

Szegedi Tudományegyetem
Informatikai Tanszékcsoporth

A PostgreSQL objektum-relációs lehetőségei

Szakdolgozat

Készítette:
Cser Lajos
programtervező informatikus
szakos hallgató

Témavezető:
Dr. Katona Endre
egyetemi docens

Szeged
2013

Feladatkiírás

- A témát kiíró oktató neve: Katona Endre
- Meghirdető egység: Képfeldolgozás és Számítógépes Grafika Tanszék
- Témakör: szoftverfejlesztés/adatbázis
- Típus: szakdolgozat
- A feladat megnevezése: A PostgreSQL objektum-relációs lehetőségei
- A feladat angol megnevezése: Object-relational features of PostgreSQL
- Milyen szakos hallgatók számára: programtervező informatikus BSc

A feladat rövid leírása:

A PostgreSQL adatbázis-kezelő rendszer objektum-relációs eszközeinek megismerése (típus definiálás, konstruktorok, tömbök, öröklődés). A hallgató készítsen mintaalkalmazást ezek használatára, értékelje az eszközök hatékonyságát és használhatóságát - összevetve az Oracle hasonló lehetőségeivel, lásd az alábbi szakdolgozatot.

Előismeretek, feltételek:

Az Adatbázis alapú rendszerek kurzus teljesítése legalább jó (4) érdemjeggyel.

Szakirodalom:

- Vasas Szabolcs: Objektum-relációs eszközök hatékonyságának vizsgálata Oracle környezetben. Szakdolgozat, SZTE, 2010.

A téma kiírását engedélyezte: Kató Zoltán, tanszékvezető, 2011. november 30.

Tartalmi összefoglaló

- **A téma megnevezése:**

A PostgreSQL objektum-relációs lehetőségeinek ismertetése és analízise

- **A megadott feladat megfogalmazása:**

A célunk egy jól áttekinthető összefoglalót adni a PostgreSQL objektum-relációs lehetőségeiről, és megállapítani, mennyiben érdemes kihasználnunk azokat; ezt követően összevetjük a megismert funkcionalitást az Oracle hasonló eszközeivel, mind hatékonyság, mind a gyakorlati alkalmazásokkor tapasztalható kifejezőkészség szempontjából.

- **A megoldási mód:**

A dolgozat egy mintaalkalmazáson keresztül bemutatja az elérhető lehetőségeket, majd különböző szempontok alapján kiértékeljük azok teljesítményét nagy mennyiségű adat feldolgozásakor, valamint a megvalósításhoz szükséges utasítások komplexitását.

- **Alkalmazott eszközök, módszerek:**

1. PostgreSQL 9.2.1
2. pgAdmin III 1.16.0
3. Microsoft Visual Studio 2010
4. Npgsql2 2.0.1
5. ArgoUML 0.28.1

- **Elért eredmények:**

A rendszerben fellelhető objektum-relációs eszközök kielégítő hatékonyságúak, de nem áll rendelkezésre olyan sokféle lehetőség, mint az Oracle-ben; a gyakorlat szempontjából azonban a legjellemzőbb problémák megoldásához elegendő a PostgreSQL kínálata is.

- **Kulcsszavak:**

PostgreSQL, adatbázis-kezelés, objektumok, objektum-relációs eszközök

Tartalomjegyzék

Feladatkiírás	2
Tartalmi összefoglaló	3
Tartalomjegyzék	4
1. Bevezetés	6
2. A PostgreSQL legfőbb objektum-relációs eszközei	7
2.1. Alapvető tudnivalók	7
2.2. A PostgreSQL megközelítése	8
2.3. Táblák és öröklődés	8
2.3.1. Tábla létrehozása	8
2.3.2. Az ONLY kulcsszó	9
2.4. Saját típusok	10
2.4.1. Elemi adattípusok definiálása	10
2.4.2. Összetett adatstruktúrák	11
2.5. Tömbök	11
2.5.1. Tömb deklarálása	11
2.5.2. Adatok beszúrása	12
2.5.3. Adatok lekérdezése	13
2.5.4. Elemek manipulációja	14
2.5.5. Elemek törlése	15
2.5.6. Teljes tömbök manipulációja	16
2.6. Metódusok	18
2.6.1. Tárolt eljárások létrehozása	18
2.6.2. Visszatérési típusok	19
2.6.3. Lekérdezések optimalizálása	20
2.6.4. Adattáblával visszatérés a gyakorlatban	20
2.6.5. Egy rövidített szintaxis	22
3. Egy mintaalkalmazás	23
3.1. Célok	23
3.2. A program fontosabb funkciói	23
3.2.1. Menüsor	23
3.2.2. Eszköztár	24

3.2.3. Munkaterület	25
3.2.4. További űrlapok	26
3.3. A fejlesztésről	27
3.3.1. Öröklődés és az integritás fenntartása.....	27
3.3.2. Összetett adatstruktúrák és tömbök	31
3.3.3. Tárolt függvények alkalmazhatósága	33
3.3.4. Tesztadatok generálása	36
3.3.5. PostgreSQL alapú alkalmazásfejlesztés C#-ban.....	38
4. Az objektum-relációs eszközök értékelése.....	40
4.1. Tesztelési eredmények	40
4.1.1. Értékelés programozói szempontból.....	41
4.1.2. A hatékonyság analízise.....	42
4.1.3. Összesítés.....	46
4.2. A PostgreSQL és az Oracle eszközei.....	46
4.2.1. PostgreSQL tömbök és az Oracle Varray, beágyazott tábla kollekciók	46
4.2.2. Oracle objektumtípusok, metódusok és a PostgreSQL eszközei	48
4.3. Betekintés a jövőbe	50
4.4. Záró gondolatok	52
Irodalomjegyzék.....	53
Nyilatkozat.....	55
Függelék	56
Tömböt felhasználó PL/pgSQL eljárás.....	56
A mintaalkalmazás adatbázisa	57

1. Bevezetés

A számítástechnika egy több évtizede létező, igen fontos feladata a valós világból vett adatok ésszerű tárolása, osztályozása és értékelése – tulajdonképpen tekinthetjük ezt az adatbázis-kezelés alapfeladatának is. Az élet mind több területén találkozhatunk olyan rendszerekkel, amik egy mögöttes adatbázisra támaszkodnak – gondoljunk akár a nagy banki adminisztrációs rendszerekre, vagy az élelmiszerboltokban használt számlázási alkalmazásokra.

A feladat univerzalitása és szerteágazósága már régóta komoly motivációt jelent a szoftverfejlesztésben arra, hogy különböző szoftverek és adatbázis-kezelési paradigmák jelenjenek meg időről-időre. Összességében elmondható, hogy napjainkban is a relációs adatmodellre támaszkodó megoldások a meghatározók a piacon, azonban több olyan rendszer is létezik, ami korunk objektumorientált szoftverfejlesztési technikáihoz jobban kíván alkalmazkodni a korábban említett „klasszikus” megközelítésnél.

Ez utóbbi csoportba sorolhatók az objektum-relációs adatbázisok (ORDB) valamint az objektum-relációs adatbáziskezelő-rendszerek (ORDBMS.) A bennük felfedezhető paradigma nem tisztán objektumorientált; sokkal inkább beszélhetünk a hagyományos relációs modell objektumokkal, valamint objektumorientált technikákkal való kibővítésére.

Mire is kell gondolnunk elsődlegesen? Megtarthatjuk mindazt, ami előnyös a relációs modellben, miközben beépíthetők olyan megoldások is, amelyeket csak OO programozásnál tudtunk eddig kihasználni. A teljesség igénye nélkül a következők bevezetéséről van szó ezek alapján: típusok, típusok származtatása, metódusok; táblák származtatása, tömbök, listák, halmazok, multihalmazok beépített kezelése; vagy éppen objektumhivatkozások.

Máig meghatározó szerepkörrel bír a piacon az *Oracle Database*, vagy csak röviden *Oracle*, az Oracle Corporation népszerű – nem nyílt forrású – ORDBMS-e. Az utóbbi években azonban egyre növekvő konkurenciát jelentenek számára a különböző ingyenes alternatívák, köztük a *PostgreSQL* is.

A *PostgreSQL* története az 1990-es évek közepére nyúlik vissza, amikor a Kaliforniai Egyetemen már huzamosabb ideje fejlesztett PostgreSQL a számottevő bővítések és a nyílt forráskód melletti állásfoglalás jeleként átkeresztelték az említett új névre. Egészen máig töretlenül folyik a rendszer finomhangolása, és ezzel párhuzamosan a népszerűsége is emelkedik. A korábban említetteknek megfelelően ingyenesen is hozzáférhető, de

amennyiben az alkalmazó nem kíván gondot és időt fordítani az üzemeltetésre, számos cég nyújt széles körű *PostgreSQL* támogatást díjazás ellenében.

Hogy láthatóvá váljék, miért is érdemes szót ejtenünk a *PostgreSQL*-ről, meg kell említeni, hogy az objektum-relációs technikákon kívül az *Oracle*-höz hasonlóan tartalmaz egy belső procedurális programozási nyelvet, a PL/pgSQL-t, sőt, akár más nyelveket is felhasználhatunk ilyen célra; támogatja a többverziós konkurencia-vezérlést, amivel eliminálhatók az olvasásra várakozások; valamint alkalmazásprogramozási interfészeivel (API) szinte tetszőleges algoritmikus programozási környezetből elérhetjük.

Joggal vetődik fel a kérdés ezek tükrében, hogy vajon érdemes-e elgondolkodnunk azon, hogy tegyük-e le a voksunkat a *PostgreSQL* mellett (például az *Oracle* helyett.) A következő fejezetekben ismertetésre kerülnek az alapvető tudnivalók a *PostgreSQL* objektum-relációs lehetőségeiről, látni fogunk mintát ezek alkalmazhatóságára, és végül megkísérlek párhuzamot vonni a nagy riválissal, az *Oracle*-lel.

2. A PostgreSQL legfőbb objektum-relációs eszközei

2.1. Alapvető tudnivalók

Az objektumorientált szervezés központi gondolata, hogy a programoknak egymással kommunikáló *objektumpéldányokból* kell állnia – minden objektumpéldány valamilyen általános sémának felel meg; ezen „sémáknak” az ún. *osztályokat* nevezhetjük. Minden objektum rendelkezik az ő egyedi jellegét meghatározó módon kitöltött adattagokkal és a kommunikációt megvalósító metódusokkal, ezzel mintegy *egységbe zárva* az adatokat és a rajtuk végzett műveleteket. Szintén általános a *származtatás*, amikor egyes speciálisabb jellegű objektumok egyes tulajdonságait valamilyen univerzálisabb objektum(ok)ból örököltetjük. Kitűnő lehetőségünk van így az *aggregáció* és *kompozíció* (rész-egész kapcsolatok) modellezésére is.

Szinte magát kínálja a lehetőség, hogy ezt a megközelítést alkalmazzuk az adatbázisokra is – ugyanis egy adatbázis megtervezésének egy lehetséges első lépése annak felmérése, milyen valós életbeli szereplők, egyedek fordulnak elő. A tisztán relációs modellnél ez után számos, sokszor kényelmetlen módosítást kell alkalmaznunk, amik javarészt elkerülhetők, ha objektum-relációs megközelítéssel élünk, ugyanis az imént említett egyedek könnyen megfeleltethetők objektumoknak. Jelen fejezet célja, hogy bemutassa, mennyire alkalmas a *PostgreSQL* legújabb, 9.2.1-es változata e törekvés

megvalósítására, mik a leglényegesebb beépített eszközök, és milyen szintaktikai sajátosságok érvényesülnek a DBMS SQL-ének vonatkozó utasításaiban.

2.2. A PostgreSQL megközelítése

Fontos leszögezni, hogy a tulajdonképpeni adattárolás a PostgreSQL-lel kezelt adatbázisokban mindig relációs elveket követ; e tekintetben úgy érezhetjük, hogy kissé szűkösebbek a lehetőségeink, mint példának okáért Oracle-ben, holott ez nem feltétlenül teljesül. Először is, nagyon széles körben támogatott az összetett adattípusok kezelése, ami a klasszikus relációs adatbázisokban nem elfogadott technika: E. Codd ajánlásai szerint egy relációs adatbázisban kerülendők az olyan attribútumok, melyek *önmagukban* halmazok. Másrészt, szofisztikált eszközök állnak a rendelkezésünkre a relációs elven tárolt adatok olyan transzformációira már az adatbázison belül, amelyekkel az objektum-relációs modell megvalósítható.

2.3. Táblák és öröklődés

A PostgreSQL-ben az objektum-relációs gondolkodás legtriviálisabb megjelenése a táblák közti öröklődés. Tulajdonképpen arról van ilyenkor szó, hogy egy valamely, speciálisabb objektumokat tároló táblában az adattagok, vagy pontosabban fogalmazva attribútumok egy részét egy vagy akár több másik táblából származtatjuk. Ennek folyamán a részt vevő táblák közt szülő-gyermek kapcsolat jön létre. A leszármazottban meg fog jelenni az összes oszlop és megszorítás a szülőből, valamint saját attribútumokkal is bővíthető.

2.3.1. Tábla létrehozása

Leszármazott táblát szokásosan a `CREATE TABLE SQL`-utasítással hozhatunk létre, ha felhasználjuk benne az `INHERITS` alparancsot. Utóbbi az `INHERITS` kulcsszón kívül azon táblák felsorolásából áll, amelyekből a gyermek örökölni fog [1].

Összességében tehát a szintaxis:

```
CREATE TABLE táblanév definíció
INHERITS (szülő [, szülő2, ..., szülőn]);
```

Ebben a megfogalmazásban táblanév alatt a létrehozandó új tábla elnevezése értendő, definíció alatt pedig a hagyományos értelemben vett `CREATE TABLE` alparancsok.

Például hozzunk létre egy táblát, amiben személyek adatait tárolhatjuk el, majd ebből származtassunk egy olyan táblát, amiben dolgozókról jegyzünk fel információkat:

```
CREATE TABLE ember (  
    id INTEGER,  
    nev CHARACTER VARYING,  
    cim CHARACTER VARYING,  
    CONSTRAINT pk PRIMARY KEY (id)  
);  
  
CREATE TABLE dolgozo (  
    fizetes INTEGER,  
    beosztas CHARACTER VARYING  
) INHERITS (ember);
```

Ilyenkor az ember táblának három, míg a dolgozó táblának a várt módon öt oszlopa lesz. Ezt megállapítván beszúrunk néhány minta adatot is a tábláinkba. Könnyen megtapasztalható, hogy az őstáblában ('ember') megjelenik az összes beszúrt „objektum”, míg a gyermek 'dolgozo' táblában csak azok, amiket konkrétan ebbe a táblába szántunk.

2.3.2. Az ONLY kulcsszó

Felvetődik a kérdés, hogy mit tehetünk, ha éppenséggel azt akarjuk elérni, hogy az őstípusnak eleget tevő rekordok kerüljenek csak lekérdezésre az őstáblából – ilyenkor a lekérdezésben használnunk kell az ONLY kulcsszót. Például, a

```
SELECT * FROM ember;
```

utasítással meg fog jelenni az eredményben az 'ember' táblában tárolt összes rekord, valamint azon rekordok is, amelyek a 'dolgozo' táblában vannak tárolva, viszont a

```
SELECT * FROM ONLY ember;
```

parancs hatására a származtatott táblákban tárolt rekordok nem kerülnek be a visszaadott eredmény soraiba.

Szintén aggályaink lehetnek a redundancia-mentességgel kapcsolatban, azonban ezek indokolatlanok. Azok a rekordok, amelyeket a leszármazott táblába szúrunk be, valójában fizikailag is csak a leszármazott táblában tárolódnak, és az ősből kizárólag a szülő-gyermek kapcsolat miatt jelennek meg. Úgy érezhetjük, hogy néha az öröklődés miatt egyes megszorítások is sérülhetnek. Például egy UNIQUE-kal megszorított őstáblabeli oszlopnál úgy tűnhet, mintha a megszorítás ellenére mégis többszörösen szerepelnének

azonos értékek – ez azonban csak úgy jöhet létre, ha valójában a gyermek táblából „felhozott” adatok közt van egy (vagy több) azonos érték. Emiatt lekérdezésekkor érdemes körütekintőnek lenni, ha élnek megszorítások az őstáblára – az ONLY kulcsszó használatával ezt a kellemetlenséget is elkerülhetjük.

2.4. Saját típusok

Csakúgy, mint a legtöbb objektum-relációs adatbáziskezelő-rendszerben, a PostgreSQL-ben is lehetséges úgymond saját típusokat felvenni. A típusfelvétel minden esetben a CREATE TYPE utasítással történik [3].

Egyaránt van lehetőség új *elemi* és *összetett* adattípusok felvételére is, valamint támogatottak a *felsorolások* és az *intervallumok*.

2.4.1. Elemi adattípusok definiálása

Az elemi adattípusok létrehozása csak adminisztrátori jogosultságokkal lehetséges. Először kettő függvényt kell kötelező jelleggel definiálnunk (a CREATE FUNCTION utasítással), ezek az adattípus úgymond *bemeneti* és *kimeneti* függvényei. A bemeneti függvény az adatot leíró szöveges konstansokat alakítja át a valós belső adatszerkezetre (tehát a típusra definiált operátorok és függvények fogják igénybe venni a programszöveg feldolgozásakor), a kimeneti függvény pedig a fordított transzformációt végzi el.

A bemeneti függvény kétféle paraméterezéssel hozható létre: vagy egy CSTRING típusú paramétert vár, vagy három argumentuma van, melyek rendre CSTRING, OID és INTEGER típusúak. Az első paraméter itt a típushoz tartozó értékek leírása, mint C szövegkonstans; a második a típushoz rendelt objektumazonosító, a harmadik pedig a típus deklarációihoz hozzáfűzhető ún. típusmódosító. Visszatérési értékének típusa minden esetben pontosan a definiálni kívánt adattípus.

Példa okáért létrehozunk egy olyan adattípust, ami római számokat tud kezelni.

```
CREATE TYPE romai;  
CREATE FUNCTION romai_be(CSTRING) RETURNS romai AS ...;  
CREATE FUNCTION romai_ki(romai) RETURNS CSTRING AS ...;  
CREATE TYPE romai (  
    INTERNALLENGTH = 2,  
    INPUT = romai_be,  
    OUTPUT = romai_ki
```

);

2.4.2. Összetett adatstruktúrák

Összetett adattípusokból könnyen készíthető tábla is. Például, felvehetünk egy szorzat-rekordot a következő módon:

```
CREATE TYPE dolgozo_tipus (  
    nev CHARACTER VARYING,  
    fizetes INTEGER  
);
```

Ebből előállítható egy olyan tábla, aminek a sémája tulajdonképpen a dolgozo_tipus adattagjai által meghatározottak:

```
CREATE TABLE dolgozo OF dolgozo_tipus (  
    PRIMARY KEY (nev)  
);
```

2.5. Tömbök

A rendszerben nagyon széles körű a tömbök támogatottsága. A tömb fogalmán tulajdonképpen egy tetszőleges PostgreSQL adattípus kiterjesztését kell értenünk – segítségükkel könnyen megvalósítható, hogy egy adott rekord adott mezőjébe azonos típusú értékek egy halmaza kerüljön. Jól látható, hogy ez a megközelítés kifejezetten az objektum-relációs modellre jellemző, hiszen a relációs adatbázisoknál szükségszerűen betartandó 1. normálformát sértené a megoldás. Jobban átgondolva azonban van létjogosultsága a technikának, hiszen sokszor – elsősorban programozói szempontból – nagyban növelhetjük így az átláthatóságot.

2.5.1. Tömb deklarációja

Ahhoz, hogy egy táblában valamely oszlop tömböket tároljon, egyszerűen szögletes zárójeleket – [] – kell írni az oszlop adattípusának megnevezése után a CREATE TABLE utasításon belül [1][3]. Például, ha egy könyv adatait kívánjuk eltárolni az adatbázisunkban, praktikusán élhetünk a következő megvalósítással:

```
CREATE TABLE konyv (  
    isbn CHARACTER VARYING,  
    kiadasi_ev INTEGER,
```

```

    cim CHARACTER VARYING,
    szerzo CHARACTER VARYING[],
    CONSTRAINT pk PRIMARY KEY (isbn)
);

```

Ha többdimenziós tömböt szeretnénk felvenni, akkor sincs nehezebb dolgunk, csupán többszöröznünk kell a zárójeleket. Ennek illusztrálására a következő módon tárolhatunk el tetszőleges kiterjedésű, kétdimenziós valós mátrixokat:

```
matrix REAL[][]
```

2.5.2. Adatok beszúrása

Az adatok beszúráshoz először definiálnunk kell egy olyan szintaxist, amivel a többértékű attribútumok konstansként történő megadása kivitelezhető. Tömböt a következő formátumban lehet megadni:

```
'{ ertek1 [, ..., ertekk] }'
```

$$k \geq 1$$

Fontos megjegyezni, hogy amennyiben az adattípus szöveges, a szövegkonstansokat a fenti konstrukción belül az SQL-hagyományoktól eltérően dupla idézőjelek közt kell megadni, azaz mondhatjuk úgy is, hogy szövegekre a fenti szintaxis speciálisan a következő:

```
'{ "ertek1" [, ..., "ertekk" ] }'
```

$$k \geq 1$$

Ezek alapján be is szúrhatunk néhány új rekordot a 'konyv' táblába:

```

INSERT INTO konyv VALUES (
    'ISBN 978-963-545-481-5',
    2009,
    'Adatbázisrendszerek',
    '{"Jeffrey D. Ullman", "Jennifer Widom"}'
);

INSERT INTO konyv VALUES (
    'ISBN 1-56592-846-6',
    2002,
    'Practical PostgreSQL',
    '{"John C. Worsley", "Joshua D. Drake"}'
);

```

2.5.3. Adatok lekérdezése

Beszúrás után próbáljuk meg lekérni az előbbieken feltöltött tömb tartalmát. Kissé kiábrándítónak találhatjuk azonban a következő SQL lekérdezés által visszaadott eredményt:

```
SELECT szerzo FROM konyv;
```

	szerzo <i>character varying[]</i>
1	{"Jeffrey D. Ullman","Jennifer Widom"}
2	{"John C. Worsley","Joshua D. Drake"}

Azaz, amint az látható, a tömböt tartalmazó attribútum lekérdezésekor a korábban vázolt konstansként kapjuk vissza az adatokat. Ez bizonyos esetekben hasznos lehet, de sokszor sokkal inkább arra lenne szükségünk, hogy a tömbben (vagy, ha az összes rekordot tekintjük, a tömbökben) lévő értékeket tábla formájában kapjuk vissza.

A cél elérése érdekében a tömböket lehet indexelni, valamint szeletelni. Az első esetben egy egész típusú értéket kell az oszlopnév után írunk a szokásos szögletes zárójelek közé lekérdezéskor; az indexelés a C és C-alapú nyelvekben megszokottól eltérően 1-től indul. Például, a

```
SELECT szerzo[2] FROM konyv;
```

utasítás le fogja kérdezni az összes könyv 2. szerzőjének nevét (azaz a fenti adatok esetén egy olyan relációt adna eredményül, amelynek egy oszlopa és két sora van – „Jennifer Widom”, valamint „Joshua D. Drake”).

A tömbök *szeletelése* is roppant hasonlóan történik [1]. Lényegében ekkor arról van szó, hogy egy adott index-intervallumot választunk ki a tömbből.

Szeleteléskor továbbra is a tömb azonosítója után írt [] operátorral dolgozunk, azonban a szögletes zárójelek közé ezúttal nem egyszerűen egy egész szám kerül, hanem az első és utolsó olyan indexérték, amelyet ki akarunk nyerni; a két szám közé pedig : (kettőspont) írandó. Például, a következőképp választható ki a könyvek második, harmadik és negyedik szerzője:

```
SELECT szerzo[2:4] FROM konyv;
```

Joggal vetődik fel a kérdés, hogy hogyan védjük ki a NULL értékeket a lekérdezéskor. Elképzelhető ugyanis, hogy a fenti lekérdezések végrehajtása egyes könyvekre definiálatlan értékkel tér vissza (lévén nem szükségszerű, hogy a könyveknek legyenek

társszerzői.) Egész egyszerűen annyit kell tennünk, hogy a lekérdezés WHERE klózában kiszelektáljuk azokat a rekordokat, ahol *nem* definiálatlan a tömb adott indexű eleme. Például, a második szerző lekérése történhetne a következőképpen:

```
SELECT szerzo[2] FROM konyv WHERE szerzo[2] IS NOT NULL;
```

A problémát ezzel a konstrukcióval elhárítottuk.

2.5.4. Elemek manipulációja

A tömbökben tárolt adatok manipulálására is teljes körű funkcionalitás érhető el. A PostgreSQL egyaránt képes az egyes tömbelemek módosítására, vagy akár egész tömbök lecserélésére.

A „módosítás” szó alatt jelen esetben nem csak a már létező indexeken található adatok lecserélését érthetjük, hanem az új tömbelemek beillesztését, vagy akár egyes tömbelemek törlését is. Az adatmódosításra példaként cseréljük le a Practical PostgreSQL második szerzőjének (Joshua D. Drake) nevét Joshua Drake-re:

```
UPDATE konyv
SET szerzo[2] = 'Joshua Drake'
WHERE cim = 'Practical PostgreSQL';
```

Az új adatok tömbbe beszúrása nyilván nem működhet a klasszikus INSERT utasítással, hiszen az egy teljesen új rekordot hozna létre. Ezt belátva logikusnak tűnik, hogy ilyenkor is az UPDATE utasítást használjuk, hiszen tulajdonképpen ez a művelet a tömb szempontjából továbbra is csak egy *módosítás*. Mielőtt azonban belevágna, még valamit tisztázni kell – ha visszagondolunk a tömbök CREATE TABLE utasításon belüli deklarációjára, látható, hogy nem szükséges számukra fix méretet megadni, és valóban – a tömbök (elméletileg) tetszőleges méretűre bővíthetők dinamikusan. Tehát, amennyiben jelenleg $n \geq 0$ elem van a tömbben, nem kell mást tennünk, mint az $n+1$ -edik, pillanatnyilag még kihasználatlan cella tartalmát a kívánt adatra módosítani. Ez egy további problémát vet fel – szükséges lenne ismernünk, hogy a módosítás előtt mennyi n értéke, azaz hány már felhasznált indexünk van. A rendszer ennek meghatározására biztosít egy eljárást:

```
array_length(tomb ANYARRAY, melyik_dimenzio INTEGER)
```

Ez az eljárás egész értéként visszaadja a tömb adott dimenziója mentén vett terjedelmét. Ezek ismeretében a következő triviális megoldás juthat eszünkbe például a Practical PostgreSQL szerkesztőjének szerzőként való bejegyzésére:

```
UPDATE konyv
SET szerzo[array_length(konyv.szerzo, 1)+1] = 'Jonathan Gennick'
WHERE cim = 'Practical PostgreSQL';
```

Némiképp cserbenhagyottnak érezhetjük magunkat, hogy egy ilyen gyakori feladatra mindig meg kell hívnunk a korábban említett eljárást – holott ennél sokkal kényelmesebb megoldás is ígérkezik a tömbökre kiterjesztett operátorok ismeretében.

Legyenek t_1 , t_2 tömbök, v egy elemi érték, amelyek rendre azonos típusúak. Ekkor a következő operátorokat alkalmazhatjuk:

$t_1 = t_2$	Meghatározza elemenként, hogy a két tömb azonos-e
$t_1 \parallel t_2$	Összefűzi a két tömböt
$v \parallel t_2$	a v értéket a tömb elé fűzi
$t_1 \parallel v$	A v értéket a tömb mögé fűzi

Ezek alapján a korábbi problémára adhatunk egy sokkal elegánsabb megoldást a \parallel operátor segítségével:

```
UPDATE konyv
SET szerzo = szerzo || '{"Jonathan Gennick"}'
WHERE cim = 'Practical PostgreSQL';
```

Amennyiben zavarónak találjuk azt, hogy a szöveges adatoknál továbbra is a korábban ismertetett formában kell megadni a konstanst, alternatíván használható az alábbi függvény:

```
tomb = array_append(tomb ANYARRAY, uj ANYELEMENT);
```

2.5.5. Elemek törlése

Elemek *törlésére* sincs konkrét utasítás – sajnos csak az eddigiekhez hasonló, „trükkös” módszereket vethetünk be, azonban a dolgunk sokkal nehezkesebb, mint korábban. Összességében – jelenleg – nem lehetséges univerzális megoldást adni, és talán nem is célszerű, így itt is csak egy példa szerepelhet.

Tegyük fel, hogy a cél egy adott indexű tömbelem eltávolítása egy egydimenziós tömbből (ilyen a 'konyv' tábla szerzo[] attribútuma is.) Az alapötlet a következő: meg kell adni egy olyan eljárást, ami paraméterül egy tetszőleges típusú tömböt és egy i index

értéket vár. Ellenőrzi, hogy a tömb megfelelő dimenziójú-e, és ha igen, bejárja azt. A bejárás során egy új tömböt állít elő, amiből kihagyja az eredeti tömb *i*-edik elemét; végül ezzel a tömbbel visszatér.

Erre a következő függvényt definiáltam:

```
CREATE FUNCTION array_delete(arr ANYARRAY, INTEGER) RETURNS ANYARRAY
AS
$$
DECLARE
    hossz INTEGER = 0;
    i INTEGER = 1;
    kimenet arr%TYPE;
BEGIN
    IF array_ndims(arr) <> 1 THEN
        BEGIN
            RETURN arr;
        END;
    END IF;
    hossz = array_length(arr, 1);
    WHILE i <= hossz LOOP
        BEGIN
            IF i <> $2 THEN
                kimenet = kimenet || arr[i];
            END IF;
            i = i + 1;
        END;
    END LOOP;
    RETURN kimenet;
END;
$$ LANGUAGE plpgsql;
```

Ezen eljárás birtokában pl. a Practical PostgreSQL 3. szerzőjeként korábban hozzáadott Jonathan Gennick nevét törölhetjük a következőképp:

```
UPDATE konyv
SET szerzo = array_delete(szerzo, 3)
WHERE cím = 'Practical PostgreSQL';
```

Ez alapján nem nehéz elképzelni egy olyan függvényt, ami nagyobb dimenziószámok esetén is boldogulhat.

2.5.6. Teljes tömbök manipulációja

Természetesen nem csak egyes tömbelemek manipulálására van lehetőség, hanem az egész tömbére is. Ekkor nem kell mást tennünk, mint az UPDATE utasítással lecserélni az egész tömböt valamilyen új változatra. Például, az összes könyv szerzőit V. A. Lacky-ra állíthatjuk az alábbi paranccsal:


```
UPDATE konyv
```

```
SET szerzo = '{"V. A. Lacky"}';
```

Azonban még ennyivel sem kell beérnünk – a PostgreSQL tömbkezelésében lehetséges a szeletelés technikájának köszönhetően egész intervallumokat lecserélni. Ha a tömbünk egy szeletét kívánjuk módosítani, ügyelnünk kell arra, hogy az értékül adott új szeletben ugyanannyi elem legyen, mint amennyit a szeleteléskor kijelöltünk.

Ennek demonstrálására először felveszünk egy sok szerzővel rendelkező könyvet, majd annak 3-5. szerzőit rendre A, B, és C-vel helyettesítjük.

```
INSERT INTO konyv VALUES ( 'ISBN 978-963-9664-45-6', 2007,  
'Szövegbányászat', '{"Tikk Domonkos", "Farkas Richárd", "Kardkovács  
Zsolt", "Kovács László", "Répási Tibor", "Szarvas György", "Szaskó  
Sándor", "Vázsonyi Miklós"}');
```

```
UPDATE konyv
```

```
SET szerzo[3:5] = '{"A", "B", "C"}'
```

```
WHERE cim = 'Szövegbányászat';
```

```
SELECT isbn, UNNEST(szerzo) AS modositott_szerzok FROM konyv WHERE  
cim='Szövegbányászat';
```

A lekérdezésben egyúttal egy praktikus tömbkezelő függvény felhasználására is látható példa – az unnest(ANYARRAY) több sorból álló táblává fejt ki a neki átadott tömböt. Az eredmény:

	isbn <i>character varying</i>	modositott_szerzok <i>character varying</i>
1	ISBN 978-963-9664-45-6	Tikk Domonkos
2	ISBN 978-963-9664-45-6	Farkas Richárd
3	ISBN 978-963-9664-45-6	A
4	ISBN 978-963-9664-45-6	B
5	ISBN 978-963-9664-45-6	C
6	ISBN 978-963-9664-45-6	Szarvas György
7	ISBN 978-963-9664-45-6	Szaskó László
8	ISBN 978-963-9664-45-6	Vázsonyi Miklós

A tömbök egy lehetséges felhasználási területét mutatja be a függelékben közzétett „konyvszerzok()” eljárás.

2.6. Metódusok

A korábbiakban láthattuk, hogy számos olyan eszköz áll rendelkezésünkre, amelyekkel a klasszikus, relációs értelemben vett táblákat jócskán kibővíthetjük objektum-relációs megoldásokkal. Ahhoz azonban, hogy a tábláinkban tárolt rekordok igazán objektumszerűen viselkedjenek, megkerülhetetlen az egységbe záras fenntartása érdekében néhány metódus, vagy másként tárolt eljárás felvétele is; az adatokon végzett műveleteket is mellékelnünk kell valamilyen úton.

2.6.1. Tárolt eljárások létrehozása

Eljárások létrehozására lényegében egy lehetőségünk van – a meglehetősen univerzális CREATE [OR REPLACE] FUNCTION utasítás, amely egyszerűsített általános alakja a következő:

```
CREATE [OR REPLACE] FUNCTION elnevezés ( [ argumentum_típus [ , ... ] ]  
)  
RETURNS visszatérési_típus  
AS 'definíciós rész'  
LANGUAGE 'definíciós rész programnyelve';
```

Lényegében ezzel a megoldással egy hagyományos, relációs adatbázis-kezelőkben is megszokott tárolt eljárás deklarálható, azonban némi absztrakcióval fel tudjuk majd használni a sorobjektumok viselkedését implementáló metódusok megadására is.

A legtöbb esetben célszerű az opcionális [OR REPLACE] alparancsot is felhasználni, ugyanis ha már korábban létrehoztuk az eljárást, így az felülíródik. Ha a paraméterlistán változtattunk a korábbi verzióhoz képest, a csere megghiúsul; ilyenkor a

```
DROP FUNCTION eljárás_neve;
```

parancs kiadása indokolttá válhat.

Az eljárás elnevezése után szükséges zárójelek közt a formális paraméterlistát megadni; a zárójelek kiírása akkor sem maradhat el, ha az üres. Az egyes paramétereket vesszővel elválasztva kell felsorolni. Minden paraméterhez meg kell adni annak típusát, valamint szabadon dönthetünk arról, hogy el kívánjuk-e nevezni azt. Például, a két egész szám összegét kiszámoló „összeg” eljárás paraméterlistája megadható két módon is:

```
CREATE FUNCTION osszeg ( INTEGER, INTEGER ) AS ...; (1)
```

```
CREATE FUNCTION osszeg ( elso INTEGER, masodik INTEGER ) AS ...; (2)
```

Az (1) esetben az eljárás törzsében egy adott paraméterre annak sorszámmal hivatkozhatunk (például a második számot \$2-ként érhetjük el), míg a (2) sorban szereplő konstrukciónál a „masodik” névvel találhatjuk meg.

2.6.2. Visszatérési típusok

A visszatérés típusa vagy egy tetszőleges *PostgreSQL* típus, egy *saját típus*, valamely típusból képzett *tömb*, vagy *adattábla* lehet, a PostgreSQL típusok közé értve egyaránt az alap, összetett- és doméntípusokat. (Szükség esetén a visszatérési típus VOID (semmis) is lehet, ekkor tulajdonképpen nem függvényt, hanem procedúrát hozunk létre.) Az első három eset meglehetősen triviális, azonban a negyedik lehetőséget érdemes bővebben tárgyalni.

Ha táblával kívánunk visszatérni, a RETURNS után a TABLE kulcsszót kell kiírunk, majd pedig meg kell adnunk a visszatérési tábla oszlopait:

```
... RETURNS TABLE ( oszlopnév1 oszloptípus1 [, oszlopnév2 típus2, ...] )  
AS ...
```

A visszatérési tábla adattartalmának előállítás az eljárás törzsében történik – mielőtt ennek mikéntjét tárgyalnánk, célszerű egyáltalán a törzs helyes megadási módját és a LANGUAGE alparancs értelmét tisztázni.

A PostgreSQL tárolt eljárások törzsét számos különböző programozási nyelven implementálhatjuk, vagy akár azt a megoldást is választhatjuk, hogy egy külső modulból töltjük be a végrehajtandó kódot. Akár C nyelven írt függvényekhez is biztosíthatunk interfészt, azonban sokkal jellemzőbb, hogy SQL-ben vagy PL/pgSQL-ben írjuk meg a kódot. A LANGUAGE kulcsszó után tehát azt kell meghatározni, hogy a törzs milyen nyelven adott – ez vagy a már említett nyelvek egyike, vagy a CREATE LANGUAGE utasítással a rendszerbe integrált saját nyelvek valamelyike lehet. Például, a következő módon adnánk meg egy konstans szöveggel visszatérő, PL/pgSQL nyelvű eljárást:

```
CREATE FUNCTION konstszoveg ()  
  RETURNS CHARACTER VARYING AS  
  $$  
  BEGIN  
      RETURN "konstans_szöveg";  
  END;  
  $$ LANGUAGE 'plpgsql' IMMUTABLE;
```

} PL/pgSQL blokk

2.6.3. Lekérdezések optimalizálása

Szokásos megadni a CREATE FUNCTION utasítás végén azt is, hogy a lekérdezés-optimalizáló miként kezelje az eljárást. Az egyik lehetőség a korábbi példában is használt IMMUTABLE módosító; ezen kívül a STABLE és VOLATILE használható. A VOLATILE-ként deklarált függvényeknél semmilyen optimalizálás nem végezhető, ugyanis ilyenkor az az érvényes feltételezés, hogy a függvény más és más értékekkel térhet vissza azonos paraméterezéssel, akár azonos táblára hívva is, valamint lehetnek bizonyos mellékhatásai a futásnak az adatbázisra nézve (rekordok módosulása.) A rendszer automatikusan VOLATILE-nak tekinti azon függvényeket, amelyeknél nem adjuk meg az optimalizációs módosítót. A STABLE eljárásoknál már jobb a helyzet; azzal a feltételezéssel élünk, hogy garantáltan nem fognak semmilyen mellékhatással lenni az adatbázisra, és egy adott táblára végrehajtva azonos bemeneti paraméterekkel azonos eredményt adnak vissza, noha más és más SQL utasításoknál eltérhet a visszaadott érték. A legjobb optimalizálási eredményt IMMUTABLE eljárásoknál kaphatjuk. Ezen módosító megadásával azt határozzuk meg, hogy a függvény nem módosít semmit az adatbázison, valamint adott argumentumértékre mindig azonos választ ad (azaz a kimenete nem függ pl. a törzsön belüli olyan lekérdezésektől, amik más és más értékeket adhatnak vissza azonos paraméterre.) Ettől az adatunk jól indexelhetővé válik [3][4].

2.6.4. Adattáblával visszatérés a gyakorlatban

Térjünk vissza a korábban felvázolt problémára – az adattáblát visszaadó függvény kimenetének előállítására (a törzs PL/pgSQL-ben történő megadását feltételezzük.)

Több megoldási lehetőség közül választhatunk [6]. A legegyszerűbb, ha az eljárásban a RETURN utasítás után a QUERY SELECT ... parancsot adjuk meg – például, megadhatunk egy függvényt arra, hogy a korábban bevezetett **konyv**(isbn, kiadasi_ev, cím, szerzo[]) táblából (meglehetősen csavartan) előállítsunk egy táblát egy bizonyos évnél újabb könyvek címével és kiadási évével:

```
CREATE OR REPLACE FUNCTION konyvlista (INTEGER)
RETURNS TABLE (ev INTEGER, cim CHARACTER VARYING) AS $$
BEGIN
    RETURN QUERY SELECT k.kiadasi_ev, k.cim FROM konyv k
        WHERE k.kiadasi_ev > $1;
END;
```

```
$$ LANGUAGE 'plpgsql' VOLATILE;
```

```
SELECT k.ev, k.cim FROM konyvlista(1990) k;
```

Ez a konstrukció elsősorban akkor lehet hasznos, ha a paraméterezéstől függően más és más, de azonos adatszerkezettel rendelkező kimenetű lekérdezések eredményét szeretnénk megkapni egy elegáns (és redundancia-mentes) megoldással.

Érdekesebb megközelítés, ha a tábla leendő rekordjait egyesével állítjuk elő. A korábbi eljárásnak megadható így egy alternatív változata:

```
CREATE OR REPLACE FUNCTION konyvlista_b (INTEGER)
RETURNS TABLE (ev INTEGER, cim CHARACTER VARYING) AS $$
DECLARE
    sor RECORD;
BEGIN
    FOR sor IN
    (
        SELECT k.cim, k.kiadasi_ev FROM konyv k
        WHERE k.kiadasi_ev > $1
    )
    LOOP
        ev = sor.kiadasi_ev;
        cim = sor.cim;
        RETURN NEXT;
    END LOOP;
END;
$$ LANGUAGE 'plpgsql' VOLATILE;
```

```
SELECT (konyvlista_b(2005)).cim, (konyvlista_b(2005)).ev;
SELECT k.ev, k.cim FROM konyvlista_b(2005) k;
```

Az eljárás meghívásának két lehetséges módját is megfigyelhetjük a példában; a SELECT utasításon belül a tábla megadásának helyén is megtehetjük ezt (második megoldás), vagy rögtön az oszlopok felsorolásánál (első megoldás.) Utóbbi esetben, ha csak egyszerűen SELECT konyvlista_b(2005); lett volna kiadva, egy egyetlen RECORD

típusú attribútummal rendelkező táblát kaptunk volna vissza, amiből már jól látható, milyen megfontolásból használható a példabeli konstrukció.

2.6.5. Egy rövidített szintaxis

A technikai részletekkel foglalkozó rövid kitérő után visszatérhetünk a tárolt eljárások objektum-relációs megoldásokban való felhasználásához.

Legegyszerűbb esetben azt szeretnénk elérni, hogy egy adott táblának legyen egy „virtuális oszlopa”, ami a valódi mezők alapján számolt értékkel rendelkezik. Ekkor tulajdonképpen egy olyan tárolt eljárást kell létrehoznunk, amely paraméterül egy adott táblatípust vár – a gyakorlatban pedig a PostgreSQL egy kényelmes megoldást biztosít arra, hogy a SELECT utasításon belül a metódushívás szinte megkülönböztethetetlen legyen a szokványos értelemben vett projekciótól [4].

Például, a 'konyv' táblánál maradvá megadhatunk egy olyan metódust, ami az adott könyvre kiszámítja annak korát (azaz a jelenlegi évszám és a kiadási év különbségét.) A következő CREATE FUNCTION utasítást adjuk ki:

```
CREATE OR REPLACE FUNCTION kor(konyv) RETURNS INTEGER AS $$  
SELECT CAST(EXTRACT(YEAR FROM NOW()) AS INTEGER)-$1.kiadasi_ev;  
$$ LANGUAGE SQL IMMUTABLE;
```

A metódusunk létrehozása után szeretnénk kipróbálni azt a gyakorlatban is. A futtatás és annak eredménye:

```
SELECT k.isbn, k.cim, k.kor FROM konyv k;
```

	isbn <i>character varying</i>	cim <i>character varying</i>	kor <i>integer</i>
1	ISBN 978-963-545-481-5	Adatbázisrendszerek	3
2	ISBN 1-56592-846-6	Practical PostgreSQL	10

Valóban, megkaptuk minden egyes könyvre annak pillanatnyi korát úgy, hogy az érték konkrétan sehol sincs feljegyezve a rendszerben; valamint azt is elértük, hogy a gyakorlat szempontjából ne legyen különválasztva a könyv típusú egyedek ezen tulajdonságának számítása az adattárolástól.

3. Egy mintaalkalmazás

A korábbi fejezetben ismertetett lehetőségeket célszerű egy valódi alkalmazás segítségével demonstrálni. A fejezet célja a program és a benne alkalmazott megoldások ismertetése.

3.1. Célok

Elsődleges cél, hogy átfogó és életszerű képet sikerüljön adni az előbbieken vázolt funkciók gyakorlati használhatóságáról. Ezért az alkalmazást a valós élethelyzetekkel meredek ellentétben a fejlesztőeszközökhöz kell igazítani, és *nem* a tulajdonképpeni funkcióhoz – a program témája így egy mezőgazdasági termelő és szolgáltató vállalkozás elképzelt adat-nyilvántartási igényeit kielégítő szoftver lett. Kitűnő lehetőség kínálkozik a táblák közti öröklődés, saját típusok, vagy épp beágyazott táblák felhasználására.

3.2. A program fontosabb funkciói

Szemléletét tekintve tulajdonképpen a felhasználói program egy kliens, ami csatlakozni képes PostgreSQL adatbázisszerverekhez. Az alkalmazás elindítása után egy ablakot kapunk, amelyben egy menüsor, egy eszköztár és egy munkaterület látható. A munkaterületen az egyes lekérdezések eredményét lehet megjeleníteni táblázatos formában.

3.2.1. Menüsor

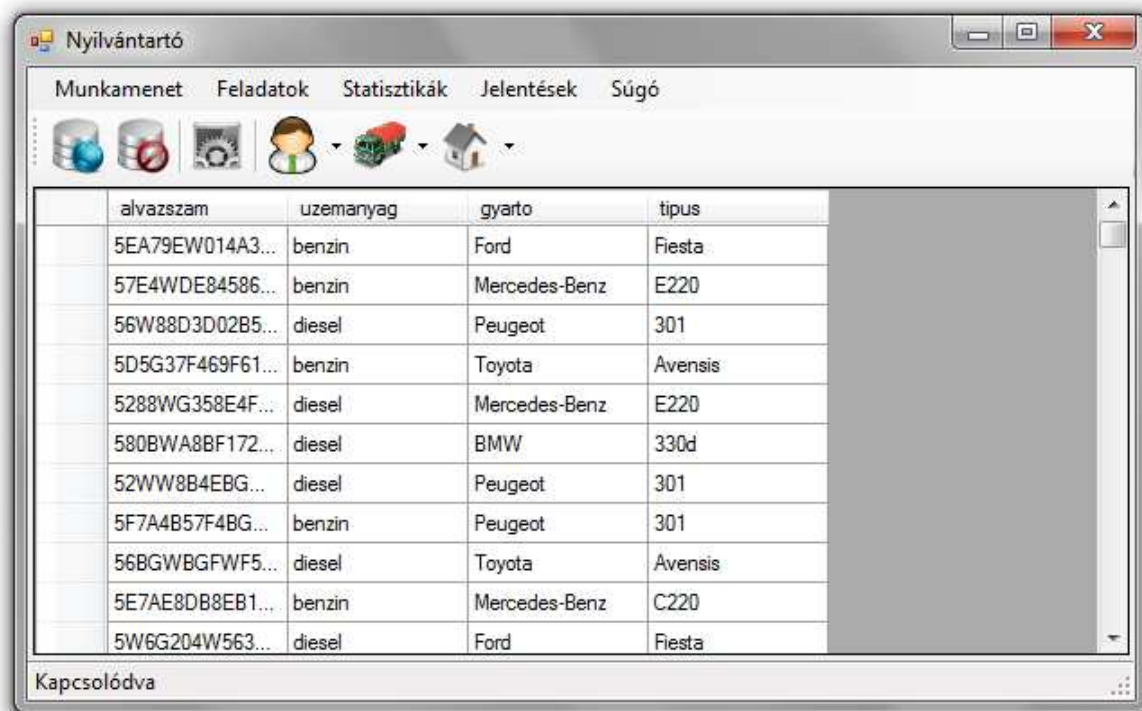
A menüsor öt pontból áll: Munkamenet, Feladatok, Statisztikák, Jelentések, Súlyó. Az első lehetőséget választva a szerverhez kapcsolódáshoz szükséges adatokon módosíthatunk (alapértelmezés szerint a saját gépünkön keresi a kliens az adatbázist, de igény esetén ezen változtatni lehet.) A kapcsolat kezdetben nem nyitott, nekünk kell azt megnyitnunk a „Kapcsolat nyitása” menüpontot választva. Ha egy másik szerverhez szeretnénk később csatlakozni, vagy valamilyen egyéb okból bontanunk kell a kapcsolatot, a „Kapcsolat bontása” menüponttal ez elvégezhető.

A Feladatok menüben az egyes tárolt adatok manipulációjához szükséges programrészek érhetők el (adatok beszúrása, törlése, módosítása, listázása.) Szintén innen indítható keresés az adatbázisban.

A Statisztikák menüben néhány érdekes adat kérhető le az adatbázissal kapcsolatban (például a dolgozói létszám, vagy a járművek átlagfogyasztása.) Az itt elérhető funkciók leginkább az objektum-relációs elven tárolt adatok kezelésének néhány érdekesebb részletét próbálják feltárni.

A Jelentések menüben hasonló összesítéseket kérhetünk, de szöveges (TXT) formában kimenthetjük azokat a háttértárra.

A Súgó menüben segítség kapható a program kezelésével kapcsolatban.



3.2.1.1 ábra. Az alkalmazás főablaka.

3.2.2. Eszköztár

Az eszköztár a menüsor alatt található, a legfontosabb funkciók gyors elérésére biztosít gombokat. Az egyes ikonok csoportokba rendezettek.

A legelső két gombbal az adatbázis-kapcsolat megnyitása illetve zárása lehetséges. Adatbázis-műveletek csak nyitott kapcsolat mellett végezhetők, ellenkező esetben ilyen kísérletnél hibaüzenetet kapunk.

Az adatbázis-kapcsolat tulajdonságai innen is állíthatóak, az eszköztár harmadik gombjára kattintás után megjelenő űrlap segítségével.

A 4-6. gombok a főbb műveletekhez nyújtanak hozzáférést. Egyszerűen rájuk kattintva listázások indíthatók, de a mellettük elhelyezkedő, lefelé mutató háromszögek

segítségével az adott objektumtípushoz kapcsolódó további feladatokat is elvégezhetjük a felugró helyi menüvel.

A negyedik gomb (stilizált emberalak) a fiktív vállalat dolgozóinak listázását teszi lehetővé. A legördíthető helyi menüből ezen kívül kiadhatunk utasítást dolgozók létrehozására vagy módosítására illetve törlésére is.

Az ötödik gomb a járműállomány kezelésére szolgál. Hasonlóan a dolgozókhoz, a gombra történő egyszerű kattintás hatására egy összesítő listát kapunk az összes járműről. Fontos megjegyezni, hogy az adatbázisban a járművek *több* táblában tárolódnak, ugyanis egyaránt léteznek személyautók, teherautók és munkagépek is az elképzelt vállalatnál. Mivel mindegyik rendelkezik néhány közös jellemzővel (például bármelyiknek van alvázszáma, adott teljesítményű motorja, stb.), kihasználtam az öröklődést, mint objektum-relációs lehetőséget, azaz egy közös őstáblából származik a fenti három járműtípus külön-külön táblája. Az eszköztári gombbal egy olyan lekérdezés indítható el, amely az összes járműt megjeleníti. Ezt egy hagyományos relációs megoldásnál viszonylag bonyolult lenne SQL-ben megfogalmazni (projekciót kell végrehajtani az egyes táblákon belül, hogy kompatibilis táblákat kapjunk, majd azoknak venni kell az unióját úgy, hogy az akár egyező rekordok is megmaradjanak az eredményhalmazban.) Ezzel szemben a származtatásnak köszönhetően elegendő volt egy roppant egyszerű szelekciót végrehajtani az őstáblán. (Bővebb összefoglalás a járművek adatait tároló adatbázisrész implementációjáról a soron következő, 3.3-as , „A fejlesztésről” című alfejezetben olvasható.)

Jelen gomb mellé is elhelyezésre került egy lenyitható helyi menü, ami hasonlóan a járművekkel kapcsolatos egyéb feladatok elvégzésére biztosít lehetőséget. Különösen érdekes – az objektum-relációs eszközök vizsgálata szempontjából – a járművek **tankolásainak** kezelésére szolgáló megoldás (erről szintén esik szó a 3.3-as alfejezetben is, valamint a jelen szakaszban is taglalom felhasználói szempontból a program kapcsolódó szolgáltatásait.)

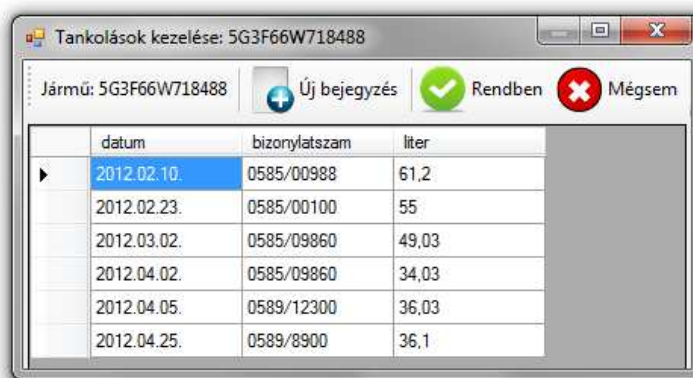
Az eszköztár legutolsó gombjával az ingatlan-nyilvántartás kezelhető. Talán nem meglepő, hogy a gombra kattintás az ingatlanok listázását teszi lehetővé, míg a legördíthető menüben a további kapcsolódó funkciók érhetők el.

3.2.3. Munkaterület

A munkaterület az eszköztár alatti része az ablaknak. Alapjában véve két részre osztható: egy adatmegjelenítésre szolgáló (kezdetben szürke) felület, és egy állapotsor.

Utóbbi kizárólag visszajelzést biztosít a különböző tevékenységeinket követően (például megjeleníti a „Kapcsolódva” szöveget, ha sikeres volt a kapcsolatlétrehozási kísérletünk, vagy a „Kapcsolat bontva” felirattal rendelkezik offline üzemmódban.)

A fontosabb rész a munkaterületen a képernyő nagy részét kitöltő adatmegjelenítő. Egy lekérdezés eredményét táblázatos formában tárja elénk, azaz minden attribútum egy oszlopként jelenik meg, és egy sor természetesen egy rekordot takar. Az adattartalom azon múlik, hogy az eszköztári gombokkal milyen lekérdezést hajtottunk végre legutoljára.



The screenshot shows a window titled 'Tankolások kezelése: 5G3F66W718488'. Below the title bar, there is a label 'Jármű: 5G3F66W718488' and three buttons: 'Új bejegyzés' (with a plus icon), 'Rendben' (with a green checkmark icon), and 'Mégsem' (with a red X icon). Below these is a table with the following data:

	datum	bizonylatszám	liter
▶	2012.02.10.	0585/00988	61,2
	2012.02.23.	0585/00100	55
	2012.03.02.	0585/09860	49,03
	2012.04.02.	0585/09860	34,03
	2012.04.05.	0589/12300	36,03
	2012.04.25.	0589/8900	36,1

3.2.3.1 ábra. A tankolásokat kezelő űrlap.

A rekordok manipulációja is részben a segítségével történik. Lehetőségünk van kijelölni egyes rekordokat, és azokra az eszköztárral műveleteket végrehajtani. Egyetlen rekordot kijelölni meglehetősen egyszerű: csupán rá kell kattintanunk, ekkor a teljes sor kiemelődik. Ha több rekordot is ki kívánunk jelölni, akkor érdemes lenyomva tartani a Ctrl billentyűt, és rákattintani a megfelelő sorokra. Ezt követően kell kiválasztanunk a menüből vagy az eszköztárból a végrehajtani kívánt aktualizálási műveletet vagy adatlekérést. A szoftver ügyel arra, hogy az adott típusú adatra csak a rá értelmezhető feladatokat lehessen végrehajtani (azaz például csak jármű tankolásait lehet lekérni, természetesen egy dolgozóra ez nem értelmezett.)

3.2.4. További űrlapok

A teljes funkcionalitás biztosításához számos további ablakot is tartalmaz az alkalmazás. Ezek konkrétabban a következők:

- űrlap ingatlan adatok felvételéhez, módosításához
- űrlap jármű adatok felvételéhez, módosításához
- űrlap dolgozói adatok felvételéhez, módosításához
- űrlap jármű tankolásainak nyilvántartásához.

Az adatbeviteli- és módosítási űrlapok használata jobbra magától értetődő. Szövegbeviteli mezők segítségével megadható a kívánt adattartalom, majd a Mentés gombra kattintva az adatbázis aktualizálódik.

Külön részletezni csak a jármű tankolásait kezelő űrlapot érdemes (összhangban a korábbiakkal.)

A főablakban először le kell kérnünk a járművek listáját. Jelöljük ki a kezelni kívánt jármű(veke)t, majd keressük meg az eszköztáron a megfelelő menüpontot („Jármű(vek) tankolásainak kezelése”). Ekkor minden egyes kijelölt járműhöz létrejön egy űrlap, amelyben az eddigi üzemanyag-vételezések listáját látjuk, valamint egy eszköztárat.

Új tankolást (dátum, bizonylatszám, vásárolt mennyiség) az „Új bejegyzés” gombra kattintással lehet hozzáadni.

A tankolási adatok tárolása figyelemre méltó módon bemutatja az objektum-relációs eszközök széles skáláját; erről szintén értekezem bővebben a következő, 3.3-as alfejezetben, „Összetett adatstruktúrák és tömbök” címen, valamint találhatunk mérési eredményeket az aktualizálási művelet végrehajtási sebességét illetően is (meglepően inkább a gyors futás jellemző.)

3.3. A fejlesztésről

A kliens fejlesztéséhez a Microsoft Visual Studio 2010-et választottam, amely a .NET Framework révén tartalmazza a napjainkban növekvő népszerűségnek örvendő C# programozási nyelvet is. Az alkalmazás minél több részét ezzel szemben PL/pgSQL-ben írtam meg, hogy nagyobb hangsúly legyen az adatbázisrétegen, és mélyebb betekintést nyerhessünk a témába. Az adatbázis sematikus ábrája mellékletként megtalálható.

Az első feladat az adattáblák létrehozása volt, amely szokásosan a CREATE TABLE utasítással történt. A mellékelt ábrán látható módon hangsúlyos a táblák közti öröklődés, valamint a nem atomi adatot tartalmazó adatszerkezetek felhasználása.

3.3.1. Öröklődés és az integritás fenntartása

Különösen szép példa az öröklődésre a **Jarmu**(Alvazszam, Uzemanyag, Gyarto, Tankolasok[]) reláció és leszármazottai:

- **Szemelyauto**(Jarmu.Alvazszam, Jarmu.Uzemanyag, Jarmu.Gyarto, Jarmu.Tankolasok[], UtasMax, CsomagtartoMeret)
- **Teherauto**(Jarmu.Alvazszam, Jarmu.Uzemanyag, Jarmu.Gyarto, Jarmu.Tankolasok[], Kapacitas)

- **Munkagep**(*Jarmu.Alvazszam*, *Jarmu.Uzemanyag*, *Jarmu.Gyarto*, *Jarmu.Tankolasok*[], *Fajta*, *MotorEro*)

Sejthető, hogy tulajdonképpen a Jarmu táblában nem fogunk adatokat tárolni, hanem csak annak leszármazottaiban. Két érveléssel indokolható jelen esetben az öröklődés használata:

- Szeretnénk kényelmes felületet kapni arra, hogy a különböző járművekről (személyautók, mezőgazdasági munkagépek, teherautók, stb.) egy összesítő listát állíthassunk elő. Hagyományos relációs eszközökkel (azaz az örökölt oszlopokat a megfelelő, most leszármaztatott táblák létrehozásakor külön-külön mindig létrehozva) ehhez egy meglehetősen bonyolult SELECT utasítást kell kiadni, amelyben az egyes táblákból projektáljuk a közös attribútumokat, és az egyes listarészletek unióját képezzük a UNION paranccsal. Az öröklődést használó objektum-relációs megoldásnál azonban egyszerűen elegendő a

```
SELECT * FROM Jarmu;
```

SQL-utasítás kiadása.

- Bizonyos műveleteket nem akarunk külön-külön minden egyes járműtípusra definiálni, hanem generikusan az összesre.

Látható, hogy minden egyes leszármazott táblában elsődleges kulcsként az Alvazszam attribútumot adtam meg. Ez garantálja, hogy adott típusú járműből nem lehet két azonos alvázszámú, de – gondoljunk vissza az előző fejezet öröklődésről szóló szakaszára – nem akadályozza meg, hogy egy másik típusú járműből létezessen azonos alvázszámú példány. Ez a gyakorlatban megengedhetetlen, ugyanis minden jármű egyedi azonosítóval rendelkezik, típustól függetlenül. Pusztán az elsődleges kulcsok leszármazott táblánkénti megadása (sőt, az őstáblában való megadása) megengedi például azt, hogy egy 'ABC123' alvázszámú személyautót, és egy ezzel azonos alvázszámú teherautót is eltároljunk. Ha a SELECT * FROM Jarmu; utasítással lekérdezzük a Jarmu tábla (virtuális) tartalmát, akkor két 'ABC123' kulcsú jármű is meg fog jelenni. Ez első ránézésre persze sérti a kulcs feltételt, de jobban belegondolva, az az előírásunk, miszerint egy leszármazott táblában csak egy 'ABC123' szerepelhet, továbbra is teljesül. A PostgreSQL jelenleg nem tartalmaz megoldást arra, hogy a fenti problémát feloldjuk, így saját kezűleg kell

implementálnunk az integritás fenntartásáért felelős programrészeket. Hogyan történhet mindez?

Első érdekesség a táblák létrehozása során, hogy lehet megszorításokat tenni örökölt oszlopra is.

```
CREATE TABLE Jarmu (  
    Alvazszam CHARACTER VARYING,  
    Uzemanyag CHARACTER VARYING,  
    Gyarto CHARACTER VARYING,  
    Tipus CHARACTER VARYING,  
    Tankolasok TankolasRekord[],  
    CONSTRAINT jarmu_pk PRIMARY KEY (Alvazszam)  
);  
  
CREATE TABLE Munkagep (  
    Fajta CHARACTER VARYING,  
    MotorEro INT,  
    CONSTRAINT munkagep_pk PRIMARY KEY (Alvazszam)  
) INHERITS (Jarmu);
```

Miután ezzel megvagyunk, érdemes kigondolni, hogyan lehetne elérni, hogy a Munkagep táblába ne vihessünk be olyan adatot, amely a Szemelyauto vagy Teherauto táblában már szerepel. Az alkalmazásban én egy **trigger** megadását választottam, ami minden beszúrás előtt elvégzi ezt az ellenőrzést, és fenntartja az integritást. A megoldás hátránya, hogy költséges lekérdezéseket alkalmaz, de ha feltételezzük, hogy ritkán szűrnak be új járművet, vagy módosítanak egy már létezőt, ettől a kényelmetlenségtől eltekinthetünk.

A trigger megadása előtt egy olyan függvényt kellett PL/pgSQL-ben megírni, amely egy adott járműtípusra megmondja, hogy mik a többi járműtípus által már lefoglalt, azaz nem engedélyezett alvázszámok:

```
CREATE OR REPLACE FUNCTION Jarmu_integritas(melyikhez  
    eng_jarmu_tipusok)  
RETURNS TABLE (alv CHARACTER VARYING) AS  
$$  
BEGIN  
IF melyikhez = 'szemelyauto' THEN  
    RETURN QUERY ((SELECT Alvazszam FROM Munkagep) UNION  
        (SELECT Alvazszam FROM Teherauto));  
ELSIF melyikhez = 'munkagep' THEN  
    RETURN QUERY (  

```

```

                (SELECT Alvazszam FROM Szemelyauto) UNION
                (SELECT Alvazszam FROM Teherauto));
ELSE
    RETURN QUERY (
        (SELECT Alvazszam FROM Munkagep) UNION
        (SELECT Alvazszam FROM Szemelyauto));
END IF;
END;
$$ LANGUAGE plpgsql VOLATILE;

```

A függvénynek valamely járműtípus megnevezését lehet megadni (az eng_jarmu_tipusok típus használata korlátozza a megadható értékek halmazát.) Ezt követően egy olyan sorszintű BEFORE triggert használtam, amely a módosított sorban ellenőrzi az új alvázszámot, és nem engedélyezi a tárolást, ha az érték zavarná a táblák közti integritást.

A triggerek létrehozása PostgreSQL-ben is a CREATE TRIGGER paranccsal történik, azonban a végrehajtandó kódot egy külön függvényben kell leírni. Az ilyen „trigger-függvények” nem rendelkezhetnek paraméterekkel, és a visszatérési típusuk rendre TRIGGER kell legyen. A törzsükben ennek hatására elérhetővé válnak bizonyos trigger-változók (például sorszintű trigger-eknél a régi rekordot tartalmazó OLD, vagy az újat tartalmazó NEW, stb.) A trigger tulajdonképpeni létrehozásakor már csupán csak hivatkozunk egy ilyen függvényre, és megadjuk az elsütési feltételeket.

A Munkagep táblára például az alábbi trigger-függvényt és triggert adtam meg:

```

CREATE OR REPLACE FUNCTION integritas_helyreallit()
RETURNS TRIGGER AS $$
DECLARE
    tipus eng_jarmu_tipusok;
BEGIN
    tipus := 'munkagep';
    IF NEW.alvazszam IN (SELECT Jarmu_integritas(tipus)) THEN
        RETURN NULL;
    ELSE
        RETURN NEW;
    END IF;
END;
$$ LANGUAGE plpgsql VOLATILE;

CREATE TRIGGER Munkagep_integritas
BEFORE INSERT OR UPDATE
ON Munkagep
FOR EACH ROW
EXECUTE PROCEDURE Integritas_helyreallit();

```

Vizsgáljuk meg alaposabban az integritas_helyreallit() trigger-függvényt. Működése meglehetősen egyszerű: ha az új rekord (NEW) alvázszáma szerepel a

jarmu_integritas(eng_jarmu_tipusok) által visszaadottak közt, NULL értékkel tér vissza, különben NEW-val.

Ha egy trigger-függvény NEW-val tér vissza, akkor az aktualizálási művelet végrehajtódik, az új értékek bekerülnek a táblába (NEW.attribútum formában hivatkozhatunk természetesen a rekord egyes mezőire, és át is írhatjuk őket még a trigger futása során.) A NULL-lal visszatérés esetén az aktualizálási művelet nem hajtódik végre.

A fenti függvényt kiterjeszthetjük úgy, hogy az összes járműtípus kezelésére alkalmas legyen, és minden típusra létrehozhatunk egy-egy trigger-t is. Végeredményben ezzel sikerült garantálni, hogy egy alvázszám csak egyszer szerepelhessen az adatbázisban.

3.3.2. Összetett adatstruktúrák és tömbök

A fejlesztés során kimondottan törekedtem minél inkább 1NF-sértő konstrukciókat (is) alkalmazni, ugyanis az objektum-relációs modell – ahogy az a bevezető fejezetből már kitűnt – erősen hagyatkozik az efféle megoldásokra.

A legegyszerűbb, ha egy olyan attribútumot használunk (egy táblában), amely valamely általunk definiált szorzat-rekord típus alapján lett definiálva. Némiképp érdekesebb, ha tömböket alkalmazunk; sőt, e kettő kombinációjával tulajdonképpen létrehozhatunk „beágyazott táblákat”, azaz rekordokból álló tömböket.

Ennek egy kitűnő példája a Jarmu tábla Tankolasok[] attribútuma, amely TankolasRekord saját típusú rekordokat képes eltárolni.

Először érdemes szemügyre venni, miként definiáltam a fenti rekordtípust:

```
CREATE TYPE TankolasRekord AS (  
    Datum DATE,  
    Bizonylatszam CHARACTER VARYING,  
    Liter REAL  
);
```

A tábla definíciójával már a korábbiakban találkozhattunk, valamint az alkalmazást leíró alfejezetben szó esett arról, hogy a programban miként lehet üzemanyag-vételezéseket eltárolni. Tekintsünk a dolgok mélyére – először érdemes megvizsgálni, hogyan működik az új tankolások eltárolása.

A tömbben új elem hozzáadásához használható az array_append() eljárás, de én a feladat szempontjából érdekesebbnek, és bizonyos szempontból kényelmesebbnek találtam, ha más megoldáshoz folyamodom. Az első probléma, hogy meg kell határozni, hogy *jelenleg* hány elem található a tömbben. Erre praktikusán definiáltam egy tárolt függvényt:

```
CREATE OR REPLACE FUNCTION UjTankolas(Jarmu) RETURNS INTEGER AS $$
SELECT COALESCE(array_length($1.Tankolasok, 1)+1, 1);
$$ LANGUAGE SQL;
```

Amint az látható, a függvény törzse egyetlen SQL-lekérdezés, amely az array_length() eljárást használja a tömb elemszámának megadására. Példát láthatunk a PostgreSQL SQL-ének COALESCE() függvényére is, amely az első nem definiálatlan értékű paraméterével tér vissza (hasonló az Oracle NVL()-éhez.)

Ezután már csak meg kellett adni egy olyan SQL utasítást, amely elvégzi a tömb kiegészítését a fenti eljárás segítségével. Ez a következőképp fest (egy konkrét inputtal a szemléletesség kedvéért:)

```
UPDATE Jarmu
SET Tankolasok[Jarmu.UjTankolas] =
    ROW('2012-04-25', '0589/8900', 36.10)
WHERE Alvazszam = '5G3F66W718488';
```

Ez a parancs tehát egy adott alvázszámú járműhöz bejegyzi a ROW() segítségével konstruált rekordot a tankolásokat tároló tömb végére. A megoldás legérdekesebb és legelgondolkodtatóbb része az, ahogyan megadtuk, melyik indexű helyre kell beilleszteni az új bejegyzést. Az első fejezetben közöltek alapján látható, hogy a tömbindexelésben szereplő **Jarmu.UjTankolas** konstrukció valójában nem mást takar, mint hogy az aktualizálási művelet által érintett **összes** Jarmu rekordra meghívódik a korábban elkészített UjTankolas(Jarmu) eljárás.

A ROW() nevezhető rekord-konstruktornak is, a paraméterül adott értékekből egy megfelelő típusú rekord-példányt állít elő.

A programban (vagy a korábbiakban) láthattuk, hogy a tankolások egy adott járműre szép, táblázatos formában jelennek meg, holott a tárolás láthatóan egyáltalán nem relációs (táblázatos) módon történik. Meg kellett adni egy olyan lekérdezést, ami a kívánt formában jeleníti meg az adatokat; ez pedig a következő:

```
SELECT
    (UNNEST(Tankolasok)).Datum,
    (UNNEST(Tankolasok)).Bizonylatszam,
    (UNNEST(Tankolasok)).Liter
FROM Jarmu WHERE Alvazszam = '5G3F66W718488';
```

Az UNNEST() felhasználása már ismerős lehet; SELECT utasításokon belül arra használhatjuk, hogy tömböket táblára „törjön szét”. Mivel a tömbünk elemei rekordok, így egy szöveges literálként kapnánk meg csak az egyes elemeket. Erre van megoldás:

referálni kell az egyes konkrét mezőkre (ahogy az a példában is látható), és máris a kívánt, több oszlopból és több sorból álló eredményt kapjuk.

Első ránézésre meglehetősen körülményesnek tűnhetnek a fenti kódrészletek. Ezzel szemben a futtatásukkor meglepően tapasztaltam, hogy a végrehajtásuk meglehetősen gyors, mindössze néhány (kb. 12) milliszekundum alatt végrehajtódott például az új rekord tankolási bejegyzés beszúrása. Bővebben olvashatunk minderről az „Elért eredmények” című fejezetben.

3.3.3. Tárolt függvények alkalmazhatósága

Egy objektum-relációs megoldásokat felvonultató implementációban szinte garantált, hogy szükség lesz tárolt függvények használatára. A legtöbb nagyobb adatbázis-kezelő tartalmaz olyan bővítményeket, amelyekkel programokat hozhatunk létre, és a programozói kánonban erősen változó annak a megítélése, hogy érdemes-e ilyen megoldásokhoz folyamodni.

Érvként hangozhat el az alkalmazásuk mellett a teljesítmény-optimalizáltság, valamint az, hogy az alkalmazások implementációs nyelvétől független megoldást adunk, és jellemzően az adatok hosszú életűek a vállalat szempontjából; a programok, és különösen a fejlesztés során használt nyelv gyakrabban cserélődik. (Például néhány évvel ezelőtt a C# korántsem örvendett akkora népszerűségnek, mint manapság, vagy a '90-es évek népszerű RAD fejlesztőkörnyezete, a Borland Delphi és az Object Pascal nyelv mára már szinte teljesen feledésbe merült.) Ha PL/pgSQL-ben írjuk meg azt az absztrakciós réteget, amely a lényegi adatmanipulációs műveleteket végzi, egységbe zárjuk azokat az adatokkal, elkerülhetjük, hogy a különböző alkalmazásokban redundáns és nehezen áttekinthető programrészleteket hozzunk létre.

További pozitívuma a tárolt függvények felhasználásának, hogy csökkenthetjük a hálózati adatforgalmat is (jellemzően külön szerveren helyezkedik el a DBMS, mint pl. a webkiszolgáló, kliens alkalmazásokról nem is beszélve.) Egyáltalán nem mindegy – egy nagy alkalmazásra gondolva – hogy például 14 millió rekordot kell átvinni a hálózaton, és azt feldolgozni egy alkalmazásprogramozási nyelven, vagy a feldolgozást már a szerveren elvégezzük, és egyetlen sort – az eredményt – küldjük csak el a lekérdezést indító programnak. [12][13]

Összességében tehát mindenképpen fontosnak éreztem, hogy a mintaalkalmazás tartalmazzon példát e részterületre is, és a fenti rövid fejtegetés után vissza is térünk a konkrétumokhoz.

Az egyszerűség kedvéért továbbra is a Jarmu relációhoz kapcsolódó függvényeket tárgyalom első ízben.

A legtriviálisabb példa talán az lehetne, amikor egy jármű összefogyasztását akarjuk meghatározni. Ez nem jelent mást, mint a járműhöz kapcsolódó összes eddigi tankolás bejegyzett mennyiségét összesíteni kell.

Gondoljunk vissza a korábbiakban elhangzottakra – a tankolások objektum-relációs elven, egy „beágyazott tábla” jellegű konstrukcióban (tömb+rekord) tárolódnak. A SELECT utasítással gyakorlatilag azt tehetjük meg, hogy előállítjuk a vásárolt mennyiségek listáját a következő módon:

```
SELECT
  (UNNEST(Tankolasok)).Liter
FROM Jarmu WHERE Alvazsam = '5G3F66W718488';
```

Ezt követően feldolgozhatnánk a listát az alkalmazásunkban, vagy megkísérelhetnénk a parancs további bonyolítását SQL-ben. A SUM() aggregát függvény nem használható egyből, ugyanis a fenti lekérdezés eredményét nem fogadja el feldolgozható halmazként. Persze megpróbálkozhatunk egy még összetettebb lekérdezéssel:

```
SELECT SUM(liter)
FROM (
  SELECT
    (UNNEST(Tankolasok)).Liter
  FROM Jarmu
  WHERE Alvazsam = '5G3F66W718488'
) AS belso;
```

Ez korántsem „szép”, és minden egyes alkalommal az alkalmazásban felhasználni nehézkesnek mondható a probléma méretéhez képest. (Ha egy valós életbeli felhasználásra gondolunk, ami feltételezhetően jóval komplexebb, mint az itt közölt mintaalkalmazás, hasonló adatszerkezetek esetén olyan komplex lekérdezéseket kellene írunk, amelyek teljességgel áttekinthetetlenek.) A nemrégiben feltárt észrevételek miatt is (hálózati forgalom, élettartam stb.) létjogosultsága van tehát egy „metódus”, tárolt függvény megadásának.

Ez a következő:

```
CREATE OR REPLACE FUNCTION OsszFogyasztas(Jarmu) RETURNS REAL AS $$
DECLARE
  sum REAL := 0.0;
  i INTEGER := 1;
BEGIN
  WHILE i <= array_length($1.Tankolasok, 1) LOOP
```

```

BEGIN
    sum := sum + $1.Tankolasok[i].Liter;
    i := i + 1;
END;
END LOOP;
RETURN sum;
END;
$$ LANGUAGE plpgsql;

```

Amint láthatjuk, maga az algoritmus roppant egyszerű, és természetesebben adódik, mint a korábbi SQL-lekérdezés. Még ennél is jobb, hogy innentől kezdve a járművünk összefogyasztása lekérhető a következő paranccsal:

```

SELECT j.OsszFogyasztas
FROM Jarmu j
WHERE j.Alvazszam = '5G3F66W718488';

```

Gondoljuk tovább a problémát. Érdekes lenne felparaméterezni a metódust – tegyük fel, hogy egy *adott év* összefogyasztására vagyunk kíváncsiak. Nem sokat kell változtatni a korábbi függvényen:

```

CREATE OR REPLACE FUNCTION Fogyasztas(Jarmu, INTEGER) RETURNS
REAL AS $$
DECLARE
    sum REAL = 0.0;
    i INTEGER = 1;
BEGIN
    WHILE i <= array_length($1.Tankolasok, 1) LOOP
        BEGIN
            IF (EXTRACT(YEAR FROM $1.Tankolasok[i].Datum) = $2)
            THEN
                sum = sum + $1.Tankolasok[i].Liter;
            END IF;
            i = i + 1;
        END;
    END LOOP;
    RETURN sum;
END;
$$ LANGUAGE plpgsql;

```

A példában mintát láthatunk az EXTRACT() eljárásra. Ez az időkezelő függvények egyike; a dátum tetszőleges mezőjét megadja (paraméterként egy kifejezést vár, amellyel meghatározzuk, mely mezőre van szükségünk mely dátumból.)

A módosított lekérdezés végrehajtása hasonlóan történik a korábbiéhoz, de itt már nem alkalmazhatjuk a rövidített hívási szintaxist:

```

SELECT Fogyasztas(j, 2011)

```

```
FROM Jarmu j
WHERE j.Alvazszam = '5G3F66W718488';
```

Eredményét tekintve ez az eljárás is hasonló formában állítja elő az adatokat, mint a korábban látott példaprogram.

Metódusokat az itt bemutatott eseteken kívül is törekedtem felhasználni az alkalmazásban, egyebek mellett például így oldódott meg a tesztadatok generálása is.

3.3.4. Tesztadatok generálása

Nincs jó adatbázis alapú mintaalkalmazás megfelelő tesztadatok nélkül – a felhasznált eszközök teljesítményét aligha lehet másként kiértékelni, mint ha lehetőleg minél nagyobb adathalmazra hajtunk végre műveleteket.

Bár a mintaalkalmazás egy nyilvántartó rendszer csupán, és egy átlagos magyar vállalkozásnál sem dolgozóból, sem gépjárműből nincs százezres nagyságrendű mennyiség, az eredeti célkitűzést – a minél nagyobb demonstratív erőt – szem előtt tartva én mégis jelentős mennyiségű tesztadatot generáltam.

Szem előtt próbáltam tartani, hogy az adatok minél inkább életszerűek legyenek. Így például a dolgozók adatait a következőképpen generáltam:

- Felvettem néhány gyakori magyar vezeték- és keresztnévet.
- Eldöntöttem, mik a lehetséges munkakörök a fiktív vállalatnál.
- Meghatároztam, hogy mikortól meddig kell terjednie a dolgozók születési dátumának.
- Személyi számként egy olyan karaktersorozatot adtam meg, amely a születési dátumból és egy véletlenszerűen választott számból tevődik össze.

A tulajdonképpeni megoldáshoz egy tárolt eljárást adtam meg, amely erősen támaszkodott a véletlenszám-generálásra. Ez a következő (dolgozók esetén:)

```
CREATE OR REPLACE FUNCTION dolgozo_generator(integer)
  RETURNS void AS $$
DECLARE
  ev INTEGER; ho INTEGER; nap INTEGER; nev CHARACTER VARYING;
  beosztasa CHARACTER VARYING; szuldat DATE;
  szemszam CHARACTER VARYING; i INTEGER = 0;
  n INTEGER; m INTEGER; o INTEGER;
  veznev CHARACTER VARYING[] = '{"Tóth", "Kovács"}';
  szemnev CHARACTER VARYING[] = '{"Péter", "Balázs"}';
  beoszt CHARACTER VARYING[] = '{"takarító", "titkár"}';
BEGIN
  WHILE i < $1 LOOP
```

```

BEGIN
    ev = 1930+CAST(random()*60 +1 AS INTEGER);
    ho = CAST(random()*11 +1 AS INTEGER);
    nap = CAST(random()*27 +1 AS INTEGER);
    n = CAST(random()*(array_length(veznev,1)-1) +1 AS INTEGER);
    m = CAST(random()*(array_length(szemnev,1)-1) +1 AS
        INTEGER);
    o = CAST(random()*(array_length(beoszt,1)-1) +1 AS
        INTEGER);
    nev = veznev[n] || ' ' || szemnev[m];
    beosztasa = beoszt[o];
    szemszam = CAST(ev AS CHARACTER VARYING) ||
        CAST(ho AS CHARACTER VARYING) || CAST(nap AS CHARACTER
        VARYING) || '-' || CAST(CAST(random()*1000 AS INTEGER)
    AS
        CHARACTER VARYING) || '-1';
    szuldat = CAST((CAST(ev AS CHARACTER VARYING) || '-' ||
    CAST(ho
        AS CHARACTER VARYING) || '-' || CAST(nap AS CHARACTER
        VARYING)) AS DATE);
    i = i + 1;
    INSERT INTO Dolgozo(nev, személyiszam, születesidatum,
        beosztas, szerzodese) VALUES (
        nev,
        szemszam,
        szuldat,
        beosztasa,
        ROW('Szerz. szöveg.', FALSE, 1)
    );
END;
END LOOP;
END;
$$LANGUAGE plpgsql VOLATILE; 1

```

Az eljárás egyetlen egész számot vár paraméterként, a generálni kívánt dolgozók mennyiségét. Az adatok generálása teljesen véletlenszerű; ha véletlenül olyan adatot próbál meg beilleszteni, ami már szerepel a táblában (és így sérülne az elsődleges kulcs általi megszorítás), a végrehajtás megghiúsul; azonban a gyakorlatban ez szinte soha sem fordult még elő.

Objektum-relációs szempontból a *Dolgozo* táblában leginkább a *szerzodese* attribútum érdekes (összetett és egyéni adattípus lévén), valamint a relációra definiálható metódusok.

Azonos elven lett létrehozva a többi adattábla tartalma is, így a nagyfokú hasonlóságból kifolyólag azok előállítási módjának közlésétől eltekintek (a programkód megtalálható a szakdolgozat lemezmellékletén.)

¹ Az eljárást a jobb olvashatóság érdekében rövidítettem, a valódi alkalmazásban például több nevet és foglalkozást tartalmaz, hogy fokozódjon a generált adatok életszerűsége.

3.3.5. PostgreSQL alapú alkalmazásfejlesztés C#-ban

A Microsoft Visual Studio és a .NET keretrendszer nagyszerű eszközöket tartalmaz gyors alkalmazásfejlesztéshez. A C# programozási nyelv segítségével a programozói rutinfeladatok meglehetősen lerövidülnek – választásom emiatt esett rá. [10]

Ahhoz, hogy kapcsolódni tudjunk egy PostgreSQL-t futtató adatbázis-szerverhez, először szükséges egy megfelelő illesztőprogramot (data provider) beszerezni. C#-hoz a legjobb választás az Npgsql csomag, ami az adatbázis-kapcsolatot a Microsoft OleDb csomagjához hasonló eszközökkel valósítja meg, de természetesen PostgreSQL-specifikus módon. A csomag beszerzése után egy dinamikus csatolású függvénykönyvtárat (dll) kell a projekthez adni. A megfelelő névterek hivatkozása után (*Npgsql* és *NpgsqlTypes*) használhatóvá válnak az eszközök. [8]

Kapcsolat definiálása és megnyitása a következő módon történik:

```
NpgsqlConnection conn = new NpgsqlConnection();
String connectionString = String.Format("Server={0};Port={1};"
    + "User Id={2};Password={3};Database={4};",
    "localhost", "5432", "postgres",
    "aaa", "teszt");

conn.Open();
```

A fenti kód hatására a localhoston (saját számítógépünkön) az 5432-es porton futó szerverhez kapcsolódhatunk, „postgres” felhasználónévvel, „aaa” jelszóval, a „teszt” adatbázist választva.

Lekérdezések végrehajtásához alapjában véve a következő parancsokat kell kiadnunk:

```
NpgsqlCommand stmt = conn.CreateCommand();
stmt.CommandText = 'SELECT * FROM Jarmu';
NpgsqlDataReader reader = stmt.ExecuteReader();
```

A lekérdezés által visszaadott tábla a *reader* nevű változóba kerül. Ennek tartalmát egy előletesztelő ismétléssel feldolgozhatjuk.

```
while(reader.Read())
{
    Console.WriteLine("{0}\t{1}\n", reader[0], reader[1]);
}
conn.Close();
```

Az alkalmazásban – tekintettel arra, hogy grafikus felhasználói felülettel rendelkezik – rendszerint azonban nem ilyen módon történik az adatok feldolgozása és megjelenítése. Használható egy DataGridView nevű vezérlő, amely táblázatos formában képes

adatforrásokat megjeleníteni. A következő módon dinamikusan változtatható a megjelenő tartalom:

```
/* Létehoztunk egy DataGridView-t korábban az osztályban dtgv néven */
NpgsqlDataAdapter da = new NpgsqlDataAdapter(sql, conn);
DataSet ds = new DataSet();
ds.Reset();
da.Fill(ds);
DataTable dt = ds.Tables[0];
dtgv.DataSource = dt;
```

Összességében elmondható, hogy az Npgsql csomag jól használható, különösebb fejtörést nem okoz az alkalmazása.

4. Az objektum-relációs eszközök értékelése

4.1. Tesztelési eredmények

Jelen fejezet célja, hogy egy minél hitelesebb értékelést adjunk a PostgreSQL korábban tárgyalt funkcióiról. Alapvetően három szempont mentén kell vizsgálatokat végezni:

- Milyennek mondhatók az objektum-relációs megközelítés leprogramozására szolgáló eszközök? Egyszerűbbé teszik a programozói munkát (feltéve, ha jól átgondolt koncepciónk van, megtervezzük az alkalmazást)? Könnyen érthető és ellenőrizhető kódot kapunk, vagy sem?
- A végrehajtási sebesség kielégítő? Jobb, vagy rosszabb eredményt kapnánk, ha tisztán relációsan oldanánk meg a feladatot?
- Mit mondhatunk összességében – a két korábbi szempont közül kapunk-e legalább az egyiknél olyannyira bízató választ, hogy a gyakorlatban is az objektum-relációs implementáció mellett döntsünk?

Az első kérdés megválaszolása viszonylag egyszerű a bevezető fejezet és a mintaalkalmazás megírása során tapasztaltak alapján. Sokkal nehezebb ügy jó választ adni a második felvetésre. Az, hogy egy SQL-utasítás mennyi idő alatt hajtodik végre, nagyban függ az adatbázist futtató számítógép teljesítményétől és a feldolgozandó adatok mennyiségétől is.

Jobb lehetőség híján a programhoz tartozó adatbázison végeztem el a tesztelést, a saját számítógépen futtatva. A technikai paraméterek:

- Intel® Core™ 2 Duo CPU P8600 @ 2.40 GHz
- 4 GB RAM
- Windows 7 Professional Service Pack 1.

Minden esetben a mérés hitelessé tételéhez többször is lefuttattam ugyanazt a lekérdezést az adatokon, majd az átlagos teljesítményből vontam le konzekvenciát. További problémát jelentett azonban, hogy hogyan lehetne az önmagukban igen semmitmondó számokhoz valós értelmet rendelni – szükségessé vált egy megfelelő viszonyítási alap keresése.

A feladat nem annyira triviális, mint amilyennek az leírva tűnik.

4.1.1. Értékelés programozói szempontból

Személyes véleményt formálni egy rendszerről bármikor lehetséges, de nem szükségszerűen objektív; erre mindenképpen érdemes tekintettel lenni jelen sorok olvasásakor.

Véleményem szerint a PostgreSQL meggyőző teljesítményű és jó funkcionalitású adatbázis-kezelő rendszer. Természetesen nem csupán a miatt mondható ez el, mert tartalmaz OR megoldásokat – a telepítéstől kezdve az üzemeltetés eszközein át egészen a nyelvi lehetőségek megalkotásáig minden részfeladatra komoly figyelmet fordítottak fejlesztői.

A *pgAdmin* szoftver, amely PostgreSQL adatbázisok adminisztrációjához és SQL / PL/pgSQL alapú szoftverfejlesztéshez biztosít környezetet (nagyban hasonlít az *Oracle SQL Developer*-hez) , minden igényt kielégítő grafikus felhasználói felületével nagyban segítette a dolgozat megírása során a munkámat. Webes munkakörnyezetben hasonlóan jól használható a *phpPgAdmin* projekt.

A PostgreSQL adatbázisok felépítése, és az SQL nyelvi lehetőségek jól követik az SQL szabványt, és hasznos, könnyen kihasználható bővítményekkel járulnak hozzá ahhoz. Tulajdonképpen az objektum-relációs kellékeket is tekinthetjük ilyennek; a gyakorlat szempontjából ugyanis úgy találtam, hogy a jelen publikációban tárgyalt eszközök felhasználása hasznos lehet az adatbázis-programozó számára. A szoftverfejlesztésben ma elterjedt technikák némiképp persze eltérő irányultságúak (pl. object-relational mapping), de ennek nem következménye, hogy nincs más járható út.

Mindazonáltal fontos kihangsúlyozni azt is, hogy nem helyes hozzáállás az sem, ha minden felmerülő problémára megpróbáljuk „ráerőszakolni” az objektum-relációs megoldásokat [4]. Fennáll ilyenkor a veszélye, hogy az adatbázisunk nehezen átláthatóvá, vagy még rosszabb esetben, nehezen kezelhetővé válik. Például bizonyos esetekben érdemes lehet tömböket használni, míg máskor nem. A tömböket ismertető fejezetben található példában könyvek szerzőit tároltuk el egy többértékű attribútumban (bővebben lásd 2.5.1. alfejezet.) Az ilyen jellegű információkat tároló adatstruktúrák módosítása ritkán szükséges (a könyvek szerzői leginkább változatlanok, legalábbis egy adott kiadásé biztosan), és kényelmesebb lehet így tárolni őket, mint külön relációs táblában, ahol az integritás fenntartásához külső kulcs megszorításokat kell tennünk. Elsőre megijedhetünk, hogy ekkor egy adott szerző (például most John C. Worsley) könyveinek lekérése

nehézségekbe ütközhet, holott valójában ez is megoldható egy, a természetes összekapcsolást igénylő relációs lekérdezésnél nem komplexebb SQL utasítással:

```
SELECT cím, szerzo FROM
(
  SELECT k.cím, UNNEST(k.szerzo) szerzo
  FROM konyv k
)
AS sub WHERE sub.szerzo = 'John C. Worsley';
```

Kevésbé mondhatók praktikusnak viszont a tömbök akkor, ha nagy mennyiségű, gyakran változó adatot kell eltárolnunk. Ilyen esetekben érdemesebb lehet a hagyományos relációs megoldások választása.

4.1.2. A hatékonyság analízise

Szubjektív véleménynyilvánítás után időszerű objektív mértékeket keresni. Az egyik legfontosabb kérdés, ha hatékonyságról beszélünk, hogy egy kérést milyen sebességgel képes az adatbázis-szerver kiszolgálni.

Lényeges, hogy egyrészt a felhasználó gyorsan hozzájusson a lekért információhoz (a türelme jellemzően véges), másrészt bizonyos alkalmazásokban szükség lehet a gyors reakciókra. Nem elhanyagolandó tényező az sem, hogy a hosszabb végrehajtási idő a működési költségeket is megemelheti [2].

Két művelettípust vizsgáltam közelebbről: az egyik az adatok beszúrása, a másik pedig az adatok módosítása, méghozzá a mintaalkalmazás olyan tábláin, amelyek objektum-relációs megvalósítást tartalmaznak. Elgondolkodtató volt számomra, hogy az adatbázisokban széles körűen alkalmazott cache-elést küszöböljem-e ki vagy sem (például a rendszer ismételt újraindításaival.) Végül úgy döntöttem, hogy nem fogom ezt tenni, ugyanis a gyakorlat szempontjából ez életszerűtlen. Igaz ugyan, hogy ritkán előforduló utasításoknál a legelső végrehajtás időben tovább tarthat, mint egyébként, de egy program jellemzően többször és frekváltan éri el ugyanazokat a területeket az adatbázisban. Emiatt a legelső végrehajtást nem vettem figyelembe, de egy-egy tesztet között lefuttattam néhány eltérő, akkor épp nem vizsgált lekérdezést is.

További vizsgálódáshoz természetesen kiváló lehetőség a cache-elés kiküszöbölése mellett a nagyon ritkán előforduló műveletek végrehajtási hatékonyságának elemzése is, de ez már túlmutat a jelen dolgozat tervezett terjedelmén.

Rögtön érdemes megvizsgálni azokat a PL/pgSQL nyelven írt szkripteket, amelyeket a minta adatok (például gépjárművek, dolgozók) generálásához használtam. Ezen táblák

tartalmaznak összetett adatstruktúrákat, így a programok futtatásából mérhető, hogy vajon gyors-e a feldolgozás. Szintén hasznos lehet ez abból a szempontból is, hogy a PL/pgSQL programblokkok teljesítményéről is árulkodni fognak az adatok.

A tesztadatok előállításához olyan tárolt eljárásokat használtam, amelyek egy egész számot várnak paraméterül, és void értékkel térnek vissza (de futás közben természetesen eltárolják a létrejövő rekordokat az adatbázisban.) Emlékeztetőül, a dolgozók generálására szolgáló függvény fejléce:

```
CREATE OR REPLACE FUNCTION dolgozo_generator(integer)
RETURNS void AS ...
```

A paraméter segítségével az határozható meg, hogy hány rekord keletkezzen a futás során. A teljesítmény tesztelése alatt érdemes különböző értékekre is kipróbálni, hogy láthassuk, mennyire skálázódik a futási idő. Természetesen, minden esetben többször is le kell futtatni a programot adott paraméterre, hogy precízebb becslést kaphassunk. Az eredmény a következő táblázat szerinti lett:

```
SELECT dolgozo_generator(N);
```

N=	1. próba (ms)	2. próba (ms)	3. próba (ms)	4. próba (ms)	5. próba (ms)	Átlagosan (ms)	Arány (előzőhöz)
10	53	64	33	151	62	72,6	-
100	82	93	71	17	23	57,2	0,79
1000	193	270	97	90	148	159,6	2,79

Látható, hogy még a jelentősebb mennyiségű rekord előállítása sem vett igénybe túlzottan nagy időt. Bár a kért darabszámban az eltérés soronként tízszeres, azt vehetjük észre, hogy – gyakorlatilag a véletlen folytán – 100 rekord generálása kissé rövidebb ideig tartott, mint 10-é; 1000 sor esetén már szembetűnőbb a különbség, 2,79-szeres – viszont még ez sem nagyságrendi, és az adatbázisok általános tárolási mechanizmusait ismerve logikusnak ható érték.

Az Oracle rendszert hasonló szemszögből vizsgáló, hivatkozott szakdolgozatban ([2]) 30 alkalmazott PL/SQL programmal történő előállítására tisztán relációs megvalósítás esetén 3368 milliszekundumos, míg objektum-relációs implementációnál 3448 ms-os átlagos futási idő olvasható. A tesztek végrehajtása egy hasonló paraméterekkel

rendelkező számítógépen történt; fontos különbség azonban, hogy abban a tesztsorozatban a cache-elést kiküszöbölték.

Ha a fenti utasítást cache nélkül hajtottam volna végre, akkor sem lenne nagyságrendi a különbség, nagyjából 500 ms alatt futott volna le a PL/pgSQL programblokk.

Ez alapján az a bátortalan következtetés vonható le, hogy a PL/pgSQL programok végrehajtási mechanizmusa hatékonyabb, továbbá a visszafogottabb OR eszközkészlet – bár kétségkívül szűkebb – nem megy a teljesítmény rovására.

Érdemes áttekinteni a járművek létrehozásakor kapott eredményeket is:

```
SELECT jarmu_generator(N, 'szemelyauto');
```

N=	1. próba (ms)	2. próba (ms)	3. próba (ms)	4. próba (ms)	5. próba (ms)	Átlagosan (ms)	Arány (előzőhöz)
10	53	37	74	92	66	66,4	-
100	32	34	72	23	62	44,6	0,69
1000	245	205	143	143	132	173,6	3,89

A *jarmu* és a *szemelyauto* relációk közt leszámazotti viszony áll fenn. Látható, hogy az öröklődés (valamint a járművek tankolásait rögzítő tömbök inicializációja) alkalmazása nem jár számottevő teljesítményromlással a hasonló megoldásokat nem tartalmazó *dolgozo* táblába való beszúráshoz képest.

Érdemes letesztelni azt is, hogy mennyi idő alatt megy végbe a járművek új tankolásainak rögzítése. A tankolásokat egy olyan tömbben tároljuk el minden gépjárműre, amelynek az összes sora egy rekord; ezzel gyakorlatilag egy „beágyazott táblát” hoztunk létre.

```
UPDATE jarmu
SET Tankolasok[Jarmu.UjTankolas] =
ROW('2013-05-14', '1035/00078', 40.2)
WHERE Alvazszam = '1M8GDM9A_KP042788';
```

1. próba (ms)	2. próba (ms)	3. próba (ms)	4. próba (ms)	5. próba (ms)	Átlagosan (ms)
11	14	12	41	12	18

A kapott végrehajtási idő egy UPDATE utasításhoz teljesen elfogadható, figyelembe véve, hogy a végrehajtása közben még az *UjTankolas* eljárást is meghívjuk.

Eljárásokról beszélve, hogy a jelen szerény fejezet a lehetőségekhez mérten átfogó legyen, érdemes kipróbálni az OR elven tárolt adatokat feldolgozó metódusokat is.

Egy adott járműre meghatározható a *Fogyasztas* függvénnyel egy adott év összes üzemanyag-fogyasztása, míg az *OsszFogyasztas* függvénnyel a teljes élettartamra vonatkozóan megkapjuk ezt az adatot. Először a következő SQL lekérdezést hajtottam végre, ami egy konkrét jármű fogyasztását határozta meg:

```
SELECT j.Alvazszam, j.OsszFogyasztas
FROM Jarmu j
WHERE j.Alvazszam = '1M8GDM9A_KP042788';
```

Lekért sorok (db)	1. próba (ms)	2. próba (ms)	3. próba (ms)	4. próba (ms)	5. próba (ms)	Átlagosan (ms)
1	12	17	13	15	11	13,6

Ezzel az eredménnyel is lényegében elégedettek lehetünk egy olyan számítógépen, mint amelyen a tesztelés folyt. Próbáljuk ki, hogy mi történik, ha az összes adatbázisban lévő járműre végrehajtjuk ezt a lekérdezést, ráadásul úgy, hogy egyszerre lekérjük a 2012. évi fogyasztást és az összfogyasztást is:

```
SELECT j.Alvazszam, Fogyasztas(j, 2012), j.OsszFogyasztas
FROM Jarmu j;
```

Lekért sorok (db)	1. próba (ms)	2. próba (ms)	3. próba (ms)	4. próba (ms)	5. próba (ms)	Átlagosan (ms)
28811	722	639	612	724	672	673,8

A lekért adatmennyiség itt már jelentősebbnek mondható. Ahogy az a 3.3.3-as fejezetben olvasható, a fogyasztási adatokat lekérő eljárás időigénye lineáris az adott jármű tankolásainak számában. A két lekérdezés időbeli különbsége alapján az mondható, hogy a metódus végrehajtása, s így a tömb feldolgozása csak elenyésző része lehet a teljes műveletsornak, és az objektum-relációs megoldás alkalmazása nem járt teljesítményromlással.

A 3.3.3-as fejezetben olvasható egy olyan lekérdezés is, amellyel megkerüljük a metódushívást. Próbaképpen ezt a verziót is lefuttattam az '1M8GDM9A_KP042788' kulcsú jármű összfogyasztásának meghatározására. Emlékeztetőül, a lekérdezés:

```
SELECT SUM(liter)
FROM (
```

```

SELECT
  (UNNEST(Tankolasok)).Liter
FROM Jarmu
WHERE Alvazszam = '1M8GDM9A_KP042788'
) AS belso;

```

Lekért sorok (db)	1. próba (ms)	2. próba (ms)	3. próba (ms)	4. próba (ms)	5. próba (ms)	Átlagosan (ms)
1	11	17	13	12	16	13,8

Az mondható, hogy a teljesítmény szempontjából a saját függvény megkerülése és a beépített UNNEST(), valamint az alkérdés használata egy rekord lekérésekor nem okozott lényeges eltérést az eredmény kiszámításának időtartamában.

4.1.3. Összesítés

Több példát is láttunk arra, hogy milyen költséggel járhatnak a legáltalánosabban végrehajtott műveletek egy olyan PostgreSQL adatbázisban, amelyben használtunk objektum-relációs eszközöket.

Gyakorlatilag minden tesztet során az vált világossá, hogy a PostgreSQL implementációjában ezek a konstrukciók nem okoznak lényeges teljesítményromlást, így alkalmazásuk ebből a szempontból nem nevezhető aggályosabbnak, mintha egy tisztán relációs megvalósításhoz ragaszkodnánk.

4.2. A PostgreSQL és az Oracle eszközei

A korábban látottak tükrében kimondható, hogy a PostgreSQL valóban számos érdekes lehetőséget biztosít objektum-relációs megoldások implementálására. Ezek „abszolút” hatékonyságát is értékeltük a korábbi fejezetben, viszont továbbra is nyitott a kérdés, hogy vajon az Oracle Database-hez képest *relatív*e mennyire rózsás a helyzetünk.

4.2.1. PostgreSQL tömbök és az Oracle Varray, beágyazott tábla kollekciók

A tömbök párhuzamba állíthatók az Oracle Varray és beágyazott tábla kollekcióival, de a korábban látottak alapján a két adatbázis-kezelő egészen más megoldásokkal él. Míg az Oracle-ben a Varray elemszámának deklarációkor felső korlátot kell adni, PostgreSQL-ben ilyen megkötést nem kell tennünk. További fontos különbség, hogy míg PostgreSQL-ben az egyes elemeket egyesével is manipulálhatjuk, az Oracle ezt – Varray használata esetén – nem teszi lehetővé.

Először tehát érdemes áttekinteni, hogy hogyan történik Oracle-ben a Varray (*variable-size array*) deklarációja és alkalmazása.

Varray-t Oracle-ben a következő módon tudunk létrehozni:

```
CREATE TYPE tipusnev AS VARRAY(hossz) OF tipus;
```

Ezután a tábladefiníciókban megadhatjuk ezt, mint attribútum típust, és beszúrhatunk elemeket az INSERT utasítással.

```
CREATE TABLE tbl (  
    attr1 VARCHAR2(100),  
    attr2 tipusnev  
);  
  
INSERT INTO tbl VALUES  
(adat1, tipusnev(elem1, elem2, ..., elemhossz));
```

Ahogy azt már korábban olvashattuk, a PostgreSQL tömbökkel ellentétben az egyes elemek egyenkénti kezelése nem lehetséges, tehát csak a teljes varray tömböt tudjuk lecserélni. A lekérdezéskor azonban alkalmazható a PostgreSQL UNNEST() függvényéhez hasonló megoldás, azaz itt is elérhető, hogy az összetett adatstruktúra elemszámának megfelelő mennyiségű sorra törjünk szét egy rekordot a már ismert módon:

```
SELECT t.attr1, nt.* FROM tbl t, TABLE(t.attr2) nt;
```

A fenti szintaxis véleményem szerint nehezebben átlátható, mint a PostgreSQL megoldása, de az eredmény szempontjából nincs különbség. Súlyos hiányossággént inkább az róható fel az Oracle-nek, hogy már létrehozott varray-ben nem módosíthatunk vagy törölhetünk elemet, valamint sokszor a fix méret is hátrányos tulajdonság. Néhány esetben persze ez a megközelítés is hasznos lehet, de a PostgreSQL tömbök összességében jobban átgondolt és kényelmesebb eszköznek bizonyulnak.

Térjünk át arra, hogy az Oracle beágyazott tábláival kapcsolatban mit állapíthatunk meg. Beágyazott táblát a következőképpen hozhatunk létre:

```
CREATE TYPE tipusnev AS TABLE OF tipus;
```

A varray-ekhez nagyon hasonlóan ezt is használhatjuk táblák definíciójánál, azonban fontos különbség, hogy ilyenkor a megfelelő attribútumban *típus* típusú elemek egy tetszőleges méretű, rendezetlen halmazát tárolhatjuk, amelyen végrehajtható elemenkénti

adatmanipuláció is. A legfőbb eltérés, hogy a beágyazott tábla tulajdonképpen teljesen külön, valódi adattáblaként tárolódik (a varray nem); emiatt a kollekción tartalmazó tábla definíciójának végén meg kell határoznunk, hogy milyen néven hivatkozzunk erre a külön adatbázis-objektumra.

```
CREATE TABLE tbl (  
    attr1 VARCHAR2(100),  
    attr2 típusnev  
) NESTED TABLE attr2 STORE AS fizikai;
```

Ha ezzel megvagyunk, a „közönséges” INSERT, UPDATE, DELETE utasítások segítségével tudjuk manipulálni magát az egész beágyazott táblát, vagy a TABLE() függvény kihasználásával akár az *egyes elemeket is* [2].

Tulajdonképpen tehát Oracle munkakörnyezetben sem szenvedünk semmiféle értelemben hiányt, de a megvalósítás sokkal körülményesebb; a PostgreSQL tömbjei egyaránt képesek helyettesíteni a varray és beágyazott tábla kollekciónkat. Esetleg egyedül egy szempontból vannak hátrányban: ahogyan a 2.5.5. fejezetben olvasható, a törlés nem kimondottan jól implementált, beágyazott táblánál viszont a DELETE utasítással könnyedén elvégezhető ugyanez a feladat.

4.2.2. Oracle objektumtípusok, metódusok és a PostgreSQL eszközei

Az Oracle-ben az objektumtípusok – tulajdonképpen osztályok – kezelése jóval kiforrottabb szintaktikai szempontból, mint PostgreSQL-ben; szemantika és tárolás szempontjából azonban nincs lényeges eltérés. Túlmutatna a jelen dolgozat tematikáján, hogy teljes körű leírást adjunk arról, miként kezelhetők az objektumok Oracle-ben; ezért elegendő a leglényegesebb részletekre fókuszálni.

Ha sorobjektumokból álló táblát kívánunk létrehozni, először létre kell hoznunk egy osztályt a következő utasítással:

```
CREATE TYPE típusnev AS OBJECT (  
    [attribútumok [,]]  
    [MEMBER PROCEDURE proc1 [, ... ]]  
);
```

Ezután ebből tábla készíthető:

```
CREATE TABLE tbl OF típusnev;
```

Látható, hogy Oracle-ben a PostgreSQL-nél elegánsabban valósul meg az enkapszuláció, azaz az adatok és a rajtuk végzett műveletek egységbe zárása; a

metódusokat rögtön az attribútumok felsorolása után listázzuk [2]. A metódusok törzsét természetesen külön utasítással adhatjuk meg:

```
CREATE TYPE BODY típusnev AS  
  MEMBER PROCEDURE proc1 IS {PL/SQL blokk};
```

Mindezzel szemben a jelen dolgozatban tárgyalt rendszernél tulajdonképpen csak tárolt eljárásokat tudunk létrehozni, amik paraméterlistájuk alapján képesek együttműködni azzal a típussal, amelyikre értelmezni szeretnénk őket.

Érdekesebb kérdés, hogy az öröklődés kétféle megvalósítását is összevegyük. Emlékeztetőül, a PostgreSQL-ben a tábladefiníciók végén elhelyezett INHERITS kulcsszóval tudunk leszármaztatni. A szintaxis:

```
CREATE TABLE os ( ... );  
CREATE TABLE leszarmazott ( ... ) INHERITS os;
```

Ekkor az összes olyan attribútum, ami *os*-ben szerepelt, megjelenik a leszármazottban is, viszont a megszorítások (kulcs, elsődleges kulcs stb.) nem öröklődnek. Ha az őstípusba eső egyedeket kérdezzük le, a belőle származó táblákban tárolt rekordok is megjelennek az eredményhalmazban [1]. Nem zárható ki táblánkénti kulcs megszorításokkal, hogy egy ilyen lekérdezés során kapjunk látszólag teljesen azonos sorokat még akkor is, ha az őstáblára él egy elsődleges kulcs megszorítás.

Akárcsak a PostgreSQL-ben, az Oracle-ben is tetszőleges mélységű öröklődési fát kreálhatunk [2]. Új objektumtípus létrehozásánál az UNDER kulcsszó után kell megadni, hogy mely ősből kívánjuk leszármaztatni azt. Ekkor az őst a NOT FINAL módosítóval kell ellátni, amely jelzi, hogy belőle származtatunk. Az öröklődés csakis egyszeres lehet, azaz egy osztálynak egyetlen ős adható meg. A polimorfizmus (többalakúság) szépen megvalósított, őstípusra hivatkozásnál felhasználható konkrét értékként tetszőleges leszármazott típusba eső példány. A metódusok is öröklődnek, és felül is bírálhatók (OVERRIDING kulcsszóval) a leszármazottakban. Vegyünk egy egyszerű példát, amikor az általános emberből származtatjuk a dolgozót (ugyanennek a PostgreSQL megvalósítása olvasható a jelen dolgozat 2.3.1. fejezetében:)

```
CREATE TYPE ember AS OBJECT (  
  id NUMBER,  
  nev VARCHAR2(100),  
  cím VARCHAR2(100)
```

```
) NOT FINAL;
```

```
/
```

```
CREATE TYPE dolgozo UNDER ember (
```

```
    fizetes NUMBER,
```

```
    beosztas VARCHAR2(100)
```

```
);
```

```
/
```

```
CREATE TABLE embertbl OF ember;
```

```
CREATE TABLE dolgozotbl OF dolgozo;
```

A PostgreSQL megvalósításában, ha beszúrtunk egy rekordot a dolgozók táblájába, a rekord (csonkolt) formában megjelent az emberek adatait őrző tábla lekérdezésekor is, vagy, ha ez számunkra nemkívánatos, használhattuk az ONLY módosítót. Az Oracle stratégiája ebben is más – a létrehozott táblák közt semmiféle ilyen jellegű kapcsolat nem jön létre, ha a *dolgozotbl* táblába beszúrunk egy rekordot, az nem fog megjelenni az *embertbl* táblában. Ez – figyelembe véve az eltérő alapelgondolásokat – teljesen logikus megfontolásnak is mondható, de ismét igaz, hogy néha pedig pont a PostgreSQL megoldása praktikus.

Látható, hogy az Oracle a (sor)objektumok kezelését és az öröklődést sokkal kifinomultabban implementálja, mint a dolgozatban vizsgált rendszer; ebből a szempontból az mondható, hogy a „relációs adatbázis-kezelő” és „objektum-relációs adatbázis-kezelő” fogalmak közti skálán a PostgreSQL sokkal inkább közelebb esik az elsőhöz, mint a másodikhoz. Nem elhanyagolandó azonban, hogy az OR eszközöket Oracle környezetben is csak ritkán használják ki a maguk teljességében, így a leggyakrabban felmerülő, objektum-relációsan megoldani kívánt problémákhoz a PostgreSQL kínálta lehetőségek is elégségesek lehetnek.

4.3. Betekintés a jövőbe

A PostgreSQL magát a saját honlapján „a világ legfejlettebb nyílt forrású adatbázis-kezelő rendszerének” aposztrofálja („*The world's most advanced open source*

database.”) Szakmaiatlan lenne ezen állítás igazságtartalmát boncolgatni, azonban tény, hogy a rendszer fejlesztése napjainkig töretlen és igen nagy lendületű.

A dolgozat írása során a PostgreSQL 9.2.1-es verziót használtam, azonban röviddel a munka elkészülte előtt megjelent a 9.3-as verzió. Ebben számos újítást fedezhetünk fel, amelyek közt egy az objektum-relációs lehetőségek témáját is érinti: ez a JSON adattípus kezelésének széleskörűen támogatottá tétele.

A JSON a JavaScript Object Notation (JavaScript objektumleírás) rövidítése. Tulajdonképpen nem másra szolgál, mint összetett adatstruktúrák ember által is olvasható szöveges reprezentációjára. Összetett adatstruktúra alatt itt egyaránt érthetünk rekordokat tetszőleges számú mezővel, valamint kulcs-érték párokból álló asszociatív tömböket, amelyek szinte akármilyen típusú elemi adatot tartalmazhatnak.

A JSON az utóbbi években általános csereformátummá vált, amelyet előszeretettel alkalmaznak például kliens-szerver rendszerekben; felfogható úgy is, mint az XML egy alternatívája.

Tegyük fel, hogy egy dolgozó adatait tároló rekordot szeretnénk JSON-ben kódolni. Ekkor a következőt kapjuk:

```
{
  "nev": "Kis Aladár",
  "fizetes": 140000,
  "cim": {
    "irsz": "6500",
    "varos": "Baja",
    "utca": "Cserkész utca 65."
  },
  "kiadottKulcsok": [
    {
      "hova": "iroda",
      "sorszam": "123"
    },
    {
      "hova": "raktár",
      "sorszam": "29"
    }
  ]
}
```

A fenti kódban egyaránt láthatjuk, hogy a rekordon belül elhelyezhető összetett adatszerkezet is (cím), valamint tömb is (kiadottKulcsok.)

Az ilyen, úgymond (félig)strukturált adatok tárolása adatbázisban jellemzően relációs eszközökkel megoldott, azonban a PostgreSQL-ben felvehető JSON típusú attribútum is egy táblában:

```
CREATE TABLE json_pelda (  
    ...,  
    pelda JSON  
);
```

A PostgreSQL 9.3 nagy újdonsága, hogy beépítetten kezeli az ilyen típusú összetett adatokat, hivatkozhatunk például egy-egy mezőre és kinyerhetjük az ott tárolt értékeket. A JSON attribútumok használatának további előnye, hogy az adatok manipulációjakor történik validáció, azaz a rendszer ellenőrzi, hogy érvényes objektumleírást adtunk-e meg (szemben például az egyszerű, TEXT-ként történő tárolással) [14].

Számos jól felhasználható JSON-operátor került bevezetésre. Ezek közül néhány a következő:

<code>j -> text</code>	A <i>j</i> JSON-kódolt struktúra <i>text</i> mezőjének értékét adja
<code>j ->> text</code>	a <i>j</i> struktúra <i>text</i> mezőjét szöveggé konvertálja
<code>j #> text_array</code>	a <i>j</i> struktúrából a <i>text_array</i> PostgreSQL szövegtömbbel megadott úton lévő struktúrát adja vissza
<code>j #>> text_array</code>	ua., mint a <i>#></i> operátor, de szöveget ad vissza

Mindezen túl több olyan beépített függvényt is bevezettek, amelyek a JSON-kódolt adatok feldolgozását és adatbázisban tárolását nagyban leegyszerűsítik.

4.4. Záró gondolatok

Mindent összevetve hasznos és érdekes témának bizonyult a PostgreSQL beható tanulmányozása. Nem pusztán az objektum-relációs eszközei miatt mondható ez el, hanem azért is, mert egy jól kidolgozott, könnyen kezelhető, meglepően funkciógazdag rendszert ismerhettem meg, amely ráadásul teljesen ingyenes, nyílt forrású.

Ahogy az a 4.3-as alfejezetben olvashattuk, a fenti jó tulajdonságokon túl a PostgreSQL fejlődése sem torpant meg, és még napjainkban is bővül további objektum-relációs eszközökkel (valamint egyéb, jelen dolgozat témájától idegen lehetőségekkel.)

Végítéletként az mondható el, hogy amennyiben egy új, nagyobb lélegzetvételű projekthez mögöttes adatbázist keresünk, véleményem szerint érdemes fontolóra venni a PostgreSQL-t, mint egy opciót az adatbázis-kezelők színes piacán.

Irodalomjegyzék

- [1] John C. Worsley – Joshua D. Drake: Practical PostgreSQL. O'Reilly & Associates, 2002. ISBN 1-56592-846-6
- [2] Vasas Szabolcs: Objektum-relációs eszközök hatékonyságának vizsgálata Oracle környezetben. Szakdolgozat, SZTE, 2010.
- [3] PostgreSQL 9.2.1 Documentation. The PostgreSQL Global Development Group, 2012.
- [4] Chris Travers: Intro To PostgreSQL as Object-Relational Database Management System. *Elérhető a <http://ledgersmbdev.blogspot.hu/2012/08/intro-to-postgresql-as-object.html> címen (megtekintve 2012. december 10.)*
- [5] Jeffrey D. Ullman – Jeniffer Widom: Adatbázisrendszerek – alapvetés. Panem, 2009. ISBN 978-963-545-481-5
- [6] Leo Hsu – Regina Obe: Using RETURN TABLE vs. OUT parameters. Postgres OnLine Journal. *Elérhető a <http://www.postgresonline.com/journal/archives/201-Using-RETURNS-TABLE-vs.-OUT-parameters.html> címen (megtekintve 2013. február 20.)*
- [7] Simon Riggs – Hannu Krosing: PostgreSQL 9 Administration Cookbook. Packt Publishing, 2010. ISBN 978-1-849510-28-8
- [8] Npgsql: User's Manual. The Npgsql Development Team, 2009. *Elérhető a <http://npgsql.projects.pgfoundry.org/docs/manual/UserManual.html> címen (megtekintve 2013. április 16.)*
- [9] Oracle Database Online Documentation 11g Release 1 (11.1). Oracle Corporation, 2013. *Elérhető a <http://www.oracle.com/pls/db111/homepage> címen (megtekintve 2013. április 16.)*
- [10] Visual C# Documentation (Microsoft Visual Studio 2010.) Microsoft, 2013. *Elérhető a [http://msdn.microsoft.com/en-us/library/vstudio/kx37x362\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/vstudio/kx37x362(v=vs.100).aspx) címen (megtekintve 2013. április 16.)*

- [11] Kevin Loney: ORACLE DATABASE 10g Teljes Referencia. Panem Kiadó, 2006. ISBN 978-963-5454-396
- [12] Paul Nielsen: Why use stored procedures? SQLBlog.com. *Elérhető a http://sqlblog.com/blogs/paul_nielsen/archive/2009/05/09/why-use-stored-procedures.aspx címen (megtekintve 2013. április 26.)*
- [13] Tony Patton: Determine when to use stored procedures vs. SQL in the code. TechRepublic. *Elérhető a <http://www.techrepublic.com/article/determine-when-to-use-stored-procedures-vs-sql-in-the-code/5766837> címen (megtekintve 2013. április 26.)*
- [14] JSON Functions and Operators, PostgreSQL 9.3 Documentation. The PostgreSQL Global Development Group, 2013. *Elérhető a <http://www.postgresql.org/docs/9.3/static/functions-json.html> webcímen (megtekintve 2013. május 14.)*

Nyilatkozat

Alulírott **Cser Lajos** programtervező informatikus BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Tanszékcsoport **Képfeldolgozás és Számítógépes Grafika Tanszékén** készítettem, programtervező informatikus BSc diploma megszerzése érdekében. **Kijelentem**, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel. **Tudomásul veszem**, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Informatikai Tanszékcsoport könyvtárában, a helyben olvasható könyvek között helyezik el.

Szeged, 2013. május 14.

Cser Lajos

Függelék

A jelen szakaszban olyan kiegészítéseket helyeztem el, amelyek zavaróak lettek volna a szövegtörzsben, de mindenképpen helyet érdemelnek a dolgozatban.

Tömböt felhasználó PL/pgSQL eljárás

A következő eljárás a tömbökről szóló fejezetben megadott 'konyv' táblából lekérdezi egy adott könyv szerzőit és azok számát úgy, hogy egy felsorolást tartalmazó szöveggel tér vissza (ami pl. egyből kiíratható a felhasználó képernyőjére.)

```
CREATE OR REPLACE FUNCTION konyvszerzok(melyik CHARACTER VARYING) RETURNS TEXT AS $$
DECLARE
    szerzo_sz INTEGER = 0;
    i INTEGER = 1;
    egy_sor CHARACTER VARYING;
    osszesites TEXT = 'A választott könyv szerzői: ';
BEGIN
    IF (SELECT COUNT(*) FROM konyv WHERE isbn = melyik) <> 0 THEN
        BEGIN
            SELECT array_length(szerzo, 1) INTO szerzo_sz FROM konyv WHERE isbn =
melyik;
            WHILE i <= szerzo_sz LOOP
                SELECT szerzo[i] INTO egy_sor FROM konyv WHERE isbn = melyik;
                osszesites = osszesites || egy_sor;
                IF i <> szerzo_sz THEN
                    osszesites = osszesites || ', ';
                ELSE
                    osszesites = osszesites || ', összesen: ';
                END IF;
                i = i + 1;
            END LOOP;
            osszesites = osszesites || szerzo_sz;
            END;
        ELSE
            osszesites = 'Nem létezik ilyen könyv!';
        END IF;
        RETURN osszesites;
    END;
$$ LANGUAGE plpgsql;
```

Futtatás és eredmény:

```
# SELECT konyvszerzok('ISBN 1-56592-846-6');
-----konyvszerzok-----
A választott könyv szerzői: John C. Worsley, Joshua D. Drake, összesen: 2
```