesíthetők*. A COMMIT utasítás végrehajtása előtt a módosítások nem véglegesítődnek, az általuk okozott adatmódosítások, illetve adatbázis-módosítások a párhuzamosan futó tranzakciók elől akár el is lehetnek takarva (ekkor ezek a módosítások csak a tranzakció véglegesítése után válnak láthatóvá a párhuzamosan futó tranzakciók számára).

[6] Egyes adatbázis-kezelőkben a tranzakciók végrehajtásával szemben támasztott alapértelmezés szerinti követelmények gyengébbek a szigorú sorbarendezhetőségnél.

Hogyan módosul az adatbázis egy tranzakció közben

A tranzakciók implementációja a különböző adatbázis-kezelő rendszerekben másképpen történhet. Előfordulhat, hogy a tranzakciók már véglegesítésük előtt is módosítják az adatbázis tartalmát. Előfordulhat, hogy egy ilyen tranzakció sikertelenül fejeződik be, és lehetséges, hogy a közbülső lépéseiben végzett adatmódosításokat más párhuzamosan futó tranzakciók felhasználták. Az adatbázis-kezelők ezt a problémát általában úgy oldják meg, hogy a tranzakciókban módosított adatelemeket lefoglalják, elzárják más tranzakciók elől egészen addig, amíg a tranzakció végre nem hajtja a végét jelző COMMIT vagy ROLLBACK utasítást. Ilyen zárakat vagy ezekkel ekvivalens eszközöket kell használnunk, ha a felhasználók sorbarendezhető módon kívánják tranzakcióik lefutását.

Mint azt a 7.2.4. alfejezetben látni fogjuk, az SQL2 számos más lehetőséget is kínál a tranzakciók közbülső lépéseiként elvégzett adatbázis-módosító műveletek eredményének kezelésére. Az is lehetséges, hogy a közbülső lépések során módosított adatelemek nem lesznek elzárva még akkor sem, ha később a módosításokat végző tranzakció sikertelenül fejeződik be, így mások is láthatják azok módosításait. A tranzakció tervezőjének a feladata annak eldöntése, hogy a közbülső lépések során megváltoztatott adatok láthatóságát meg kell-e akadályozni. Abban az esetben, ha ezt meg kell akadályozni, akkor a tranzakció implementálója megteheti ezt az SQL rendszerek által ilyen célokra biztosított valamelyik eszközzel, például adatelzárással (de egyes SQL rendszerekben léteznek ezzel ekvivalens más eszközök is).

2. A ROLLBACK utasítással egy tranzakció sikertelen befejezését jelezzük. Egy így befejezett tranzakció SQL utasításai által végrehajtott módosításokat az SQL rendszer meg nem történtekké teszi (azaz *visszacsinálja*), azok nem jelennek meg többé az adatbázisban. (Úgy is mondjuk, hogy a tranzakció *abortál*, módosításai *visszagörgetésre kerülnek*.)

7.13. példa: Most módosítsuk a 7.10. ábra `átutal()` nevű függvényét úgy, hogy a benne levő SQL utasítások egyetlen tranzakcióba tartozzanak. A tranzakció a 8. sorban kezdődik, amikor beolvassuk az első számla egyenlegét. Ha a 11. sorban látható feltétel értéke logikai igaz, akkor a 12. és 17. sorok között elvégezzük az átutalást, majd ha a módosításokat véglegesíteni akarjuk, akkor a 17. sor mögé oda kell tennünk az

```
EXEC SQL COMMIT;
```

SQL utasítást.

Ha viszont a 11. sorban látható feltétel értéke – például a fedezet hiánya miatt – logikailag hamis, akkor a 19. sorban látható `else` ágat kiegészíthetjük az

```
EXEC SQL ROLLBACK;
```

SQL utasítással (az `else` ág végén). Mivel az említett programban az `else` ágon nem végeztünk módosításokat az adatbázison, ezért mindegy, hogy kiírjuk-e oda ezt az SQL utasítást, hiszen nincs mit „visszacsinálnunk" az adatbázison. □

7.2.4. Csak olvasó tranzakciók

A 7.11. és 7.12. példákban olyan tranzakciókkal foglalkoztunk, amelyek beolvasnak, majd esetleg módosítanak valamilyen adatokat az adatbázisban. Az ilyen tranzakciók esetében szükség van a sorbarendezhetőségi problémák vizsgálatára. A 7.11. példában azt láttuk, hogy mi történik akkor, ha két program ugyanakkor próbál meg lefoglalni egy helyet egy járaton; a 7.12. példában pedig azt, hogy mi történne, ha a rendszer összeomlana egy adatbázis-módosítási művelet végrehajtása során. Nagyobb viszont a szabadságunk a tranzakciók végrehajtásának párhuzamosítása során azoknál a tranzakcióknál, amelyek csak olvassák az adatbázis tartalmát, és nem módosítják azt.[7]

7.14. példa: Tegyük fel, hogy elkészítettünk egy segédfüggvényt, amely adatokat olvas az adatbázisból annak eldöntésére, hogy egy adott hely szabad-e még. Ez a függvény nagyjából úgy működhet, mint a 7.8. ábra 1.–11. sora közötti részprogramja. Egy ilyen lekérdezőfüggvényből egyszerre akárhány példányt elindíthatunk anélkül, hogy az adatbázis tartalmában bármilyen kárt okozhatnánk. Legrosszabb, ami ilyenkor történhet, hogy mialatt beolvassuk egy adott hely foglaltsági állapotát, addig valaki velünk párhuzamosan lefoglalja azt. Az általunk visszakapott információ aktualitása akár ezredmásodperceken is múlhat (vagyis ha a lekérdezési műveletet néhány ezredmásodperccel később hajtottuk volna végre, akkor már más lenne az eredménye), de a válasz a legtöbb esetben megfelelne céljainknak. □

Ha az SQL adatbázis-kezelővel közöljük egy tranzakcióról, hogy az *csak olvasó* – azaz nem módosítja az adatbázis tartalmát –, akkor az adatbázis-kezelő felhasználhatja ezt az információt a tranzakciók optimálisabb ütemezésének megszervezésére. Az ehhez felhasznált mechanizmusokkal nem foglalkozunk részletesebben, de megemlítjük, hogy egy adatbázis-kezelő rendszer egyidejűleg több olyan *csak olvasó* tranzakciót is futtathat, amelyek ugyanazt az adatot olvassák, míg ugyanezek a tranzakciók nem futhatnának egy időben egy olyan másik tranzakcióval, amely az említett adatelemet módosítja.

Az alábbi SQL utasítással közölhetjük az SQL adatbázis-kezelővel, hogy a következő tranzakció *csak olvasó* tranzakció lesz:

```
SET TRANSACTION READ ONLY;
```

[7] Megjegyezzük, hogy felfedezhetünk bizonyos hasonlóságot a tranzakciók és a sormutatók kezelésének párhuzamosíthatósága között. Például a 7.1.10. alfejezetben említettük, hogy nagyobb mértékű párhuzamosítást tesz lehetővé a csak olvasásra használt sormutatók alkalmazása, az író-olvasó sormutatók alkalmazásához képest. Hasonlóan a csak olvasó tranzakciók is nagyobb mértékű párhuzamosságot biztosítanak az adatbázis-kezelő rendszernek.

A fenti utasítást a tranzakció első művelete előtt végre kell hajtani. Ha például lenne egy függvényünk, amely a 7.8. ábra 1.–11. sorát tartalmazná, akkor a 9. sor elé helyezett

```
EXEC SQL SET TRANSACTION READ ONLY;
```

beágyazott SQL utasítással jelezhetnénk az adatbázis-kezelő rendszer felé, hogy egy csak olvasó tranzakció következik. Ezt az utasítást nem tehetnénk a 9. sor után, mivel a 9. sorban már végrehajtott művelet során az adatbázis-kezelő a megkezdett tranzakciót író-olvasó tranzakciónak feltételezi.

Azt az egyébként alapértelmezésnek tekintett tényt is közölhetjük az adatbázis-kezelővel, hogy a következő tranzakció írni is és olvasni is fog. Ehhez az alábbi SQL utasítást használjuk:

```
SET TRANSACTION READ WRITE;
```

Ez azonban felesleges, mivel ez az alapértelmezés szerinti viselkedés.

7.2.5. Piszkos adatok olvasása

Piszkos adatoknak nevezik a még nem véglegesített tranzakciók által módosított adatokat. *Piszkos adatbeolvasási műveletnek* nevezik azt az adatbeolvasó műveletet, amely egy piszkos adatot olvas be. A piszkos adatok beolvasásának az a veszélye, hogy az azt kiíró tranzakció lehet, hogy sikertelenül fejeződik be (abortál), és ilyenkor az illető tranzakció által kiírt piszkos adatok törlődnek az adatbázisból – mintha az időközben más tranzakciók által már beolvasott adatok sosem lettek volna az adatbázisban. Ha egy időközben visszavont piszkos adatot beolvasó tranzakció véglegesítve lesz, akkor az adatbázis tartalma egy olyan adatra épül, amelyet időközben már visszavontak.

Egyes alkalmakkor a piszkos adatbeolvasási műveletek nem jelentenek problémát, de van, amikor problémákat okozhatnak. Az adatbázis-kezelő rendszernek általában sok időt és munkát okoz a piszkos adatok beolvasásának megakadályozása, ezért minden alkalmazásnál egyedi mérlegelést igényel annak eldöntése, hogy az adatbázis-kezelőnek meg kell-e akadályoznia ezt. Tekintsünk néhány példát a piszkos adatok beolvasásának lehetséges következményeiről.

7.15. példa: Tekintsük a 7.12. példabeli átutalási műveletet. Most azonban az ott megismertekkel ellentétben tételezzük fel, hogy az átutalást végző *P* program a következő lépések sorozatát hajtja végre:

1. Hozzáadja az átutalni kívánt összeget a 2. számla egyenlegéhez.

2. Ellenőrzi, hogy van-e elegendő pénz az 1. számlán.

   a) Ha az 1. számlán nincs meg a megfelelő fedezet, akkor levonja az előző lépésben a 2. számlára írt összeget, és a tranzakciót abortálja.

b) Ha az 1. számlán van fedezet, akkor kivonja az átutalni kívánt összeget az 1. számla egyenlegéből és véglegesíti a változtatást.

Ha a fenti *P* program más tranzakciókhoz viszonyítva sorba rendezhetően hajtódik végre, nem számít, hogy időlegesen pénzt írtunk a 2. számla egyenlegéhez: utólag senki sem fog tudni erről, ha az átutalás fedezet hiánya miatt meghiúsulna.

Most viszont tegyük fel, hogy megengedjük a piszkos adatok beolvasását. Tekintsünk három számlát: *A*1-et, *A*2-t és *A*3-at, egyenlegük rendre legyen 100 $, 200 $ és 300 $. Tegyük fel, hogy egy *T*1 nevű tranzakció végrehajtja a *P* programot, hogy átutaljon 150 $-t *A*1-ről *A*2-re. Nagyjából ugyanekkor egy másik – *T*2 nevű – tranzakció megkísérel átutalni szintén az említett *P* programmal 250 $-t *A*2-ről *A*3-ra. Előfordulhat, hogy az egyes résztvevők az alábbi sorrendben hajtják végre lépéseiket:

1. A *T*2 tranzakció végrehajtja az 1. lépést és hozzáad 250 $-t *A*3 egyenlegéhez, melyen így 550 $ lesz.
2. A *T*1 tranzakció végrehajtja az 1. lépést és hozzáad 150 $-t *A*2 egyenlegéhez, melyen így 350 $ lesz.
3. *T*2 végrehajtja a 2. lépés ellenőrzési részét, és látja, hogy *A*2 tartalmazza a 250 $ átutalásához szükséges fedezetet (350 $-t).
4. *T*1 végrehajtja a 2. lépés ellenőrzési részét, és látja, hogy *A*1-en nincs fedezet arra, hogy 150 $-t átutaljunk *A*1-ről *A*2-re.
5. *T*2 végrehajtja a 2/b) lépést: levonja a 250 $-t *A*2 egyenlegéből, amelyen most már csak 100 $ van, végül *T*2 véglegesítve lesz.
6. *T*1 végrehajtja a 2/a) lépést: levonja a 150 $-t *A*2 egyenlegéből, amelyen –50 $ lesz, majd *T*1 abortál.

A számlákon levő pénz mennyisége nem változott (az egyenlegek összege 600 $). Minthogy *T*2 piszkos adatot olvasott, nem tudtuk megakadályozni, hogy egy számlán negatív egyenleg képződjön, pedig ez volt a célja annak az ellenőrzésnek, mellyel az átutalás fedezetét vizsgáltuk. □

7.16. példa: Tekintsük a 7.11. példában megismert helyfoglalási feladatot, amit most a következő lépésekkel fogunk megoldani:

1. Megkeresünk egy szabad helyet és lefoglaljuk a `foglalt` mező értékének 1-re állításával. Ha nem találtunk szabad helyet, akkor abortáljuk a tranzakciót.
2. Megkérdezzük a felhasználót, hogy megfelel-e a lefoglalt hely. Ha megfelel, akkor véglegesítjük a tranzakciót; ha nem felel meg, akkor felszabadítjuk az illető helyet a `foglalt` mező értékét 0-ra állítva, majd megismételjük az 1. lépést egy új helyet keresve.

Ha a fenti módon nagyjából ugyanabban az időben két tranzakció próbál meg helyet lefoglalni, akkor előfordulhat, hogy egyikük lefoglal egy *S* helyet, amit a felhasználó visszautasít. Ha a másik tranzakció akkor hajtja végre az 1. lépést, amikor még az adatbázis azt tartalmazza, hogy az *S* hely foglalt, akkor a program az illető helyet nem ajánlja fel a másik felhasználónak annak ellenére, hogy az a hely már nem foglalt.

Hasonlóan a 7.15. példához, itt is a piszkos adatok olvasása okoz gondot: a második tranzakció olyan adatot olvasott, amelyet az első időlegesen kiírt, de később visszavont. □

Vajon mennyire fontos az a tény, hogy egy beolvasott adat piszkos? A 7.15. példában nagyon fontos volt: egy számla egyenlege negatívba ment miatta, pedig megpróbáltuk megelőzni. A 7.16. példában nem volt ilyen komoly a probléma, de ennek ellenére előfordulhatna, hogy a második utas nem kapja meg kedvenc helyét, vagy esetleg alaptalanul mondják neki azt, hogy nincs hely a gépen. Ez a probléma könnyen feloldható, ha a tranzakciót újból lefuttatjuk (ekkor az *S* hely szabad volta kiderül, ha időközben más nem foglalta le). Látható, hogy ez utóbbi esetben igenis van értelme megengedni a piszkos adatok olvasását, mivel az így elérhető munkagyorsítás a legtöbb helyfoglalás esetében messze kárpótol az említett ritkán előforduló probléma okozta kényelmetlenségekért.

Az SQL2 lehetővé teszi, hogy egy-egy tranzakcióra vonatkozóan kijelöljük, megengedhető-e piszkos adatok olvasása. Erre a 7.2.4. alfejezetben már említett `SET TRANSACTION` SQL utasítást használhatjuk. Egy, a 7.16. példában leírtakhoz hasonlóan működő tranzakció esetén, a következő utasítást írhatjuk:

```
1) SET TRANSACTION READ WRITE
2)                  ISOLATION LEVEL READ UNCOMMITTED;
```

Az előbbi utasítás két dolgot jelöl ki:

Az 1. sor azt közli az adatbázis-kezelővel, hogy a tranzakció adatokat írni is és olvasni is fog.

A 2. sor azt közli, hogy a tranzakciót ún. piszkos adatok olvasását megengedő elkülönítési szinten is szabad futtatni. A 7.2.6. alfejezetben foglalkozunk a 4 lehetséges elkülönítési szinttel, melyek közül eddig kettőt ismertünk meg, a sorba rendezhetőt és a piszkos adatok olvasását megengedőt.

Figyeljük meg, hogy ha egy tranzakció nem csak olvasó (vagyis legalább egy adatelemet ír vagy módosít az adatbázisban), és az elején megadjuk a `READ UNCOMMITTED` elkülönítési szintet, akkor ugyanitt meg kell adnunk a `READ WRITE` kulcsszavakat is. Emlékezzünk a 7.2.4. alfejezetből, hogy alapértelmezés szerint a tranzakciók író-olvasók, de az SQL2 esetében kivételt képeznek e szabály alól a piszkos adatok olvasását végző tranzakciók: ezek alapértelmezés szerint csak olvasók, mivel az ilyen típusú író-olvasó tranzakciók végrehajtása túlságosan kockázatos. Ezért ha egy író-olvasó, piszkos adatok olvasására is képes tranzakciót akarunk kezdeményezni, akkor a `READ WRITE` kulcsszavakat a fenti módon le kell írnunk.

7.2.6. További elkülönítési szintek

Az SQL2 összesen négy *elkülönítési szintet* definiál, melyek közül mi már kettőt megismerhettünk: a sorba rendezhetőt és a piszkos adatok olvasását megengedő szinteket. A maradék két elkülönítési szint a *véglegesített olvasást* és az *ismételhető olvasást* biztosító szintek. Egy tranzakciónál ezeket rendre a

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

valamint a

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

SQL utasításokkal jelölhetjük ki. Mindkét esetben a tranzakciók alapértelmezés szerint író-olvasók, így szükség esetén a fenti utasítások mögé kell írni a READ ONLY kulcsszavakat, ha csak olvasó tranzakciót akarunk kezdeményezni a megfelelő elkülönítési szinten. Továbbá lehetőségünk van az SQL2-ben alapértelmezésnek tekinthető sorba rendezhető végrehajtási módot választanunk a

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

SQL utasítással (de mivel az SQL2-ben ez az alapértelmezés, ezért ezt nem szükséges megadni).

A véglegesített olvasás elkülönítési szintjén – az elnevezésének megfelelően – nem megengedett a piszkos adatok beolvasása, de a párhuzamosan működő véglegesített tranzakciók módosításai láthatóak lesznek. Ezért ennél a szintnél előfordulhat, hogy egy tranzakció ugyanazt a lekérdezést többször egymás után végrehajtja, és az illető lekérdezés különböző végrehajtásainak alkalmával más-más eredményt kapjon vissza mindaddig, amíg a válaszok véglegesített tranzakciók eredményeként létrejött vagy módosult értékeket tükröznek.

7.17. példa: Tekintsük a 7.16. példában említett helyfoglaló műveletet, de tételezzük fel, hogy a tranzakciót a véglegesített olvasás elkülönítési szinten futtatjuk. Amikor a tranzakció átnézi a helyfoglalások adatbázisát, az 1. lépésben nem látja azokat a helyeket, amelyeket éppen abban a pillanatban próbálnak meg más tranzakciók lefoglalni.[8] Azonban, ha az utas visszautasítja valamelyik felajánlott helyet, akkor egy következő próbálkozás során előfordulhat, hogy más szabad helyeket fog találni, amint más tranzakciók sikeresen lefoglalnak egy-egy helyet, vagy éppen úgy döntenek, hogy a felajánlott hely nekik sem megfelelő, ezért új helyet próbálnak keresni. □

[8] Itt csak az egyes elkülönítési szintek jellemzőit rögzítjük, nem adunk meg ezeket megvalósító módszert. E példa elkülönítési szintjének egy lehetséges megvalósítása a következő: ha két tranzakció próbálja meg lefoglalni ugyanazt a helyet, akkor az adatbázis-kezelő rendszer az egyiküket abortálni fogja még akkor is, ha az illető tranzakciónak nem áll szándékában egy ROLLBACK utasítást végrehajtani.

Most tekintsük az ismételhető olvasás elkülönítési szintjét. Az elnevezés félrevezető, hiszen ha egy tranzakción belül egy lekérdezést többször végrehajtunk, akkor más-más választ kaphatunk az egyes végrehajtásokkor. Az ismételhető olvasás elkülönítési szintjén, ha egy lekérdezés egy adott sort visszaad az első végrehajtás alkalmával, akkor biztosak lehetünk abban, hogy az illető sort a lekérdezés másodszori végrehajtásakor is visszakapjuk. Ezen a szinten az is előfordulhat, hogy a lekérdezés egy következő végrehajtásakor ún. *fantom sorokat* is visszakapunk, amelyeket párhuzamosan működő tranzakciók illesztettek be az adatbázisba.

7.18. példa: Vizsgáljuk tovább a 7.16. és 7.17. példában megismert helyfoglalási problémát. Ha ezt a függvényt a megismételhető olvasás elkülönítési szintjén futtatjuk, akkor az első lekérdezés 1. lépése során visszakapott összes szabad helyet visszakaphatjuk a további lekérdezések során is.

Amennyiben eközben a `Járatok` adatbázisába újabb járatokat vesznek fel, mert például a légitársaság egy adott járatán repülőgépet váltott, és ezért több az utasok rendelkezésére álló férőhely, vagy valaki lemondta a helyfoglalását, akkor a megismételhető olvasás elkülönítési szinten futó tranzakciók az újonnan bekerült üléseket is látni fogják a további helyfoglalási kísérleteik során. □

7.2.7. Feladatok

7.2.1. feladat: Az ebben és a következő feladatban olyan programokat vizsgálunk, melyek az alábbi két reláción dolgoznak:

```
Termék(gyártó, modell, típus)
PC(modell, sebesség, memória, merevlemez, cd, ár)
```

A relációk a korábban már használt PC-adatbázisunkból származnak.

A feladat az egyes alfejezetekben specifikált program megvalósítása vagy megtervezése valamely programnyelven és beágyazott SQL utasításokkal. Ne felejtsük el a COMMIT, illetve a ROLLBACK utasításokat a megfelelő helyen kitenni, és ha van rá lehetőség, akkor csak olvasó tranzakciókat használjunk, és jelezzük azt az adatbázis-kezelőnek is.

a) Adott egy sebességérték és egy RAM mennyiség (mondjuk a megvalósítandó függvény paramétereként). Keressük ki az adatbázisból a megadott sebességű és megadott mennyiségű RAM memóriával rendelkező PC-k adatait, és írjuk ki azok modellazonosítóit és az áraikat.

* b) Egy megadott modellazonosítójú termékhez tartozó sorokat mindkét táblából töröljük ki.

c) Csökkentsük 100 $-ral egy megadott modellazonosítójú PC árát.

d) Adott egy számítógép gyártója, modellazonosítója, sebessége, RAM mérete, winchesterének mérete, CD-jének a sebessége és ára. Ellenőrizzük, hogy az illető PC benne van-e már az adatbázisban. Ha már benne van, akkor írjunk ki egy figyelmeztető üzenetet; ha még nincs benne, akkor vegyük fel a fenti két táblába.

! **7.2.2. feladat:** Vizsgáljuk meg a 7.2.1. feladat programjainál esetleg felmerülő olyan problémákat, melyek a végrehajtott műveletek atomiságával kapcsolatosak egy esetleges rendszerösszeomlás lehetőségét figyelembe véve.

! **7.2.3. feladat:** Tegyük fel, hogy a 7.2.1. feladat négy programjának valamelyikét végrehajtjuk egy *T* tranzakcióként, mialatt nagyjából ugyanabban az időben más tranzakciók a 7.2.1. feladat másik programjainak valamelyikét hajtják végre. Miben különbözne a *T* tranzakció működése azon két esetben, amikor minden tranzakció `READ UNCOMMITTED` elkülönítési szinten futna, illetve amikor `SERIALIZABLE` elkülönítési szinten futna minden tranzakció. A 7.2.1. feladat minden programjára vonatkozóan gondoljuk végig, mi történne, ha az illető program futna a *T* tranzakcióban.

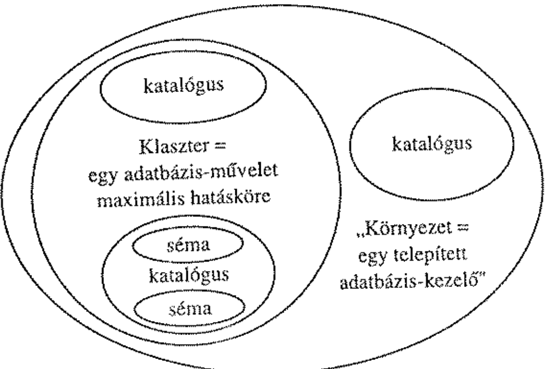
*!! **7.2.4. feladat:** Tegyük fel, hogy van egy állandóan futó *T* tranzakciónk, mely óránként ellenőrzi, hogy van-e az adatbázisban egy 200-nál nem kisebb sebességű PC, melynek ára 1000 $ alatt van. Ha talál egy ilyen PC-t, akkor kiírja a jellemzőit, és befejeződik a futása. Ezalatt az idő alatt a 7.2.1. feladatban megismert programok futhatnak az adatbázisban. Vizsgáljuk meg mind a négy elkülönítési szinten, mi a hatása az említett *T* tranzakció futására annak a ténynek, hogy az az illető elkülönítési szinten fut.

7.3. Az SQL környezet

Ebben a részben megvizsgáljuk egy adatbázis-kezelő rendszer környezetét, magukat az adatbázisokat, és azokat a programokat, amelyek az adatbázisok feldolgozását segítik. Megmutatjuk, hogy az adatbázisok hogyan lesznek klaszterekbe, katalógusokba, illetve sémákba szervezve, továbbá megvizsgáljuk a programok és az általuk feldolgozott adatok kapcsolatát. Mivel egy-egy adatbázis-kezelő vizsgálatakor ezen a területen sok implementációfüggő tényezővel kell számolni, ezért itt az SQL2 szabványban felmerülő ötleteket próbáljuk meg bemutatni.

7.3.1. Környezetek

Az *SQL környezet* egy olyan keretrendszer, amelyben adatokról és az azokat feldolgozó programokról beszélhetünk – az elnevezés a gyakorlatban általában egy telepített adatbázis-kezelő rendszerre vonatkozik. Például ha egy ABC nevű vállalat megvásárolja a Dundi-Soft vállalat SQL adatbázis-kezelő rendszerét, hogy saját számítógépein azt a rendszert futtassa, akkor az SQL környezet az adatbázis-kezelőt futtató számítógépeket és magukat az adatbázis-kezelő szoftverpéldányokat takarja.



7.11. ábra. *A környezetet alkotó adatbáziselemek struktúrája*

Az eddigiekben tárgyalt adatbáziselemek – így a táblák, nézettáblák, tartományok és önálló megszorítások – egy SQL környezeten belül vannak definiálva. Ezen adatbáziselemek egy hierarchikus rendszerbe vannak szervezve úgy, hogy a hierarchiában minden egyes elem egy jól meghatározható szereppel bír. Az SQL2-ben definiált struktúrákat a 7.11. ábrán szemléltetjük, szerepüket pedig itt röviden összefoglaljuk.

1. *Sémák*[9]: táblák, nézettáblák, önálló megszorítások, tartományok és más – e könyvben részletesen nem tárgyalt – típusú információk gyűjteménye (l. még a 7.3.2. alfejezet „Mi minden lehet még egy sémában" részét). A sémák az adatszervezés elemi összetevői. A séma fogalom talán a hétköznapi „adatbázis" fogalomhoz áll a legközelebb, de mint azt a (3)-as alfejezetben látni fogjuk, kevesebb annál.

2. *Katalógusok:* sémák csoportjai. Szerepük az egységes terminológia biztosítása szempontjából lényeges. Minden katalógus alá egy vagy több séma tartozik, egy katalóguson belül a sémákat egyedi nevükkel lehet megkülönböztetni egymástól. Minden katalógus tartalmaz egy speciális `INFORMATION_SCHEMA` nevű sémát, amely a katalógusban levő sémákról tartalmaz további információkat.

3. *Klaszterek:* katalógusok csoportjai. Minden felhasználóhoz hozzá van rendelve egy klaszter: azok a katalógusok, amelyeket az illető felhasználó elérhet (a katalógusokhoz és más elemekhez való hozzáférési jogosultságok beállítását lásd a 7.4. alfejezetben). Az SQL2 szabvány nem specifikálja pontosan a klaszter fogalmát, például nem határozza meg, hogy a különféle felhasználók klaszterei átfedhetik-e egymást anélkül, hogy azonosak lennének. Számunkra elég annyit rögzíteni, hogy egy lekérdezés során csak egy klaszterben tárolt információhoz férhetünk hozzá, ezért tekintik sokan a klasztert az „adatbázisnak".

[9] Ebben a környezetben a séma kifejezést adatbázisséma értelemben használjuk, nem pedig relációsémaként.

7.3.2. Sémák

Egy sémadefiníció legegyszerűbb formája a következő:

1. A definíciót a CREATE SCHEMA kulcsszavak vezetik be.
2. Ezt követi a séma neve.
3. Ezt követik a sémaelemek deklarációi: alaptáblák, nézettáblák, önálló megszorítások és tartományok deklarációi.

Eszerint egy séma definíciójának alakja a következő:

```
CREATE SCHEMA <sémanév> <elemdeklarációk>
```

Az elemdeklarációs rész szintaxisát az 5.7., 5.8. alfejezetekben, valamint a 6. fejezetben már megismertük. Mint említettük, vannak egyéb sémaelemek is, amelyeket az SQL2 definiál, de mi nem foglalkozunk velük.

7.19. példa: Deklarálhatnánk egy sémát, amely tartalmazza a filmekkel kapcsolatos példákban megismert öt relációt, és tartalmazhatna további nézettáblákat és más elemeket egyaránt. A 7.12. ábrán szemléltetjük egy ilyen deklaráció szerkezetét. □

Egyáltalán nem fontos a teljes sémát egyszerre deklarálnunk, mivel a CREATE, DROP, ALTER utasítások segítségével a sémák később is megváltoztathatók. Ilyenkor problémát jelent annak eldöntése, hogy az illető utasítás mely sémára vonatkozzon (hiszen különböző sémákban előfordulhatnak azonos nevű elemek, ezért ez alapján nem határozható meg egyértelműen az utasítás végrehajtásához használatos séma neve).

Az „aktuális" sémát a SET SCHEMA paranccsal jelölhetjük ki. Például a

```
SET SCHEMA FilmSéma;
```

utasítás a 7.12. ábrán leírt sémát teszi aktuális sémává, hogy azután a séma tartalmát módosító utasítások erre a sémára vonatkozzanak.

```
CREATE SCHEMA FilmSéma
CREATE DOMAIN AzonÉrtékek ... mint a 6.8. példában
                 További sémadeklarációk.
CREATE TABLE FilmSzínész ... mint a 6.4. ábrán
                 A másik 4 táblához tartozó definíciós utasítások.
CREATE VIEW FilmProd ... mint az 5.40. példában
                 További nézettáblák definíciói.
CREATE ASSERTION GazdagElnök ... mint a 6.10. példában
```

7.12. ábra. *Egy séma deklarációja*

Mi minden lehet még egy sémában?

A táblákon, nézettáblákon, tartományokon és önálló megszorításokon kívül más elemek is lehetnek egy sémában (további négy elemet fogunk megismerni).

Egy séma definiálhat egy *karakterkészletet*, ami jelek és kódjaik halmazából áll. Az ASCII a legelterjedtebb karakterkészlet, de egy SQL2 implementáció más – például nemzeti – karakterkészleteket is definiálhat.

Továbbá egy séma definiálhatja egy *karakterkészlet elemein a rendezést*. Az 5.1.3. alfejezetben azt mondtuk, hogy a karakterláncok lexikografikusan lesznek összehasonlítva, ahol a rendezés alapját a „kisebb, mint" reláció definiálja. Egy karakterkészlet elemeire vonatkozóan ezt a „kisebb, mint" összehasonlítást lehet definiálni.

Harmadszor a sémák definiálhatnak *karakterkészletek közötti átalakító* módszereket is. Végül egy séma tartalmazhat jogosultságokat kiosztó elemeket is, azt rögzítve, hogy mely elemhez ki férhet hozzá. Ezzel a 7.4. alfejezetben fogunk foglalkozni.

7.3.3. Katalógusok

Ahogyan a sémaelemeket egy sémában létrehozhatjuk vagy módosíthatjuk, úgy a sémákat egy feljebb levő hierarchiaszinten belül hozhatjuk létre vagy módosíthatjuk. E feljebb lévő szint a katalógusok szintje. Bár azt várnánk, hogy a katalógusok kezelése a sémák kezeléséhez hasonlóan történik, ez mégsem így van. Sajnos az SQL2 nem definiál egy

```
CREATE CATALOG <katalógusnév>
```

jellegű utasítást új katalógus létrehozására. Ennek ellenére az SQL2 biztosít egy

```
SET CATALOG <katalógusnév>
```

formájú utasítást, amellyel kiválaszthatjuk az „aktuális" katalógust, amelybe az újonnan definiált sémák kerülnek, illetve amelyre a sémamódosító műveletek vonatkoznak, ha nem adunk meg más egyértelmű katalógusnevet.

7.3.4. Kliensek és szerverek az SQL környezetben

Egy SQL környezet több katalógusok és sémák gyűjteményénél. Olyan elemeket is tartalmaz, amelyeknek célja a katalógusok és sémák által reprezentált adatbázison vagy adatbázisokon végezhető műveletek támogatása. Egy SQL környezetben alapvetően kétféle folyamatról beszélünk: SQL kliensekről és SQL szerverekről. Egy szerver lehetővé teszi egy adatbázis elemeinek módosítását, míg egy kliens feladata az, hogy

Sémaelemek teljes neve

A sémaelemek – mint például a táblák, nézettáblák – nevei a következő komponenseket tartalmazzák: az illető sémaelemet tartalmazó katalógus nevét, az illető elemet tartalmazó séma nevét, valamint magának a sémaelemnek a nevét pontokkal elválasztva. Eszerint például a `FilmSéma` sémabeli `Film` tábla neve a `FilmKatalógus` katalógusban az alábbi:

```
FilmKatalógus.FilmSéma.Film
```

Ha egy sémaelem az aktuális katalógusban van, akkor a megnevezésekor elhagyhatjuk a katalógusnevet. Ezen felül ha egy elem az aktuális sémában van, akkor a megnevezésekor a sémanevet is elhagyhatjuk, és ilyenkor csak a sémaelem nevét használhatjuk a sémaelem megnevezésére. Természetesen minden sémaelem megnevezésekor használhatjuk az előbb bemutatott három komponensből álló teljes nevet, ha az illető sémaelem nem az aktuális sémában vagy katalógusban van.

felhasználójának lehetővé tegye valamelyik szerver elérését. Általában a szerver egy nagy mennyiségű adat tárolására képes nagygépen fut, míg a kliensek futtatása inkább a kisebb kapacitású felhasználói munkaállomások feladata szokott lenni. Persze az is előfordulhat, hogy a kliens és a szerver ugyanazon a számítógépen fut.

7.3.5. Kapcsolatteremtés

Ha egy olyan számítógépen akarunk SQL alapú alkalmazásokat futtatni, amelyen egy SQL kliens érhető el, akkor kapcsolatot kell teremtenünk valamelyik SQL szerverrel az alábbi SQL utasítás végrehajtásával:

```
CONNECT TO <szerver neve> AS <kapcsolat neve>
```

A szerver neve SQL telepítésenként más és más lehet;[10] a legtöbb adatbázis-kezelőnél a `DEFAULT` kulcsszót beírhatjuk szerver névnek, és ekkor a kliens alkalmazás megpróbál megkeresni egy alapértelmezés szerint elérhető szervert (ha az adatbázis rendszergazdája megadott egy ilyen adatot).

A `CONNECT` utasításban megadott kapcsolatnevet[11] használhatjuk a későbbiekben a megnevezett szerverrel való kapcsolatra hivatkozáskor. Erre azért van szükség, mert

[10] A fordító megjegyzése: a szerver nevét megkérdezhetjük például a szervert karbantartó rendszergazdától.

[11] A fordító megjegyzése: a kapcsolatnevek általában addig élnek, amíg az SQL kliens programunk futása be nem fejeződik.

az SQL2 lehetőséget nyújt egyidejűleg több kapcsolat felépítésére, de ezek közül egyszerre csak egy lehet aktív (amelyen SQL parancsokat hajthatunk végre).

A kapcsolatok közül például a `kapcs1` nevű kapcsolatot az alábbi paranccsal tehetjük aktívvá:

```
SET CONNECTION kapcs1;
```

Ilyenkor a korábban aktív kapcsolat *alvó kapcsolattá* válik, amíg ismét nem aktiváljuk a `SET CONNECTION` SQL utasítással.

Egy korábban felépített kapcsolatot megszüntethetjük az alábbi SQL paranccsal:

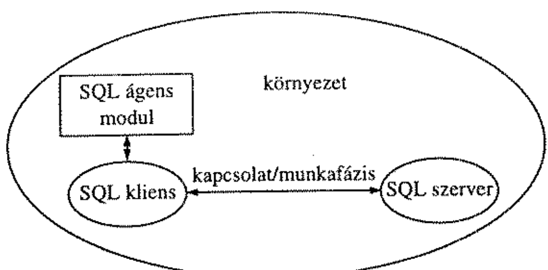
```
DISCONNECT kapcs1;
```

Ahol `kapcs1` a megszüntetni kívánt kapcsolat neve. Ezután a megnevezett kapcsolat se nem aktív, se nem alvó: a továbbiakban nem hivatkozhatunk rá.

Ha egy kapcsolatfelépítési művelet után nem akarunk a létrehozott kapcsolat nevére hivatkozni (mert például az az egyetlen, alapértelmezés szerint elérhető kapcsolat), akkor a `CONNECT TO` utasítás `AS` kulcsszavát és a mögötte következő kapcsolatazonosító nevét elhagyhatjuk. Sőt a kapcsolatfelépítési műveletet teljesen el is hagyhatjuk: ilyenkor ha végrehajtunk egy SQL utasítást, akkor az SQL parancsértelmező automatikusan felépít egy kapcsolatot az alapértelmezés szerint elérhető SQL adatbázis-kezelővel.

7.3.6. Munkafázisok

Egy kapcsolat aktív állapota alatt végrehajtott SQL utasítások egy ún. *munkafázist* képeznek. Egy munkafázis az őt létrehozó kapcsolathoz van kötve: amikor a kapcsolatot alvó állapotba helyezik, akkor a hozzá tartozó munkafázis is alvó állapotba kerül, majd amikor a kapcsolatot újra aktiválják, akkor a hozzá tartozó munkafázis is aktiválva lesz. Ez alapján a 7.13. ábrán a kliens és a szerver kommunikációját két különböző szemszögből vizsgálva nevezhetjük kapcsolatnak, illetve munkafázisnak.



7.13. ábra. *Az SQL kliensek és szerverek együttműködése*

Minden munkafázishoz hozzátartozik egy aktuális katalógus és ezen belül egy aktuális séma (ezeket módosíthatjuk a `SET SCHEMA`, valamint a `SET CATALOG` utasításokkal), és munkafázisonként történik az adatbázist elérni kívánó felhasználó igazoltatása is.

7.3.7. Modulok

A *modul* elnevezést az SQL2-ben az alkalmazói programok fogalmára használják. Az SQL2 szabvány háromféle modult különböztet meg, de a szabvány csak annyit követel meg, hogy az SQL implementációknak ezek közül legalább egyet biztosítaniuk kell a felhasználók számára.

1. *Általános SQL felület.* Ebben a használati módban a felhasználó egy SQL parancsértelmező előtt ül, amelynek SQL utasításokat írhat be, és a parancsértelmező a beadott utasításokat végrehajtja. Ekkor minden beadott SQL utasítás önmagában egy modulnak minősül. Bár ebben a könyvben példaprogramjainknál leggyakrabban egy ilyen SQL felületet feltételeztünk, a gyakorlatban ezt ritkán használják alkalmazások fejlesztésekor.

2. *Beágyazott SQL.* Ezzel foglalkoztunk a 7.1. alfejezetben, amikor SQL utasításokat ágyaztunk be valamilyen befogadó nyelven megírt programba (ezeket az SQL utasításokat vezettük be az `EXEC SQL` kulcsszavakkal). A beágyazott SQL utasításokat általában egy előfeldolgozó program alakítja befogadó nyelvi utasításokká, és a program futása során a megfelelő – számukra kijelölt – helyen lesznek végrehajtva.

3. *Valódi modulok.* Az SQL2 által támogatott legáltalánosabb modell, mely különféle tárolt eljárásokra és függvényekre épül (ezeket elkészíthetik akár valamilyen befogadó nyelven, akár SQL nyelven). Maga a program ezeknek az eljárásoknak és függvényeknek az együttműködéseként jön létre, ahol az egyes komponensek paraméterek átadásával és közösen elért változókkal kommunikálnak egymással.

Egy modul egy végrehajtását nevezik *SQL ágensnek*. A 7.13. ábrán egy SQL ágenst és egy modult egyetlen alkotóelemként tüntettünk fel (és az ábrán ez az alkotóelem épp egy SQL kliens segítségével kommunikál egy SQL szerverrel). Ennek ellenére ügyeljünk az ágens és a modul közt meglevő különbségekre, ami leginkább az operációs rendszerek program és folyamat absztrakciója közti különbségekre hasonlít: a modul egy végrehajtható kód, egy ágens pedig egy végrehajtás alatt álló modul.

7.4. Biztonság és felhasználói jogok az SQL2-ben

Az SQL2 kiköti az *engedélyazonosítók* létezését, amelyek lényegében felhasználói nevek. Az engedélyazonosítókat fel lehet ruházni különféle jogokkal, hasonlóan ahhoz, ahogyan ezt például egy operációs rendszer fájlrendszerében is megtehetjük. Van egy

speciális engedélyazonosító, a `PUBLIC`, amelyet bármelyik felhasználó használhat, azaz a mögötte levő jogosultsággal minden felhasználó fel van ruházva.

Egy UNIX-szerű fájlrendszerhez hasonlítva az alábbi analógiát szokás felhozni: egy fájlrendszerben általában értelmezve van az írás, olvasás és programvégrehajtási jogosultság. E három jogosultság megléte elegendő is és indokolt is, mivel a UNIX operációs rendszerben általában minden erőforrást fájlokkal reprezentálnak, és ez a három művelet jól jellemzi azt, amit egy-egy fájllal tenni szoktak.[12] Azonban az adatbázisok lényegesen bonyolultabbak a fájlrendszereknél, ezért az SQL2 szabványban definiált jogosultságok is ennek megfelelően összetettebbek.

Ebben a részben megismerjük, hogy milyen jogosultságokat biztosít az SQL2 az adatbáziselemeken, hogyan ruházhatunk fel felhasználókat (pontosabban engedélyazonosítókat) különféle jogosultságokkal, illetve hogyan vonhatunk vissza korábban kiadott jogosultságokat.

7.4.1. Jogosultságok

Az SQL2 hatféle jogosultságot definiál:

1. `SELECT`
2. `INSERT`
3. `DELETE`
4. `UPDATE`
5. `REFERENCES`
6. `USAGE`

Ezek közül az első négy relációkra vonatkozik: alaptáblákra vagy nézettáblákra. Amint erre a nevükből is következtethetünk, az illető jogosultság tulajdonosának joga van az illető típusú műveletet végrehajtani azon a táblán, amelyre a megfelelő jogosultsággal rendelkezik (az egyes jogosultságok lehetővé teszik rendre egy tábla vagy nézettábla lekérdezését, új sor beszúrását, sor törlését és sor módosítását). Egy SQL utasítást tartalmazó modul nem hajtható végre a modulban levő SQL utasítások végrehajtásához szükséges jogosultságok hiányában. Például egy SELECT-FROM-WHERE utasítás csak akkor hajtható végre, ha az összes benne hivatkozott táblára van `SELECT` jogosultsága. Hamarosan látni fogjuk, hogy a modulok hogyan ruházhatók fel különféle jogosultságokkal.

---

[12] A fordító megjegyzése: a UNIX operációs rendszerben egyes fájltípusoknál az említett három művelet speciális jelentéssel bír (gondoljunk például a könyvtárakon levő végrehajtásjelző bitekre).

```
1) INSERT INTO Stúdió (név)
2)   SELECT DISTINCT stúdióNév
3)   FROM Film
4)   WHERE stúdióNév NOT IN
5)     (SELECT név
6)     FROM Stúdió);
```

7.14. ábra. *Új stúdiók felvitele*

A REFERENCES jogosultság lehetővé tesz egy adott relációra történő hivatkozást egy épségi megszorítási feltételben. E megszorítások a 6. fejezetben megismert lehetséges megszorítások lehetnek: például önálló megszorítások, sor vagy attribútum alapú megszorítások, vagy hivatkozási épséget ellenőrző megszorítások. Egy megszorítás csak akkor ellenőrizhető, ha az ellenőrzéséhez szükséges összes adatbáziselemre megvan a REFERENCES jogosultság.

A USAGE jogosultság egy tartományon, illetve számos sémaelemen (a relációk és önálló megszorítások kivételével – l. 7.3.2. alfejezetet) van értelmezve, és azt jelenti, hogy a birtokosa az illető tartományt vagy sémaelemet felhasználhatja saját adatbáziselemeinek a definíciójában.

Az INSERT, UPDATE és REFERENCES jogosultságok paraméterezhetők is egy attribútummal. Ebben az esetben a megnevezett jog csak a megadott attribútumra vonatkozik. Mivel a birtokolható jogosultságok mennyisége nincs korlátozva, ezért egy reláció oszlopaiból álló tetszőleges részhalmazhoz adhatunk hozzáférési felhatalmazást.

7.20. példa: Tekintsük például, hogy milyen jogosultságok szükségesek az 5.12. ábrán látható INSERT utasítás végrehajtásához (az illető utasítást a 7.14. ábrán is láthatjuk). Először is, mivel ez a Stúdió relációba szúr be új sorokat, a végrehajtásához szükség van a Stúdió táblán INSERT jogra. Minthogy csak a beszúrni kívánt nevet határozzuk meg, ezért elegendő lenne csak a név attribútumra vonatkozó INSERT jogosultság is, amely csak olyan sorok beszúrását engedélyezi, amelyeknek csak a név attribútumának értékét adjuk meg, az újonnan beszúrni kívánt sor többi attribútuma NULL értékű lesz.

Vegyük észre, hogy a 7.14. ábrán látható SQL utasítás tartalmaz két beágyazott lekérdezést is a 2., illetve 5. soroktól kezdődően. Ahhoz, hogy ezeket a lekérdezéseket is kiértékelhessük, szükségünk van a Film és a Stúdió táblákra vonatkozóan egy SELECT jogosultságra. Természetesen az, hogy a Stúdió táblára INSERT jogosultságunk van, nem vonja maga után a SELECT jogosultság birtoklását is. □

7.4.2. Jogosultságok kialakítása

Már láttuk, hogy az SQL2 szabványban mit jelentenek a jogosultságok, és milyen jogosultságok szükségesek SQL utasítások végrehajtásához. Most azt fogjuk áttekinteni, hogyan adhatók jogosultságok egy művelet végrehajtására: hogyan lehet új jogokat létrehozni, illetve hogyan lehet már létrehozott jogosultságokat másik felhasználóknak

is engedélyezni. Ez a fejezet új jogosultságok létrehozásával foglalkozik, míg a jogosultságok engedélyezésével a 7.4.4. alfejezetben fogunk foglalkozni.

Minden SQL elemnek – mint például a sémáknak, moduloknak – van egy tulajdonosa. Egy elem tulajdonosa minden jogosultsággal rendelkezik az illető elem felett. Az SQL2-ben a tulajdonosi viszony a következő módok valamelyikén létesül:

1. Egy séma létrehozásakor a séma és a benne levő elemek alapértelmezés szerint a létrehozó felhasználó tulajdonába kerülnek. Ennek a felhasználónak tehát minden jogosultsága megvan a létrehozott elemekre vonatkozóan.

2. Egy munkafázist kezdeményező CONNECT utasításnál lehetőségünk van a munkafázist kezdeményező felhasználó nevének a megadására egy USER záradékban. Például a

   ```
   CONNECT TO Matild-sql-szerver AS kapcsl USER kirk;
   ```

   utasítás létrehoz egy kapcsolatot a Matild-sql-szerver SQL szerverrel, a kapcsolatra a későbbiekben kapcsl néven hivatkozhatunk. A fenti utasításban megadtuk a kapcsolatot kezdeményező felhasználó nevét: kirk. Az SQL adatbázis-kezelők általában valamilyen módon ellenőrzik, hogy a felhasználó valóban az-e, akinek kiadja magát. Ezt megtehetik például egy jelszó bekérésével.

3. Egy modul létrehozásakor az AUTHORIZATION kulcsszót követő záradékban adhatjuk meg a létrehozott modul tulajdonosát. A modulok létrehozásának részleteivel nem foglalkozunk, de megemlítjük, hogy az SQL2 ilyen irányú lehetőségei nagyon sokrétűek. Az említett AUTHORIZATION záradék alakja a következő:

   ```
   AUTHORIZATION picard;
   ```

   a fenti záradékkal kiegészített moduldefiníciós utasítással létrehozott modul tulajdonosa picard nevű felhasználó lesz. Előfordulhat, hogy egy modulnak egyáltalán nincs megadva a tulajdonosa. Ezek a modulok mindenki által végrehajthatóak, de a sikeres végrehajtáshoz szükség van a modulban levő SQL utasítások végrehajtásához szükséges jogosultságokra is, amikkel például a végrehajtást kezdeményező felhasználó rendelkezhet.

7.4.3. Jogosultságok ellenőrzése

Mint azt korábban már láthattuk, minden modulnak, sémának és munkafázisnak van egy tulajdonosa: SQL terminológiával ezt úgy mondhatnánk, hogy mindegyikhez hozzá van rendelve egy engedélyazonosító. Egy SQL művelet végrehajtásában általában két elem vesz részt:

1. A művelet által elért és módosított adatbáziselemek.

2. A művelet végrehajtását kezdeményező ágens.

A végrehajtó ágens a végrehajtáshoz szükséges jogokat egy bizonyos engedélyazonosító, az ún. *aktuális engedélyazonosító* alapján kapja. Az aktuális engedélyazonosító

a) vagy a modul engedély azonosítója, ha a végrehajtás alatt álló modulnak van engedélyazonosítója

b) vagy pedig a munkafázis-engedély azonosítója.

Egy SQL művelet csak akkor hajtható végre, ha az aktuális engedélyazonosító biztosítja az ehhez szükséges összes jogosultságot.

7.21. példa: A jogosultságok ellenőrzési mechanizmusának áttekintéséhez tekintsük újra a 7.20. példát. Feltételezhetjük, hogy a hivatkozott `Film` és `Stúdió` táblák a `FilmSéma` sémában lettek létrehozva, és a séma létrehozója – így tulajdonosa is – `janeway`. A tulajdonos, `janeway` minden jogosultsággal rendelkezik az illető táblák felett. A tulajdonos másokat is felruházhat az említett adatbáziselemekre vonatkozó jogosultságokkal, ahogyan azt a 7.4.4. alfejezetben majd megismerjük, de most tegyük fel, hogy még nem adott másoknak semmilyen jogosultságot ezekre a táblákra. Számos mód van a 7.20. példa beszúrási műveletének végrehajtására.

1. Az említett beszúrási művelet végrehajtódhat egy olyan modulban, amelyet `janeway` hozott létre oly módon, hogy a létrehozó utasítás tartalmazta az `AUTHORIZATION janeway` záradékot. Ilyenkor a modult bárki is hajtja végre (vagyis bármi is az aktuális engedélyazonosító), az aktuális engedélyazonosító a modulhoz eltárolt engedélyazonosító lesz. Ez azt jelenti, hogy az SQL utasítás végrehajtása során pontosan `janeway` jogosultságaival fut, aki – mivel `janeway` tulajdonos – minden jogosultságot birtokol a `Film` és `Stúdió` táblákra vonatkozóan.

2. Az említett beszúrási művelet végrehajtható egy olyan modulból is, amelynek nincs tulajdonosa. Ha ilyenkor a `janeway` azonosítójú felhasználó bejelentkezik egy `CONNECT TO` utasítással a `USER janeway` záradékot is megadva, akkor az aktuális engedélyazonosító `janeway` lesz, az ő jogosultságaival kezd el futni az illető modul, így minden jogosultsága megvan az illető beszúrás elvégzésére.

3. Most tegyük fel, hogy a tulajdonos, `janeway` felhasználó a szóban forgó táblákra vonatkozó összes jogosultsággal felruházza a `sisko` nevű másik felhasználót (vagy esetleg a `PUBLIC` azonosítójú, speciális felhasználót, ami lényegében az „összes felhasználót" jelenti). Ha most a szóban forgó beszúrási utasítás egy olyan modulban van, amely el van látva az

   ```
   AUTHORIZATION sisko
   ```


   záradékkal, akkor mivel a modul futtatása `sisko` felhasználó jogosultságaival történik, és neki megvannak a végrehajtáshoz szükséges jogosultságai, ezért a beszúrás sikeres lesz.

4. Mint az előző – azaz 3. – esetben, itt is tegyük fel, hogy `janeway` felhasználó az illető táblákra vonatkozó összes jogosultsággal felruházta a `sisko` nevű felhasználót, de most az említett beszúrási utasítás legyen egy olyan modulban, amelynek nincs tulajdonosa. Ha ez a modul egy olyan munkafázisban lesz végrehajtva, amelynek az engedélyazonosítója egy `USER sisko` záradékkal lett beállítva, akkor az aktuális engedélyazonosító `sisko`, ezért a végrehajtáshoz szükséges jogosultságok megvannak. □

Számos jogosultságokkal kapcsolatos alapelvet megismerhettünk a 7.21. példában, amit most röviden összefoglalunk:

- A végrehajtáshoz szükséges jogosultságok rendelkezésre állnak, ha az elérni kívánt elemek és adatok tulajdonosának az engedélyazonosítója megegyezik az aktuális engedélyazonosítóval. Erre volt példa az 1. és 2. eset.
- A végrehajtáshoz szükséges jogosultságok rendelkezésre állnak, ha az aktuális engedélyazonosítóval rendelkező felhasználónak a tulajdonos jogosultságot adott az általa elérni kívánt adatbáziselemek elérésére, vagy ha a tulajdonos a `PUBLIC` nevű felhasználónak adta meg a szóban forgó jogosultságokat. Erre volt példa a 3. és 4. eset.
- Ha egy olyan modult hajtunk végre, amelynek tulajdonosa megegyezik az elérni kívánt adatok tulajdonosával, akkor megvannak a végrehajtáshoz szükséges jogosultságok. Erre volt példa az 1. és a 3. eset.
- Ha egy nyilvánosnak minősített modult hajtunk végre egy olyan munkafázisban, amely munkafázisnak az aktuális engedélyazonosítója olyan felhasználóé, akinek jogosultsága van a szükséges adatelemek elérésére, akkor a végrehajtásnak nincsenek „jogi akadályai". Ezt szemléltettük a fenti 2. és 4. esetekben.

7.4.4. Jogosultságok megadása

A 7.21. példában láthattuk, hogy miért fontos, hogy a felhasználók (azaz az engedélyazonosítók) rendelkezzenek bizonyos jogosultságokkal, de idáig jogosultságok szerzésének csak egyetlen módjával ismerkedtünk meg: egy elemre vonatkozóan tulajdonosi minőségben megszerzett jogosultságokkal. Az SQL2 biztosítja a `GRANT` utasítást, mellyel egy felhasználó jogosultságokat ruházhat át egy másik felhasználóra – egy felhasználó az átruházott jogosultságait nem veszíti el, így itt inkább jogosultságok „lemásolásáról" beszélhetünk (a továbbiakban erre a célra az engedélyezés kifejezést használjuk).

Minden egyes jogosultsághoz tartozhat egy *engedélyezési képesség* is, ami a jogosultságok továbbadásakor („lemásolásakor") nem kerül automatikusan továbbadásra. Ha például egy felhasználónak van `SELECT` jogosultsága a `Film` táblára engedélyezési képességgel együtt, egy másik felhasználónak pedig ugyanezen jogosultsága van, de engedélyezési képesség nélkül, akkor mindketten elérhetik az illető táblát SQL lekérdezési műveletekben. A két felhasználó lehetőségei abban különböznek egymástól, hogy míg az első – engedélyezési képességgel bíró – felhasználó megadhatja e jogosultságot egy harmadik személynek (esetleg az engedélyezési képességével együtt), addig a másodikként említett – engedélyezési képességgel nem bíró – felhasználó ezt a jogosultságát nem adhatja tovább másoknak.

Az *engedélyező utasítás* alakja a következő:

1. Az utasítás a `GRANT` kulcsszóval kezdődik.
2. Ezt követi a kiosztani kívánt jogosultságok listája, pl. `SELECT` vagy `INSERT(név)`. Ha az utasítás végrehajtója az összes átruházható jogosultságát át akarja adni, akkor ehelyett állhat egyszerűen az `ALL PRIVILEGES` kulcsszó is.
3. Ezt követi az `ON` kulcsszó.
4. Ezt követi egy adatbáziselemnek a neve. Az erre az adatbáziselemre vonatkozó átruházható jogosultságok közül kellett megnevezni a 2. pontban azokat a jogosultságokat, amelyeket tovább akarunk adni valakinek. Itt – a 4. pontban – adatbáziselemként egy alaptábla vagy egy nézettábla nevét szokás megadni, de megadható itt akár egy tartománynév, akár más sémaelemek nevei is (lásd a további sémaelemekről a 7.3.2. alfejezetnél látható „Mi minden lehet még egy sémában" című kiegészítést), de ezeknek nevét egy megfelelő kulcsszó, például `DOMAIN` mögött kell megadni.
5. Ezt követi a `TO` kulcsszó.
6. Ezután felhasználónevek (engedélyazonosítók) következnek (egy vagy akár több is).
7. Ezután megadhatók még a `WITH GRANT OPTION` kulcsszavak is.

   Azaz egy engedélyező utasítás formája az alábbi:

   GRANT <jogosultságok listája> ON <adatbáziselemek> TO <felhasználók nevei>

és ezt követheti még egy `WITH GRANT OPTION` rész is.

Hogy ez az utasítás sikeresen végrehajtható legyen, szükség van arra, hogy az ezt végrehajtó felhasználó rendelkezzen a 2. pontban felsorolt jogosultságokkal és a hozzájuk tartozó engedélyezési képességgel is. De megemlítjük, hogy egy felhasználó átadhatja akár saját jogosultságait is, de átadhatja akár azoknak egy leszűkített lehetősé-

geket biztosító változatát is. Például az `INSERT(név)` jogosultságot a `Stúdió` táblára vonatkozóan továbbadhatja egy olyan felhasználó, aki rendelkezik erre a táblára egy `INSERT` jogosultsággal és engedélyezési képességgel.

7.22. példa: A `FilmSéma` sématulajdonosa, `janeway` átruházza a `Stúdió` táblára vonatkozó `SELECT` és `INSERT` jogosultságát, valamint a `Film` táblára vonatkozó `SELECT` jogosultságát `kirk` és `picard` felhasználóknak.

```
Film(cím, év, hossz, színes, stúdióNév, producerAzon)
Stúdió(név, cím, elnökAzon)
```

Sőt átadja az ezekre vonatkozó engedélyezési képességét is. Az ehhez szükséges `GRANT` utasítás alakja a következő:

```
GRANT SELECT, INSERT ON Stúdió TO kirk, picard
  WITH GRANT OPTION;
GRANT SELECT ON Film TO kirk, picard
  WITH GRANT OPTION;
```

Most ha `picard` tovább akarja adni az előbb megszerzett jogosultságait `sisko` felhasználónak, de az engedélyezési képessége nélkül, akkor ezt az alábbi utasításokkal teheti meg:

```
GRANT SELECT, INSERT ON Stúdió TO sisko;
GRANT SELECT ON Film TO sisko;
```

Ezek után `kirk` is átadja `sisko`-nak a `Stúdió` táblára vonatkozó `SELECT` és `INSERT(név)` jogosultságát, és a `Film` táblára vonatkozó `SELECT` jogosultságát. Ezt az alábbi utasításokkal tette meg:

```
GRANT SELECT, INSERT(név) ON Stúdió TO sisko;
GRANT SELECT ON Film TO sisko;
```

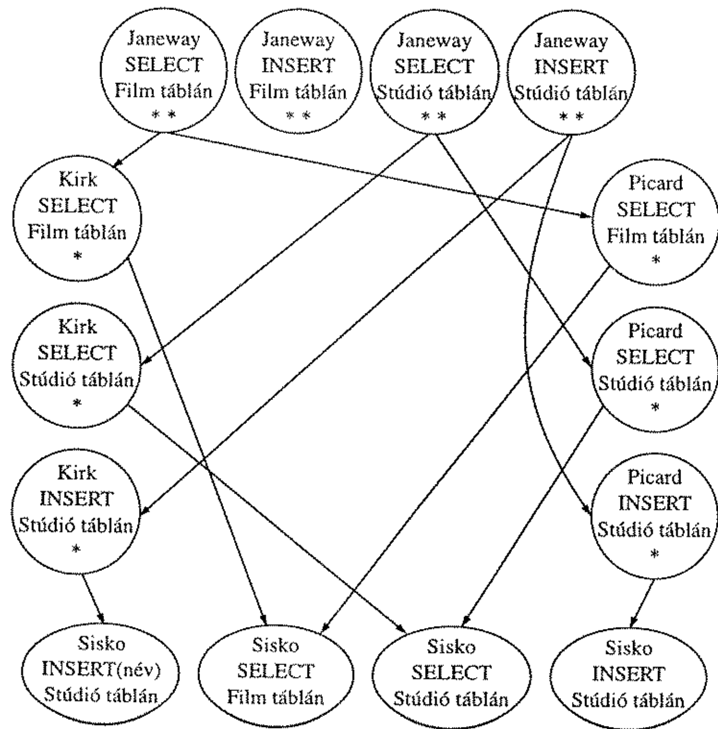
Látható, hogy `sisko` a `Film` és a `Stúdió` táblákra vonatkozó `SELECT` jogosultságát két felhasználótól is megkapta. Az `INSERT(név)` jogosultságát a `Stúdió` táblára szintén két helyről is megkapta: közvetlen `kirk`-től, valamint `picard`-tól is `picard` `INSERT` jogosultságának korlátozásával. □

7.4.5. Engedélyezési diagramok

Mivel a jogosultságok továbbadása során jogosultságok bonyolult rendszere jöhet létre, ezért a kiosztott jogosultságok áttekintésének segítségére kidolgoztak egy jól használható eszközt, az *engedélyezési diagramokat*. Egy SQL rendszer egy ilyen diagra-

mot tárol az egyes felhasználók jogosultságainak nyilvántartására, illetve annak nyilvántartására, hogy mely jogosultság mely felhasználótól származik, kitől lett átruházva (ez utóbbi különösen akkor érdekes, ha egyes jogosultságokat vissza kívánnak vonni – l. a 7.4.6. alfejezetet). Az engedélyezési diagram egy olyan gráf, amelynek csomópontjai megfeleltetésbe hozhatók egy felhasználóval és egy jogosultsággal. Ha egy *U* felhasználó átruház egy *P* jogosultságot egy *V* felhasználónak, és ezt *U* egy *Q* jogosultsága alapján teszi (ekkor *Q* lehet *P*, vagy *P*-t tartalmazó általánosabb jogosultság, mindkettő engedélyezési képességgel együtt), akkor ezt a tényt a diagramon az *U*/*Q* csomóponttól a *V*/*P* csomópontig húzott vonallal jelölhetjük.

7.23. példa: A 7.15. ábra szemlélteti a 7.22. példában megadott engedélyezési műveletek végrehajtásakor létrejött engedélyezési diagramot. A jelölési konvenciónkról megjegyezzük, hogy egy felhasználó-jogosultság pár mögé tett * karakterrel jelöljük az illető jogosultságra vonatkozó engedélyezési képességet, két csillag (azaz **) karakterrel jelöljük a tulajdonosi viszonyból származó jogosultságokat (ezek a jogosultsá-



7.15. ábra. *Egy engedélyezési diagram*

gok nem engedélyezéssel születtek). Ez a megkülönböztetés a jogosultságok visszavonásakor lesz majd érdekes, mint azt a 7.4.6. alfejezetben látni fogjuk. A tulajdonosi jogosultság maga után vonja az engedélyezési képességet. □

7.4.6. Jogosultságok visszavonása

A megszerzett jogosultságok bármikor visszavonhatók. Egy jogosultság visszavonásának *következetesnek* kell lennie, ami azt jelenti, hogy ha egy engedélyezési képességgel kiegészített jogosultságot vonnak vissza, akkor az illető jogosultsággal és engedélyezési képességgel más felhasználóknak átruházott jogosultságokat is mind vissza kell vonni. A jogosultság-visszavonási utasítás legegyszerűbb alakja a következő:

1. Az utasítást a REVOKE kulcsszó vezeti be.
2. Ezt követi a visszavonni kívánt jogosultságok listája.
3. Ezt követi az ON kulcsszó.
4. Ezt követően azoknak az adatbáziselemeknek a listáját kell megadni, amelyekre vonatkozóan a 2. alfejezetben megnevezett jogosultságokat vissza akarjuk vonni.
5. Ezt követi a FROM kulcsszó.
6. Ezt követően kell megadni azoknak a felhasználóknak (engedélyazonosítóknak) az azonosítóit, amelyektől a megnevezett jogosultságokat a megnevezett elemeken vissza akarjuk vonni.

   Azaz az utasítás alakja a következő:

   ```
   REVOKE <jogosultságlista> ON <adatbáziselemek listája>
                             FROM <felhasználók listája>
   ```

   Ezenkívül a fenti utasítást kiegészíthetjük az alábbi elemekkel:

- Az utasítás végére írhatjuk a CASCADE kulcsszót, ami következetes jogosultság-visszavonást eredményez. Ilyenkor a visszavont jogosultsággal létrehozott más jogosultságok is vissza lesznek vonva, amit tulajdonosuk más forrásból nem szerzett meg: ha egy *U* felhasználó visszavonja *V* felhasználótól a korábban *U* felhasználó *Q* saját jogosultságán ráruházott *P* jogosultságot, akkor az engedélyezési diagramnak törölve lesz az az éle, amely az *U*/*Q* csúcsból a *V*/*P* csúcsig vezet. Végül törölve lesznek azok a csomópontok is, amelyekbe nem vezet él tulajdonosi jogosultságokat reprezentáló csúcsokból.
- Az utasítás végére CASCADE kulcsszó helyett írhatnánk RESTRICT kulcsszót is, ami azt jelenti, hogy a jogosultság-visszavonási utasítás nem hajtható végre, ha az

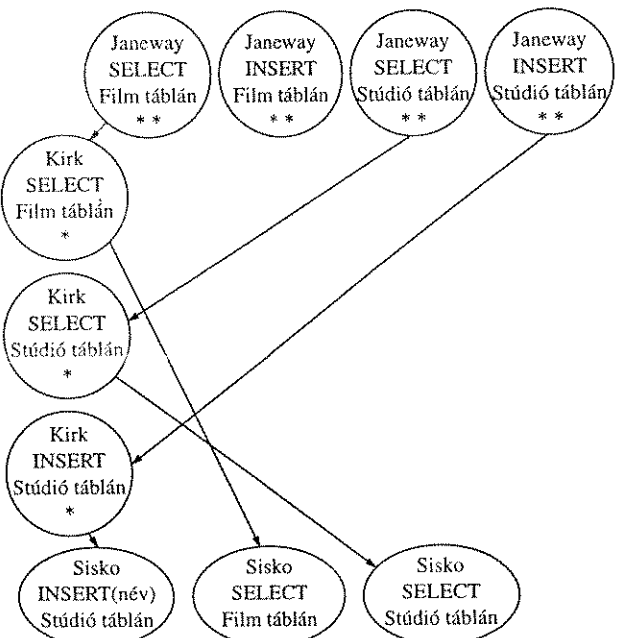
előző alfejezetben leírt következetes jogosultság-visszavonási szabályok szerint olyan jogosultságok visszavonását eredményezné, amelyek tovább lettek már adva.

- A REVOKE kulcsszó helyett írhatjuk a REVOKE GRANT OPTION FOR kulcsszavakat is, ami nem magának a jogosultságnak, hanem csak engedélyezési képességének a visszavonását eredményezi. Ez az utasítás is kombinálható az előbb említett CASCADE és RESTRICT záradékokkal, amikor az engedélyezési diagram a fenti módon át lesz vizsgálva további visszavonandó jogosultságok után.

7.24. példa: A 7.22. példát folytatva tételezzük fel, hogy janeway visszavonja picard-nak átadott jogosultságait az alábbi utasításokkal:

```
REVOKE SELECT, INSERT ON Stúdió FROM picard CASCADE;
REVOKE SELECT ON Film FROM picard CASCADE;
```

A 7.15. ábráról töröltük janeway megfelelő jogosultságaiból picard megfelelő jogosultságaihoz vezető nyilakat. Mivel a CASCADE kulcsszót is megadtuk, ezért töröltük azokat a jogosultságokat is, amelyek nem érhetők el kétcsillagos (tulajdonosi) jogosultságokat reprezentáló csúcsokból. A 7.15. ábrát megvizsgálva látható, hogy



7.16. ábra. *Az engedélyezési diagram picard jogosultságainak visszavonása után*

picard jogosultságai nem érhetők el kétcsillagos csomópontból (hiszen, ha elérhetők lennének, akkor vezetne oda út). Azt is láthatjuk, hogy sisko INSERT jogosultsága a Stúdió táblán már szintén nem érhető el. Ezért nemcsak picard jogosultságait kell törölni a diagramról, hanem sisko INSERT jogosultságát is.

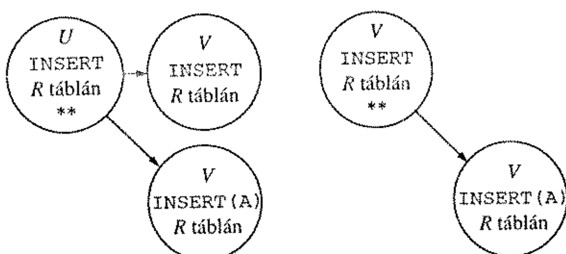
Látható, hogy nem kell törölni sisko SELECT jogosultságát a Film és Stúdió táblákra, és nem kell törölni az INSERT jogosultságát a Stúdió(név) sorokra, mivel ezek elérhetők janeway tulajdonosi jogosultságait reprezentáló csúcsokból. A diagramon az elmondott változtatásokat elvégezve a 7.16. ábrát kapjuk. □

7.25. példa: Van még néhány egyéb vonatkozás is, amit absztrakt példákkal fogunk szemléltetni. Először is, amikor visszavonunk egy általános p jogosultságot, akkor nem vonjuk vissza p specializált változatait. Tekintsük erre a következő példát, melyben U felhasználó – az R reláció tulajdonosa – ad egy INSERT jogosultságot az R relációra V felhasználónak, és ezenkívül ad egy INSERT(A) jogosultságot is ugyanerre a relációra.

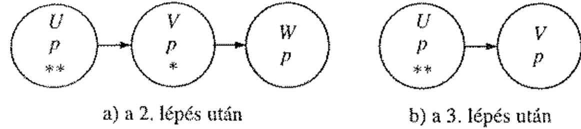
Lépés	*Végrehajtója*	*Tevékenység*
1	U	GRANT INSERT ON *R* TO *V*
2	U	GRANT INSERT(A) ON *R* TO *V*
3	U	REVOKE INSERT ON *R* FROM *V* RESTRICT

Amikor U visszavonja INSERT jogosultságát V-től, V felhasználó INSERT(A) jogosultsága megmarad. A 2. és 3. lépések után az engedélyezési diagram állapotát a 7.17. ábrán szemléltetjük.

Figyeljük meg, hogy a 2. lépés után két elválasztott csomópont marad V felhasználó két hasonló, de nem egyező jogosultságának ábrázolására. Szintén észrevehető, hogy a RESTRICT kulcsszó nem akadályozta meg a jogosultság visszavételét, mivel V nem adta tovább e jogosultságát más felhasználóknak (nem is adhatta volna tovább, mivel nem volt továbbadási képessége). □



7.17. ábra. *Egy általánosabb jogosultság visszavonásakor a specifikusabb jogosultságok megmaradnak*



7.18 ábra. *Egy adott jogosultság engedélyezési képességének visszavonása nem vonja maga után az illető jogosultság visszavonását*

7.26. példa: Most tekintsünk egy hasonló példát, ahol *U* átad *V*-nek egy *p* jogosultságot engedélyezési képességgel együtt, majd csak az engedélyezési képességet vonja vissza. Ebben az esetben módosítanunk kell *V* csomópontot az elvesztett képesség jelölése céljából, és törölni kell az összes jogosultságot, amit *p* jogosultsága alapján *V* továbbadott. Ezt úgy tehetjük meg, hogy töröljük a *V/p* csomópontból kiinduló éleket. Az elvégzett lépések sorozata a következő:

Lépés	*Végrehajtója*	*Tevékenység*
1	*U*	GRANT *p* TO *V* WITH GRANT OPTION
2	*V*	GRANT *p* TO *W*
3	*U*	REVOKE GRANT OPTION FOR *p* FROM *V* CASCADE

Az 1. lépésben *U* átadja a *p* jogosultságot *V*-nek engedélyezési képességgel. A 2. lépésben *V* az 1. lépésben szerzett engedélyezési képességét felhasználva átadja a *p* jogosultságot *W*-nek. Ezután a diagram a 7.18.a) ábra szerint néz ki.

Majd a 3. lépésben *U* visszavonja az engedélyezési képességet a *p* jogosultságra vonatkozóan *V*-től, de magát a *p* jogosultságot nem vonja vissza. Azaz a csillagot törölni kell a *V/p* csomópontról. De egy csillag nélküli csomópontból nem indulhatnak kifelé élek (hiszen ezek nem lehetnek átadott jogosultságok forrásai), ezért törölnünk kell a *V/p* csomópontból a *W/p* csomópontba húzott élt.

Most pedig a *W/p* csomópontba nem vezet él olyan kétcsillagos (**-gal ellátott) csomópontból, amely *p* jogosultság eredetét tartalmazná, ezért a *W/p* csomópontot törölni kell a diagramból. De a *V/p* csomópont megmarad: egyszerűen le lesz szedve róla a csillag, ami az engedélyezési képességet jelölte. Az eredményként kapott gráfot a 7.18.b) ábrán szemléltetjük. □

7.4.7. Feladatok

7.4.1. feladat: Adjuk meg milyen jogosultságok szükségek az alábbi lekérdezések végrehajtásához. Minden lekérdezéshez jelöljük meg mind a legspecifikusabb, mind pedig a legáltalánosabb szükséges jogosultságokat.

a) Az 5.3. ábra lekérdezése.

b) Az 5.5. ábra lekérdezése.

* c) Az 5.12. ábrán látható beszúrási művelet.

d) Az 5.29. példán látható törlési művelet.

e) Az 5.31. példán átható módosító utasítás.

f) A 6.4. ábrán látható ellenőrzés.

g) A 6.10. példában látható önálló megszorítás.

* **7.4.2. feladat:** Mutassuk meg a 4. és a 6. lépés végrehajtása után kialakult engedélyezési diagramot a 7.19. ábrán leírt műveletek végrehajtásakor. Tételezzük fel, hogy a *p* jogosultság olyan relációra vonatkozik, melynek tulajdonosa *A*.

Lépés	*Végrehajtója*	*Tevékenység*
1	*A*	GRANT *p* TO *B* WITH GRANT OPTION
2	*A*	GRANT *p* TO *C*
3	*B*	GRANT *p* TO *D* WITH GRANT OPTION
4	*D*	GRANT *p* TO *B*, *C*, *E* WITH GRANT OPTION
5	*B*	REVOKE *p* FROM *D* CASCADE
6	*A*	REVOKE *p* FROM *C* CASCADE

7.19. ábra. *A 7.4.2. feladathoz tartozó műveletsorozat*

7.4.3. feladat: Mutassuk meg az 5. és a 6. lépés végrehajtása után kialakult engedélyezési diagramot a 7.20. ábrán leírt műveletek végrehajtásakor. Tételezzük fel, hogy a *p* jogosultság olyan relációra vonatkozik, melynek tulajdonosa *A*.

Lépés	*Végrehajtója*	*Tevékenység*
1	*A*	GRANT *p* TO *B*, *E* WITH GRANT OPTION
2	*B*	GRANT *p* TO *C* WITH GRANT OPTION
3	*C*	GRANT *p* TO *D* WITH GRANT OPTION
4	*E*	GRANT *p* TO *C*
5	*E*	GRANT *p* TO *D* WITH GRANT OPTION
6	*A*	REVOKE GRANT OPTION FOR *p* FROM *B* CASCADE

7.20. ábra. *A 7.4.3. feladathoz tartozó műveletsorozat*

! **7.4.4. feladat:** Mutassuk meg az alábbi lépések végrehajtása után kialakult engedélyezési diagramot. Tételezzük fel, hogy a *p* jogosultság egy olyan táblára vonatkozik, amelynek tulajdonosa az *A* felhasználó.

Lépés	*Végrehajtója*	*Tevékenység*
1	*A*	GRANT *p* TO *B* WITH GRANT OPTION
2	*B*	GRANT *p* TO *B* WITH GRANT OPTION
3	*A*	REVOKE *p* FROM *B* CASCADE

7.5. Összefoglalás

- *Beágyazott SQL:* Az általános SQL felület helyett használható olyan módszer, mely gyakran hatékonyabb utasítás-végrehajtást tesz lehetővé SQL utasításoknak más nyelven megírt programokba történő beágyazásával.

- *Az SQL és a programozási nyelvek különbözőségéből eredő problémák:* Az SQL adatmodellje lényegesen különbözik a hagyományos programozási nyelvek által támogatott adatmodellektől. Ezért az SQL és a befogadó nyelven írt programok közötti adatcsere olyan közös elérésű változókon keresztül történhet, amelyek a program SQL nyelven elkészített részéből is elérhetőek.

- *Sormutatók:* A sormutató egy olyan SQL változó, amely egy reláció egy sorára hivatkozik. Az SQL és a beágyazó programnyelven megírt program közötti adatcsere úgy történik, hogy egy sormutatóval egyenként végigmegyünk egy reláció sorain, és eközben az egyes sorok tartalmát beolvassuk közös elérésű változókba, és a befogadó nyelven megírt programmal feldolgozzuk azokat.

- *Dinamikus SQL:* Egy befogadó nyelven elkészített programba ahelyett, hogy konkrét SQL utasításokat ágyaznánk, lehetőség van SQL utasításoknak karakterlánc típusú változókban történő előállítására, és lehetőség van az így összeállított SQL utasítások végrehajtására.

- *Párhuzamosság kezelése:* Az SQL2 kétféle mechanizmust is biztosít párhuzamosan futó műveletek kölcsönhatásából származó problémák elkerülésére: a tranzakciókat és a sormutatók egyes jellemzőinek a megszorítását. E megszorítások lehetővé teszik például, hogy sormutatókat változtatásokra érzéketlennek nyilvánítsunk. Az ilyen sormutatók esetében az alaprelációkon végzett módosítások nem befolyásolják azt a relációt, amelynek sorait a sormutatóval feldolgozzuk.

- *Tranzakciók:* Az SQL lehetővé teszi utasítások tranzakcióba szervezését. Egy tranzakció véglegesíthető vagy hatása visszavonható (abortálható). Ez utóbbi esetben a tranzakció által végzett összes adatbázis-módosítási művelet visszavonásra kerül.

- *Elkülönítési szintek:* Az SQL2 a tranzakciók futtatásához négy különböző elkülönítési szintet biztosít, melyek a legszigorúbbtól a leggyengébbig rendre a következők: sorba rendezhető (egy ilyen szinten futó tranzakció úgy viselkedik, mintha vele egy időben nem futna másik tranzakció), ismételhető olvasást biztosító szint (egy ezen a szinten futó tranzakcióban egy lekérdezés során beolvasott összes sort újból megkapjuk a lekérdezés megismételt végrehajtásakor), véglegesített olvasás szintje (ezen a szinten a párhuzamosan futó módosító tranzakcióknak először le kell futniuk, és a véglegesített módosításokat látja a tranzakció). A negyedik, leggyengébb konzisztenciát biztosító szint nem támaszt megszorításokat azzal kapcsolatban, hogy a tranzakció más tranzakciók által elvégzett milyen módosításokat láthat.

- *Csak olvasásra használt sormutatók és csak olvasó tranzakciók:* A sormutató és a tranzakció is lehet csak olvasó. Ezt a tulajdonságát a deklarációjában kell rögzíteni, és azt garantálja az adatbázis-kezelő rendszer felé, hogy az illető sormutató vagy tranzakció nem módosítja az adatbázis tartalmát úgy, hogy az bármiféle szinkronizációs problémát okozna a párhuzamosan működő tranzakcióknál vagy sormutatóknál. Szinkronizációs problémán itt például az érzéketlenség biztosításának, vagy a sorbarendezhetőség biztosításának a problémáját értjük.

- *Egy adatbázis szervezése:* Egy SQL2 adatbázis-kezelő telepítésekor kialakul egy új SQL környezet. Egy környezeten belül az egyes adatbáziselemek csoportosítva lesznek (adatbázis) sémákra, katalógusokra és klaszterekre. Egy katalógus sémák gyűjteménye, míg a klaszter az olyan elemek legnagyobb gyűjtőcsoportja, amelyeket egy felhasználó még elérhet.

- *Kliens/szerver rendszerek:* Egy SQL kliens összekapcsolódik egy SQL szerverrel, kettejük között létrehozva egy kapcsolatot, valamint létrehozva egy munkafázist (műveletek sorozatát). Egy munkafázis alatt általában egy modul lesz végrehajtva. Egy végrehajtás alatt álló modulra SQL ágens kifejezéssel hivatkozhatunk.

- *Jogosultságok:* Biztonsági követelmények teljesítése érdekében egy SQL2 rendszer sokféle jogosultságot biztosít az egyes adatbáziselemeken: olvasási (lekérdezési), beszúrási, törlési vagy módosítási jogosultságokat relációkon, valamint megszorításokban a relációnak más relációkból való hivatkozásának jogosultságát. A beszúrási, módosítási, valamint a más relációból való hivatkozás jogosultsága megadható általánosan (egy reláció összes oszlopára vonatkozóan), vagy csak egy bizonyos oszlopra vonatkozóan is.

- *Engedélyezési diagram:* Az egyes jogosultságokat azok tulajdonosai adhatják tovább a rendszer felhasználóinak. Továbbadható az engedélyezési képesség is, és az így megszerzett jogosultságokat azok megszerzői tetszésük szerint átruházhatják másokra is. A jogosultságok akár vissza is vonhatók. Az engedélyezési diagram hasznos szemléltető eszköz. Segítségével könnyen számon tartható az, hogy egy adott adatelemre vonatkozóan ki milyen jogosultságokkal rendelkezik, és kitől szerezte ezeket a jogosultságokat.

7.6. Irodalomjegyzék

Az olvasónak az SQL2 szabvánnyal kapcsolatos hivatkozásként ajánljuk az 5. fejezetben már ismertetett hivatkozásokat. Az [1] hivatkozásban találhatunk további tudnivalókat a szabvány hiányosságairól a tranzakció-kezelési vonatkozásában.

A tranzakciók implementálására használt legismertebb technikáról, a kétfázisú erőforrás elzárási technikáról [3]-ban olvashatunk. További hasznos információkat tartalmaz [2] és [5] a tranzakció-kezelésről. Az SQL2 jogosultság ellenőrzési rendszere mögött lévő alapötletekről [6]-ban és [4]-ben olvashatunk.

1. Berenson, H., P. A. Bernstein, J. N. Gray, J. Melton, E. O'Neil és P. O'Neil, „A critique of ANSI SQL isolation levels", *Proceedings of ACM SIGMOD Intl. Conf. On Management of Data*, pp. 1–10, 1995.

2. Bernstein, P. A., V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, MA, 1987.

3. Eswaran, K. P., J. N. Gray, R. A. Lorie és I. L. Traiger, „The notions of consistency and predicate locks in a database system", *Communications of the ACM* **19**:11, pp. 624–633, 1976.

4. Fagin, R.: „On an authorization mechanism", *ACM Transactions on Database Systems* **3**:3, pp. 310–319, 1978.

5. Gray, J. N. és A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan-Kaufmann, San Francisco, 1993.

6. Griffiths, P. P. és B. W. Wade, „An authorization mechanism for a relational database system", *ACM Transactions on Database Systems* **1**:3, pp. 242–255, 1976.

8. fejezet

Objektumorientált lekérdezőnyelvek

Ebben a fejezetben két olyan kísérletet tárgyalunk, amelyek az objektumorientált programozásnak az adatbázisvilágba történő bevezetésére irányulnak. Mindkét nyelv – az OQL és az SQL3 – inkább kialakulóban lévő szabvány, semmint széles körben implementált nyelv, de mindkettő egyre inkább elfogadottá válik, és a bennük szereplő elképzelések gyorsan beszivárognak a kereskedelmi forgalomban lévő rendszerekbe.

Az OQL *(Object Query Language – Objektum Lekérdezőnyelv)* egy arra irányuló kísérlet, hogy az objektumorientált lekérdezőnyelvek szabványosításával egy olyan nyelvet alakítson ki, amely az SQL magas szintű, deklaratív programozási modelljét az objektumorientált programozási mintával egyesíti. Ebben a fejezetben először a metódusokat és az objektumkészleteket tekintjük át az ODL-ben, az objektumdefiníciós nyelvben, amelyet a 2. fejezetben vezettünk be mint egy modellezési eszközt. Ez a két jellegzetesség jelentős hatással van az OQL lekérdezőnyelvre. Ezután az OQL-ben történő programozás különféle területeit tárgyaljuk.

Amennyiben az OQL azt célozza meg, hogy az SQL legjavát bevigye az objektumorientált világba, úgy az SQL3 akként jellemezhető, hogy az objektumorientáltság legjavát viszi a relációs világba. A két nyelv bizonyos értelemben „középen találkozik", vannak azonban lényeges megközelítésbeli különbségek, amelyek egyes dolgokat könnyebbé tesznek az egyik nyelvben, mint a másikban. Ezért az SQL3 szabványtervezet objektumorientált vonásainak bevezetése után összehasonlítjuk a két nyelv lehetőségeit.

Lényegében az objektumorientáltság két megközelítési módja a következő kérdésre adott válaszban különbözik: „mennyire fontos a reláció?" Az ODL és OQL köré csoportosult objektumorientált közösség számára a válasz az, hogy „nem nagyon". Ennélfogva ebben a felfogásban mindenféle típusú objektumokat találunk, amelyek között vannak struktúrákból képzett halmazok vagy multihalmazok (azaz relációk) is. Az SQL3 szerinti megközelítésben, amelyet gyakran *objektumrelációsnak* neveznek, a relációs modellt terjesztjük ki oly módon, hogy a relációk sorai és a relációk attribútumaihoz tartozó értéktartományok összetett típusúak is lehetnek. Ezáltal objektumokat és osztályokat vezetünk be a relációs modellbe, de azok mindig valamely reláción belül szerepelnek.

8.1. Lekérdezésekkel kapcsolatos lehetőségek az ODL-ben

Ebben az alfejezetben a 2. fejezetben már bevezett ODL tárgyalását folytatjuk. Először azt tárgyaljuk, hogy az ODL osztályok hogyan vesznek részt a nagyobb programozási környezetben, amelyben szerepelnek. Aztán foglalkozunk az osztályokhoz tartozó „objektumkészletek"-kel, amelyek hasonló szerepet játszanak az OQL-ben, mint a relációk az SQL-ben.

8.1.1. ODL objektumok műveletei

Mint hamarosan látni fogjuk, az OQL egy olyan nyelv, amely lehetővé teszi, hogy relációs vagy halmazorientált műveleteket fejezzünk ki, ahogy azt az SQL is teszi. Gyakran van szükség azonban olyan műveletek elvégzésére, amelyek nem halmazorientáltak. Például ha az objektumok dokumentumok, akkor lehet, hogy azt akarjuk ellenőrizni, hogy egy adott dokumentum tartalmaz-e egy adott kulcsszót. Ha valamely objektum egy térkép vagy egy ábra egy eleme, akkor meg akarhatjuk jeleníteni az objektumot a neki megfelelő pozíción. Még hagyományos, rekordalapú adatok esetén is, mint például a mi filmes példánkban, hasznosak lehetnek bizonyos speciális műveletek, például egy grafikonnak az előállítása, amely éves bontásban mutatja, hogy egy színész hány filmben szerepelt.

Amikor SQL-t használunk, ezeket a műveleteket egy olyan program végzi, amely egy hagyományos programozási nyelvben készült, például C-ben, tehát a befogadó nyelvben, amelybe az SQL utasításokat beágyazzuk. Az SQL változói és a befogadó nyelvi változók közötti értékátadások a 7.1. alfejezetben leírt módon történnek.

Az ODL és a befogadó nyelv közötti kapcsolódás szorosabb, feltételezve, hogy a befogadó nyelv egy objektumorientált nyelv, például C++ vagy Smalltalk. Az ilyen nyelvek ugyanis kellőképpen ODL-szerűek, így az ODL-beli deklarációk közvetlenül lefordíthatók befogadó nyelvi deklarációkká. Továbbá az objektumokat reprezentáló befogadó nyelvi változók könnyen tudják reprezentálni az ODL utasításokban definiált objektumokat.[1]

Azért, hogy a befogadó nyelv kapcsolódása az ODL deklarációkhoz és OQL lekérdezésekhez még megfelelőbb legyen, az attribútumok és kapcsolatok mellett az ODL megenged egy harmadik típusú elemet is, nevezetesen a metódusokat. Egy *metódus* egy osztályhoz rendelt művelet, az osztályba tartozó objektumokra alkalmazható, és lehetnek további argumentumai. Mint látni fogjuk, a metódusokat majdnem úgy használhatjuk az OQL-ben, mintha az osztály attribútumai lennének.

[1] Emlékezzünk, hogy az SQL esetében a befogadó nyelv változóinak típusai, például egész számok, rosszul illeszkednek az SQL alapvető típusaihoz, a sorokhoz és relációkhoz. Tehát az SQL párosítása a befogadó nyelvvel meglehetősen kényelmetlen, mint ahogy ezt a 7.1. alfejezetben láttuk is.

8.1.2. Metódusok deklarálása az ODL-ben

ODL-ben megadhatjuk egy osztályhoz tartozó metódusok neveit, valamint a metódusok bemeneti/kimeneti típusait. Ezek a – *szignatúráknak* nevezett – deklarációk olyanok, mint a függvénydeklarációk a C-ben vagy a C++-ban (ellentétben a függvények *definícióival,* amelyek a függvényt implementáló kódot adják meg). Az egy metódushoz tartozó tényleges kódot a befogadó nyelvben kell megírni; egy metódus implementációja tehát nem része az ODL-nek.

A metódusokat egy interfész-deklarációban, az attribútumokkal és kapcsolatokkal együtt deklaráljuk. Mint ahogy az objektumorientált nyelvekben ez megszokott, minden metódus egy osztályhoz (azaz interfészhez) kapcsolódik, és a metódusok ennek az osztálynak az objektumaira alkalmazhatók. Az objektum így a metódus egy „rejtett" argumentuma. Ez a modell lehetővé teszi, hogy ugyanazt a metódusnevet több, különböző osztályban is használjuk, mert az objektum, amelyiken a műveletet végezzük, meghatározza, hogy pontosan melyik metódusról van szó. Az ilyen metódusnévre azt mondjuk, hogy *túlterhelt* (több osztályban jelenik meg metódusként).

A metódusok deklarálásának szintaxisa hasonló a C-beli függvénydeklarációk szintaxisához az alábbi két fontos kiegészítéssel:

1. A metódusok paraméterei `in`, `out` vagy `inout` paraméterként adhatók meg, ami azt jelenti, hogy bemenő paraméterként, kimenő paraméterként, illetve mindkettőként használjuk őket. A két utolsó típusba tartozó paramétereket a metódus módosíthatja, ellenben az `in` paramétereket nem. Valójában arról van szó, hogy `out` és `inout` paraméterek esetében hivatkozás szerinti paraméterátadás történik, míg `in` paraméterek átadhatók érték szerint. Egy metódusnak visszatérési értéke is lehet, ami szintén egy módja annak, hogy a metódus valamilyen eredményt állítson elő, azon túl, hogy egy `out` vagy `inout` paraméternek ad értéket.

2. A metódusok kiválthatnak *kivételeket,* azaz olyan speciális válaszokat, amelyek kívül esnek a normális paraméterátadási és válaszadási mechanizmuson, amelyen keresztül a metódusok kommunikálnak. Egy kivétel rendszerint egy abnormális vagy váratlan állapotot jelez, amelyet az a metódus „kezel", amely hívta (esetleg közvetve, hívások sorozatán keresztül). A nullával való osztás például egy olyan helyzet,

Miért szignatúrák?

A szignatúrák megadásának értéke abban áll, hogy amikor a sémát egy valódi programozási nyelvben implementáljuk, automatikusan meggyőződhetünk arról, hogy az implementáció megfelel-e annak az elgondolásnak, amit a séma kifejez. Azt persze nem tudjuk ellenőrizni, hogy az implementáció helyesen implementálja-e a műveletek „jelentését", de legalább arról meggyőződhetünk, hogy a bemenő és kimenő paraméterek száma és típusa megfelelő.

amely kivételként kezelhető. ODL-ben egy metódus deklarációja után megadható a `raises` kulcsszó, majd ezt követően azoknak a kivételeknek a zárójelezett listája, amelyeket a metódus kiválthat.

8.1. példa: A 8.1. ábrán látható a bővített változata a `Film` osztályhoz tartozó interfész definíciójának, amelyet legutóbb a 2.6. ábrán láttunk. Két olyan változás van, amelyik nem a metódusokhoz kapcsolódik.

1. A 2) sor egy „objektumkészlet-deklarációt" tartalmaz. Ezt a kérdést a 8.1.3. részben fogjuk tárgyalni.
2. A 3) sor annak deklarálása, hogy a `cím` és az `év` együtt kulcsot alkotnak a `Film` osztályban.

Az interfész-deklarációhoz a következő metódusokat vettük hozzá. A (10) sor a `hosszÓrákban` metódust deklarálja. Hozzáképzelhetjük, hogy annak a filmnek (mint objektumnak) a hosszát adja vissza, amelyikre alkalmazzuk, de úgy, hogy a percekben adott hosszt (melyet a `hossz` attribútum reprezentál) átalakítja egy lebegőpontos számmá, amely a film hosszának az órákban kifejezett értékét jelenti. Vegyük észre, hogy a függvénynek nincsenek paraméterei. A `Film` objektum, amelyre a metódust alkalmazzuk, a „rejtett" argumentum, és a `hosszÓrákban` egy lehetséges implementációja a film percekben megadott hosszát ettől az objektumtól kapná meg.

Azt is láthatjuk, hogy a függvény kiválthat egy `nincsHossz` nevű kivételt. Feltételezhetően akkor váltaná ki ezt a kivételt, ha az objektumban, amelyre a

```
 1)  interface Film
 2)     (extent Filmek
 3)     key (cím, év))
     {
 4)     attribute string cím;
 5)     attribute integer év;
 6)     attribute integer hossz;
 7)     attribute enumeration(színes,feketeFehér)szalagFajta;
 8)     relationship Set<Színész> szereplők
                    inverse Színész::szerepelBenne;
 9)     relationship Stúdió gyártó
                    inverse Stúdió::gyárt;
10)     float hosszÓrákban() raises(nincsHossz);
11)     színészNevek(out Set<String>);
12)     egyébFilmek(in Színész, out Set<Film>)
                    raises(nemLétezőSzínész);
     };
```

8.1. ábra. *A* `Film` *osztály bővítése metódusdeklarációkkal*

`hosszÓrákban` metódust alkalmazzuk, a `hossz` attribútumnak definiálatlan értéke lenne, vagy egy olyan értéke, amely nem lehet érvényes hossz (például egy negatív szám).

Emlékezzünk vissza, hogy a deklarációban semmi olyan nem szerepel, ami egy metódustól *megkövetelné*, hogy azt tegye, amit a neve sugall. Például a `hosszÓrákban` metódus implementációja lehetne egy olyan függvény, amely mindig 3,14159-et ad eredményül, függetlenül a `Film` objektumtól, amelyre éppen alkalmazzuk. Vagy akár úgy is implementálhatnánk, hogy a hossz négyzetével tér vissza, átalakítva azt lebegőpontos számmá. Valójában bármilyen függvény elfogadható, amelynek nincs argumentuma (kivéve az objektumot, amelyre alkalmazzuk), lebegőpontos értéket ad eredményül, és amely legfeljebb a `nincsHossz` kivételt váltja ki.

A 11) sorban láthatunk egy másik szignatúrát, a `színészNevek` nevű metódus szignatúráját. Ennek a metódusnak nincs visszatérési értéke, van viszont egy kimenő paramétere, amelynek típusa karakterláncok halmaza. Feltételezhetjük, hogy a kimenő paraméter értékét a metódus úgy számítja ki, hogy az azoknak a karakterláncoknak a halmaza, amelyek azon színészek `név` attribútumának értékei, akik abban a filmben szerepelnek, amelyikre a metódust alkalmazzuk. Arra persze nincs garancia, mint ahogy általában sincs, hogy az implementált metódus valóban pont így viselkedik.

Végül a 12) sorban van egy harmadik metódus, az `egyébFilmek`. Ennek a metódusnak van egy `Színész` típusú bemenő paramétere. A metódus egy lehetséges implementációja a következő. Feltételezhetjük, hogy az `egyébFilmek` metódus elvárja, hogy a színész a filmben szereplő valamelyik színész legyen; ha ez nem teljesül, akkor kiváltja a `nemLétezőSzínész` kivételt. Ha a színész egyike a film szereplőinek, amelyre a metódust alkalmazzuk, akkor a kimenő paraméter értéke, amelynek típusa filmek halmaza, azon további filmek halmaza lesz, amelyekben szintén szerepel a színész. □

8.1.3. Osztályhoz tartozó objektumkészlet

Minden ODL osztályhoz (interfészhez) deklarálható egy *objektumkészlet* (extent), ami az osztály aktuális objektumainak halmazát jelölő név. A deklaráció az `extent` kulcsszó, majd azt követően az objektumkészlethez választott név megadásából áll. Ennek a deklarációnak közvetlenül az interfész (osztály) nevének megadása után kell szerepelnie.

Bizonyos értelemben egy osztály objektumkészlete megfelel egy reláció nevének, míg maga az osztálydefiníció a reláció attribútumainak, illetve azok típusainak deklarálásaként tekinthetők. Amint látni fogjuk, az OQL lekérdezései egy osztály objektumkészletére hivatkoznak, nem pedig magára az osztály nevére.

8.2. példa: A 8.1. ábra 2) sora tartalmazza a `Film` osztályhoz tartozó objektumkészlet megadását. Az objektumkészlet neve: `Filmek`. A `Filmek` mindenkori értéke az a halmaz, amely az adott pillanatban az adatbázisban létező összes `Film` objektumot magában foglalja. □

8.1.4. Feladatok

8.1.1. feladat: A 8.2. ábra mutatja a termékekkel foglalkozó példánk leírását ODL-ben. A termékeknek mind a három típusát a Termék osztály alosztályaként definiáltuk. Az olvasó észreveheti, hogy egy termék típusát egyrészt megkaphatjuk a típus attribútum alapján, másrészt az alosztály alapján, amelyikbe tartozik. Ez nem éppen a legszerencsésebb tervezés, hiszen meghagyja annak lehetőségét, hogy például egy PC objektum típus attribútuma "laptop" vagy "nyomtató" legyen. Ugyanakkor viszont érdekes választási lehetőségeket biztosít a felhasználó számára abban a tekintetben, hogy a lekérdezéseit hogyan fejezi ki.

Mivel a típus attribútumot a Nyomtató osztály örökli a Termék osztálytól, a

```
interface Termék
   (extent Termékek
    key modell)
{
   attribute integer modell;
   attribute string gyártó;
   attribute string típus;
   attribute real ár;
};

interface PC : Termék
   (extent PCk)
{
   attribute integer sebesség;
   attribute integer memória;
   attribute integer merevlemez;
   attribute string cd;
};

interface Laptop : Termék
   (extent Laptopok)
{
   attribute integer sebesség;
   attribute integer memória;
   attribute integer merevlemez;
   attribute real képernyő;
};

interface Nyomtató : Termék
   (extent Nyomtatók)
{
   attribute boolean színes;
   attribute string nyomtatóTípus;
};
```

8.2. ábra. *A termékek adatbázissémája ODL-ben*

Nyomtató eredeti típus attribútumát át kellett neveznünk, így abból nyomtatóTípus lett. Ez utóbbi attribútum azt adja meg, hogy a nyomtató milyen eljárás szerint működik (például lézer vagy tintasugaras), míg a Termék osztályban definiált típus attribútumnak olyan értekei lesznek, mint például PC, laptop vagy nyomtató.

```
interface Hajóosztály
   (extent Hajóosztályok
    key név)
{
   attribute string név;
   attribute string ország;
   attribute integer ágyúkSzáma;
   attribute integer kaliber;
   attribute integer vízkiszorítás;
   relationship Set<Hajó> hajók inverse Hajó::hajóosztálya;
};

interface Hajó
   (extent Hajók
    key név)
{
   attribute string név;
   attribute integer felavatva;
   relationship Hajóosztály hajóosztálya
              inverse Hajóosztály::hajók;
   relationship Set<Kimenetel> csatái
              inverse Kimenetel::melyikHajó;
};

interface Csata
   (extent Csaták
    key név)
{
   attribute string név;
   attribute Date dátum;
   relationship Set<Kimenetel> eredmények
              inverse Kimenetel::melyikCsata;
};

interface Kimenetel
   (extent Kimenetelek
    key név)
{
   attribute enum Státus {ép,elsüllyedt,sérült} állapot;
   relationship Hajó melyikHajó inverse Hajó::csatái;
   relationship Csata melyikCsata inverse Csata::eredmények;
};
```

8.3. ábra. *A csatahajók adatbázissémája ODL-ben*

```
interface Színész
   (extent Színészek
    key név)
{
   attribute string név;
   attribute Struct Cím
        {string utca, string város} lakcím;
   relationship Set<Film> szerepelBenne
            inverse Film::szereplők;
};

interface Stúdió
   (extent Stúdiók
    key név)
{
   attribute string név;
   attribute string cím;
   relationship Set<Film> gyárt
            inverse Film::gyártó;
};
```

8.4. ábra. *Az objektumorientált filmes adatbázis sémájának egy része*

Egészítsük ki a 8.2. ábrán szereplő ODL kódot az alábbi műveletek elvégzésére alkalmas metódusok szignatúráival:

* a) Termék árának csökkentése x-szel, ahol x mint a művelet bemenő paramétere adott.

* b) Termék sebességének visszaadása, ha a termék PC vagy laptop, ellenkező esetben a `nemSzámítógép` kivétel kiváltása.

c) Laptop képernyőméretének beállítása valamely x bemenő értékre.

! d) Annak meghatározása, hogy a q termék, amelyre a metódust alkalmazzuk, nagyobb sebességű és alacsonyabb árú-e, mint egy adott bemenő p termék. A `rosszBemenet` kivétel kiváltása, ha p egy sebességgel nem rendelkező termék (azaz se nem PC, se nem laptop), illetve a `nincsSebesség` kivétel kiváltása, ha q egy sebességgel nem rendelkező termék.

8.1.2. feladat: A 8.3. ábrán látható a „csatahajók" adatbázisunk leírása ODL-ben. Egészítsük ki a következő metódusok szignatúráival:

a) Hajó tűzerejének, azaz az ágyúk száma és a kaliber köbe szorzatának kiszámítása.

b) Hajó testvérhajóinak megadása. A `nincsTestvére` kivétel kiváltása, ha a hajó az egyetlen az ő hajóosztályában.

c) Paraméterként adott b csata esetén, valamint egy s hajóra alkalmazva a metódust, azoknak a hajóknak a megadása, amelyek a b csatában elsüllyedtek, feltéve, hogy s részt vett abban a csatában. Ha s nem harcolt a b csatában, akkor a `nemVettRészt` kivétel kiváltása.

d) Paraméterként megadott névvel és felavatási évvel rendelkező hajó hozzáadása ahhoz a hajóosztályhoz, amelyikre a metódust alkalmazzuk.

8.2. Bevezetés az OQL-be

Ebben az alfejezetben bevezetjük az OQL-t, az Objektum Lekérdezőnyelvet (Object Query Language). Leírásunk ebben és a következő két alfejezetben nem lesz annyira alapos, mint amilyen az SQL-ről szóló leírásunk volt. A legfontosabb utasításokat és sajátságokat elmagyarázzuk, de sok egyéb lehetősége is van az OQL-nek. Ezek a lehetőségek gyakran hasonlítanak vagy az SQL-hez, vagy a tipikus, hagyományos objektumorientált programozási nyelvek megfelelő lehetőségeihez.

Az OQL nem teszi lehetővé tetszőleges függvények kifejezését úgy, ahogy azt a hagyományos programozási nyelvek, például C, teszi. Az OQL inkább egy olyan SQL-szerű írásmódot biztosít az egyes lekérdezések megfogalmazására, amely az absztrakció egy magasabb szintjén van, mint egy hagyományos nyelv tipikus utasításai. Az elképzelés az, hogy az OQL-t úgy használjuk, mint valamely objektumorientált *befogadó* nyelvnek egy kiterjesztését, mint például a C++-nak, Smalltalknak vagy a Javának. Az objektumokat mind az OQL lekérdezések, mind a befogadó nyelv szokásos utasításai kezelik. A befogadó nyelvi utasítások és az OQL lekérdezések keveredhetnek anélkül, hogy a két nyelv között az értékek explicit átvitelére lenne szükség. Ez a lehetőség előrelépést jelent az SQL-nek valamely befogadó nyelvbe történő beágyazásához képest, amit a 7.1. alfejezetben tárgyaltunk.

8.2.1. Egy objektumorientált példa

Az OQL stílusának bemutatásához szükségünk van egy példára. Ez a már ismerős `Film`, `Színész` és `Stúdió` osztályokat fogja tartalmazni. A `Film` osztálynak a 8.1. ábrán szereplő definícióját fogjuk használni. Másrészről, a `Színész` és a `Stúdió` definícióját a 2.6. ábrából vettük át, kiegészítve azokat kulcs és objektumkészlet deklarálásával (metódusokat nem deklaráltunk); ezt a 8.4. ábra mutatja.

8.2.2. Típusok az OQL-ben

A típusokkal hasonlóan építkezhetünk az OQL-ben, mint az ODL deklarációiban (lásd 2.1.7. alfejezet). Az OQL-ben azonban nincs korlátozás arra vonatkozóan, hogy a típuskonstruktorokat milyen beágyazási mélységig alkalmazhatjuk.

Amikor egy programozási nyelv típusairól (típusrendszeréről) beszélünk, különbséget kell tenni aközött, hogy egy változó (melyet néha *változékony – mutable – objektumnak* neveznek) típusát deklaráljuk, vagy valamilyen konstans értéket (*változhatatlan – immutable – objektum*) adunk meg. Az OQL utasításokban használt vátozókat az azokat körülvevő befogadó nyelvben deklaráljuk, amihez feltehetően az ODL-nek a 2.1.7. részben ismertetett írásmódját használjuk, vagy valami hasonlót. Mivel az ODL egy adatdefiníciós nyelv, ott nincs szükség konstansok használatára. OQL programok esetében viszont igen, így meg kell néznünk, hogyan konstruálhatunk tetszőleges típusú konstansokat az OQL-ben. Konstansokat az alábbiakból építkezve hozhatunk létre:

1. *Alaptípusok*, amelyek a következők:

   a) *Atomi típusok*: egész számok, valós számok, karakterek, karakterláncok és logikai értékek. Ezek ábrázolása úgy történik, mint az SQL-ben, azzal a kivétellel, hogy a karakterláncokat idézőjelek zárják közre.

   b) *Felsorolások*. Egy felsorolásban szereplő értékeket valójában az ODL-ben adjuk meg. Ezen értékek bármelyike használható konstansként.

2. *Összetett típusok*, amelyek felépítéséhez a következő típuskonstruktorokat használhatjuk:

   a) `Set(...)`

   b) `Bag(...)`

   c) `List(...)`

   d) `Array(...)`

   e) `Struct(...)`

   Az első négyet ezek közül *kollekciótípusoknak* nevezzük. A kollekciótípusok és a `Struct` tetszés szerint alkalmazhatók bármilyen típusú értékekre, legyenek azok alaptípusok vagy összetett típusok. A `Struct` operátor alkalmazásakor azonban a mezőneveket és a megfelelő értékeket is meg kell adni. Mindegyik mezőnév után egy kettőspont majd a megfelelő érték megadása következik, a mező-érték párokat pedig vesszővel választjuk el egymástól.

8.3. példa: A `bag(2,1,2)` kifejezés azt a multihalmazt jelenti, amelyikben a 2 egész szám kétszer, az 1 egész szám pedig egyszer szerepel. A

```
struct(mező1: bag(2,1,2), mező2: "baz")
```

kifejezés egy struktúrát jelöl, amely két mezőből áll. Az egyik a `mező1`, amelynek értéke a fenti multihalmaz, a másik a `mező2`, amelynek értéke a `"baz"` karakterlánc. □

8.2.3. Útkifejezések

Az összetett típusú változók komponenseinek elérésekor a pontot használjuk mint jelölési módot. Ez hasonló ahhoz, ahogy a pontot a C-ben használjuk, és a pontnak az SQL-beli használatához is kapcsolódik. Az általános szabály a következő. Ha *a* egy *C* osztályba tartozó objektumot jelöl, és *p* az osztály valamely összetevője – az osztály egy attribútuma, kapcsolata vagy metódusa –, akkor *a.p* annak az eredményét jelöli, hogy *p*-t „alkalmazzuk" *a*-ra. Bővebben kifejtve:

1. Ha *p* egy attribútum, akkor *a.p* ennek az attribútumnak az értéke az *a* objektumban.

2. Ha *p* egy kapcsolat, akkor *a.p* az *a* objektummal a *p* kapcsolaton keresztül kapcsolatban álló objektum vagy objektumok kollekciója.

3. Ha *p* egy metódus (esetleg paraméterekkel), akkor *a.p* a *p* *a*-ra való alkalmazásának az eredménye.

8.4. példa: Tegyük fel, hogy `kedvencFilm` egy befogadó nyelvi változó, amelynek értéke egy `Film` objektum. Ekkor:

- A `kedvencFilm.hossz` értéke az adott film hossza, azaz a `hossz` attribútum értéke abban a `Film` objektumban, amelyet a `kedvencFilm` változó jelent.

- A `kedvencFilm.hosszÓrákban()` értéke egy valós szám, a film hossza órákban kifejezve, amit úgy kapunk, hogy a `hosszÓrákban` metódust alkalmazzuk a `kedvencFilm` objektumra.

- A `kedvencFilm.szereplők` azoknak a `Színész` objektumoknak a halmaza, amelyek a `szereplők` kapcsolaton keresztül a `kedvencFilm` filmmel kapcsolatban állnak.

- A `kedvencFilm.színészNevek(kedvencSzínészek)` kifejezésnek nincs visszatérő értéke (vagyis C++-ban a kifejezés típusa `void`). Van azonban egy ún. mel-

---

Pontok helyett nyilak

Az OQL a `->` nyilat a pont szinonimájaként használja. Ez a szabály valamelyest megfelel a C filozófiájának, ahol a ponttal és nyíllal egyaránt egy struktúra komponenseihez lehet hozzáférni. C-ben azonban a nyíl és a pont operátorok némileg eltérő jelentéssel bírnak, jóllehet OQL-ben ugyanazt jelentik. C-ben az `a.f` kifejezés azt várja el, hogy az a egy struktúra legyen, míg `p->f` azt várja el `p`-től, hogy egy mutató legyen, amely egy struktúrára mutat. Mindkét kifejezés a struktúra `f` mezőjének értékét állítja elő.

---

lékhatása, mégpedig az, hogy a `színészNevek` metódus `kedvencSzínészek` kimenő változójának értékül adja azt a karakterláncokból álló halmazt, amely a filmben szereplő színészek neveit tartalmazza.
□

Kifejezéseket a pont többszöri használatával is képezhetünk, amennyiben ennek van értelme. Ha például a `kedvencFilm` változó értéke egy film (objektum), akkor a `kedvencFilm.gyártó` azt a `Stúdió` objektumot jelenti, amelyik a filmet készítette, a `kedvencFilm.gyártó.név` pedig egyenlő azzal a karakterlánccal, amely ennek a stúdiónak a neve.

8.2.4. Select-from-where kifejezések az OQL-ben

Az OQL-ben lehetőség van olyan kifejezések megfogalmazására, ahol az SQL-ből ismerős select-from-where szintaxist használjuk. Íme egy példa, amelyben az *Elfújta a szél* című film gyártási évét kérdezzük:

```
SELECT m.év
FROM Filmek m
WHERE m.cím="Elfújta a szél"
```

Észrevehetjük, hogy eltekintve az idézőjelek használatától a konstans karakterlánc megadásában, ez a lekérdezés akár SQL-beli lekérdezés is lehetne. Az egyetlen nem nyilvánvaló részlet az, hogy SQL-t használva arra számítanánk, hogy a `FROM` záradékot az alábbi módon adjuk meg:

```
FROM Filmek AS m
```

Mindamellett az `AS` kulcsszó opcionálisan megadható az OQL-ben is, mint ahogyan az SQL-ben is opcionális. Értelmesebbnek tűnik azonban annak elhagyása az OQL-ben, mivel a „`Filmek m`" rész jelentése az, hogy `m` egy változó, amely a `Filmek` objektumkészletben szereplő objektumokon végigfut, ahol az objektumkészlet a `Film` osztályba tartozó aktuális objektumok halmaza.

Az OQL-ben a select-from-where kifejezés általános alakja a következő részekből áll:

1. A `SELECT` kulcsszó és azt követően kifejezések egy listája.
2. A `FROM` kulcsszó, amit egy vagy több változó deklarációja követ. Egy változó deklarálása annyit jelent, hogy megadjuk az alábbiakat:
   a) egy kifejezést, amelynek értéke valamilyen kollekciótípusú, például halmaz vagy multihalmaz;

   b) opcionálisan az `AS` kulcsszót; és
   c) a változó nevét.

   Sok esetben az a)-ban említett kifejezés nem más, mint egy osztálynak az objektumkészlete, mint például a `Film` osztályhoz tartozó `Filmek` objektumkészlet a fenti példában. Az objektumkészletek hasonlóak az SQL-beli `FROM` záradékokban szereplő relációkhoz. Egy változó deklarálásakor azonban tetszőleges olyan kifejezést használhatunk, amely kollekciót ad eredményül, tehát például egy másik select-from-where kifejezést is. Ennek a lehetőségnek a közvetlen analógiája nincs meg az SQL2-ben, jóllehet néhány kereskedelmi SQL megengedi az alkérdések használatát a `FROM` záradékokban.

3. A `WHERE` kulcsszó és egy logikai értékű kifejezés. A `SELECT` után szereplő kifejezésekhez hasonlóan a kifejezésben kizárólag olyan operandusok szerepelhetnek, amelyek csak konstansokat és a `FROM` záradékban deklarált változókat tartalmaznak. Az összehasonlító operátorok ugyanazok, mint az SQL-ben, azzal a kivétellel, hogy a „nem egyenlő" kifejezésére a <> helyett a `!=` használatos. A logikai operátorok az ÉS (`AND`), a VAGY (`OR`) és a NEM (`NOT`), csakúgy mint SQL-ben.

A lekérdezés egy objektumokból álló multihalmazt állít elő. Ezt a multihalmazt úgy számítjuk ki, hogy beágyazott ciklusok segítségével a `FROM` záradékban előforduló változók összes lehetséges értékeit tekintetbe vesszük. Ha ezeknek a változóknak valamely értékkombinációja kielégíti a `WHERE` záradékban megadott feltételt, akkor a `SELECT` záradéknak megfelelő objektum bekerül abba a multihalmazba, amely a select-from-where utasítás eredménye lesz.

8.5. példa: Az alábbi OQL lekérdezés illusztrálja a select-from-where kifejezések szerkezetét.

```
SELECT s.név
FROM Filmek m, m.szereplők s
WHERE m.cím = "Casablanca"
```

Ebben a lekérdezésben a *Casablanca* című film szereplőinek neveit kérdezzük. Vegyük észre, hogy a `FROM` záradékban több kifejezés szerepel. Először azt adjuk meg, hogy `m` a `Film` osztály egy tetszőleges objektuma. Ezt azáltal definiáljuk, hogy azt mondjuk, hogy `m` ennek az osztálynak az objektumkészletéből (`Filmek`) való. Majd azt írjuk elő, hogy `m` minden egyes értéke esetén az `s` egy olyan `Színész` objektum lehet, amely az `m.szereplők` halmazban előfordul, tehát abban a halmazban, amely az `m` filmben szereplő színészekből áll. Vagyis két ciklus segítségével tekintjük az összes olyan `(m, s)` párt, ahol `m` egy film és `s` egy szereplője annak a filmnek. A kiértékelés lényege tehát:

```
minden m-re a Filmek-ben
  minden s-re az m.szereplők-ben
    ha m.cím = "Casablanca", akkor
      az s.név kerüljön be az eredményül
      visszaadandó multihalmazba
```

A WHERE záradék leszűkíti a figyelembe vett párok halmazát azokra, ahol az m megegyezik azzal a Film objektummal, amelynek címe *Casablanca*. Majd a SELECT záradék előállítja azt a multihalmazt (melynek esetünkben inkább halmaznak kellene lennie), amely a WHERE záradékot kielégítő (m,s) párokban szereplő s színész objektumok név attribútumainak értékeit tartalmazza. Ezek a nevek a m_c.szereplők halmazban előforduló színészek nevei, ahol m_c a *Casablanca* film objektum. □

8.2.5. Ismétlődések megszüntetése

A 8.5. példában szereplő és a hozzá hasonló lekérdezések válaszként multihalmazokat állítanak elő és nem halmazokat. Az OQL tehát követi az SQL megközelítését, amely szerint az ismétlődéseket nem szünteti meg alapértelmezés szerint, hanem csak abban az esetben, ha ezt előírjuk. Az SQL-hez hasonlóan az ismétlődések kiküszöbölése a SELECT után a DISTINCT kulcsszó megadásával érhető el.

8.6. példa: Tegyük fel, hogy a Disney-filmek szereplőinek neveit akarjuk megtudni. A következő lekérdezés elvégzi ezt a feladatot, ráadásul úgy, hogy csak egyszer tartja meg azokat a neveket, amelyek olyan színészek nevei, akik több Disney-filmben is szerepelnek.

```
SELECT DISTINCT s.név
FROM Filmek m, m.szereplők s
WHERE m.gyártó.név = "Disney"
```

Ennek a lekérdezésnek a kiértékelése hasonló az 8.5. példáéhoz. Két ciklus segítségével ismét tekintjük az összes párt, ahol egy film a benne szereplő színészekkel alkot párokat, mint ahogy ezt a 8.5. példában is tettük. Most azonban az ilyen (m,s) párokra vonatkozó feltétel az, hogy „Disney" a neve annak a stúdiónak, amelyet az m.gyártó Stúdió objektumként kapunk. □

8.2.6. Összetett típusú eredmények

A SELECT záradékban szereplő kifejezés vagy kifejezések nemcsak egyszerű változók lehetnek, hanem bármilyen kifejezések, beleértve a típuskonstruktorok használatával képzett kifejezéseket is. Alkalmazhatjuk például a Struct típuskonstruktort

valahány kifejezésre, amivel olyan select-from-where lekérdezést kapunk, amely struktúrák egy halmazát vagy multihalmazát eredményezi.

8.7. példa: Tételezzük fel, hogy azokat a színészekből alkotott párokat keressük, akik ugyanazon a címen laknak. Az ilyen párok halmazát megkaphatjuk az alábbi lekérdezés végrehajtásával:

```
SELECT DISTINCT Struct(színész1: s1, színész2: s2)
FROM Színészek s1, Színészek s2
WHERE s1.cím = s2.cím AND s1.név < s2.név
```

Vesszük tehát a színészekből alkotott összes (s1,s2) párt. A WHERE záradék ellenőrzi, hogy a két színésznek ugyanaz-e a címe. Továbbá azt is vizsgálja, hogy az ábécé szerint az első színész neve megelőzi-e a második színész nevét. Ezzel biztosítja, hogy az olyan párok nem kerülnek be az eredménybe, ahol a pár mindkét tagja ugyanaz a színész, valamint hogy az ugyanabból a két színészből álló párokat nem állítjuk elő kétszer a két különböző sorrendnek megfelelően.

Minden olyan pár esetén, amelyek a fenti két feltételt kielégítik, egy struktúra jön létre. Ennek a struktúrának a típusa egy két mezőből álló rekordtípus, a mezők nevei színész1 és színész2. Mindkét mező típusa a Színész osztály, mivel ez az osztály a típusa mind az s1, mind az s2 változónak, amelyek a két mezőhöz tartozó értékeket adják. Ha N valamilyen név, akkor formálisan a struktúra típusa:

```
Struct N {színész1: Színész, színész2: Színész}
```

A lekérdezés eredményének típusa az N típusú struktúrákból alkotott halmaztípus, vagyis:

```
Set<Struct N {színész1: Színész, színész2: Színész}>
```

Vegyük észre, hogy ez a típus egy olyan típusra példa, amely OQL programok esetében megjelenhet, de nem lehet a típusa valamely ODL-ben deklarált attribútumnak vagy kapcsolatnak. □

Megjegyezzük még, hogy a 8.7. példáéval azonos eredményt kapunk, ha a struktúra típusának definícióját nem adjuk meg teljesen, hanem a SELECT kulcsszó után csak a komponenseket és a hozzájuk rendelt mezőneveket soroljuk fel. Ennek megfelelően a 8.7. példában szereplő lekérdezés első sorát az alábbi módon is írhattuk volna:

```
SELECT DISTINCT színész1: s1, színész2: s2
```

8.2.7. Alkérdések

Select-from-where kifejezést minden olyan helyen használhatunk, ahol egy kollekció (például egy halmaz) helyénvaló. Egy ilyen hely az alkérdések számára például a FROM záradék, ahol egy változó tartományát jelentő kollekciót egy select-from-where kifejezés segítségével is megkonstruálhatunk. Mellékesen megjegyezzük, hogy az ennek megfelelő lehetőség – táblát definiáló kifejezés használata táblanév helyén – már az SQL3 szabványtervezetnek és néhány kereskedelmi SQL rendszernek is része.

8.8. példa: Nézzük ismét a 8.6. példában feltett kérdést, ahol a Disney-filmekben szereplő színészek neveit kérdeztük. A lekérdezésünket most másként fogalmazzuk meg. A Disney-filmek halmazát megadja a következő lekérdezés:

```
SELECT m
FROM Filmek m
WHERE m.gyártó.név = "Disney"
```

Ezt a lekérdezést használhatjuk mint alkérdést annak a halmaznak a definiálására, amelynek elemein a Disney-filmeket reprezentáló d változó végigfut.

```
SELECT DISTINCT s.név
FROM (SELECT m
    FROM Filmek m
    WHERE m.gyártó.név = "Disney") d,
   d.szereplők s
```

A „Kik a Disney-filmek szereplői?" kérdésnek ez a megfogalmazása nem tömörebb, mint a 8.6. példában adotté, sőt talán kevésbé tömör, de bemutatja egy új formáját annak, ahogyan OQL-ben a lekérdezéseinket felépíthetjük. A fenti lekérdezésben a FROM záradék tartalma két ciklusnak felel meg. Az elsőben a d változó végigfut az összes Disney-filmen, vagyis a FROM záradékban szereplő alkérdés eredményeként kapott halmaz elemein. Az első ciklusba beágyazott második ciklusban pedig az s változó a d film szereplőin fut végig. Észrevehetjük, hogy WHERE záradékra nincs is szükség. □

8.2.8. Az eredmény rendezése

Az OQL-ben egy select-from-where kifejezés eredménye vagy egy multihalmaz, vagy egy halmaz (ha a DISTINCT kulcsszót megadjuk). Az eredményt egy rendezett listává alakíthatjuk, és egyben azt is megválaszthatjuk, hogy a lista elemei milyen szempont szerint legyenek rendezve, ha a select-from-where végén egy ORDER BY záradékot használunk. Az ORDER BY záradék olyan az OQL-ben, mint az SQL-ben. Az ORDER BY kulcsszavak után kifejezések egy listáját kell megadni. A lekérdezés eredményében lévő minden objektumra kiértékelődik az első kifejezés, majd az így kapott értékek alapján kialakul az objektumok egy sorrendisége. „Döntetlenek" esetén a sorrend a második kifejezés kiértékelése során kapott értékek alapján alakul, és így tovább.

8.9. példa: Adjuk meg a Disney-filmek halmazát, de úgy, hogy az eredmény a filmeknek egy hossz szerint rendezett listája legyen. Az azonos hosszal rendelkező filmek az ábécé szerint legyenek rendezve. A lekérdezés az alábbi:

```
SELECT m
FROM Filmek m
WHERE m.gyártó.név = "Disney"
ORDER BY m.hossz, m.cím
```

Az első három sor megegyezik a 8.8. példában található alkérdéssel. A negyedik sor azt írja elő, hogy a select-from-where kifejezés eredményeként kapott *m* objektumokat először az m.hossz értéke (vagyis a film hossza) szerint kell rendezni, majd az m.cím (vagyis a film címe) szerint az egyforma hosszú filmek esetén, ha vannak ilyenek. Következésképpen ennek a lekérdezésnek a végső eredménye Film objektumok listája lesz. Az alapértelmezés növekvő sorrendiséget jelent, de megválaszthatjuk azt, hogy a sorrend növekvő vagy csökkenő legyen. Csakúgy mint az SQL-ben, ezt itt is az ASC vagy DESC kulcsszó megadásával lehet jelezni az ORDER BY záradékban. □

8.2.9. Feladatok

8.2.1. feladat: Tekintsük a 8.1.1. feladatban és a 8.2. ábrán szereplő ODL-beli sémát, és fogalmazzuk meg a következő lekérdezéseket OQL-ben:

* a) Keressük meg a 2000 $-nál olcsóbb PC-k modellszámait.

b) Keressük meg azoknak a PC-knek a modellszámait, amelyek legalább 32 megabájt (MB) memóriával rendelkeznek.

*! c) Keressük meg azokat a gyártókat, akik legalább két különböző modellű lézernyomtatót gyártanak.

d) Keressük meg az olyan (r, h) párok halmazát, amelyekre teljesül, hogy van olyan PC vagy laptop, amelynek r megabájt (MB) memóriája és h gigabájt (GB) kapacitású merevlemeze van.

e) Készítsünk egy listát, amely a PC-ket (az objektumokat, tehát nem a modellszámokat) tartalmazza processzorsebesség szerint növekvő sorendben.

! f) Készítsünk egy listát, amely a legalább 16 megabájt (MB) memóriával rendelkező laptopok modellszámait tartalmazza képernyőméret szerint csökkenő sorrendben.

! **8.2.2. feladat:** Fogalmazzuk meg újra a 8.2.1. feladat lekérdezéseit úgy, hogy minden lekérdezésben használunk legalább egy alkérdést.

8.2.3. feladat: Tekintsük a 8.1.2. feladatban és a 8.3. ábrán szereplő ODL-beli sémát, és fogalmazzuk meg a következő lekérdezéseket OQL-ben:

a) Keressük meg a legalább kilenc ágyúval rendelkező hajók hajóosztályainak neveit.

b) Keressük meg a legalább kilenc ágyúval rendelkező hajókat (objektumokat, tehát nem hajóneveket).

c) Keressük meg azoknak a hajóknak a neveit, amelyek vízkiszorítása kevesebb, mint 30 000 tonna. Az eredmény egy lista legyen, ahol a hajók a felavatás évei szerint növekvő sorrendben szerepeljenek, egyezés esetén pedig a hajónevek ábécé szerinti sorrendje határozza meg a lista rendezettségét.

d) Keressük meg azokat az objektumpárokat, ahol a pár tagjai testvérhajók (vagyis ugyanannak a hajóosztálynak a hajói). Objektumokat keresünk tehát, és nem hajóneveket.

! e) Keressük meg azoknak a csatáknak a neveit, amelyekben elsüllyedtek legalább két különböző országhoz tartozó hajók.

!! f) Keressük meg azoknak a csatáknak a neveit, amelyekben egyetlen hajó sem sérült meg.

8.3. További lehetőségek OQL kifejezések képzésére

Ebben az alfejezetben az OQL-ben biztosított néhány további operátort mutatunk be, melyeket kifejezések képzésére használhatunk. Ezekben a műveletekben szerepelnek a logikai kvantorok – a *minden* és a *létezik* –, az összesítő függvények, a csoportosítást végző group-by operátor, valamint az egyesítés, metszet és különbség halmazműveletek.

8.3.1. Kvantort használó kifejezések

Megvizsgálhatjuk, hogy egy halmaznak az összes eleme, illetve legalább egy eleme eleget tesz-e valamilyen feltételnek. Annak ellenőrzésére, hogy egy S halmaznak minden eleme kielégíti a $C(x)$ feltételt, ahol x egy változó, a következő OQL kifejezést használjuk:

FOR ALL x IN S : $C(x)$

A kifejezés értéke IGAZ (TRUE), ha minden S-beli x kielégíti a $C(x)$ feltételt, egyébként HAMIS (FALSE).

Ennek mintájára az

EXISTS x IN S : $C(x)$

kifejezés értéke IGAZ, ha létezik legalább egy S-beli x, amely kielégíti a $C(x)$ feltételt, egyébként HAMIS.

8.10. példa: A „Keressük meg a Disney-filmek szereplőit" lekérdezés megfogalmazásának egy újabb módja látható a 8.5. ábrán. Itt a színészekre összpontosítunk, akiket az s reprezentál, és azt kérdezzük, hogy egy színész szereplője-e valamely m filmnek, ahol m egy Disney-film. A 3) sor azt mondja, hogy tekintjük az összes m filmet az `s.szerepelBenne` értékeként adott filmek halmazából, vagyis azoknak a filmeknek a halmazából, amelyekben az s színész szerepel. A 4) sorban azt viszgáljuk, hogy az m film egy Disney-film-e. Ha legalább egy ilyen filmet találunk, akkor a 3) és 4) sorban megadott EXISTS kifejezés értéke IGAZ, különben HAMIS. □

```
1) SELECT s
2) FROM Színészek s
3) WHERE EXISTS m IN s.szerepelBenne :
4)     m.gyártó.név = "Disney"
```

8.5. ábra. A létezik *kvantor használata*

8.11. példa: Használjuk most a FOR ALL operátort annak a lekérdezésnek a megfogalmazására, amelyben azokat a színészeket keressük, akik csak Disney-filmekben szerepelnek. Ez a halmaz azokat a színészeket is tartalmazza, akik egyetlen filmben sem szerepelnek, legalábbis az adatbázisunk tekintetében. A lekérdezés természetesen kiegészíthető egy további feltétellel, amelyben megköveteljük, hogy a színész legalább egy filmben szerepeljen, de ezt a finomítást meghagyjuk feladatnak. A mi lekérdezésünk megfogalmazása a 8.6. ábrán látható. □

```
1) SELECT s
2) FROM Színészek s
3) WHERE FOR ALL m IN s.szerepelBenne :
4)     m.gyártó.név = "Disney"
```

8.6. ábra. A minden *kvantor használata*

8.3.2. Összesítő kifejezések

Az OQL ugyanazt az öt összesítő operátort használja, mint az SQL, ezek az AVG, COUNT, SUM, MIN és a MAX. Ugyanakkor míg az SQL-ben ezek az operátorok lényegében egy tábla valamely kijelölt oszlopára vonatkoznak, ugyanezeket az operátorokat

az OQL-ben olyan kollekciókra alkalmazzuk, amelyeknek elemei megfelelő típusúak. Ennek alapján a COUNT bármilyen kollekcióra vonatkozhat. A SUM és az AVG aritmetikai típusú (például egész számokból álló) kollekciókra alkalmazható. Végül pedig a MIN és a MAX olyan kollekciók esetén használható, amelyek olyan típusú elemekből állnak, hogy azokon értelmezett az összehasonlítás. Ilyenek például az aritmetikai értékek vagy a karakterláncok.

8.12. példa: Az adatbázisban szereplő filmek átlagos hosszának kiszámításához az összes filmhosszt tartalmazó multihalmazt kell létrehoznunk. Hangsúlyozzuk, hogy a filmek hosszainak nem a halmazát kell előállítanunk, mert abban az esetben két azonos hosszal rendelkező filmet csak egyszer vennénk figyelembe az átlag kiszámításakor. A lekérdezés a következő:

```
AVG(SELECT m.hossz FROM Filmek m)
```

Egy alkérdést használunk tehát a filmek hosszainak megszerzésére. Ennek eredménye a filmek hosszaiból álló multihalmaz, és az AVG operátort erre a multihalmazra alkalmazzuk, ami aztán a kívánt választ megadja. □

8.3.3. Csoportosító kifejezések

Az OQL átvette az SQL GROUP BY záradékát, de nem annak eredeti formájában, hanem érdekes módon átformálva, amivel új perspektívához juttatta azt. Az OQL-ben a GROUP BY záradék az alábbi részekből áll:

1. A GROUP BY kulcsszavak.
2. Egy vesszővel elválasztott lista, amely valahány partíciót definiál. Egy ilyen definíció (a lista egy tagja) tartalmaz
   a) egy mezőnevet,
   b) egy kettőspontot és
   c) egy kifejezést.

   Vagyis a GROUP BY záradék alakja:

   GROUP BY $f_1{:}e_1, f_2{:}e_2, \ldots, f_n{:}e_n$

   Egy GROUP BY záradék egy select-from-where lekérdezés után állhat. Az

$e_1, e_2, \ldots, e_n$

kifejezések hivatkozhatnak a FROM záradékban felsorolt változókra. Hogy a GROUP BY működését jól érthetően megmagyarázhassuk, szorítkozzunk arra a gyakori esetre,

amikor egyetlen változó áll a FROM záradékban. Legyen ez a változó x, amely egy C kollekcióban szereplő értékeken fut végig. A C kollekció minden i tagjára, amely kielégíti a WHERE záradékban megadott feltételt, kiértékeljük a GROUP BY után felsorolt kifejezések mindegyikét. Az így kapott $e_1(i), e_2(i), \ldots, e_n(i)$ értékek listája (értékkombináció) alkotja azt a csoportot, amelyikbe az i érték tartozik.

A GROUP BY tényleges visszatérő értéke struktúrák egy halmaza, amely a következő alakú elemekből áll:

Struct $(f_1{:}v_1, f_2{:}v_2, \ldots, f_n{:}v_n,$ partition:$P)$

Az első n mező egy csoportot jelent. A $v_1, v_2, \ldots, v_n$ egy olyan értéklista, amelyet a C kollekcióban szereplő legalább egy – a WHERE záradék feltételét kielégítő – i érték esetén megkapunk az $e_1(i), e_2(i), \ldots, e_n(i)$ kifejezések kiértékelésekor.

Az utolsó mezőnek speciális neve van: partition. Ennek értéke (P) az ebbe a csoportba tartozó i értékeket tartalmazza. Pontosabban szólva P egy multihalmaz, amely

```
Struct(x:i)
```

alakú struktúrákból áll, ahol x a FROM záradékban deklarált változó.

Egy GROUP BY záradékkal rendelkező select-from-where kifejezés SELECT záradékában csak a GROUP BY eredményében szereplő struktúrák mezőire lehet hivatkozni, amelyek tehát $f_1, f_2, \ldots, f_n$, és partition. A partition mezőnéven keresztül hivatkozhatunk a hozzá tartozó P multihalmazban szereplő struktúrák x mezőjére. Hivatkozhatunk tehát a FROM záradékban megjelenő x változóra. Ezt azonban csak valamilyen összesítő műveleten belül tehetjük meg, amely a P multihalmaz elemein végez összesítést.

8.13. példa: Készítsünk egy táblázatot, amely megadja az egyes stúdiók különböző években gyártott filmjeinek összhosszát. Az OQL-ben valójában struktúrák egy multihalmazát állítjuk elő, ahol a sruktúrák három komponensből állnak, nevezetesen egy stúdióból, egy évszámból és azoknak a filmeknek az összegzett hosszából, amelyeket az adott stúdió az adott évben készített. A lekérdezést a 8.7. ábra mutatja.

Nézzük először a lekérdezés FROM záradékát. Azt látjuk, hogy az m változó befutja az összes Film objektumot, vagyis jelen esetben az m játssza azt a szerepet, amelyet az általános tárgyalás során az x játszott. A GROUP BY záradék az std és az évsz mezőket szerepelteti, amelyeknek rendre az m.gyártó.név és az m.év kifejezések felelnek meg.

A *Micsoda nő!* például egy 1990-ben készített Disney-film. Amikor m az ennek megfelelő objektum, akkor az m.gyártó.név kifejezés értéke "Disney", az

```
SELECT std, évsz, összHossz: SUM(SELECT p.m.hossz
                                 FROM partition p)
FROM Filmek m
GROUP BY std: m.gyártó.név, évsz: m.év
```

8.7. ábra. *Filmek csoportosítása stúdió és év szerint*

`m.év` értéke pedig 1990. Következésképpen a `GROUP BY` eredményeként kapott halmaznak egy eleme lesz az alábbi struktúra:

`Struct(std:"Disney", évsz:1990, partition:`P`)`

Itt P struktúrák egy halmaza, amely többek között tartalmazza a

`Struct(m:`m_{mn}`)`

struktúrát, ahol m_{mn} a *Micsoda nő!* című filmnek megfelelő `Film` objektum. A P-ben szerepelnek még a további 1990-ben gyártott Disney-filmekből alkotott egykomponensű struktúrák, ahol a mező neve `m`.

Vegyük most a `SELECT` záradékot. A `GROUP BY` záradék eredményében előforduló minden egyes struktúrához felépítünk egy másik struktúrát, amelyet a lekérdezés végeredményéül adott multihalmaz fog tartalmazni. Az első komponens az `std`, ami annyit jelent, hogy a mezőnév `std`, értéke pedig a `GROUP BY` eredményéből származó struktúra `std` mezőjének értéke. Hasonlóképpen a végeredménybe kerülő struktúra második komponensét adó mező neve `évsz`, értéke pedig egyenlő a `GROUP BY` eredményéből származó struktúra `évsz` mezőjének értékével.

A lekérdezés végeredményében szereplő struktúra harmadik összetevője:

```
SUM(SELECT p.m.hossz FROM partition p)
```

Ennek a select-from kifejezésnek a megértéséhez mindenekelőtt azt kell látnunk, hogy a p változó a `GROUP BY` eredményében lévő struktúra `partition` mezőjének elemein fut végig. A p változó minden egyes értéke tehát egy `Struct(m:`o`)` típusú struktúra, ahol o egy `Film` objektum. Ennélfogva a `p.m` kifejezés nem egyéb, mint erre az o objektumra való hivatkozás, így a `p.m.hossz` ennek a `Film` objektumnak a hossz komponensét jelenti.

A select-from kifejezés tehát azt a multihalmazt állítja elő, amely az egy adott csoportba tartozó filmek hosszaiból áll. Ha például az `std` értéke `"Disney"` és az `évsz` értéke 1990, akkor a fenti select-from eredménye minden olyan film hosszát tartalmazza, amelyet a Disney-stúdió készített 1990-ben. Alkalmazva a `SUM` operátort erre a multihalmazra, a csoportba tartozó filmek hosszainak összegét kapjuk. Ennek megfelelően a lekérdezés végeredményének egy eleme lehet például a

```
Struct(std:"Disney", évsz:1990, összHossz:1234)
```

struktúra, amennyiben az 1990-ben készített Disney-filmek összhossza ténylegesen 1234. □

Ha a `FROM` záradékban egynél több változó szerepel, a lekérdezés értelmezése néhány változtatást igényel, de az alapelvek ugyanazok maradnak, mint az eddig ismertetett egyváltozós esetben. Tegyük fel, hogy a `FROM` záradék az $x_1, x_2, \ldots, x_k$ változókat tartalmazza. Ekkor:

1. Az $x_1, x_2, \ldots, x_k$ változók mindegyike használható a `GROUP BY` záradékban felsorolt $e_1, e_2, \ldots, e_n$ kifejezésekben.
2. A `partition` mező értékét adó multihalmaz struktúrái k darab mezőből állnak, amelyekhez rendelt mezőnevek $x_1, x_2, \ldots, x_k$.
3. Tegyük fel, hogy az $i_1, i_2, \ldots, i_k$ értékek rendre az $x_1, x_2, \ldots, x_k$ változók értékei, amelyek a `WHERE` záradékot igazzá teszik. Ekkor a `GROUP BY` eredményeként adódó halmazban van egy olyan struktúra, amelynek alakja a következő:

`Struct` $(f_1{:}e_1(i_1, i_2, \ldots, i_k), \ldots, f_n{:}e_n(i_1, i_2, \ldots, i_k)$, `partition:P)`

Továbbá a P multihalmaznak eleme az alábbi struktúra:

`Struct` $(x_1{:}i_1, x_2{:}i_2, \ldots, x_k{:}i_k)$

8.3.4. `HAVING` záradékok

Az OQL-ben egy `GROUP BY` záradékot követhet egy `HAVING` záradék, amelynek az SQL `HAVING` záradékához hasonló jelentése van. A `HAVING` záradék szintaxisa a következő:

`HAVING` <feltétel>

Egy `HAVING` záradék arra szolgál, hogy a hozzá tartozó `GROUP BY` által létrehozott csoportok közül bizonyosakat elhagyjunk. A megadott feltétel a `GROUP BY` eredményében lévő minden egyes struktúra `partition` mezőjének értékére vonatkozik. Ha teljesül, akkor a struktúra bekerül abba a körbe, amelyből a 8.3.3. részben leírtak szerint a lekérdezés végeredménye kialakul. Ha nem teljesül a feltétel, akkor a struktúrát elhagyjuk, vagyis nem használjuk a lekérdezés végeredményének előállításában.

8.14. példa: Ismételjük meg az 8.13. példát azzal a változtatással, hogy egy stúdió valamely évben gyártott filmjeinek az összhosszát csak abban az esetben kérjük megadni, ha az adott stúdió az adott évben készített legalább egy 120 percnél hosszabb filmet. A 8.8. ábrán látható lekérdezés elvégzi a feladatot. Észrevehető, hogy a `HAVING` záradékban az egy adott stúdió adott évben készített filmjeinek hosszait tar-

```
SELECT std, évsz, összHossz: SUM(SELECT p.m.hossz
                   FROM partition p)
FROM Filmek m
GROUP BY std: m.gyártó.név, évsz: m.év
HAVING MAX(SELECT p.m.hossz FROM partition p) > 120
```

8.8. ábra. *A figyelembe vett csoportok körének szűkítése*

talmazó multihalmaz előállítására ugyanazt az alkérdést használtuk, mint a `SELECT` záradékban. A `HAVING` záradékban ezeknek az értékeknek képezzük a maximumát, és ezt összehasonlítjuk 120-szal. □

8.3.5. Halmazműveletek

Az egyesítés, metszet és különbség halmazműveleteket két halmaz vagy multihalmaz típusú objektumra alkalmazhatjuk. Az SQL-hez hasonlóan ezeket a műveleteket a `UNION`, `INTERSECT`, illetve `EXCEPT` kulcsszavakkal fejezzük ki.

8.15. példa: Azoknak a filmeknek a halmazát, amelyekben Harrison Ford szerepel, de nem a Disney-stúdió által készített filmek, megkaphatjuk két select-from-where lekérdezés különbségeként. Ezt a 8.9. ábra mutatja. Az 1)–3) sorok megkeresik azon filmek halmazát, amelyeknek Ford szereplője, míg az 5)–7) sorok az azokból a filmekből álló halmazt adják meg, amelyek Disney-filmek. A 4) sorban álló `EXCEPT` veszi ezek különbségét. □

Vegyük észre, hogy a 8.9. ábra 1) és 5) sorai tartalmazzák a `DISTINCT` kulcsszót. Ez biztosítja azt, hogy a két lekérdezés eredménye halmaz típusú lesz. A `DISTINCT` nélkül ezek az eredmények multihalmaz típusúak lennének. Az OQL-ben a `UNION`, `INTERSECT` és `EXCEPT` operátorok halmazokon és multihalmazokon értelmezettek. Ha mindkét argumentum halmaz, akkor a műveletek jelentése a szokásos.

Ha azonban mindkét argumentum multihalmaz típusú, vagy az egyik multihalmaz, a másik pedig halmaz, akkor a műveleteknek a multihalmazokon értelmezett jelentése a mérvadó. A 4.6.2. részben leírtak alapján tudjuk, hogy egy multihalmazban valamely objektum többször előfordulhat. A multihalmazokon végzett műveletek szabályai a következők. Tegyük fel, hogy B_1 és B_2 két multihalmaz, továbbá x egy objektum, amely B_1-ben n_1-szer, B_2-ben n_2-ször szerepel. Az n_1 és n_2 értékek bármelyike 0 is lehet.

- A $B_1 \cup B_2$ multihalmazban az x objektum $(n_1 + n_2)$-ször szerepel.
- A $B_1 \cap B_2$ multihalmazban az x objektum $\min(n_1, n_2)$-ször szerepel.

```
1)    (SELECT DISTINCT m
2)    FROM Filmek m, m.szereplők s
3)    WHERE s.név = "Harrison Ford")
4) EXCEPT
5)    (SELECT DISTINCT m
6)    FROM Filmek m
7)    WHERE m.gyártó.név = "Disney")
```

8.9. ábra. *Két halmaz különbségét képző lekérdezés*

- A $B_1 - B_2$ multihalmazban:
  1. $n_1 \leq n_2$, akkor az x objektum 0-szor szerepel (nem szerepel);
  2. $n_1 > n_2$, akkor az x objektum $(n_1 - n_2)$-ször szerepel.

A 8.9. ábrán látható konkrét lekérdezéssel kapcsolatban elmondhatjuk, hogy eleve legfeljebb egyszer fordulhat elő egy film a két alkérdés bármelyikének eredményében, így ugyanazt az végeredményt kapjuk a `DISTINCT` használatától függetlenül. Van különbség viszont az eredmény *típusát* illetően. Ha a `DISTINCT` kulcsszót mindkét alkérdésben használjuk, akkor az eredmény típusa `Set<Film>`, míg ha legalább az egyikben nem szerepel a `DISTINCT`, akkor az eredmény típusa `Bag<Film>`.

8.3.6. Feladatok

8.3.1. feladat: Vegyük a 8.1.1. feladatban és 8.2. ábrán szereplő ODL sémát, és fogalmazzuk meg az alábbi lekérdezéseket OQL-ben:

* a) Keressük meg azokat a gyártókat, amelyek PC-ket és nyomtatókat is gyártanak.

* b) Keressük meg azokat a PC-gyártókat, amelyek csak legalább 2 gigabájt (GB) kapacitású merevlemezzel rendelkező PC-ket gyártanak.

c) Keressük meg azokat a gyártókat, amelyek gyártanak PC-ket, de nem gyártanak laptopokat.

* d) Adjuk meg a PC-k átlagos sebességét.

* e) Az egyes CD-sebességekhez adjuk meg, hogy a kategóriájukba eső PC-k átlagosan mekkora memóriával rendelkeznek.

! f) Keressük meg azokat a gyártókat, amelyek gyártanak valamilyen legalább 16 megabájt memóriával rendelkező terméket, valamint 1000 $-nál olcsóbb terméket is gyártanak.

!! g) Az átlagosan legalább 150-es sebességgel rendelkező PC-ket előállító minden egyes gyártóhoz keressük meg az általa kínált PC-k memóriaméretei közül a legnagyobbat.

8.3.2. feladat: Vegyük a 8.1.2. feladatban és 8.3. ábrán szereplő ODL sémát, és fogalmazzuk meg az alábbi lekérdezéseket OQL-ben:

a) Keressük meg azokat a hajóosztályokat, amelyekre teljesül az, hogy az osztályba tartozó összes hajót 1919 előtt avatták fel.

b) Keressük meg a vízkiszorítás-értékek maximumát.

! c) Minden egyes kaliberértékhez keressük meg azt a legkorábbi évet, amelyikben felavattak egy olyan kaliberrel rendelkező hajót.

*!! d) Minden egyes hajóosztályhoz, amelynek legalább egy hajóját 1919 előtt avatták fel, adjuk meg az osztály azon hajóinak számát, amelyek valamelyik csatában elsüllyedtek.

! e) Adjuk meg, hogy egy hajóosztályba átlagosan hány hajó tartozik.

! f) Adjuk meg a hajók átlagos vízkiszorítását.

!! g) Keressük meg azokat a csatákat (objektumokat, tehát nem csataneveket), amelyekben legalább egy hajó Nagy-Britanniából részt vett, és amelyekben legalább két hajó elsüllyedt.

! 8.3.3. feladat: A 8.11. példában említettük, hogy a 8.6. ábrán bemutatott OQL lekérdezés azokat a színészeket is visszaadja az eredményben, akik egyáltalán nem is szerepelnek egyetlen filmben sem. Fogalmazzuk meg újra a lekérdezést, de most úgy, hogy csak azok a színészek kerüljenek be az eredménybe, akik legalább egy filmben szerepelnek, és akik csak Disney-filmekben szerepelnek.

8.4. Objektumok létesítése és értékül adása az OQL-ben

Ebben az alfejezetben azt nézzük meg, hogy milyen a kapcsolódás az OQL és a hozzá választott befogadó nyelv között. Példáinkban a C++-t fogjuk használni erre a célra, jóllehet bizonyos rendszerekben más objektumorientált általános célú programozási nyelvek is betölthetik a befogadó nyelv szerepét.

8.4.1. Befogadó nyelvi változókhoz történő érték-hozzárendelés

Az SQL-lel ellentétben, ahol a sorok komponensei és a befogadó nyelv változói között adatmozgatásra van szükség, az OQL természetes módon beleilleszkedik a befogadó nyelvbe. Ez annak következménye, hogy az eddigiek során is tárgyalt OQL kifejezések, mint például a select-from-where kifejezések, értékekként objektumokat állítanak elő. Egy ilyen OQL kifejezés eredményeként előállított értéket pedig bármely olyan változónak értékül adhatjuk, amely megfelelő típusú.

8.16. példa: Az alábbi lekérdezés azoknak a filmeknek a halmazát állítja elő, amelyek 1920 előtt készültek:


```
SELECT DISTINCT m
FROM Filmek m
WHERE m.év < 1920
```

A lekérdezés eredményének típusa `Set<Film>`. Ha `régiFilmek` egy ugyanilyen típusú befogadó nyelvi változó, akkor (az OQL-lel kiterjesztett C++-ban) írhatjuk a következőt:

```
régiFilmek = SELECT DISTINCT m
      FROM Filmek m
      WHERE m.év < 1920;
```

Ennek megfelelően a `régiFilmek` változó értéke a kiválasztott `Film` objektumok halmaza lesz. □

8.4.2. Hozzáférés a kollekciók elemeihez

Mivel minden select-from-where és group-by kifejezés kollekciót – halmazt vagy multihalmazt – eredményez, tennünk kell még valami pluszt, ha a kollekció valamely elemét akarjuk megkapni. Ez az állításunk akkor is helytálló, ha egy olyan kollekcióval van dolgunk, amely egyetlen elemet tartalmaz. Az OQL az `ELEMENT` operátor bevezetésével gondoskodik arról, hogy egy ilyen egyelemű halmaznak vagy multihalmaznak az egyetlen tagját előállíthassuk. Ez az operátor alkalmazható például egy olyan lekérdezés eredményére, amelyről tudjuk, hogy egyelemű kollekciót ad.

8.17. példa: Tegyük fel, hogy van egy `elfsz` változónk, amelynek típusa `Film` (azaz a `Film` osztály), és amelyhez hozzá akarjuk rendelni az *Elfújta a szél* című filmet reprezentáló objektumot. Tekintsük először az alábbi lekérdezést:

```
SELECT m
FROM Filmek m
WHERE m.cím = "Elfújta a szél"
```

A lekérdezés eredménye az a multihalmaz, amely egyedül a kérdéses objektumot tartalmazza. Ezt nem adhatjuk értékül az `elfsz` változónak, mert az típushibához vezetne. Ha viszont először alkalmazzuk az `ELEMENT` operátort, akkor az így kapott kifejezés típusa illeszkedik a változó típusához, így az értékadás elvégezhető. Tehát az

```
elfsz = ELEMENT(SELECT m
        FROM Filmek m
        WHERE m.cím = "Elfújta a szél");
```

értékadó utasítás megengedett. □

8.4.3. Egy kollekció összes elemének elérése

Egy halmaz vagy multihalmaz összes eleméhez való hozzáférés már bonyolultabb, de még mindig egyszerűbb, mint az SQL-ben szükséges kurzoralapú algoritmusok. Először a halmazt vagy multihalmazt egy listává kell alakítani. Ezt úgy valósítjuk meg, hogy `ORDER BY` záradékot tartalmazó select-from-where kifejezést használunk. A 8.2.8. részben mondottakat felidézve emlékezhetünk, hogy egy ilyen kifejezés a kiválasztott objektumok vagy értékek egy listáját adja eredményül.

8.18. példa: Tegyük fel, hogy egy listát szeretnénk előállítani, amely a `Film` osztályba tartozó összes objektumot tartalmazza. A sorrend egyértelműségének biztosítása érdekében a cím és ezen belül az év szerinti rendezettséget érdemes előírni, mivel a `(cím, év)` kombináció kulcsot alkot a `Film` osztályban. Az itt következő

```
filmLista = SELECT m
       FROM Filmek m
       ORDER BY m.cím, m.év;
```

utasítás tehát az összes `Film` objektumot magában foglaló, cím és név szerint rendezett listát hozzárendeli a `filmLista` befogadó nyelvi változóhoz. □

Amint kezünkben van egy lista, legyen az rendezett vagy nem rendezett, a lista elemeihez sorszám szerint hozzáférhetünk. Az L lista i-edik elemét az $L[i-1]$ kifejezés adja meg. A listák és tömbök elemeinek sorszámozása a C-hez vagy C++-hoz hasonlóan 0-val kezdődik.

8.19. példa: Tegyük fel, hogy egy C++ függvényt akarunk írni, amely kiírja az egyes filmek címét, készítési évét és hosszát. A függvény vázát a 8.10. ábrán láthatjuk.

Az 1) sor rendezi a `Film` osztályba tartozó objektumokat, és a kapott eredményt értékül adja a `List<Film>` típusú `filmLista` változónak. Az OQL `COUNT` operátorát használva a 2) sor kiszámítja a filmek számát. A 3)–7) sorok egy ciklust tartalmaznak, amelyben az egész típusú `i` változó végigfut a lista pozcióin. A lista i-edik elemét beteszi a `film` változóba, majd az 5)–6) sorokban megtörténik az aktuális film megfelelő attribútumainak kiíratása. □

```
1) filmLista =  SELECT m
                FROM Filmek m
                ORDER BY m.cím, m.év;
2) filmekSzáma = COUNT(Filmek);
3) for(i=0; i<filmekSzáma; i++) {
4)    film = filmLista[i];
5)    cout << film.cím << " " << film.év << " "
6)       << film.hossz << "\n";
7) }
```

8.10. ábra. *Filmek adatainak kiíratása*

8.4.4. Új objektumok létrehozása

Azt már láttuk, hogy a select-from-where típusú OQL kifejezések új objektumok létrehozását teszik lehetővé. Az így kialakult objektumok a már létező objektumokon végzett műveletek eredményeként jönnek létre. Úgy is alkothatunk új objektumokat, hogy konstansokból vagy más kifejezésekből közvetlenül építünk fel struktúrákat vagy kollekciókat. Ilyenkor valamilyen típuskonstruktort alkalmazunk bizonyos értékekre a maga nyilvánvaló módján. Amikor értékeket konstruálunk, a típusok leírásakor használatos kapcsos zárójelek helyett kerek zárójeket használunk. A 8.7. példában láttunk erre egy példát, ahol a

```
SELECT DISTINCT Struct(színész1: s1, színész2: s2)
```

sorban azt írtuk elő, hogy a lekérdezés eredménye egy halmaz legyen, amely `Struct{színész1: Színész, színész2: Színész}` típusú objektumokból áll. Ennek pontos meghatározásához a `színész1` és `színész2` mezőneveket meg kellett adnunk, a mezők típusait viszont az `s1` és `s2` változók típusaiból lehetett származtatni.

8.20. példa: Kollekciókat úgy is létrehozhatunk, hogy a kollekciótípusoknak megfelelő `Set`, `Bag`, `List` vagy `Array` típuskonstruktorok bármelyikét alkalmazzuk megegyező típusú objektumokra. Nézzük például a következő értékadásokat:

```
x = Struct(a:1, b:2);
y = Bag(x, x, Struct(a:3, b:4));
```

Az első sor az `x` változóhoz egy

```
Struct(a:integer, b:integer)
```

típusú értéket rendel, vagyis egy két egész típusú mezővel rendelkező struktúrát, ahol a mezőnevek `a` és `b`. Az ilyen típusú értékeket elképzelhetjük olyan sorokként, ahol csak az egész számok szerepelnek mint komponensek, az `a` és `b` mezőnevek viszont nem. Ennek megfelelően az `x` értéke az (1, 2) párral ábrázolható. A második sor az `y` értékét egy multihalmazként definiálja, amely az `x` típusával megegyező típusú struktúrákat tartalmaz. Ebben a mutihalmazban az (1, 2) pár kétszer, a (3, 4) pár pedig egyszer fordul elő. □

Ha van egy definiált típusunk és valamely lekérdezés ilyen típusú objektumok kollekcióját állítja elő, akkor a már létező típus nevét használhatjuk a lekérdezésben ahelyett, hogy a típust újra specifikálnánk. A 8.7. példában láttuk például, hogy színészpárok egy halmazát hogyan hozhatjuk létre. A `SELECT` záradékban egy olyan kifejezést adtunk meg, amely a `Struct` típuskonstruktort használva két mezőből álló objektumokat épített fel, ahol is mindkét mező `Színész` objektumokat tartalmazott.

Tegyük fel, hogy a `SzínészPár` típus az alábbi struktúraként definiált:

```
Struct{színész1: Színész, színész2: Színész}
```

Ekkor a 8.7. példa lekérdezését átírhatjuk úgy, hogy ezt a típust használjuk a `SELECT` záradékban, mégpedig a következőképpen:

```
SELECT DISTINCT SzínészPár(színész1: s1, színész2: s2)
FROM Színészek s1, Színészek s2
WHERE s1.cím = s2.cím AND s1.név < s2.név
```

A 8.7. példához képest az egyetlen különbség az, hogy jelen esetben a lekérdezés eredményének típusa `Set<SzínészPár>`. Következésképpen ez az eredmény egy ilyen típusúnak deklarált befogadó nyelvi változónak adható értékül.

Egy típus nevének argumentumokra történő alkalmazása különösen akkor hasznos, amikor ez a típusnév egy osztálynak a neve. Egy osztályhoz általában több különböző alakú *konstruktorfüggvény* tartozik attól függően, hogy mely összetevőket inicializáljuk explicit módon, illetve melyek kapnak valamilyen alapértelmezés szerinti értéket. Metódusokat például bizonyosan nem inicializálunk. Az attribútumok többsége általában kap valamilyen kezdeti értéket, a kapcsolatokat pedig inicializálhatjuk például az üres halmazzal, és ezeket gyarapíthatjuk a későbbiek során. Mindegyik konstruktorfüggvény neve megegyezik az osztály nevével, megkülönböztetésük pedig az argumentumokban szereplő mezőnevek alapján történik.

8.21. példa: Nézzünk egy lehetséges konstruktorfüggvényt `Film` objektumok létrehozásához. Feltevésünk szerint ez a függvény a `cím`, `év`, `hossz` és `gyártó` komponensekhez veszi a megadott értékeket, és létrehoz egy objektumot, amelyben a felsorolt mezők értékei ezek az értékek lesznek, a `szereplők` mező pedig színészek egy üres halmaza lesz. Ha `mgm` egy változó, amelynek értéke az MGM `Stúdió` objektum, akkor az *Elfújta a szél* című filmnek megfelelő objektumot létrehozhatjuk az alábbi utasítással:

```
elfsz = Film(cím: "Elfújta a szél",
        év: 1939,
        hossz: 239,
        gyártó: mgm);
```

Az utasításnak kettős hatása van:

1. Létrehoz egy új `Film` objektumot, amely a `Filmek` objektumkészlet részévé válik.
2. Ezt az objektumot az `elfsz` befogadó nyelvi változó értékévé teszi.

□

8.4.5. Feladatok

8.4.1. feladat: Az x befogadó nyelvi változónak adjuk értékül a következőképpen definiált konstansokat:

* a) Az {1, 2, 3} halmaz.

b) Az {1, 2, 3, 1} multihalmaz.

c) Az (1, 2, 3, 1) lista.

d) Az a két komponensből álló struktúra, amelyben az első mező neve `a`, amelynek értéke az {1, 2} halmaz, a második mező neve `b`, amelynek értéke az {1, 1} multihalmaz.

e) Az a multihalmaz, amely `a` és `b` mezőkből álló struktúrákat tartalmaz, és az (1, 2), (2, 1) és (1, 2) értékpároknak megfelelő három struktúrából áll.

8.4.2. feladat: Vegyük a 8.1.1. feladatban és 8.2. ábrán szereplő ODL sémát, és írjunk olyan utasításokat az OQL-lel kiterjesztett C++-ban (vagy egy szabadon választott objektumorientált befogadó nyelvben), amelyek elvégzik a következő feladatokat:

* a) Az x befogadó nyelvi változónak adjuk értékül azt az objektumot, amelyik az 1000 modellszámmal rendelkező PC-nek felel meg.

b) Az y befogadó nyelvi változónak adjuk értékül a legalább 16 megabájt memóriával rendelkező laptop objektumokból álló halmazt.

c) A z befogadó nyelvi változónak adjuk értékül az 1500 $-nál olcsóbb PC-k átlagos sebességét.

! d) Keressük meg az összes lézernyomtatót, írassuk ki a hozzájuk tartozó modellszámok és árak listáját, valamint a végén írassunk ki egy üzenetet, amely jelzi, hogy melyik modellszámhoz tartozik a legalacsonyabb ár.

!! e) Írassunk ki egy táblázatot, amely minden PC-t készítő gyártóhoz megadja a hozzá tartozó minimális és maximális árat.

8.4.3. feladat: Ebben a feladatban a 8.1.2. feladatban és 8.3. ábrán szereplő ODL sémát fogjuk használni. Feltételezzük, hogy az ott előforduló mind a négy osztályhoz van egy olyan, az osztály nevével megegyező nevű konstruktorfüggvény, amely az összes attribútum és egyértékű kapcsolat esetén a nekik megfelelő értékek megadását várja, a többértékű kapcsolatokat pedig üres halmazokkal inicializálja. Egy egyértékű kapcsolat megadásakor alapul vehetjük annak a befogadó nyelvi változónak a meglé-

tét, amelynek értéke az az objektum, amelyikkel a kapcsolatot létesítjük. Hozzuk létre, és egy befogadó nyelvi változónak adjuk értékül azokat az objektumokat, amelyeket az alábbi állítások határoznak meg:

* a) A Colorado csatahajó a Maryland hajóosztályba tartozott és 1923-ban avatták fel.

b) A Graf Spee csatahajó a Lützow hajóosztályba tartozott és 1936-ban avatták fel.

c) A malayai (Malaya nevű) csatában a Prince of Wales csatahajó elsüllyedt.

d) A malayai csata 1941. december 10-én zajlott.

e) A Hood hajóosztály brit csatahajókat jelentett, amelyek nyolc darab 15 hüvelyk kaliberű ágyúval voltak felszerelve, vízkiszorításuk pedig 41 000 tonna volt.

8.5. Sorobjektumok az SQL3-ban

Az OQL nem ismeri a reláció speciális fogalmát, egy relációnak az OQL-ben struktúrák egy halmaza (vagy multihalmaza) felel meg. Az SQL-ben azonban a reláció olyannyira fontos fogalom, hogy az SQL3 is központi fogalomként tekinti a relációkat. Az SQL3-ban kétféleképpen jelennek meg objektumok:

1. *Sorobjektumok*, amelyek lényegében sorokat jelentenek.

2. *Absztrakt adattípusok* (*Abstract Data Types*, az SQL3 dokumentációjában ezekre gyakran az *ADT* rövidítéssel hivatkoznak), olyan általános objektumok, amelyek csak sorok komponenseiként használhatók.

Ebben az alfejezetben a sorobjektumokkal foglalkozunk, az absztrakt adattípusokat pedig a 8.6. alfejezetben tárgyaljuk.

8.5.1. Sortípusok

Az SQL3-ban definiálhatunk ún. sortípusokat, és egy ilyen típus nagyjából objektumok egy osztályához hasonlítható. Egy sortípus deklarációja a következőket tartalmazza:

1. A CREATE ROW TYPE kulcsszavak.

2. A típus neve.

3. Attribútumok és típusuk zárójelezett listája.

Egy *T* sortípus definíciója tehát az alábbi alakú:

CREATE ROW TYPE *T* (<komponensek deklarálása>)

8.22. példa: A 8.4. ábrán szereplő OQL példa tartalmazza a `Színész` osztályt. Ennek analógiájára az SQL3-ban létrehozhatunk egy sortípust a filmszínészek modellezésére. Ugyanakkor viszont filmek egy halmazát nem tudjuk közvetlenül reprezentálni ennek a sortípusnak egy mezőjeként. Emiatt a `Színész` soroknak először csak a `név` és `lakcím` komponenseit vesszük figyelembe.

Mindenekelőtt észre kell vennünk, hogy a 8.4. ábrán a lakcím típusa maga is egy sor, melynek komponensei az `utca` és a `város`. Ebből az következik, hogy két típus definiálására van szükség, az egyik a lakcímek, a másik pedig a színészek modellezéséhez kell. Egy sortípus felhasználása egy másik sortípus vagy egy reláció valamely komponensének típusaként megengedett az SQL3-ban. A szükséges definíciókat a 8.11. ábra mutatja.

Egy `CímTípus` típusú sor két komponensből áll. A sor attribútumai az `utca` és a `város`, amelyek 50, illetve 20 hosszú karakterlánc típusú attribútumok. Egy `SzínészTípus` típusú sornak szintén két összetevője van. Az első a `név` attribútum, amely 30 hosszú karakterlánc típusú. A második attribútum a `lakcím`, ennek típusa a `CímTípus`, vagyis egy `utca` és `város` komponensekből álló sortípus. □

```
CREATE ROW TYPE CímTípus(
   utca    CHAR(50),
   város CHAR(20)
);

CREATE ROW TYPE SzínészTípus(
   név     CHAR(30),
   lakcím  CímTípus
);
```

8.11. ábra. *Két sortípus definíciója*

8.5.2. Reláció létrehozása sortípus felhasználásával

Miután deklaráltunk egy sortípust, létrehozhatunk egy vagy több relációt, amelyek a sortípusnak megfelelő típusú sorokat tartalmaznak. Egy ilyen relációt az 5.7.2. részben leírtakhoz hasonlóan hozhatunk létre, azzal a különbséggel, hogy az attribútumok és típusuk felsorolása helyett, ami a hagyományos SQL-beli tábladefinícióban szükséges, az alábbit adjuk meg:

OF TYPE <sortípus neve>

8.23. példa: Tegyük fel, hogy a `FilmSzínész` nevű reláció `SzínészTípus` típusú sorokat tartalmaz. Ezt a relációt a következő utasítással hozhatjuk létre:

```
CREATE TABLE FilmSzínész OF TYPE SzínészTípus;
```

Az utasítás azt eredményezi, hogy a `FilmSzínész` táblának két attribútuma lesz, a `név` és a `lakcím`. Az utóbbi típusa maga is egy sortípus, ezt az SQL3 előtti SQL szabványok nem tették lehetővé. □

Gyakori eset, hogy minden egyes sortípushoz pontosan egy reláció tartozik, amikor is a relációt tekinthetjük úgy, mint annak az osztálynak az objektumkészletét (a 8.1.3. részben mondottak értelmében), amelyik a sortípusnak megfelel. Ugyanakkor nem feltétlenül tartozik reláció valamely sortípushoz, valamint több relációt is létrehozhatunk egy adott sortípus felhasználásával.

8.5.3. Sortípus komponenseinek elérése

Mivel az SQL3-ban a komponensek maguk is lehetnek összetett szerkezetűek, szükségünk van egy olyan mechanizmusra, amelyen keresztül hozzáférhetünk komponensek komponenseihez. Az SQL3 egy két (dupla) pontot (`..`) használó jelölési módot vezetett be erre a célra, ami nagyjából az OQL és a C egy pontból álló jelölésének felel meg.

8.24. példa: A 8.12. ábrán látható lekérdezés minden Beverly Hills-ben lakó színész esetén megadja a színész nevét, valamint azt, hogy milyen nevű utcában lakik. Hogy érzékeltetni tudjuk az egy, illetve a dupla pont használata közötti különbséget, a példában a `FilmSzínész.név` és a `FilmSzínész.lakcím` teljes attribútumneveket használtuk. Mivel azonban a `név` és `lakcím` attribútumok itt egyértelműek, a `FilmSzínész` és a (nem dupla) pont használata ezeken a helyeken nem szükséges. □

```
SELECT FilmSzínész.név, FilmSzínész.lakcím..utca
FROM FilmSzínész
WHERE FilmSzínész.lakcím..város = 'Beverly Hills';
```

8.12. ábra. *Komponensek komponenseinek elérése*

8.5.4. Hivatkozások

Az objektumok azonosítását az SQL3 a *hivatkozás* (reference) fogalmának bevezetésével oldja meg. Egy sortípus valamely komponensének típusa lehet egy hivatkozás egy másik sortípusra. Ha T egy sortípus, akkor REF(T) egy olyan hivatkozás típusa, amely egy T típusú sorra hivatkozik. Ha egy sort objektumként képzelünk el, akkor egy hivatkozás arra az objektumra az ő objektumazonosítóját jelenti.

8.25. példa: A `FilmSzínész` relációban ugyan még nem tudjuk tárolni az egyes színészekhez tartozó filmek halmazát, de rögzítsük legalább azok legjobb filmjeit. Ehhez szükség van egy `Film` relációra. Definiáljunk először egy ennek megfelelő sortípust, majd a relációt ennek felhasználásával hozzuk létre. A filmek modellezésére az alábbi egyszerű típust deklaráljuk (nem tartalmazza a filmekhez tartozó színészeket, stúdiókat vagy producereket):

```
CREATE ROW TYPE FilmTípus(
  cím    CHAR(30),
  év   INTEGER,
  színes BIT(1)
);
```

Most már létrehozhatjuk az ilyen típusú sorokat tartalmazó `Film` relációt:

```
CREATE TABLE Film OF TYPE FilmTípus;
```

Következő lépésként a `FilmSzínész` sorok típusát kell kiegészíteni egy komponenssel. Ez az új mező egy adott színész esetén az ő legjobb filmjére történő hivatkozást fogja tartalmazni.[2] A `SzínészTípus` új definíciója az alábbi:

```
CREATE ROW TYPE SzínészTípus(
      név            CHAR(30),
      lakcím         CímTípus,
      legjobbFilm REF(FilmTípus)
);
```

□

8.26. példa: Tegyük fel most, hogy kezelni akarjuk a filmek és színészek között létező sok-sok kapcsolatot is: egy színész szerepel filmek egy halmazában, illetve egy filmhez tartozik színészek egy halmaza. Míg az ODL-ben egy film valamely komponense lehet színészek egy halmaza és fordítva, az SQL3 megőrzi a relációs szemléletet, amit mi is követtünk végig a könyvben.[3] Egy sok-sok kapcsolat ennek megfelelő-

[2] Az SQL3 nem tartalmaz `ALTER TYPE` vagy valami hasonló utasítást, amely lehetővé tenné már létező típusok módosítását. Emiatt, ha egy sortípus definícióját meg akarjuk változtatni, akkor először törölnünk kell a sortípust és az összes olyan táblát, amelyet annak felhasználásával hoztunk létre, majd újra kell definiálnunk a sortípust, és újra létre kell hoznunk a táblákat.

[3] Habár az SQL3 szabvány néhány korábbi előzetes specifikációja megengedte a kollekciótípusok (például hamazok vagy multihalmazok) használatát attribútumok típusaiként, nagyon valószínű, hogy az ilyen értelemben vett kollekciótípusok csak a későbbi SQL4 szabványban kapnak majd helyet.

en egy olyan külön relációban ábrázolható, amely az egymással kapcsolatban álló párokat tartalmazza.

A 8.13. ábra mutat egy megoldást arra vonatkozóan, hogy a filmek és színészek között fennálló szereplési kapcsolatot hogyan reprezentálhatjuk. Az SQL3 előtti szabványokban a sok-sok kapcsolatokat csak úgy tudtuk reprezentálni, hogy a kapcsolatban álló osztályok kulcsai alapján képeztünk párokat. Az SQL3-ban hivatkozás típusú attribútumok segítségével közvetlenül is hivatkozhatunk objektumokra (pontosabban sorokra). Először a `FilmTípus` és a `SzínészTípus` típusokat definiáljuk, amelyek a `Film`, illetve a `FilmSzínész` relációkhoz szolgáltatják a típust. A `SzínészTípus` esetében visszatértünk az eredeti definícióhoz, amely a legjobb filmnek megfelelő komponenst nem tartalmazza. A `SzerepelBenneTípus` sortípus, amely a `SzerepelBenne` relációhoz adja a típust, két hivatkozás típusú komponenst tartalmaz, amelyek egy színészre és egy filmre hivatkoznak. Egy ilyen pár azt fejezi ki, hogy az adott színész szerepel az adott filmben.

A sortípusok definiálása után következik a három tábla `Film`, `FilmSzínész` és `SzerepelBenne` létrehozása, amelyek használják a sortípusokat. Észrevehetjük, hogy a `CímTípus` sortípust egyetlen tábla típusaként sem használtuk, hanem csak mint a `SzínészTípus` sortípus `lakcím` attribútumának típusa szerepel.

```
CREATE ROW TYPE FilmTípus(
   cím      CHAR(30),
   év       INTEGER,
   színes   BIT(1)
);

CREATE ROW TYPE CímTípus(
   utca     CHAR(50),
   város    CHAR(20)
);

CREATE ROW TYPE SzínészTípus(
   név      CHAR(30),
   lakcím   CímTípus
);

CREATE ROW TYPE SzerepelBenneTípus(
   színész  REF(SzínészTípus),
   film     REF(FilmTípus)
);

CREATE TABLE Film OF TYPE FilmTípus;

CREATE TABLE FilmSzínész OF TYPE SzínészTípus;

CREATE TABLE SzerepelBenne OF TYPE SzerepelBenneTípus;
```

8.13. ábra. *Színészek, filmek és kapcsolataik*

Ha összehasonlítjuk az itteni `SzerepelBenne` relációt a 3.9. alfejezetben használt adatbázisséma ugyanilyen nevű relációjával, akkor azt látjuk, hogy ez utóbbi reláció a `filmCím` és az `év` attribútumokat használja a film sorokra történő közvetlen hivatkozások helyett, illetve a `színész` – neveket tartalmazó – attribútumot a színész sorokra történő közvetlen hivatkozások helyett. □

8.5.5. Hivatkozások követése

Miután elfogadjuk, hogy egy sor valamely komponense lehet egy másik sorra történő hivatkozás, természetes módon adódik az igény, hogy az SQL-t kibővítsük egy, a *hivatkozásokat feloldó* (dereferencing) operátorral. A hivatkozások feloldására az SQL3 a `->` operátort használja, jelentése pedig megegyezik az operátor C-beli jelentésével. Ha tehát x egy hivatkozás egy t sorra, az a pedig a t egy attribútuma, akkor `x->a` az a attribútum értéket jelenti a t sorban. Ez az operátor jól használható SQL3 lekérdezésekben, mivel helyettesíthet bizonyos összekapcsolásokat, amelyek az SQL2-ben elengedhetetlenek.

8.27. példa: Vegyük a 8.13. ábrán látható sémát, és keressük meg az összes olyan filmnek a címét, amelyben Mel Gibson játszik. Ehhez megvizsgálunk minden egyes, a `SzerepelBenne` relációban előforduló párt. Ha a hivatkozott színész Mel Gibson, akkor az eredmény részeként előállítjuk annak a filmnek a címét, amelyre a pár másik komponense hivatkozik. Az SQL3 lekérdezés az alábbi:

```
SELECT film->cím
FROM SzerepelBenne
WHERE színész->név = 'Mel Gibson';
```

Értéktartományok és sortípusok

Az 5.7.6. részben beszéltünk az értéktartományokról, amelyek egyfajta típusdeklarációk. Az értéktartományok és a sortípusok között legalább két lényeges különbséget fedezhetünk fel. Egy nyilvánvaló különbség az, hogy egy értéktartomány egy komponens típusát definiálja, míg egy sortípus teljes soroknak a típusát jelenti.

A másik különbség alapjául az szolgál, hogy az értéktartományok valójában csak rövidítések. Két különböző értéktartomány jelentheti ugyanazt a típust, amikor is a belőlük származó értékek között nincs megkülönböztetés. Ha viszont két azonos definícióval rendelkező, de különböző sortípusunk van, mondjuk T_1 és T_2, akkor két nekik megfelelő reláció sorai nem felcserélhetőek. Egy REF(T_1) típusú attribútum például nem hivatkozhat egy T_2 típusú sorra.

Hivatkozások feloldása és komponensek elérése

Az SQL3 és az OQL közötti különbségek egyike abban áll, ahogyan azok a -> és a pont operátorokat értelmezik. A 8.2.3. alfejezetben elhangzottak alapján tudjuk, hogy az OQL-ben a -> és a pont operátorok egymás szinonimái. Mindkettő egy struktúra típusú objektumra alkalmazható, és az objektum egy komponensét adja vissza. Az SQL3 a C-hez hasonlóan különbséget tesz a két operátor között. A -> operátort csak egy sorra történő hivatkozásra alkalmazhatjuk, míg a pont operátor (ami az SQL3 esetében két pontot jelent) magukra a sorváltozókra alkalmazható. Csakúgy mint C-ben, ha *r* egy hivatkozás egy *t* sorra, akkor `r->a` ugyanazt az értéket eredményezi, mint `t..a`.

A lekérdezést a következőképpen értelmezzük. Mint ahogy azt megszoktuk az SQL select-from-where lekérdezéseinél, vesszük a `FROM` záradékban feltüntetett reláció minden egyes sorát. Jelöljünk egy ilyen sort az (*s*, *m*) párral, ahol *s* egy hivatkozás egy színész sorra, *m* pedig egy hivatkozás egy film sorra. A `WHERE` záradék megvizsgálja, hogy Mel Gibson-e a `név` komponense annak a `FilmSzínész` sornak, amelyre az *s* hivatkozik. Ha igen, akkor a lekérdezés eredményébe bekerül egy egyetlen értékből álló sor, amely érték annak a `Film` sornak a `cím` komponense, amelyre az *m* hivatkozik. □

8.5.6. Hivatkozások hatásköre

Egy olyan lekérdezés megválaszolásához, mint amilyen a 8.27. példában is szerepel, egy SQL3 adatbázisrendszernek a `színész->név` hivatkozást feloldó kifejezést úgy kell értelmeznie, hogy az egy bizonyos reláció `név` mezőjére történő hivatkozás. Ehhez tekinthetjük például a `SzerepelBenne` reláció minden egyes sorát, ahol aztán követjük a `színész` hivatkozást, és megvizsgáljuk, hogy a hivatkozott sorban névként „Mel Gibson" szerepel-e. Ez azonban a válaszadásnak egy nagyon időigényes módja abban az esetben, ha a `SzerepelBenne` reláció nagy.

Hatékonyabb megoldást kaphatunk, ha indexet hozunk létre valamely *R* reláció `név` attribútumára vonatkozóan. Ekkor egy bizonyos értékből kiindulva – például „Mel Gibson" – eljuthatunk azokhoz a `SzerepelBenne` sorokhoz, amelyek az *R* olyan soraira hivatkoznak, ahol a `név` attribútum értéke „Mel Gibson". De melyik *R* relációban vagy relációkban végezzük a keresést ilyen indexet használva? Példánk esetében tudjuk, hogy mindegyik `SzerepelBenne` sor `színész` attribútumának értéke egy olyan sorra való hivatkozás, amely sor egy `SzínészTípus` típusú relációban szerepel. Csak egy ilyen relációt hoztunk létre, nevezetesen a `FilmSzínész` relációt, így arra számítunk, hogy ez az a reláció, amelynek soraira a hivatkozások történnek.

Létezhetnek azonban további `SzínészTípus` típusú relációk, amikor is a „Mel Gibson" nevet az összes ilyen relációhoz tartozó indexben kell keresni. Ez a keresés felesleges időpocsékolás ha – a séma tervezője számára ismeretes okokból – minden

hivatkozás egy bizonyos relációba mutat, mint ahogy ennek normális esetben lennie kellene. Az SQL3 biztosít egy mechanizmust, amellyel előírhatjuk, hogy egy hivatkozás típusú attribútum melyik relációba mutasson. Egy hivatkozás típusú attribútummal rendelkező reláció létrehozásakor egy plusz záradék hozzáadásával meghatározhatjuk az attribútum hatáskörét. A záradék az alábbi alakú:

```
SCOPE FOR <attribútum> IS <reláció>
```

A záradék jelentése az, hogy a megnevezett atribútum, amelynek hivatkozás típusúnak kell lennie, minden esetben a megnevett reláció egy sorára hivatkozik.

8.28. példa: Hogy garantáljuk azt, hogy a `SzerepelBenne` táblában a `színész` attribútum mindig a `FilmSzínész` tábla soraira, illetve a `film` attribútum mindig a `Film` tábla soraira hivatkozik, a `SzerepelBenne` relációt a 8.14. ábrán látható módon definiálhatjuk. □

```
CREATE ROW TYPE SzerepelBenneTípus(
   színész REF(SzínészTípus),
   film    REF(FilmTípus)
);

CREATE TABLE SzerepelBenne OF TYPE SzerepelBenneTípus
SCOPE FOR színész IS FilmSzínész,
SCOPE FOR film IS Film;
```

8.14. ábra. *Hivatkozás típusú attribútumok hatáskörének deklarálása*

8.5.7. Objektumazonosítók mint értékek

Az objektumorientált nyelvek egy általános alapelve, hogy az objektumazonosítók a rendszer belső értékei, amelyek a lekérdezőnyelven keresztül nem hozzáférhetőek. Az OQL is megfelel ennek az alapelvnek. Elvileg azonban semmi nem indokolja, hogy ne hivatkozhassunk közvetlenül az objektumazonosítókra, és az SQL3 ezt lehetővé is teszi számunkra. Egy reláció vagy a neki megfelelő sortípus deklarálásakor lehet egy olyan attribútumunk, amely ugyanolyan típusú sorokra hivatkozik. Ha egy sortípus vagy tábla deklarációjában szerepeltetjük a

```
VALUES FOR <attribútum> ARE SYSTEM GENERATED
```

záradékot, akkor a megadott attribútum értékei „önhivatkozások", azaz magukra a hivatkozásokat tartalmazó sorokra történő hivatkozások lesznek. Egy ilyen attribútum tehát a reláció elsődleges kulcsaként és sorok objektumazonosítójaként is szolgál.

8.29. példa: Írjuk át a 8.13. ábrán található deklarációkat úgy, hogy a `FilmSzínész` és a `Film` relációk tartalmazzanak egy-egy objektumazonosító attribútumot, neveze-

tesen a `színész_azon`, illetve a `film_azon` attribútumokat. A módosított sémát a 8.15. ábra mutatja. Az alábbi változtatásokat hajtottuk végre:

1. A `FilmTípus` sortípust kibővítettük a `film_azon` attribútummal.

2. A `SzínészTípus` sortípust kibővítettük a `színész_azon` attribútummal.

3. A `Film` tábla definícióját kiegészítettük egy záradékkal, miszerint ebben táblában a `film_azon` értékét a rendszer állítja elő.

4. A `FilmSzínész` tábla definícióját kiegészítettük egy záradékkal, miszerint ebben táblában a `színész_azon` értékét a rendszer állítja elő.

```
CREATE ROW TYPE FilmTípus(
   film_azon  REF(FilmTípus),
   cím           CHAR(30),
   év            INTEGER,
   színes        BIT(1)
);

CREATE ROW TYPE CímTípus(
   utca          CHAR(50),
   város         CHAR(20)
);

CREATE ROW TYPE SzínészTípus(
   színész_azon REF(SzínészTípus),
   név           CHAR(30),
   lakcím        CímTípus
);

CREATE ROW TYPE SzerepelBenneTípus(
   színész       REF(SzínészTípus),
   film          REF(FilmTípus)
);

CREATE TABLE Film OF TYPE FilmTípus
VALUES FOR film_azon ARE SYSTEM GENERATED;

CREATE TABLE FilmSzínész OF TYPE SzínészTípus
VALUES FOR színész_azon ARE SYSTEM GENERATED;

CREATE TABLE SzerepelBenne OF TYPE SzerepelBenneTípus
SCOPE FOR színész IS FilmSzínész,
SCOPE FOR film IS Film;
```

8.15. ábra. *Relációk kibővítése objektumazonosítókkal*

5. Felhasználtuk a 8.28. példában szereplő `SCOPE` deklarációkat.

Ezek után a 8.27. példában tárgyalt lekérdezést, ahol Mel Gibson filmjeit kerestük, megfogalmazhatjuk a relációs szemlélethez közelebb álló formában. A `WHERE` záradékban vizsgálhatjuk az egyenlőséget a `SzerepelBenne` reláció hivatkozásai és a `FilmSzínész`, illetve `Film` relációk objektumazonosító attribútumai között, amely attribútumok a saját sorukra történő (ön)hivatkozások. Az így megfogalmazott lekérdezés a következő:

```
SELECT Film.cím
FROM SzerepelBenne, FilmSzínész, Film
WHERE SzerepelBenne.színész =
      FilmSzínész.színész_azon AND
      SzerepelBenne.film = Film.film_azon AND
      FilmSzínész.név = 'Mel Gibson';
```

A `FROM` záradék azt fogalmazza meg, hogy a `SzerepelBenne`, `FilmSzínész` és `Film` relációk soraiból képzett hármasokat tekintjük. A `WHERE` záradék első feltétele azt fejezi ki, hogy a `SzerepelBenne` táblából vett sor `színész` komponense egy hivatkozás a `FilmSzínész` táblából származó sorra. Hasonlóképpen a `WHERE` záradék második feltétele az, hogy a `SzerepelBenne` sor `film` komponense egy hivatkozás a `Film` táblából származó sorra. Összhatásában ez a két feltétel azt követeli meg, hogy a `FilmSzínész` és a `Film` táblából vett sorok azt a színészt, illetve filmet reprezentálják, amelyeket a `SzerepelBenne` sor összepárosít.

A `WHERE` záradék harmadik feltétele azt írja elő, hogy a kérdéses színész Mel Gibson legyen, a `SELECT` záradék pedig a szóban forgó film címét állítja elő. Vegyük észre, hogy a lekérdezést most is megfogalmazhatnánk a hivatkozások követésének módszerével, mint ahogy azt a 8.27. példában tettük. Valójában ennél a lekérdezésnél az a módszer sokkal egyszerűbb. A fenti megfogalmazással az objektumazonosító attribútumok használatában rejlő lehetőségeket akartuk szemléltetni. □

8.5.8. Feladatok

8.5.1. feladat: Deklaráljuk az alábbi típusoknak megfelelő sortípusokat:

a) `NévTípus`, amely tartalmazza a családi nevet, utónevet és valamilyen megszólítást.

* b) `SzemélyTípus`, amely tartalmazza a személy nevét, valamint a hivatkozásokat azokra a személyekre, akik az ő apja és anyja. A deklarációban használjuk az a) részben definiált sortípust.

c) `HázasságTípus`, amely tartalmazza a házasság dátumát, valamint a férjre és a feleségre történő hivatkozásokat.

8.5.2. feladat: Tervezzük át a 4.1.1. feladatban szereplő termékek adatbázissémáját. Használjunk sortípusokat és hivatkozás típusú attribútumokat ott, ahol helyénvaló. A `PC`, `Laptop` és `Nyomtató` relációkban a `modell` attribútum hivatkozás legyen a modellnek megfelelő `Termék` sorra.

8.5.3. feladat: Használjuk a 8.5.2. feladat megoldásaként kapott sémát, és fogalmazzuk meg az alábbi lekérdezéseket. Ahol lehet, hivatkozásokat használjunk.

a) Keressük meg azoknak a PC-knek a gyártóit, amelyek több mint 2 gigabájt kapacitású merevlemezzel rendelkeznek.

b) Keressük meg a lézernyomtatók gyártóit.

! c) Készítsünk táblázatot, amely minden laptop modellszámhoz megadja annak a laptopnak a modellszámát, amelyiket ugyanaz a gyártó gyárt, és az ilyenek közül a legnagyobb sebességű.

! **8.5.4. feladat:** A 8.5.2. feladatban azt indítványoztuk, hogy a modellszámok helyett a `PC`, `Laptop` és `Nyomtató` táblákban a megfelelő `Termék` sorokra történő hivatkozások szerepeljenek. Megtehetjük-e azt, hogy a `Termék` tábla `modell` attribútumából olyan hivatkozást csinálunk, amely ennek a relációnak az adott típusú terméket reprezentáló sorára mutat? Ha igen, akkor miért, ha nem, akkor miért nem?

* **8.5.5. feladat:** Tervezzük át a 4.1.3. feladatban szereplő csatahajók adatbázissémáját. Használjunk sortípusokat és hivatkozás típusú attribútumokat ott, ahol helyénvaló. A 8.1.2. feladatban használt sémát támpontként tekinthetjük arra vonatkozóan, hogy hivatkozás típusú attribútumok hol lehetnek hasznosak. Keressük meg a sok-egy kapcsolatokat, és próbáljuk meg azokat hivatkozás típusú attribútumokkal reprezentálni.

8.5.6. feladat: Használjuk a 8.5.5. feladat megoldásaként kapott sémát, és fogalmazzuk meg az alábbi lekérdezéseket. Ahol lehet, próbáljuk meg a hivatkozásokat használni és elkerülni az összekapcsolásokat (vagyis egynél több sorváltozó, illetve alkérdések használatát a `FROM` záradékban).

* a) Keressük meg a 35 000 tonnánál nagyobb vízkiszorítású hajókat.

b) Keressük meg azokat a csatákat, amelyekben legalább egy hajó elsüllyedt.

! c) Keressük meg azokat a hajóosztályokat, amelyekbe tartoztak 1930 után felavatott hajók.

!! d) Keressük meg azokat a csatákat, amelyekben legalább egy amerikai hajó megsérült.

8.6. Absztrakt adattípusok az SQL3-ban

Az SQL3 sortípusai és az azokra történő hivatkozások szolgáltatják annak a funkcionalitásnak a nagy részét, amelyet az OQL objektumai nyújtanak. Ezenkívül lehetővé teszik az „objektumok" változtatását úgy, hogy az SQL erre alkalmas műveleteit használjuk, mint például a beszúrást és a törlést. Ezzel ellentétben az OQL úgy tekinti az ilyen módosításokat, hogy azokat a körülvevő objektumorientált programozási nyelv végzi, például a C++.

Nem szolgáltatják azonban a sortípusok a *betokozás (encapsulation)* fogalmát, ami az objektumorientált nyelvekben rendelkezésre áll. Elevenítsük fel az 1.3.1. alfejezetben mondottakat, miszerint egy osztályt azért „tokozunk be", hogy biztosítsuk azt, hogy annak objektumai csak az osztállyal együtt definiált műveletek egy rögzített halmazán keresztül változzanak. A betokozás azt célozza meg, hogy meggátoljuk az olyan programozási hibákat, amelyek akkor keletkezhetnek, amikor az adatokat oly módokon használjuk, amelyek az adatbázis tervezőjének szándékán vagy előrelátásán kívül esnek.

A sortípusok nem betokozottak, vagyis egy sortípus sorainak manipulálásához az SQL3-ban kifejezhető bármilyen műveletet felhasználhatunk. Az ODL interfészei (osztályai) sem teljes mértékben betokozottak, ugyanis az objektumok komponenseinek elérésére használhatjuk az OQL lekérdezéseit. Másrészt az is igaz, hogy az objektumok belső szerkezetének lekérdezése kevesebb veszéllyel jár, mint az objektumok nem betervezett módon történő változtatása. Az OQL-ben nem módosíthatjuk az objektumokat, kivéve amikor ezt metódusokon keresztül tesszük (lásd 8.1.2. alfejezet). A metódusokat feltételezhetően a körülvevő hagyományos nyelvben, például C++-ban, implementáljuk, és csak a szóban forgó osztály objektumaira alkalmazhatjuk.

A sortípusok mellett az SQL3-ban létezik egy olyan „osztály" fogalom, amely támogatja a betokozást. Ezt *absztrakt adattípusnak (abstract data type* – rövidítve *ADT)* nevezzük. Egy absztrakt adattípus objektumait nem sorokként, hanem sorok komponenseiként használjuk. Mindamellett ezek az objektumok tipikusan sorszerkezetűek, mint ahogy általában az ODL objektumai is komponensekből álló struktúrával rendelkeznek.

8.6.1. Absztrakt adattípusok definiálása

Egy absztrakt adattípus definíciójának általános szintaxisát a 8.16. ábra mutatja. Az 1) sor az ADT létrehozását jelző kulcsszavakat és az ADT nevének megadását tartalmazza. A 2) sor az attribútumok és típusuk felsorolását reprezentálja.

```
1)  CREATE TYPE <típus neve> (
2)          attribútumok és típusuk listája
3)          opcionálisan a típushoz tartozó = és < függvények deklarálása
4)          a típushoz tartozó függvények (metódusok) deklarálása
5)  );
```

8.16. ábra. *Absztrakt adattípusok definiálása*

A 8.16. ábra 3) sora azt reprezentálja, hogy opcionálisan deklarálhatjuk az = és a < összehasonlító operátorokat. Az egyenlőség függvény deklarációja az alábbi alakú:

EQUALS <az egyenlőséget definiáló függvény neve>

A < függvényt hasonlóan deklarálhatjuk, azzal a különbséggel, hogy az EQUALS kulcsszót a LESS THAN kulcsszavakkal kell helyettesíteni.[4] A további négy összehasonlító operátor e kettő segítségével kifejezhető, így azokat nem kell külön definiálni. A ≤ például azzal egyenértékű, hogy „= vagy <", a > pedig azzal, hogy „nem ≤". Ha az = és a < operátorokat definiáljuk, akkor a létrehozott absztrakt adattípusnak az értékeit a WHERE záradékokban ugyanúgy összehasonlíthatjuk, mint a hagyományos SQL típusokba tartozó értékeket, például egész számokat vagy karakterláncokat.

A 4) sor az ADT-hez tartozó további függvények (vagyis metódusok) deklarálását sugallja. Az SQL3 minden ADT-t ellát bizonyos „beépített" függvényekkel is, amelyeket nem kell deklarálni vagy definiálni. A következő függvények tartoznak ide:

1. Egy *konstruktorfüggvény,* amely a típus egy új objektumával tér vissza. Ezen objektum mindegyik attribútumának kezdeti értéke NULL. Ha T az ADT neve, akkor $T()$ a konstruktorfüggvény.
2. Egy ún. *hozzáférési* (observer) függvény minden egyes attribútum számára, amely ennek az attribútumnak az értékét adja vissza. Ha A egy attribútum neve, és X egy változó, amelynek értéke az ADT egy objektuma, akkor $A(X)$ az X objektum A attribútumának értéke. Ugyanennek a kifejezésére a hagyományosabb $X.A$ jelölést is használhatjuk.
3. Egy ún. *változtató* (mutator) függvény minden egyes attribútum számára, amely ennek az attribútumnak az értékét egy új értékre állítja be. Ezeket rendszerint értékadó utasítások bal oldalán használjuk a 8.6.2. részben tárgyalásra kerülő módon. Megjegyezzük, hogy a betokozás megvalósítása érdekében ezeket a függvényeket a nyilvános használat elől el kell zárni. Ezt az SQL3 úgy oldja meg, hogy a függvényekre vonatkozóan EXECUTE jogosultságot követel meg. Ez a jogosultság ugyanúgy engedélyezhető vagy visszavonható, mint az SQL2-nek a 7.4.1. részben tárgyalt hat jogosultsága.

A további függvényeket vagy a CREATE TYPE utasításon belül, vagy azon kívül definiálhatjuk. Ha egy ilyet kívül definiálunk, akkor az csak a belül definiált függvényeket használhatja, beleértve a fentebb felsorolt beépített függvényeket is.

8.30. példa: A 8.22. példában a címek modellezésére egy sortípust definiáltunk, amely az utca és a város komponenseket tartalmazta. Ugyanerre a célra egy azonos

[4] Elképzelhető, hogy az SQL3 szabvány fejlődése során ezek a függvények elveszítik speciális jellegüket, és ugyanúgy kell majd őket definiálni, mint a típus bármelyik másik függvényét.


```
1) CREATE TYPE CímADT (
2)    utca        CHAR(50),
3)    város       CHAR(20),
4)    EQUALS címEq,
5)    LESS THAN címLT
      itt további függvények deklarálhatók
   );
```

8.17. ábra. *Egy címeket modellező ADT definíciója*

szerkezetű absztrakt adattípust is létrehozhatnánk, amivel elérnénk a címek betokozását. Ez azt jelenti, hogy az utca és a város komponensekhez nem tudnánk hozzáférni, hacsak nem tennénk a hozzáférési és a változtató függvényeket nyilvánossá.

A 8.17. ábrán látható a CímADT absztrakt adattípus definíciója, eltekintve az ADT-hez kapcsolt függvények vagy metódusok tényleges definícióitól. Az 1) sor az ADT nevet adja meg.

A 2) és 3) sorok definiálják a reprezentációt, ami egy sor, amely az utca és a város komponensekből áll. Ezek a komponensek ugyanolyan típusúak, mint a 8.22. példában, vagyis 50, illetve 20 hosszú karakterláncok.

A 4) és 5) sorok azt közlik, hogy a CímADT számára az egyenlőség operátort a címEq nevű függvény, a < összehasonlító operátort pedig a címLT nevű függvény definiálja. Azt még nem tudjuk, hogy ezek a függvények mit csinálnak, hiszen azok definícióiról még nem gondoskodtunk. Az általunk választott definíciókat a 8.32. példában fogjuk megadni. A címeket lexikografikusan fogjuk rendezni, először város, majd utca szerint. □

A következő példában azt mutatjuk be, hogy az absztrakt adattípusok segítségével hogyan lehet az SQL programokba olyan típusokat bevezetni, amelyekkel nem számoltak az arra vonatkozó klasszikus feltételezések, hogy az adatbázis-kezelő rendszerek mire is lennének használatosak. Ma már nagyon nagy objektumok – mint például képek, hanganyagok vagy filmek – adatbázisban való tárolása is megvalósítható. Az ilyen objektumokon végzett műveletek azonban nem olyanok, mint a „szabványos" SQL műveletek, mint például az összehasonlítások, összesítések stb. Ezeket az objektumokat inkább meg kell tudni „mutatni" valahogy, amihez gyakran egy összetett dekódoló algoritmus szükséges. A jövőben még az is lehetségessé válhat, hogy képek között bonyolult összehasonlításokat végezzünk, vagy hogy képeket jellemző vonásokat ismerjünk fel.

8.31. példa: Tegyük fel, hogy van egy Mpeg absztrakt adattípusunk, amely egy filmnek az MPEG kódolását reprezentálja (az MPEG a videók tömörítésének egy szabványos formája; az ezzel kapcsolatos részleteket lásd a bekeretezett részben). Mivel egy MPEG kódolás formailag egy karakterláncot ad, azt gondolhatnánk, hogy a reprezentálásra a VARCHAR típus megfelelő lehetne. Az MPEG-kódolt videó hossza azonban gyakran olyan nagy (gigabájtokban mérhető), hogy a videók karakterláncos kezelése nem lehetséges.

```
1) CREATE TYPE Mpeg (
2)    videó         BLOB,
3)    hossz         INTEGER,
4)    szerzőiJog VARCHAR(255),
5)    EQUALS DEFAULT,
6)    LESS THAN NONE
      itt függvények definíciói jönnek
   );
```

8.18. ábra. *Egy MPEG ADT definíciója*

A videók és más nagyon nagy adattételek kezelhetősége érdekében a modern adatbázisrendszerek támogatnak egy speciális adattípust, amelyet BLOB-nak nevezünk. Egy *BLOB* (Binary Large OBject – Bináris Nagy Objektum) egy olyan bitsorozat, amely nagyon hosszú lehet, szükség esetén akár több gigabájt is. Ebben a példában azzal a feltevéssel élünk, hogy a BLOB típus az adatbázisrendszer egy beépített típusa. Az `Mpeg` ADT egy megfelelő definícióját a 8.18. ábrán láthatjuk.

A 2) sor a `videó` attribútumot BLOB típusúnak definiálja. Ez az attribútum tartalmazza az MPEG-kódolt videót. A 2) és 3) sorok két további, „hagyományos" attribútumot deklarálnak: a videó hosszát (lejátszási idejét) és a szerzői jogra vonatkozó figyelmeztetést. Az 5) sor azt mondja, hogy az `Mpeg` típusú értékek egyenlőségének vizsgálata az alapértelmezés szerinti módon történjen, vagyis azonosság alapján. Ez azt jelenti, hogy két `Mpeg` objektum akkor és csak akkor egyenlő, ha azok az egymásnak megfelelő attribútumokban teljesen (bitről bitre) megegyezőek. Az egyenlőséget egy olyan összetettebb módon is definiálhatnánk, ami azt az elképzelést tükrözné, hogy két `Mpeg` érték akkor „egyenlő", ha a dekódolás után egy elfogadható felbontású képernyőn megjelenítve ugyanúgy néznek ki, de ezzel a kérdéssel itt nem foglalkozunk tovább. A 6) sor azt jelenti, hogy az `Mpeg` értékek között a < összehasonlítást nem definiáljuk, azaz ha A és B `Mpeg` típusú értékek, akkor nem írhatjuk azt, hogy $A < B$. □

8.6.2. Absztrakt adattípusok metódusainak definiálása

Egy absztrakt adattípus attribútumainak felsorolása után tetszőleges számú függvényt deklarálhatunk. Egy függvénydeklaráció szintaxisa a következő:

```
FUNCTION <név> ( <argumentumok> ) RETURNS <típus> ;
```

Minden argumentum egy változó nevének és típusának megadásából áll. Az argumentumokat vesszővel választjuk el egymástól.

A függvényeknek két fajtáját különböztetjük meg: belső és külső függvényeket. A külső függvények a befogadó nyelvben megírt függvények, és az ADT definíciójában az ilyeneknek csak a szignatúrája jelenik meg. A külső függvényeket a 8.6.3. részben fogjuk tárgyalni. A belső függvények egy kiterjesztett SQL-ben megírt függvények. A következő felsorolás néhány olyan lehetőséget sorol fel, amelyek mind az SQL2-nek, mind az SQL3 lekérdezőnyelvi részének kibővítéséhez hozzátartoznak:

Video MPEG kódolása

Mivel a video tárolása nagyon sok helyet igényel, azt rendszerint egy vagy több szabványos tömörítő eljárás segítségével kódolják. A legelterjedtebb ilyen eljárás az MPEG, amely azt a tényt használja ki, hogy egy mozgóképnek egy filmkockája nagyon hasonló az őt megelőző filmkockához. Ebből kifolyólag egy filmkocka egy területét az előző filmkocka egy területére irányuló mutatóval lehet reprezentálni. A megelőző filmkocka területe vagy ugyanazon a pozíción helyezkedik el (ha egy mozdulatlan háttér része), vagy egy más pozíción (ha egy mozgó objektum része).

Noha az MPEG a videót sokkal jobban tömöríti, mint a szabványos tömörítő algoritmusok a szövegeket, egy MPEG-tömörített egyórás videó még így is körülbelül egy gigabájt helyet igényel. Azonkívül, mivel a módszer az egymásnak megfeleltetett régiók közötti apró különbségeket tolerálja, a videó minősége sok esetben romlik egy keveset. A megjelenítéshez a tömörített videót ki kell bontani, ami szintén egy meglehetősen bonyolult eljárás. Mindezen problémák ellenére az MPEG egy jó kompromisszumot képvisel a képminőség, a helyigény és a szükséges számítási teljesítmény tekintetében.

- A `:=` használható mint értékadó operátor.
- Egy függvénynek egy lokális változóját úgy deklarálhatjuk, hogy megadjuk a változó nevét és típusát. A változó nevét egy kettőspont előzi meg.
- A pont operátor használható egy struktúra komponenseinek elérésére.
- Logikai értékeket úgy fejezhetünk ki, mint a `WHERE` záradékokban.
- Egy függvény törzsét alkotó utasítások sorozatát `BEGIN` és `END` zárja közre.

8.32. példa: Folytassuk most a 8.30. példát, ahol egy absztrakt adattípust definiáltunk a címek modellezésére. A 8.19. ábrán néhány függvény szerepel, amelyeket belevehetünk a 8.17. ábrán látható, a típust létrehozó utasításba.

Az 1)–6) sorok a `CímADT` absztrakt adattípus egy konstruktorfüggvényét definiálják. Emlékezzünk vissza, hogy az SQL3 egy argumentumok nélküli beépített konstruktort biztosít, amelynek neve `CímADT` (magának az ADT-nek a neve). Mi viszont egy olyan konstruktort szeretnénk, amely argumentumként az `utca` és a `város` attribútumok számára vesz egy-egy értéket. Ezt a függvényt bárhogyan elnevezhetjük, és az is megengedett és helyes, ha az osztály nevét használjuk.

Az 1) sorban az új konstruktorfüggvény deklarációját látjuk. A függvénynek két argumentuma van, s és c, amelyek az utcát és a várost reprezentálják. Ezek 50, illetve 20 hosszú karakterlánc típusú argumentumok. A függvény egy `CímADT` típusú értékkel tér vissza. A 2) sor az a lokális változót deklarálja, amelynek típusa `CímADT`.

Bináris nagy objektumok

A felhasználó számára a BLOB értékek nagy bitsorozatoknak tűnhetnek, a kulisszák mögött azonban azok implementációja sokkal bonyolultabb, mint mondjuk a maximum 255 hosszú karakterláncok implementációja. Nincs értelme például annak, hogy az óriási bitsorozatokat sorok részeiként tároljuk, így azokat külön kell tárolni, rendszerint a környező fájlrendszer segítségével.

Vagy például a 7.3.4. alfejezetben tárgyalt kliens-szerver modell azt feltételezi, hogy az értékek és a sorok kisméretűek, és egy lekérdezésre adott válaszként a szerver a klienshez teljes sorokat juttat el. Nincs viszont értelme annak, hogy a kliensnek rögtön egy egész BLOB-ot átadjunk. Ha például a kliens a szervertől egy videoklippet kér, akkor elég, ha a szerver egyszerre annak csak egy kis darabját adja át, mondjuk egy néhány másodpercnyi részt. Így a kliens elkezdheti lejátszani a filmet anélkül, hogy a több gigabájtnyi videót lokálisan kellene tárolnia, vagy hogy meg kellene várnia, hogy a teljes videót megkapja.

A függvény törzsét a 3)–6) sorok alkotják. A 3) sorban a `CímADT()` beépített konstruktor használatával létrehozunk egy új objektumot, és azt a `:a` változónak értékül adjuk. Láthatjuk, hogy a beépített konstruktort nem téveszthetjük össze azzal, amelyiket éppen írunk, mert az argumentumokat illetően a két függvény különbözik. Vagyis a 3) sort nem lehet egy rekurzív hívásként félreérteni. A 4) sor az első argumentumot bemásolja az *a* `utca` komponensébe, az 5) sor pedig a második argumentumot másolja be az *a* `város` komponensébe. Végül a 6) sor visszatér a létrehozott *a* értékkel.

A 7) és 8) sorok a `CímADT` típushoz rendelt egyenlőség függvényt definiálják. A 8.17. ábra 4) sorában erről a függvényről azt mondtuk, hogy a neve `címEq`, így ezt a nevet kell használnunk. A függvény egyszerű, akkor és csak akkor tér vissza az `IGAZ` értékkel, ha a két vizsgált értéknek mind az utca, mind a város komponensei megegyeznek. Valójában ez a függvény az egyenlőség alapértelmezés szerinti definícióját – ami az értékek azonosságát jelenti – valósítja meg, így elég lett volna ezt a tényt deklarálni, úgy ahogy a 8.31. példában tettük.

A 9) és 10) sorok definiálják a < függvényt, amelynek neve `címLT`. Definícióul azt választottuk, hogy az első cím megelőzi a másodikat, ha az első városa lexikografikusan kisebb, mint a második városa (az ábécé szerint megelőzi). Ha a két város ugyanaz, akkor az utcaneveket hasonlítjuk ugyanígy.

A 11)–14) sorok egy `teljesCím` nevű függvényt definiálnak, amely vesz egy `CímADT` típusú objektumot, és visszaadja a teljes címet, vagyis az utcát (pontosabban az utcacímet, ami a házszámot is tartalmazza), a várost és az irányítószámot. A 12) sor az irányítószámok ideiglenes tárolására szolgáló `:z` lokális változót deklarálja. A 13) sorban az `irszámKeres` függvényt hívjuk. Ez egy olyan külső függvény, amelynek egy utcacímet és egy várost reprezentáló karakterlánc típusú argumentuma van. A külső függvények deklarálásának módját a 8.6.3. részben tárgyaljuk.


```
1)  FUNCTION CímADT(:s CHAR(50), :c CHAR(20))
        RETURNS CímADT;
2)    :a CímADT;
      BEGIN
3)      :a := CímADT();
4)      :a.utca := :s;
5)      :a.város := :c;
6)      RETURN :a;
      END;

7)  FUNCTION címEq(:a1 CímADT, :a2 CímADT)
        RETURNS BOOLEAN;
8)    RETURN (:a1.utca = :a2.utca AND
        :a1.város = :a2.város);

9)  FUNCTION címLT(:a1 CímADT, :a2 CímADT)
        RETURNS BOOLEAN;
10)   RETURN ((:a1.város < :a2.város) OR
        (:a1.város = :a2.város AND :a1.utca < :a2.utca));

11) FUNCTION teljesCím(:a CímADT) RETURNS CHAR(82);
12)   :z CHAR(10);
      BEGIN
13)     :z = irszámKeres(:a.utca, :a.város);
14)     RETURN (:a.utca ||' '|| :a.város ||' '|| :z);
      END;
```

8.19. ábra. *A CímADT típus néhány függvénye*

Az `irszámKeres` függvény valamilyen bonyolult eljárás alapján, ami lehet például egy másik adatbázisból való kikeresés vagy döntések egy jól kidolgozott sorozata, megadja a szóban forgó utcához és városhoz tartozó helyes irányítószámot. Az `irszámKeres` függvény megvalósításával itt most nem foglalkozunk. Végül a 14) sorban összefűzzük a `:a` objektum `utca` és `város` komponenseit, valamint a `:z` változóban tárolt irányítószámot. A cím e három komponensét szóközzel választjuk el egymástól. □

8.6.3. Külső függvények

Az absztrakt adattípusoknak olyan metódusai is lehetnek, amelyeket nem az SQL3-ban, hanem valamely befogadó nyelvben implementálunk. Ha egy ilyen függvényt használunk, akkor az ADT definíciójában csak a függvény szignatúráját adjuk meg, valamint azt, hogy a függvényt melyik nyelvben valósítottuk meg. Egy külső függvény deklarációja az alábbi alakú:

```
DECLARE EXTERNAL <függvény neve> <szignatúra>
LANGUAGE <nyelv neve>
```

8.33. példa: Hogy az `irszámKeres` külső függvényt a 8.32. példában használhassuk, azt a `CímADT` típus definíciójában deklarálnunk kell. A függvénynek két argumentuma van, amelyek 50, illetve 20 hosszú karakterlánc típusúak, és egy 10 hosszú karakterlánc a visszatérési értéke, így a helyes deklaráció a következő:

```
DECLARE EXTERNAL irszámKeres
CHAR(50) CHAR(20) RETURNS CHAR(10)
LANGUAGE C;
```

Az a tény, hogy a C-t adtuk meg mint az implementálás nyelvét, azt jelenti, hogy az `irszámKeres` függvény számára az argumentumok a C programoknak megfelelően adódnak át. □

8.6.4. Feladatok

* **8.6.1. feladat:** Definiáljuk a „PC" absztrakt adattípust, amelynek objektumai személyi számítógépeket reprezentálnak. A processzor sebességét, a memóriát (RAM), a merevlemez méretét, a CD sebességét és az árat tartalmazza a típus.

8.6.2. feladat: A belső függvényeket támogató SQL kiterjesztést használva, írjuk meg a következő függvényeket a 8.6.1. feladatban szereplő ADT-hez:

* a) Egy `újPC` nevű konstruktorfüggvényt, amely a `PC` ADT öt attribútumához vesz értékeket, és egy új `PC` típusú objektummal tér vissza. Emlékezzünk, hogy a függvény definiálásához a `PC()` beépített konstruktort lehet (kell) használni.

* b) Egy `érték` nevű függvényt, amelynek argumentuma egy `PC` objektum, és a PC egy „értékelését" adja vissza eredményül, ami egy valós szám, amely azt mondja meg, hogy mennyire „jó" a PC. A PC „értékét" úgy számítjuk ki, hogy összeadjuk a processzor sebességét, a (megabájtokban kifejezett) memória 5-szörösét, a merevlemez (gigabájtokban kifejezett) méretének 50-szeresét és a CD sebességének 10-szeresét.

c) Egy `jobb` nevű függvényt, amely argumentumként vesz egy `PC` objektumot, és egy olyan másik `PC` objektummal tér vissza, amely kétszer akkora processzorsebességgel, memóriával, merevlemezzel és CD-sebességgel rendelkezik, de ugyanannyiba kerül. Az (a) pontban definiált konstruktort használhatjuk.

d) Egy `egyenlőPC` nevű függvényt, amely a `PC` objektumok közötti egyenlőség definiálására szolgál. Ez a fügvény két PC-t akkor tekint „egyenlőnek", ha ugyanolyan sebességű processzorral és ugyanolyan kapacitású merevlemezzel rendelkeznek, függetlenül a további komponensek értékeitől.

e) Egy `kisebbPC` nevű függvényt, amely a `PC` ADT-hez a kisebb (<) függvényt szolgáltatja. Ez a függvény akkor tekint egy p_1 PC-t egy p_2 PC-nél kisebbnek

($p_1 < p_2$), ha a p_1 „értéke" kisebb, mint a p_2 „értéke", ahol „érték" alatt a b) pontban megfogalmazott „érték" fogalmat értjük. A b) pontban definiált `érték` függvényt használhatjuk.

8.6.3. feladat: A hajók számára hozzunk létre egy `Hajó` absztrakt adattípust, amely magában foglalja a hajó nevét, a felavatás dátumát, az ágyúk számát, az ágyúk kaliberét, a vízkiszorítást, egy MPEG-kódolt videót, amely a hajót egy akció közben mutatja, és egy postscript dokumentumot, amely a hajó történetét tartalmazza.

8.6.4. feladat: Deklaráljuk és definiáljuk a következő függvényeket a 8.6.3. feladatban szereplő `Hajó` ADT-hez:

a) Egy `tűzerő` nevű fügvényt, amelynek argumentuma egy `Hajó` objektum, és a hajó tűzerejét adja eredményül, ami az ágyúk számának és a kaliber köbének a szorzata.

b) Egy `videóLejátszás` nevű függvényt, amelynek argumentuma egy `Hajó` objektum, és amely lejátssza a hajóhoz tartozó videót. Ehhez egy külsőként definiált függvényt használ (amit deklarálni kell). `mpegLejátszás` a külső függvény neve, és egy MPEG fájl lejátszására szolgál.

c) Egy `újHajó` nevű konstruktorfüggvényt, amely argumentumként vesz egy értéket a név komponenshez (a többi komponenshez nem), és egy ilyen nevű új `Hajó` objektummal tér vissza.

d) Egy `egyenlőHajó` nevű függvényt, amely a `Hajó` objektumok közötti egyenlőség definiálására szolgál. Ez a függvény két hajót akkor tekint „egyenlőnek", ha ugyanolyan névvel és felavatási dátummal rendelkeznek, függetlenül a további komponensek értékeitől.

e) Egy `kisebbHajó` nevű függvényt, amely a `Hajó` ADT-hez a kisebb (<) függvényt szolgáltatja. Ez a függvény akkor tekint egy h_1 hajót egy h_2 hajónál kisebbnek ($h_1 < h_2$), ha a h_1 neve betűrendben megelőzi a p_2 nevét, illetve ha a nevek megegyeznek, akkor a h_1-et előbb avatták fel, mint a h_2-t.

8.7. Az ODL/OQL és az SQL3 összehasonlítása

Miután az objektumorientált adatbázis-kezelés két jelentős szabványába – az ODL/OQL-be és az SQL3-ba – némi betekintést nyertünk, itt az ideje, hogy azt is lássuk, hogy milyen szempontokból különböznek ezek a megközelítések. Az is igaz ugyanakkor, hogy a hasonlóságok számottevőbbek, mint a különbségek. Annak ellenére, hogy a két javaslat erősen eltérő modellekből – objektumorientált programozási

nyelvekből, illetve relációs adatbázisnyelvekből – indul ki, sok mindent eredményesen magukévá tettek abból, ami a másik modelljében alapvető fontosságú.

Ebben az alfejezetben a két megközelítés közötti leglényegesebb különbségeket vesszük sorra. Ezzel egyidejűleg arra is rámutatunk, hogy az SQL3 sortípusai és absztrakt adattípusai miben különböznek egymástól és az ODL interfészeitől (osztályaitól). Így tehát valójában az objektumorientáltság három különböző megközelítését hasonlítjuk össze: az ODL/OQL-t, az SQL3 sortípusait és az SQL3 absztrakt adattípusait.

1. *Programozási környezet.* Az OQL azzal a feltételezéssel él, hogy utasításait egy olyan programozási nyelvbe ágyazzuk be, amelyik ugyanazzal a programozási és adatmodellel dolgozik. Ez feltehetően egy objektumorientált nyelv, például C++, Smalltalk vagy Java. Másrészt, az SQL3 azt feltételezi, hogy az ő objektumai nem objektumai a körülvevő befogadó nyelvnek. Mint minden SQL szabványban és implementációban, itt is egy meghatározott interfész teszi lehetővé a tárolt SQL adatok és a befogadó nyelvi változók közötti értékátadásokat. A külső függvények használata az SQL3 absztrakt adattípusaiban egy további kommunikációs mechanizmust jelent, ami kiegészíti a 7.1. alfejezetben bemutatott szokásos interfészt az SQL és a befogadó nyelv között.

2. *A relációk szerepe.* Az SQL3 adatszemléletében a relációk megtartották központi szerepüket. A sortípusok igazából relációkat írnak le, az absztrakt adattípusok pedig az attribútumok számára definiálnak új típusokat. Másrészt, az OQL-ben az objektumok vagy struktúrák halmazai és multihalmazai a select-from-where utasításokban betöltött szerepük miatt alapvető fontosságúak. Az ODL/OQL struktúrákból alkotott kollekciói nagyon közel állnak az SQL3 relációihoz.

3. *Betokozás.* A sortípusok nem betokozottak. A sortípus segítségével definiált relációk, sorok vagy azok komponensei az SQL által nyújtott lehetőségek keretein belül tetszés szerint lekérdezhetők és módosíthatók. Az SQL absztrakt adattípusai a szokásos értelemben betokozottak. A betokozás szempontjából az ODL osztályok nagyon közel állnak az SQL3 absztrakt adattípusaihoz.

4. *Osztályokhoz tartozó objektumkészletek.* Az OQL egy osztályhoz pontosan egy objektumkészlet fenntartását feltételezi. A hivatkozások (az OQL terminológia szerint kapcsolatok) mindig ennek az objektumkészletnek valamely elemére vagy elemeire vonatkoznak. Az SQL3-ban fenntarthatunk egy objektumkészletet egy sortípus számára, vagyis egy relációt, amely az összes olyan típusú létező sort tartalmazza, de nem vagyunk arra kötelezve, hogy így tegyünk. Ha egy sortípushoz nem tartozik ilyen objektumkészlet, akkor problémás lehet annak a relációnak a megtalálása, amelyik az egy adott hivatkozás által hivatkozott sort tartalmazza, mint ahogy ezt a hivatkozások hatásköre kapcsán a 8.5.6. részben tárgyaltuk.

5. *Objektumok változékonysága.* Egy objektum *változhatatlan* (immutable), ha a létrehozás után annak semmilyen része nem változhat. Az elemi típusú objektumok, mint például az egész számok vagy karakterláncok, változhatatlanok ebben az értelemben. Ha egy objektum komponensei változhatnak mialatt az objektum megőrzi az identitását, akkor az objektumot *változékonynak* (mutable) mondjuk. Az ODL osztályai és az SQL3 sortípusai változékony objektumok osztályait definiálják, bár az ODL/OQL azt feltételezi, hogy az objektumok módosítása nem az OQL-en keresztül, hanem a körülvevő programozási nyelven belül történik. Az SQL3 absztrakt adattípusai nem egészen változhatatlanok. Egy komponensre alkalmazott változtató függvény viszont egy új értéket eredményez, amely a régit lecserélheti, hasonlóan egy egész típusú attribútumra vonatkozó SQL `UPDATE` utasításhoz, ami egy új egész számot állít elő, amely az adott sorban lecserélheti a régi egész értéket.

6. *Objektumok azonosítása.* Mind az OQL, mind az SQL3 absztrakt adattípusai megfelelnek az objektumazonosítás szokásos értelmezésének, miszerint egy objektumazonosító egy rendszer által generált belső érték, amelyet a felhasználó nem tárolhat vagy manipulálhat. Az SQL3 sortípusai és hozzájuk szorosan kapcsolódó hivatkozások viszont nem követik ezt az alapelvet. A felhasználó egy relációban létesíthet egy olyan oszlopot, amely a sorok objektumazonosítóit tartalmazza magukban a sorokban, mintha azok közönséges értékek volnának. Következésképpen egy ilyen reláció sorainak objektumazonosítói a reláció kulcsaként szolgálhatnak. Annak ellenére, hogy a módosítások és a törlések lógó hivatkozásokat vezethetnek be az adatbázisba, van bizonyos értelme ennek a lehetőségnek. Ha ugyanis az objektumazonosítót nem engednénk meg mint attribútumot, akkor a relációknak tipikusan két kulcsa lenne: az objektumazonosító és az azonosítást pótló valamilyen érték, mint például a TAJ szám vagy valami egyéb.

8.8. Összefoglalás

- *Metódusok az OQL-ben:* A 2. fejezetben tárgyalt attribútumok és kapcsolatok mellett az ODL egy interfész specifikációjának részeként metódusok deklarálását is megengedi. Egy metódusnak csak a szignatúráját, azaz a bemenő és kimenő paramétereinek a típusát definiáljuk. Magukat a metódusokat a körülvevő programban definiáljuk, és az objektumorientált befogadó nyelvben írjuk meg.
- *Az OQL típusrendszere:* Az OQL-ben a típusokat osztálynevekből és atomi típusokból építjük fel. A típuskonstruktorok között szerepel a struktúrák definiálására szolgáló `Struct` típuskonstruktor, valamint a halmazok, multihalmazok, listák és tömbök definálására szolgáló kollekciótípusok. A típusrendszer tehát ugyanaz, mint az ODL-ben, azzal a különbséggel, hogy az OQL-ben nincs korlátozás a típuskonstruktorok beágyazási fokára nézve.
- *Select-From-Where utasítások az OQL-ben:* Az OQL egy olyan select-from-where kifejezést kínál, amely az SQL-éhez hasonlít. A `FROM` záradékban változókat dek-

larálhatunk, amelyek tetszőleges kollekciókat futnak be, beleértve osztályok objektumkészleteit (amelyek a relációknak felelnek meg) és olyan kollekciókat, amelyek objektumokban szereplő attribútumok értékei.

- *Az OQL egyszerű operátorai:* Az OQL tartalmazza a minden, a létezik, az `IN`, az egyesítés, a metszet, a különbség és az összesítéseket végző operátorokat, amelyek szellemükben az SQL megfelelő oprátoraihoz hasonlóak. Egy összesítés azonban mindig egy kollekcióra vonatkozik, nem pedig egy reláció valamelyik oszlopára.

- *Az OQL Group-By operátora:* Az SQL-hez hasonlóan az OQL egy `GROUP BY` záradékot is megenged a select-from-where utasításokban. Az OQL-ben azonban az egy csoportot alkotó objektumok kollekciója a `partition` nevű mezőn keresztül közvetlenül elérhető.

- *Hozzáférés az OQL kollekciók elemeihez:* Egy egyelemű kollekciónak az egyedüli elemét az `ELEMENT` operátor alkalmazásával kaphatjuk meg. Egy több elemből álló kollekció elemeihez úgy férhetünk hozzá, hogy egy `ORDER BY` záradékot tartalmazó select-from-where utasítást használva a kollekciót először egy listává alakítjuk, majd ennek a listának az egyes elemeit a körülvevő befogadó nyelvi programban egy ciklus segítségével megszerezzük.

- *Objektumok az SQL3-ban:* Az SQL3 az objektumoknak két fajtáját nyújtja: sortípusokat és absztrakt adattípusokat. A sortípusok sorok típusaként, az absztrakt adattípusok pedig komponensek típusaként szolgálnak.

- *Sortípusok és objektumazonosítók:* Minden sortípushoz kapcsolódik egy hivatkozási típus, és ennek a hivatkozási típusnak egy értéke egy sorhoz tartozó objektumazonosító. Az SQL3 megengedi, hogy egy attribútum típusa a saját relációjának a sortípusára vonatkozó hivatkozási típus legyen, és hogy ennek az attribútumnak az értéke egy adott sorban magának a sornak az objektumazonosítója legyen. Ezáltal az objektumazonosító a reláció kulcsaként is szolgál.

- *Absztrakt adattípusok az SQL3-ban:* Az SQL3-ban absztrakt adattípusokat deklarálhatunk a `CREATE TYPE` utasítás használatával. Egy absztrakt adattípus értékei egy vagy több komponensből álló rekordszerkezetek, és saját metódusokkal rendelkezhetnek.

- *Metódusok az SQL3-ban:* Egy absztrakt adattípushoz függvényeket (metódusokat) deklarálhatunk. Az ilyen függvényeket vagy megírjuk egy SQL-szerű programnyelvben, vagy úgy deklaráljuk mint a befogadó nyelvben megírt külső függvények.

8.9. Irodalomjegyzék

Az OQL-hez kapcsolódó hivatkozás ugyanaz, mint az ODL-hez tartozó, vagyis az [1]. Az SQL3-mal kapcsolatos anyagokhoz az 5. fejezet Irodalomjegyzékében leírt módon juthatunk hozzá. Emellett a sorobjektumokhoz a [3] szolgáltatja a forrást, a [2] pedig egy korai ismertetése az SQL3 absztrakt adattípusainak, amiből a szabvány már több irányba továbbfejlődött.

1. Cattell, R. G. G. (ed.), *The Object Database Standard: ODMG–93 Release 1.2*, Morgan-Kaufmann, San Francisco, 1996.

2. Melton, J., J. Bauer, and K. Kulkarni, „Object ADT's (with improvements for value ADT's)", ISO WG3 report X3H2–91–083, April, 1991.

3. Kulkarni, K., M. Carey, L. DeMichiel, N. Mattos, W. Hong, and M. Ubell, „Introducing reference types and cleaning up SQL3's object model", ISO WG3 report X3H2–95–456, Nov., 1995.

Szofi

IT CONSULTING

Informatikai képzések magyar és angol nyelven

- System Engineer (Rendszermérnök)
- System Programmer (Rendszerprogramozó)
- Programmer, Application Developer (Programozó)
- System Administrator (Rendszeradminisztrátor, rendszergazda)
- System Analyst and Designer (Információrendszer-szervező)
- Database Administrator (Adatbázis-adminisztrátor)
- Posztgraduális továbbképzések informatikusok számára
- Egyetemi oktatatás melletti kiegészítő képzések

Megszerezhető végzettségek:
CIT (Certified Information Technologist)* amerikai és magyar állami felsőfokú szakképesítések

ITCM (IT Career Management) rendszer a CIT fokozatot megszerző informatikusok részére

* A CIT a Szofi USA (New York) és a Szofi Algorithmic Research (Hungary) bejegyzett védjegye

Szofi Magyar–Amerikai Informatikai Oktató- és Továbbképző Központ

WWW.SZOFI.HU

Tárgymutató

., 457, 484
.., 480, 484
:=, 493
->, 457, 483, 484

3NF, 174, 175
4NF, 183

absztrakt adattípus, 478, 489–96, 498
adatbázis épsége, 414
Adatbázisséma, 46, 105
Adatdefiníciós nyelv, 48
Adatmanipulációs nyelv, 48
ADD, 313
Additivitás, 156
ágens, 430
aktív kapcsolat, 429
aktuális előfordulás, 109
aktuális engedélyazonosító, 434
alapértelmezett, 304, 313, 323, 351
Alkérdés, 287, 289, 292, 462
ALL, 289, 293. *Lásd* FOR ALL
alosztály, 78, 81
alosztályok, 132
általános megszorítás, 85
ALTER, 313
alvó kapcsolat, 429
anomáliák, 158
ANY, 289
Armstrong-axiómák, 155
Astrahan, M. M., 353
átlag, 297
Átnevezés, 207
atom, 215
  aritmetikai atom, 216
  relációs atom, 216

atomi típus, 57, 456
attribútum, 49, 50, 58, 62, 70, 105, 448, 457, 478, 489
attribútumhalmazok lezárása, 150
AVG, 465

bal oldali külső összekapcsolás, 337
Batini, C., 103
Bauer, J., 501
bázis, 154
BCNF, 161
beágyazott ciklus, 281
beágyazott SQL, 393
befogadó nyelv, 393, 394, 448, 455, 472, 498
BEGIN, 493
belső függvény, 492
beszúrás, 304
betokozás, 489, 498
biztonságossági feltétel, 219
BLOB, 492, 494
Boyce-Codd normálforma, 161
bővítés, 155

C++, 47, 48, 54, 57, 448, 455, 472, 498
Carey, M., 501
Cattell, R. G. G., 103, 501
Celko, J., 353
Ceri, S., 103
Chamberlin, D. D, 353
Chen, P. P., 103
CORBA, 47
COUNT, 298, 465
CREATE, 312, 314, 316, 319, 478, 489
CROSS JOIN, 333
csoportosítás, 464

Darwen, H., 353
Datalog, 215, 282, 340, 343
Date, C. J., 353
dátum, 274
deklarációs rész, 396
dekompozíció, 158, 159
DeMichiel, L., 501
Descartes-szorzat, 201
dinamikus SQL, 407
direkt SQL, 393
DISTINCT, 295, 298, 307, 460, 462, 470, 471
DROP, 313, 315, 317

E/K kapcsolatok átírása relációkká, 125
E/K modell, 123
egész, 311
egyedhalmaz, 62, 124
egyedhalmazok átírása relációkká, 124
egyed-kapcsolat diagram, 46, 62
[illegible], 56
[illegible], 117
egyértékűség megszorítás, 85, 89
egyesítés, 197, 284, 296, 464, 470
egyesítési szabály, 188
egyetlen sort eredményező lekérdezések, 398
egyszerűség, 73
ekvivalens kifejezés, 206
El Masri, R., 103
ELEMENT, 473
elkülönítési szintek, 421, 422
elkülönítési szintek, ismételhető olvasási, 422, 423
elkülönítési szintek, piszkos adatok olvasását megengedő, 421
elkülönítési szintek, sorba rendezhető, 421
elkülönítési szintek, véglegesített olvasási, 422
előfeldolgozó program, 393
első normálforma, 174
elsődleges kulcs, 88, 142
END, 493
engedély-azonosító, 430
engedélyezési diagram, 437
engedélyezési képesség, 436
engedélyező utasítás alakja, 436
EQUALS, 490, 492

értékadás, 472
értéktartomány, 107, 483
értelmezési tartomány, 315
értelmezésitartomány-megszorítás, 85, 91

Escape karakter, 273
EXEC SQL kulcsszavak, 395
EXISTS, 289, 465

fantom sorok, 423
fej, 217
felsorolás, 456
Felsorolás típus, 57
fetch utasítás, 400
Finkelstein, S. J., 353
FOR ALL, 464
FROM, 267, 269, 458–60
funkcionális függőség, 138
függőségi gráf, 343

GROUP BY, 299, 300, 466–69
gyenge egyedhalmaz, 93, 127

halmaz, 57, 112, 460, 462, 467, 470–71, 474, 498
hálós modell, 46, 98
hányados, 215
harmadik normálforma, 172, 175
háromértékű logika, 332
hatáskör, 485, 498
HAVING, 301, 302, 469
hierachikus modell, 46, 100
hivatkozás feloldása/követése, 483, 498
hivatkozásiépség-megszorítás, 85, 90
hivatkozási típus, 480–87
Hong, W., 501
hozzáférési függvény, 490

IDL, 47
igazságértékek táblázata, 332
IN, 289, 291
index, 316, 484
interfész, 50, 449, 451, 498
interfész típus, 57
inverzkapcsolat, 52
ISMERETLEN, 331, 333
ismétlődő sorok, 295

Java, 455, 498
jobb oldali külső összekapcsolás, 337
jogosultság, 431, 435, 437
jogosultság engedélyezése, 433
jogosultság létrehozása, 432
jogosultság visszavonása, 439
jogosultság visszavonása, következetes, 439
jogosultság, tulajdonosi, 433, 435
jogosultságok típusai, 431

kapcsolat, 49, 52, 58, 62, 125, 448, 457
kapcsolat és inverze, 121
kapcsolathalmaz, 63
kapcsolattípus, 56, 63
kapcsoló egyedhalmaz, 68
kapocs, 98
karakter, 270, 273
karakterkészlet, 427
karakterlánc, 271, 272, 274
katalógus, 424, 425, 427
katalógus, aktuális, 427, 430
kiválasztás, 200, 267, 270
kivétel, 449
klaszter, 424, 425
kliens, 427
kölcsönös rekurzió, 343
kollekciótípus, 58, 456, 475
komplementer szabály, 182
komponens, 106
konjunkció, 225
konstruktorfüggvény, 476, 490, 493
környezet, 424, 427
korrelált alkérdés, 292
kulcs, 84, 86, 88, 141
Kulkarni, K., 501
különbség, 197, 284, 464, 470
különbség szabály, 188
külső függvény, 492, 495, 498
külső összekapcsolás, 330, 333, 335
kvantor, 464

legkisebb fixpont, 233
lekérdezés, 197
  Datalog lekérdezés, 218
lekérdezések beágyazása, 397
LESS THAN, 490, 492
levezetési szabályok, 146
levezetett függőség, 154

lezárt, 150
LIKE, 272, 273
lineáris rekurzió, 340, 342
lista, 58, 112, 116, 462, 474
literál, 225
logikai rekordtípus, 98
lógó sor, 203, 336, 337

másodlagos kulcs, 88
másodnév, 269, 280, 342, 458
Mattos, N., 353, 501
MAX, 465
maximum, 298
megosztott elérésű változó, 395
megszorítás, 84
Melton, J., 353, 501
metódus, 49, 448, 449–51, 457, 492–95
metszet, 197, 284, 464, 470
metszet szabály, 188
MIN, 465
minimum, 298
minta, 273
Módosítási problémák, 159
módosítható nézettábla, 323
modul, 430
monoton operátor, 214
monotonitás, 347, 348
MPEG, 493
multihalmaz, 58, 112, 116, 296, 459, 460, 462, 470–71, 474, 498
Mumick, I. S., 353
munkafázis, 429

nagy tömegben feltöltés, 312
Navathe, S.B., 103
negáció
  rétegzett negáció, 243
negyedik normálformában, 183
Nem triviális, 149
nézet, 262
nézettábla, 319, 320, 322
Null, 57, 89, 135
nullérték, 135, 263, 304, 313, 330

objektum, 47
objektumazonosító, 48, 49, 120, 485, 499
objektumkészlet, 451, 459, 498

objektumorientált adatbázis-kezelő rendszer, 47
objektumorientált programozás, 447
objektumrelációs rendszer, 447
ODL, 46, 47, 448–51, 497–99
ODL sémák, 110
OQL, 447, 455–76, 497–99
ORDER BY, 275, 302, 462, 474
osztály, 48, 447

önhivatkozás, 485
öröklés, 79
öröklődés, 81
összekapcsolás, 202, 277. Lásd CROSS JOIN, természetes összekapcsolás, külső összekapcsolás, théta-összekapcsolás
  természetes összekapcsolás, 202
  théta-összekapcsolás, 203
összesítés, 297, 299, 464, 465–70
összesítő műveletek, 262
összetett attribútum, 112
összetett típus, 456
összevonhatósági szabály, 148

paraméter, 449
partition, 466–69
Pirahesh, H., 353
piszkos adatok, 419
predikátum, 215
  extenzionális predikátum, 221
  intenzionális predikátum, 221
Pszeudotranzitivitás, 156
PUBLIC jogosultság, 431

redundancia, 72, 159
reflexivitás, 155
rekordszerkezet, 112
rekurzió, 231, 340, 343
  bal oldali rekurzió, 238
  jobb oldali rekurzió, 238
  nem lineáris rekurzió, 238
reláció, 447, 451, 479, 498
relációelőfordulás, 108
relációk, 104
relációk *felbontása*, 159
relációs adatbázis-kezelő rendszer, 47
relációs adatbázisséma, 105
relációs adatmodell, 46
relációs algebra, 196, 267, 277, 284, 326
relációs modell, 104
relációséma, 105
rendezési szempontok, 404
részcél, 217
  negált részcél, 219
rétegzett negáció, 343

SELECT, 267, 269, 458–60
séma, 424, 425, 426
séma, aktuális, 426, 430
sémaelemek elnevezése, 428
SET, 308. *Lásd* ALL, DISTINCT
Simon, A., 353
Smalltalk, 47, 48, 448, 455, 498
sokágú kapcsolat, 64
sok-egy kapcsolat, 56
sok-sok kapcsolat, 56
sor, 478
sor(objektum), 478
sorbarendezhetőség, 411, 413, 414, 416, 418
sormutató, 398, 399, 404
sormutató opciók, 403
sormutató, csak olvasó, 405
sormutató, mozgatási irány, 406
sormutató, pozicionálás, 406
sormutatók érzéketlensége, 404
sorok, 106
sortípus, 478–87, 498
sorváltozó, 280, 283
SQL, 455
SQL ágens. *Lásd* ágens
SQL könyvtár, 394
SQL környezet. *Lásd* környezet
SQL2, 266, 323, 330, 333
SQL3, 266, 340, 342, 346, 447, 478–500, 497–99
SQLSTATE, 395, 402
struktúra, 51, 58, 456, 461, 467, 475, 498
SUM, 298, 465
szabály, 217
származtatott alosztály, 79
szerep, 65
szerver, 427
szétvághatósági szabály, 148
szignatúra, 449–51
szorzat, 201, 277, 282
szuperkulcsok, 142

szuperosztály, 78

tábla, 312, 313
tag típus, 98
Teljes bővítés, 156
Teljesen nem triviális, 149
természetes összekapcsolás, 287, 335
típuskonstruktor, 455, 456, 460, 475
típusrendszer, 456
többértékű függőség, 177, 179
többértékű függőség *nem triviális*, 183
többértékű függőségekre vonatkozó szabályok, 181
többértékű kapcsolat, 89
többértékű kapcsolatok, 118
többszörös öröklődés, 79
tömb, 58, 116, 474
törlés, 304, 306, 308, 324, 347
törlési problémák, 159
törzs, 217
tranzakció, 416
tranzakció, csak olvasó, 418
tranzakció, író-olvasó, 419
tranzakciók megvalósítása, 417
tranzakciók véglegesítése, 416
tranzitivitás, 155
tranzitivitási szabály, 153, 181
triviális függőségi szabály, 150, 181
triviális többértékű függőségek, 188
tulajdonos típus, 98
túlterhelt metódus, 449

Ubell, M., 501
útkifejezés, 457–58

valósághű modellezés, 71
változékony objektum, 456, 499
változhatatlan objektum, 456, 498
változó hosszúságú karaktersor, 312
változtató függvény, 490, 499
veszteségmentes összekapcsolás, 170
vetítés, 199, 268, 282, 328
virtuális típus, 100

WHERE, 267, 268, 270, 271, 458–60
WITH, 340, 344