

Capitolul 19. Obiectual-Relațional în SQL

Fuziunea modelelor relațional și obiectual de organizare a datelor este o temă ce preocupă teoreticienii și practicienii bazelor de date de la finalul anilor optzeci. Ca răspuns la cerințele dezvoltatorilor de aplicații și concurența unor limbaje precum OQL (Object Query Language), versiunea SQL:1999 a standardului a adus noutăți radicale în materie de gestiune a obiectelor în bazele de date relaționale. De la această versiune, putem spune SQL înglobează elementele esențiale ale modelului obiectual-relațional (O-R). Perioada de redactare a primei versiuni obiectual-relaționale a standardului coincide cu cea de efervescență O-R în rândul producătorilor de SGBD-uri. A doua parte a anilor nouăzeci și imediat după 2000 sunt anii în care producătorii au inclus în serverele BD *lame* și *cartușe de date* (*data blades*, *data cartridges*) – extensii ale tipurilor primitive dedicate stocării și manipulării de texte, imagini, sunete, hărți de mari dimensiuni. Apoi... liniște. După ce au făcut atâta tam-tam pe seama lipsei opțiunilor orientate pe obiecte (OO) în BD relaționale, și, în fine, le-au dobândit, dezvoltatorii de aplicații au ajuns să se mulțumească cu MySQL-ul. Astăzi, apatia practicienilor față de bazele de date O-R este evidentă, iar lipsa documentației este cronică. SQL Server este cel mai rece la avansurile tipurilor, ierarhiei de tipuri, metodelor, moștenirii și polimorfismului. PostgreSQL are implementată doar o cvasi-ierarhie de tabele case se regăsește, pe un plan dezvoltat, și în DB2. DB2-ul este mai generos în materie de opțiuni O-R, dar nu am reușit să aflu prea multe surse informaționale dedicate subiectului. Cel mai bine plasat este Oracle, atât ca opțiuni de lucru, cât și ca documentație¹.

Melton și Simon se opresc asupra a două tipuri utilizator, *distincte* și *structurate*². La acestea adăugă tipurile referință (*reference*) și masiv (*array*) care pot fi incluse în tipurile structurate. *Tipurile distincte* sunt particularizări ale unor tipurilor sistem „clasice” (primitive). *Tipurile structurate* sunt cele mai apropiate de percepția obișnuită asupra tipurilor definite utilizator (UDT – User Defined Types), fiind ingredientul esențial pentru crearea tabelelor „tipate”³ (*typed table*) pe care le vom discuta în paragraful 19.3. În tabelele „tipate” fiecare linie reprezintă o instanță a unui tip (clasă) structurat, iar atributele tabelului sunt, de fapt, atributele tipului pe care a fost definită. O instanță a unui *tip referință* este un pointer către un obiect dintr-o clasă. Pentru o bază de date, este important de știut că țința unei valori referință este o linie dintr-o tabelă „tipată”. Tipurile referință

¹ [Oracle 2008-1],

² [Melton & Simon], pp.45-47

³ Îmi cer scuze și pentru barbarismul lingvistic *tabele tipate*, dar n-am găsit ceva mai rezonabil.

pot fi folosite pentru instituirea unor legături părinte-copil dintre două tipuri, fiind similare legăturii părinte-copil dintre două tabele din modelul relațional, fără însă a include restricția referențială propriu-zisă. Un *masiv* este o structură ce conține o colecție ordonată de elemente (date) în care fiecare element poate fi referit/accesat prin poziția (ordinea) sa în cadrul colecției. Fiecare element/componentă este definit pe un alt tip de date, primitiv (simplu) sau structurat. Un masiv are o dimensiune maximă, definită explicit sau implicit (de limitele sistemului).

19.1. Tipuri distincte și structurate de date

Tipurile distincte au o importanță redusă în economia aplicațiilor cu baze de date, nefiind implementate decât în DB2 și SQL Server, iar în SQL Server, chiar dacă sunt implementate, nu pot fi folosite la crearea tabelelor. Ideea de bază este de a crea subtipuri ale unor tipuri de bază, pentru a evita compararea lor sau combinarea lor accidentală în expresii. Astfel, definim două tipuri structurate, *t_cantitate* și *t_pret*, pe care le folosim la crearea tablei TAB_TEST:

```
CREATE TYPE t_cantitate AS DECIMAL (10,3) FINAL ;
CREATE TYPE t_pret AS DECIMAL(9,2) FINAL ;
```

```
CREATE TABLE tab_test (
    produs VARCHAR(30),
    cantitate t_cantitate,
    pret_unit t_pret
);
```

Fiind de tipuri diferite, atributele Cantitate și Pret_Unit nu pot fi comparate, interogarea următoare soldându-se cu un mesaj de eroare:

```
SELECT *
FROM tab_test
WHERE cantitate > pret_unit
```

Din păcate, nici interogarea următoare nu ar duce-o mai bine:

```
SELECT t.*, cantitate * pret_unit AS val
FROM tab_test t
```

produsul valorilor celor două atribute fiind posibil doar cu ajutorul funcției CAST:

```
SELECT t.*, CAST (cantitate AS DECIMAL(10,3)) *
           CAST (pret_unit AS DECIMAL(9,2)) AS val
FROM tab_test t
```

19.1.1. Tipuri distincte și structurate în PostgreSQL

Comanda de creare a unui nou tip de date pentru BD curentă este CREATE TYPE. Cel care lansează (cu succes) comanda devine proprietarul tipului, tip care poate fi partajat de către toți utilizatorii bazei. În afara colecțiilor, pe care le discutăm în paragraful 19.2.1, documentația PostgreSQL face trimitere la trei categorii de tipuri utilizator:

- tipuri compozite – echivalentele tipurilor structurate;
- tipuri enumerate (ENUM)
- tipuri de bază, derivate și care modifică tipurile implementate standard în PostgreSQL.

Tipurile *enumerated* (ENUM) sunt utile atunci când valorile posibile ale atributelor/variabilelor definite pe această categorie fac parte dintr-o listă, cum ar fi sexul, starea civilă, zilele săptămânii etc.

```
CREATE TYPE t_sex AS ENUM ('F', 'M');
```

```
CREATE TYPE t_regiune AS ENUM ('Banat', 'Dobrogea', 'Moldova',  
                                'Muntenia', 'Oltenia', 'Transilvania');
```

În privința tipurilor structurate, să începem cu un exemplu simplu, dedicat județelor din țara noastră:

```
CREATE TYPE t_judet AS (  
    jud CHAR(2),  
    judet VARCHAR(30),  
    regiune t_regiune  
);
```

Continuăm cu un tip de date care ne va ajuta într-o funcție ce furnizează un set de înregistrări necesar unui raport:

```
CREATE TYPE facturi_detaliat AS (  
    nrfact INTEGER,  
    datafact DATE,  
    codcl SMALLINT,  
    dencl VARCHAR(30),  
    linie SMALLINT,  
    codpr SMALLINT,  
    denpr VARCHAR(30),  
    cantitate NUMERIC(10),  
    pretunit NUMERIC(9,2),  
    valfaratva NUMERIC(14,2),  
    tva NUMERIC(13,2),  
    valcutva NUMERIC(15,2)  
);
```

Funcția care folosește acest tip în clauza RETURNS este *f_facturi_detaliat_clienti* și este prezentată în listing 19.1. Nu există niciun element de dificultate care să merite discuții suplimentare.

Listing 19.1. O funcție de folosește tipul nou definit

```
CREATE OR REPLACE FUNCTION f_facturi_detaliat_clienti (codcl_clienti.codcl%TYPE)
    RETURNS SETOF facturi_detaliat AS $$
    SELECT f.nrfact, datafact, f.codcl, dencl, linie, codpr,
        f.denpr(codpr), cantitate, pretunit, cantitate * pretunit,
        cantitate * pretunit * f.proctva(codpr),
        cantitate * pretunit * (1 + f.proctva(codpr))
    FROM clienti c
        INNER JOIN facturi f ON c.codcl=f.codcl
        INNER JOIN liniifact lf ON f.nrfact=lf.nrfact
    WHERE f.codcl = $1
    $$ LANGUAGE SQL ;
```

Folosirea funcției într-o interogare este, după cum știți, banală:

```
SELECT * FROM f_facturi_detaliat_clienti (CAST (1002 AS SMALLINT)) t
```

Următorul exemplu are, oarecum, un caracter reparator. În capitolul 11 PostgreSQL a stat pe tușă, nefiind dotat cu nicio opțiune/funcție OLAP. De aceea definim un tip și o funcție (listing 19.2) care să se apropie de logica operatorului CUBE.

```
CREATE TYPE t_cub_client_produs_zi AS (
    client VARCHAR(35),
    produs VARCHAR(35),
    zi VARCHAR(35),
    vinzari NUMERIC (14,2),
    nr INTEGER
);
```

Listing 19.2. O funcție cât un CUBE

```
CREATE OR REPLACE FUNCTION f_cub_clienti_produs_zi (
    datai DATE, dataf DATE) RETURNS SETOF t_cub_client_produs_zi AS
    $$
    SELECT ' ' || DenCl, ' ' || DenPr, ' ' || DataFact,
        SUM(Cantitate * PretUnit * (1+ProcTVA)), CAST (NULL AS INTEGER)
    FROM clienti
        INNER JOIN facturi ON clienti.CodCl=facturi.CodCl
        INNER JOIN liniifact ON facturi.NrFact=liniifact.NrFact
        INNER JOIN produse ON liniifact.CodPr=produse.CodPr
    WHERE DataFact BETWEEN COALESCE($1, DATE'2000-01-01') AND
        COALESCE($1, DATE'2010-12-31')
    GROUP BY DenCl, DenPr, DataFact
    UNION
    SELECT ' ' || DenCl, ' ' || DenPr, 'subtotal client/produs',
        SUM(Cantitate * PretUnit * (1+ProcTVA)), CAST (NULL AS INTEGER)
    FROM clienti
        INNER JOIN facturi ON clienti.CodCl=facturi.CodCl
        INNER JOIN liniifact ON facturi.NrFact=liniifact.NrFact
        INNER JOIN produse ON liniifact.CodPr=produse.CodPr
    WHERE DataFact BETWEEN COALESCE($1, DATE'2000-01-01') AND
        COALESCE($1, DATE'2010-12-31')
    GROUP BY DenCl, DenPr
```

```

        UNION
SELECT ' ' || DenCl, 'subtotal client/zi' , ' ' || DataFact,
      SUM(Cantitate * PretUnit * (1+ProcTVA)), CAST (NULL AS INTEGER)
FROM clienti
      INNER JOIN facturi ON clienti.CodCl=facturi.CodCl
      INNER JOIN liniifact ON facturi.NrFact=liniifact.NrFact
      INNER JOIN produse ON liniifact.CodPr=produse.CodPr
WHERE DataFact BETWEEN COALESCE($1, DATE'2000-01-01')
      AND COALESCE($1, DATE'2010-12-31')
GROUP BY DenCl, DataFact
        UNION
SELECT ' subtotal produs/zi', ' ' || DenPr, ' ' || DataFact,
      SUM(Cantitate * PretUnit * (1+ProcTVA)), CAST (NULL AS INTEGER)
FROM facturi
      INNER JOIN liniifact ON facturi.NrFact=liniifact.NrFact
      INNER JOIN produse ON liniifact.CodPr=produse.CodPr
WHERE DataFact BETWEEN COALESCE($1, DATE'2000-01-01') AND
      COALESCE($1, DATE'2010-12-31')
GROUP BY DenPr, DataFact
        UNION
SELECT ' ' || DenCl, 'total client' , '****', SUM(Cantitate * PretUnit * (1+ProcTVA)),
      CAST (NULL AS INTEGER)
FROM clienti
      INNER JOIN facturi ON clienti.CodCl=facturi.CodCl
      INNER JOIN liniifact ON facturi.NrFact=liniifact.NrFact
      INNER JOIN produse ON liniifact.CodPr=produse.CodPr
WHERE DataFact BETWEEN COALESCE($1, DATE'2000-01-01') AND
      COALESCE($1, DATE'2010-12-31')
GROUP BY DenCl
        UNION
SELECT ' total produs ' , ' ' || DenPr , '****',
      SUM(Cantitate * PretUnit * (1+ProcTVA)), CAST (NULL AS INTEGER)
FROM facturi
      INNER JOIN liniifact ON facturi.NrFact=liniifact.NrFact
      INNER JOIN produse ON liniifact.CodPr=produse.CodPr
WHERE DataFact BETWEEN COALESCE($1, DATE'2000-01-01') AND
      COALESCE($1, DATE'2010-12-31')
GROUP BY DenPr
        UNION
SELECT '****', ' total zi' , ' ' || DataFact,
      SUM(Cantitate * PretUnit * (1+ProcTVA)), CAST (NULL AS INTEGER)
FROM facturi
      INNER JOIN liniifact ON facturi.NrFact=liniifact.NrFact
      INNER JOIN produse ON liniifact.CodPr=produse.CodPr
WHERE DataFact BETWEEN COALESCE($1, DATE'2000-01-01') AND
      COALESCE($1, DATE'2010-12-31')
GROUP BY DataFact
        UNION
SELECT '==TOTAL GENERAL==', '=====', '=====',
      SUM(Cantitate * PretUnit * (1+ProcTVA)), CAST (NULL AS INTEGER)
FROM facturi
      INNER JOIN liniifact ON facturi.NrFact=liniifact.NrFact
      INNER JOIN produse ON liniifact.CodPr=produse.CodPr
WHERE DataFact BETWEEN COALESCE($1, DATE'2000-01-01') AND
      COALESCE($1, DATE'2010-12-31')
ORDER BY 1,2,3 ;
$$ LANGUAGE sql ;

```

```
SELECT * FROM f_cub_clienti_produs_zi (NULL, NULL)
```

Suntem obligați să facem două precizări vecine cu dezmințirea. Mai întâi, funcția de mai sus nu este câtuși de puțin generală, referindu-se, strict, la facturi/vânzări. În al doilea rând, în loc de a crea tipul pentru a-l folosi în clauza RETURNS a funcției, puteam scrie direct RETURNS SETOF RECORD.

Ei, și acum ajungem la ceva mai interesant, ceea ce în limbajul contabililor se numește „fișă-șah”. Noi o să-i spunem *situație matriceală a vânzărilor fiecărui produs pentru fiecare client*, adică ceva de genul raportului din figura 19.1.

client character varying	vnz_prod_1 numeric	vnz_prod_2 numeric	vnz_prod_3 numeric	vnz_prod_4 numeric	vnz_prod_5 numeric	vnz_prod_6 numeric	vnz_prod_8 numeric
Client 1 SRL	2361555.0000	3352077.0000			34025670.0000		571200.0000
Client 2 SA		956475.0000	11900.0000				
Client 3 SRL		332668.0000	266560.0000	568653.0000	45169425.0000		
Client 4		697000.5000			27740328.0000		
Client 5 SRL		746595.5000	397162.5000				
Client 6 SA		244923.0000		167260.5000	17652936.0000		
Client 7 SRL		1064385.0000					
Client destul de nou							

Figura 19.1. Un raport interesant

Mai întâi, redactăm o funcție care să primească doi parametri, un cod de client și un cod de produs și să returneze suma vânzărilor din acel produs către clientul respectiv – vezi listing 19.3.

Listing 19.3. Funcție simplă ce calculează vânzările dintr-un produs către un client

```
CREATE OR REPLACE FUNCTION f_client_produs (codcl SMALLINT, codpr SMALLINT,
datai DATE, dataf DATE) RETURNS NUMERIC AS
$$
SELECT SUM(cantitate * pretunit * (1 + f_proctva($2)))
FROM facturi f INNER JOIN liniifact lf ON f.nrfact=lf.nrfact
WHERE codcl = $1 AND codpr = $2 AND
DataFact BETWEEN COALESCE($3, DATE'2000-01-01') AND
COALESCE($4, DATE'2010-12-31');
$$ LANGUAGE sql;
```

Chestiunea cea mai dificilă ține de faptul că, în timp, în baza de date pot apărea clienți și produse noi, deci raportul din figura 19.1 nu are o structură fixă. De aceea, realizăm o funcție-procedură care să creeze dinamic un tip care stă la baza raportului, tip pe care-l vom denumi *t_temp*. Acesta va avea un prim atribut fix, *client*, și un număr variabil de attribute, în funcție de câte produse există în tabela cu același nume la momentul execuției funcției. Pentru asigurarea generalității, ne înfruptăm din SQL-ul dinamic.

Listing 19.4. Funcție-procedură ce crează, dinamic tipul T_TEMP

```
CREATE OR REPLACE FUNCTION f_create_type_t_temp () RETURNS VOID AS
$$
DECLARE
v_sir VARCHAR (1000);
```

```

rec_prod RECORD ;
BEGIN
EXECUTE 'DROP TYPE IF EXISTS t_temp CASCADE ' ;
v_sir := 'CREATE TYPE t_temp AS (client VARCHAR ' ;
FOR rec_prod IN SELECT CodPr FROM produse ORDER BY CodPr LOOP
v_sir := v_sir || ', vnz_prod_' || LTRIM(TO_CHAR(rec_prod.CodPr,'999999')) || ' NUMERIC ' ;
END LOOP ;
v_sir := v_sir || ') ' ;
EXECUTE v_sir ;
END ;
$$ LANGUAGE plpgsql ;

```

Apelăm funcția:

```
SELECT f_create_type_t_temp()
```

Clientul pgAdmin nu este prea darnic, așa că, pentru a vedea structura tipului (figura 19.2) folosim dicționarul de date:

```

SELECT * FROM pg_attribute
WHERE attrelid = (SELECT typrelid FROM pg_catalog.pg_type
WHERE typename='t_temp')

```

```

SELECT * FROM pg_attribute WHERE attrelid =
(SELECT typrelid FROM pg_catalog.pg_type WHERE typename='t_temp')

```

	attrelid oid	attname name	atttypid oid	attstattarget integer	attlen smallint	attnum smallint
1	35058	client	1043	-1	-1	1
2	35058	vnz_prod_1	1700	-1	-1	2
3	35058	vnz_prod_2	1700	-1	-1	3
4	35058	vnz_prod_3	1700	-1	-1	4
5	35058	vnz_prod_4	1700	-1	-1	5
6	35058	vnz_prod_5	1700	-1	-1	6
7	35058	vnz_prod_6	1700	-1	-1	7
8	35058	vnz_prod_8	1700	-1	-1	8

Figura 19.2. Atributele tipului T_TEMP

Acum putem redacta o funcție, pe care o vom denumi *f_dinamic_cl_prod*, - vezi listing 19.5. Mai întâi, construim șirul de caractere care, executat, va extrage conținutul raportului. Apoi vom declara o variabilă cursor (*cursor1 REFCURSOR*) și o deschidem specificând imensă frază *SELECT* stocată în șir (*OPEN cursor1 FOR EXECUTE v_sir*). Furnizarea, pe rând, a fiecărei înregistrări din variabila cursor se realizează prin *RETURN NEXT*.

Listing 19.5. Funcție ce returnează, la execuție, raportul fin figura 19.1

```

CREATE OR REPLACE FUNCTION f_dinamic_cl_prod() RETURNS SETOF t_temp AS
$$
DECLARE
v_sir VARCHAR (1000) ;
rec_prod RECORD ;
rec_RECORD ;

```

```

    cursor1 REFCURSOR ;
BEGIN
    v_sir := 'SELECT DenCl ' ;
    FOR rec_prod IN SELECT CodPr, DenPr FROM produse ORDER BY DenPr LOOP
        v_sir := v_sir || ', f_client_produ (codcl, CAST ('|| rec_prod.CodPr ||
            ' AS SMALLINT), null, null) AS "vnz_prod_"||rec_prod.DenPr ||"' ;
    END LOOP ;
    v_sir := v_sir || ' FROM clienti ' ;

    OPEN cursor1 FOR EXECUTE v_sir ;
    LOOP
        FETCH cursor1 INTO rec_ ;
        IF NOT FOUND THEN
            EXIT ;
        END IF ;
        RETURN NEXT rec_ ;
    END LOOP ;

    RETURN ;
END ;
$$ LANGUAGE plpgsql ;

```

Acum, la apelul funcției:

```
SELECT * FROM f_dinamic_cl_prod ()
```

avem toate șansele să obținem raportul râvnit.

PostgreSQL nu permite definirea ierarhiilor de tipuri, ci numai ierarhii de tabele, subiect pe care-l vom ignora, probabil pe nedrept, și în DB2. Dacă în paragrafele următoare vom defini un tip rădăcină – *tip_adresa_generala* și câteva subtipuri ale acestuia – *tip_adresa_sat*, *tip_adresa_oras*, *tip_adresa_bloc* – în PostgreSQL, pentru evidența analitică a tuturor componentelor unei adrese vom crea unul singur în care vom „îngrămădi” toate atributele/proprietățile:

```

CREATE TYPE tip_adresa AS (
    codpostal CHAR(6),
    localitate VARCHAR(25),
    comuna VARCHAR(30),
    judet t_judet,
    strada VARCHAR(35),
    nr VARCHAR(10),
    bloc VARCHAR(20),
    scara VARCHAR(10),
    etaj VARCHAR(10),
    apartament NUMERIC(4)
);

```


19.1.2. Tipuri simple în Oracle

Oracle candidează cu cele mai mari șanse la categoria „cel mai darnic server BD în materie de lucru cu obiecte”, după cum vedea până la finalul acestui capitol. În cele ce urmează vom crea doar câteva tipuri simple, fără metode asociate. Începem un tip banal, asociat codului numeric personal:

```
CREATE OR REPLACE TYPE tip_cnp AS OBJECT (
    cnp NUMBER(13)
) FINAL
```

Pentru a mai câștiga în atractivitate, introducem în discuție o ierarhie de tipuri. Cazul asupra căruia ne vom opri este cel al adreselor localităților din țara noastră. Există diferențe semnificative între adresele de la sat și cele de la oraș. Toate, însă, au un cod poștal, o denumire de localitate și o denumire de județ, așa că vom defini un tip rădăcină denumit *tip_adresa_generala* cu trei proprietăți (atribute):

```
CREATE OR REPLACE TYPE tip_adresa_generala AS OBJECT (
    codpostal CHAR(6),
    localitate VARCHAR2(25),
    judet VARCHAR2(25)
) NOT FINAL NOT INSTANTIABLE
```

Tipul este *non-final*, adică urmează să apară, cât de curând, sub-tipuri ale sale, și *neinstantiabil*, în sensul că nicio variabilă sau atribut de tabelă nu vor putea fi definite pe *tip_adresa_generala*. Figura 19.3 surprinde eroarea declanșată la execuția blocului anonim din listing 19.6 în care variabila *a* este declarată de *tip_adresa_generala* și se încearcă inițializarea sa.

Listing 19.6. Bloc anonim ce testează vigilența type-istă a Oracle

```
-- bloc anonim care folosește acest tip - generează eroare, deoarece tipul este NE-INSTANTIABIL
DECLARE
    a tip_adresa_generala := tip_adresa_generala ('700505', 'Iasi', 'Iasi') ;
BEGIN
    DBMS_OUTPUT.PUT_LINE(a.Localitate) ;
END ;
/
```

Continuăm cu un subtip al *tip_adresa_generala* dedicat reprezentării adreselor din mediul rural în care, de obicei, nu apare strada și numărul, dar apare comuna. Acesta se va numi *tip_adresa_sat*:

```
CREATE OR REPLACE TYPE tip_adresa_sat UNDER tip_adresa_generala (
    comuna VARCHAR2(30)
) FINAL
```

```
-- bloc anonim care foloseste acest tip - genereaza eroare, deoarece tipul este NE-
DECLARE
a tip_adresa_generala := tip_adresa_generala ('700505', 'Iasi', 'Iasi') ;
BEGIN
  DBMD_OUTPUT.PUT_LINE(a.Localitate) ;
END ;
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Error starting at line 2 in command:
 DECLARE
 a tip_adresa_generala := tip_adresa_generala ('700505', 'Iasi', 'Iasi') ;
 BEGIN
 DBMD_OUTPUT.PUT_LINE(a.Localitate) ;
 END ;
 Error report:
 ORA-06550: linia 2, coloana 27:
 PLS-00713: attempting to instantiate a type that is NOT INSTANTIABLE

Figura 19.3. Încercare de a instanția un tipul declarat NOT INSTANTIABLE

Celălalt subordonat direct al *tip_adresa_generala* este *tip_adresa_oras* care, la rândul lui, are un tip-fiu – *tip_adresa_bloc*. Folosirea celor două tipuri ar fi justificată de faptul că în orașe există persoane care locuiesc atât la bloc, cât și la case (sau vile).

```
CREATE OR REPLACE TYPE tip_adresa_oras UNDER tip_adresa_generala (
    strada VARCHAR2(35),
    nr VARCHAR2(10)
) NOT FINAL
/
CREATE OR REPLACE TYPE tip_adresa_bloc UNDER tip_adresa_oras (
    bloc VARCHAR2(20),
    scara VARCHAR2(10),
    etaj VARCHAR2(10),
    apartament VARCHAR2(10)
) FINAL
/
```

Listingul 19.7 conține un alt bloc PL/SQL anonim în care două variabile sunt declarate pe cele două (sub)tipuri, se instanțiază și li se afișează (vezi figura 19.4) câte una dintre proprietăți.

Listing 19.7. Bloc anonim cu două variabile de *tip_adresa_oras* și *tip_adresa_bloc*

```
DECLARE
a tip_adresa_oras := tip_adresa_oras ('700515', 'Pascani',
'ias', 'Independentei', '12bis') ;
b tip_adresa_bloc := tip_adresa_bloc ('700515', 'Pascani',
```

```
'Iasi', 'Independentei', '12bis', 'H4', 'B', 4, 54) ;
BEGIN
  DBMS_OUTPUT.PUT_LINE(a.strada);
  DBMS_OUTPUT.PUT_LINE(b.apartament);
END ;
```

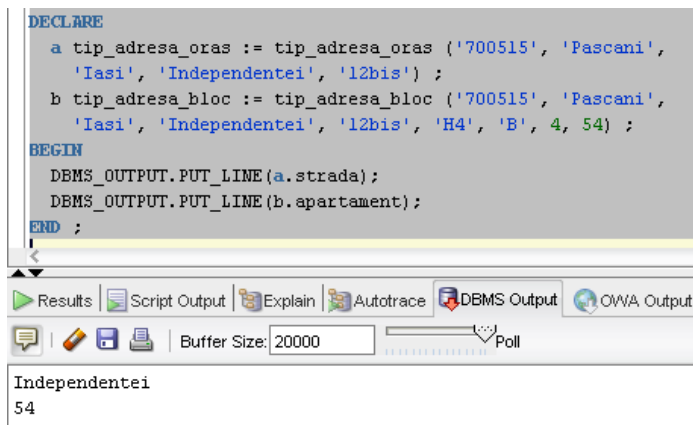


Figura 19.4. Două variabile de *tip_adresa_oras* și *tip_adresa_bloc*

Ierarhia declarată poate servi la definirea atributelor (proprietăților) unei tabele sau unui alt tip. Spre exemplu, *tip_persoana1* are patru proprietăți, ultima, *Adresa*, fiind definită pe tipul rădăcină al ierarhiei:

```
CREATE OR REPLACE TYPE tip_persoana1 AS OBJECT (
  cnp tip_cnp,
  nume VARCHAR2(30),
  prenume VARCHAR2(30),
  adresa tip_adresa_generala
) NOT FINAL
/
```

19.1.3. Tipuri simple în SQL Server

Versiunea folosită aproape exclusiv pe parcursul acestei lucrări, SQL 2005, este NULL-ă în materie de tipuri structurate. Tipurile pot fi create în limbaje ale platformei .NET și folosite în SQL Server, dar ceea ce ne interesează pe noi sunt opțiunile SQL (T-SQL) prin care să declarăm tipuri mai simple sau mai complexe. SQL Server 2008 mai spală din rușinea obiectual-relațională a Microsoftului, însă noile opțiuni sunt de fandoseală, întrucât nici tipurile distincte, nici cele structurate, nu pot fi folosite la definirea atributelor în tabele, ceea ce este esențial în BD O-R. Pentru tipurile distincte, sintaxa este destul de simplă:

```
CREATE TYPE t_sex FROM CHAR(1) ;
CREATE TYPE t_cantitate FROM NUMERIC(9,3) ;
CREATE TYPE t_pret FROM NUMERIC(10,2) ;
```

Și tipurile structurate au sunt particulare SQL Server 2008, fiind declarate cu ajutorul clauzei TABLE:

```
CREATE TYPE t_coduri_cl AS TABLE (CodCl SMALLINT) ;
```

```
CREATE TYPE t_facturi AS TABLE (
    NrFact INTEGER,
    DataFact DATE,
    CodCl SMALLINT NOT NULL ,
    Obs VARCHAR(50) ,
    ValTotala NUMERIC(10, 2) NULL,
    TVA NUMERIC(10, 2) NULL,
    ValIncasata NUMERIC(10, 2) NULL,
    Reduceri NUMERIC(9, 2) NULL,
    Penalizari NUMERIC(9, 2) NULL
);
```

Ca și în PostgreSQL, nu sunt permise ierarhiile de tipuri, deci nu putem vorbi de moștenire sau alte ingrediente O-R. Vom mai discuta câte ceva despre aceste tipuri în paragraful 19.2.3.

19.1.4. Tipuri simple în DB2

Conform documentației IBM⁴, în DB2 există patru tipuri de date definite de utilizator: distincte (distinct), structurate (structured), referință (reference) și masiv (array). Pentru comparație cu exemplele din Oracle, în DB2 vom începe cu *tip_cnp*:

```
CREATE TYPE tip_cnp AS (
    cnp DECIMAL(13)
) MODE DB2SQL FINAL
```

Continuăm cu tipul-rădăcină al ierarhiei adreselor:

```
CREATE TYPE tip_adresa_generala AS (
    codpostal CHAR(6),
    localitate VARCHAR(25),
    judet VARCHAR(25)
) MODE DB2SQL NOT FINAL NOT INSTANTIABLE
```

și subtipul său *tip_adresa_sat*:

```
CREATE TYPE tip_adresa_sat UNDER tip_adresa_generala AS (
    comuna VARCHAR(30)
) MODE DB2SQL FINAL ;
```

⁴ [IBM 2007-2]

De data aceasta, pentru testarea funcționalității subtipurilor vom redacta câteva funcții simple. Prima este *f_afisare_adresa_sat* și face obiectul listing-ului 19.8.

Listing 19.8. Funcție SQL PL de afișare a unui șir de caractere ce conține o adresă rurală

```
CREATE FUNCTION f_afisare_adresa_sat(adresa_tip_adresa_sat)
RETURNS VARCHAR(500)
LANGUAGE SQL
NO EXTERNAL ACTION
RETURN 'CodPostal ' || adresa_..CodPostal || ', Comuna ' || adresa_..Comuna ||
', Sat ' || adresa_..Localitate || ', Judet ' || adresa_..Judet
```

La apelarea la, trebuie procedat cu grijă la specificarea instanței-parametru:

```
SELECT f_afisare_adresa_sat ( tip_adresa_sat() ..codpostal('600515')
..localitate('Jorasti') ..judet('Vrancea') ..comuna('Vinatori') )
FROM sysibm.dual
```

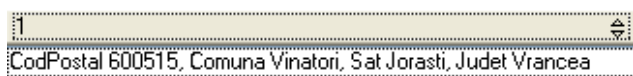


Figura 19.5. Apelarea funcției pentru afișarea unei adrese rurale

Continuăm cu *tip_adresa_oras* și funcția destinată afișarea unei adrese urbane de tip casă/vilă (listing 19.9):

```
CREATE TYPE tip_adresa_oras UNDER tip_adresa_generala AS (
    strada VARCHAR(35),
    nr VARCHAR(10)
) MODE DB2SQL NOT FINAL
```

Listing 19.9. Funcție SQL PL de afișare adresei urbane de tip casă/vilă

```
CREATE FUNCTION f_afisare_adresa_oras(adresa_tip_adresa_oras)
RETURNS VARCHAR(500)
LANGUAGE SQL
NO EXTERNAL ACTION
RETURN
CASE WHEN adresa_..strada IS NOT NULL THEN 'Strada ' || adresa_..strada ELSE " END ||
CASE WHEN adresa_..nr IS NOT NULL THEN ' nr ' || adresa_..nr ELSE " END ||
', CodPostal ' || adresa_..CodPostal || ', Localitate ' || adresa_..Localitate ||
', Judet ' || adresa_..Judet
```

Apelul funcției este similar celei precedente:

```
SELECT f_afisare_adresa_oras ( tip_adresa_oras()
..codpostal('700515') ..localitate('Pascani')
..judet('Iasi') ..strada('Independentei') ..nr('12bis') )
FROM sysibm.dual
```

În fine, *tip_adresa_bloc* are și cea mai „voluminoasă” funcție de afișare (listing 19.10):

```
CREATE TYPE tip_adresa_bloc UNDER tip_adresa_oras AS (
    bloc VARCHAR(20),
    scara VARCHAR(10),
    etaj VARCHAR(10),
    apartament DECIMAL(3)
) MODE DB2SQL FINAL
```

Listing 19.10. Funcție SQL PL de afișare adresei urbane de tip apartament

```
CREATE FUNCTION f_afisare_adresa_bloc(adresa_ tip_adresa_bloc)
    RETURNS VARCHAR(500)
LANGUAGE SQL
NO EXTERNAL ACTION
RETURN
CASE WHEN adresa_..strada IS NOT NULL THEN 'Strada ' || adresa_..strada ELSE " END ||
CASE WHEN adresa_..nr IS NOT NULL THEN ' nr ' || adresa_..nr ELSE " END ||
CASE WHEN adresa_..bloc IS NOT NULL THEN ' Bl. ' || adresa_..bloc ELSE " END ||
CASE WHEN adresa_..scara IS NOT NULL THEN ' Sc. ' || adresa_..scara ELSE " END ||
CASE WHEN adresa_..etaj IS NOT NULL THEN ' Etaj ' || adresa_..etaj ELSE " END ||
CASE WHEN adresa_..apartament IS NOT NULL THEN ' Ap. ' ||
    CAST (adresa_..apartament AS CHAR(3)) ELSE " END ||
', CodPostal ' || adresa_..CodPostal || ', Localitate ' || adresa_..Localitate ||
', Judet ' || adresa_..Judet
```

```
SELECT f_afisare_adresa_bloc ( tip_adresa_bloc() ..codpostal('700515')
    ..localitate('Pascani') ..judet('Iasi') ..strada('Independentei') ..nr('12bis')
    ..bloc('H4') ..scara('B') ..etaj ('4') ..apartament(54) )
FROM sysibm.dual
```

Încheiem paragraful cu *tip_persoana1* ce folosește ierarhia de adrese pentru unul dintre atributele sale:

```
CREATE TYPE tip_persoana1 AS (
    cnp tip_cnp,
    nume VARCHAR(30),
    prenume VARCHAR(30),
    adresa tip_adresa_generala
) REF USING INTEGER MODE DB2SQL
```

19.2. Colecții

Standardele SQL nu au fost grozav de generoase cu tipurile colecție. Abia în SQL:1999 a apărut tipul ARRAY, iar în SQL:2003 este introdus tipul MULTISSET.

Dintre acestea doar primul este ceva mai prezent în SGBD-urile actuale. SQL Server este absent la capitolul colecții, iar Oracle exagerat de generos.

19.2.1. Colecții în PostgreSQL

În prezentarea colecțiilor PostgreSQL pornim la o problemă plauzibilă. Să presupunem că avem nevoie de o listă a facturilor, filtrată pe clienți. Până acum puteam să obținem acest raport fie pentru toți clienții, fie numai pentru cei care îndeplineau condiția formulată în clauza WHERE. În aplicații, utilizatorul ar putea dispune de o listă a clienților, din care ar bifa numai unul, doi, trei... clienți care trebuie să apară în raportul respectiv. Cum folosirea operatorului IN poate fi anevoioasă (ce-ar fi dacă utilizatorul dorește includerea în raport a 50 de clienți din cei 1500 pe care îi are firma?), putem recurge la masive. Funcția din listing 19.11 este una simplă, însă atrage atenția prin două noutăți: parametrul de intrare este de tip ARRAY iar clauza WHERE folosește opțiunea ANY.

Listing 19.11. Funcția SQL (PostgreSQL) **f_facturi_filtrate** cu parametru de tip masiv

```
CREATE OR REPLACE FUNCTION f_facturi_clienti (coduri_clienti ANYARRAY )
  RETURNS SETOF facturi AS $$
  SELECT * FROM facturi WHERE codcl = ANY ( $1 )
  $$ LANGUAGE SQL ;
```

Apelul funcției într-o interogare se face de maniera:

```
SELECT *
FROM f_facturi_clienti ( ARRAY[1002, 1003, 1004] ) t
```

Putem, astfel, rescrie și funcția din listing 19.1 - *f_facturi_detaliat_clienti*, înlocuind parametrul simplu de intrare cu unul de tip ARRAY (listing 19.12). Faptul că parametrul de intrare este de tip vector se indică prin clauza ANYARRAY, conferă un mare grad de generalitate (deși nu era chiar necesar în exemplul nostru). Clauza WHERE a frazei SELECT ce constituie, de fapt, corpul funcției (SQL) prezintă predicatul *codcl = ANY (\$1)*., echivalentul lui *codcl IN (coduri_clienti(1), coduri_client(2)... coduri_clienti(n))*.

Listing 19.12. O funcție de folosește și tipul nou definit și un parametru-colecție

```
CREATE OR REPLACE FUNCTION f_facturi_detaliat_clienti (coduri_clienti ANYARRAY )
  RETURNS SETOF facturi_detaliat AS $$
  SELECT f.nrfact, datafact, f.codcl, dencl, linie, codpr,
         f.denpr(codpr), cantitate, pretunit, cantitate * pretunit,
         cantitate * pretunit * f_proctva(codpr),
         cantitate * pretunit * (1 + f_proctva(codpr))
  FROM clienti c
       INNER JOIN facturi f ON c.codcl=f.codcl
       INNER JOIN liniifact lf ON f.nrfact=lf.nrfact
  WHERE f.codcl = ANY ( $1 )
  $$ LANGUAGE SQL ;
```

Folosirea sa într-o interogare este, de acum, banală:

```
SELECT * FROM f_facturi_detaliat_clienti ( ARRAY[1001, 1002, 1004] ) t
```

În PostgreSQL tipurile ARRAY pot fi folosite la declararea proprietăților/atributelor altor tipuri:

```
CREATE TYPE tip_persoana AS (
    cnp CHAR(13),
    nume VARCHAR(30),
    prenume VARCHAR(30),
    adresa tip_adresa,
    telefoane CHAR(13) ARRAY[5],
    faxuri CHAR(13) ARRAY[5],
    emailuri VARCHAR(35) ARRAY[5]
);
```

sau:

```
CREATE TYPE tip_telefoane AS (nr_tel CHAR(10) ARRAY[10]);
```

```
CREATE TYPE tip_persoana2 AS (
    cnp CHAR(13),
    nume VARCHAR(30),
    prenume VARCHAR(30),
    adresa tip_adresa,
    telefoane tip_telefoane
);
```

19.2.2. Colecții în Oracle

În Oracle există trei categorii de colecții: vectori asociativi (INDEX BY), tabele incluse (NESTED TABLE) și vectori de mărime variabilă (VARRAY)⁵. Vectorii asociativi au fost utilizați în paragraful 17.3 pentru stocarea facturilor și liniilor șterse. Cel mai important dezavantaj al lor este că nu pot fi folosiți la definirea atributelor în tabele. În schimb, tabelele incluse și vectorii de mărime variabilă, da.

Mai întâi, să lichidăm cu datoria din capitolul 16 legată de funcții care returnează seturi de înregistrări. De fapt, cele două funcții pe care le vom discuta îndată returnează colecții de tip NESTED TABLES care pot fi convertite în seturi propriu-zise de înregistrări prin funcția TABLE. Definim tipul *t_r_facturi* care are aceleași atribute ca tabela FACTURI:

```
CREATE TYPE t_r_facturi AS OBJECT (
    NrFact NUMBER(8),
```

⁵ Pentru detalii, vezi (și) [Fotache s.a.2003]


```

DataFact DATE,
CodCI NUMBER(6),
Obs VARCHAR2(50),
ValTotala NUMBER(10,2),
TVA NUMBER(10,2),
    ValIncasata NUMBER(10,2),
Reduceri NUMBER(9,2),
Penalizari NUMBER(9,2)
);

```

Un tip colecție NESTED TABLE se definește analog unui vector asociativ, eliminând clauza INDEX BY. Fiecare element al colecției *t_facturi* este un obiect de tipul *t_r_facturi*:

```
CREATE TYPE t_facturi AS TABLE OF t_r_facturi
```

Tipul colecție este folosit de cele două funcții din noul *pac_arhivare2* ce lucrează în tandem cu *pac_arhivare* din capitolul precedent. Ambele funcții returnează tipul colecție. Prima, *f_fuziune_facturi2*, returnează tot masivul odată, ceea ce poate congestiona traficul din rețea (dacă dimensiunea masivului este foarte mare). A doua este recomandată de documentația Oracle, întrucât componentele colecției se returnează una câte una (clauza PIPELINED).

Listing 19.13. Un pachet ce „continuă” *pac_arhivare*

```

=====
-- pachetul pac_arhivare2 continuă operațiunile demarate în paragraful 18.2 (SQL dinamic)
=====
CREATE OR REPLACE PACKAGE pac_arhivare2 AUTHID CURRENT_USER AS
-- urmeaza alte doua functii pentru "recompunerea" dinamica a inregistrarilor arhivate

-- varianta 2 de reconstituire facturi : functie ce returneaza un NESTED TABLE;
-- in prealabil trebuie create tipurile T_R_facturi si T_facturi
FUNCTION f_fuziune_facturi2(datai DATE, dataf DATE) RETURN t_facturi ;

-- varianta 3 este o noua versiune a F_FUZIUNE_FACTURI2 cu variabile cursor si PIPELINED
FUNCTION f_fuziune_facturi3(datai DATE, dataf DATE) RETURN t_facturi PIPELINED ;

END pac_arhivare2 ;
=====
/
=====
CREATE OR REPLACE PACKAGE BODY pac_arhivare2 AS
=====
FUNCTION f_fuziune_facturi2(datai DATE, dataf DATE) RETURN t_facturi
IS
    v_facturi t_facturi := t_facturi();
    v_sir VARCHAR2(2056) := '';
    TYPE trefcursor IS REF CURSOR;
    vrefcursor trefcursor;
    v_o_factura facturi%ROWTYPE;
BEGIN
    v_sir := pac_arhivare.f_sir_facturi(datai, dataf);
    -- ar fi fost frumos, dar aceasta comanda nu functioneaza !!!

```

```
--EXECUTE IMMEDIATE 'SELECT * BULK COLLECT INTO v_facturi FROM (' || v_sir || ') ' ;

-- asa ca folosim REFCURSOR
OPEN vrefcursor FOR v_sir;
LOOP
  FETCH vrefcursor INTO v_o_factura ;
  EXIT WHEN vrefcursor % NOTFOUND;
  v_facturi.extend;
  v_facturi(v_facturi.COUNT) := t_r_facturi(v_o_factura.NrFact, v_o_factura.DataFact,
  v_o_factura.CodCl, v_o_factura.Obs, v_o_factura.ValTotala,
  v_o_factura.TVA, v_o_factura.ValIncasata, v_o_factura.Reduceri, v_o_factura.Penalizari);
END LOOP;
CLOSE vrefcursor;

RETURN v_facturi;
END f_fuziune_facturi2;

-----
FUNCTION f_fuziune_facturi3(datai DATE, dataf DATE) RETURN t_facturi PIPELINED
IS
  --v_facturi t_facturi ; --:= t_facturi() ; -- nu mai avem nevoie de variabila colectie,
  -- deoarece liniile se returneaza pe rind (PIPELINED)
  v_sir VARCHAR2(2056) := ' ';
  TYPE tRefCursor IS REF CURSOR;
  vRefCursor tRefCursor;
  v_o_factura facturi%ROWTYPE ;
  v_factura_p t_r_facturi ;
BEGIN
  v_sir := pac_arhivare.f_sir_facturi(datai, dataf);
  OPEN vrefcursor FOR v_sir;
  LOOP
    FETCH vRefCursor INTO v_o_factura;
    EXIT WHEN vRefCursor%NOTFOUND ;
    v_factura_p := t_r_facturi(v_o_factura.NrFact, v_o_factura.DataFact,
    v_o_factura.CodCl, v_o_factura.Obs, v_o_factura.ValTotala,
    v_o_factura.TVA, v_o_factura.ValIncasata, v_o_factura.Reduceri, v_o_factura.Penalizari) ;
    PIPE ROW(v_factura_p);
  END LOOP;
  CLOSE vrefcursor;
  RETURN ;
END f_fuziune_facturi3;

END pac_arhivare2;
/
```

Dacă vă gândiți la o modalitate de a „pasa” ceva mai simplu o linie din tabela facturi într-o instanță a tipului *t_facturi*, aflați că m-am gândit și eu, dar fără spor. Înainte de a demonstra funcționalitatea celor două funcții (vezi figura 19.6), trebuie să ne asigurăm că formatul datei calendaristice este cel ISO:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD' ;
SELECT * FROM TABLE (pac_arhivare2.f_fuziune_facturi2(
DATE'2007-09-05',
DATE'2007-11-10')) ORDER BY 1
```

```

ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD'

SELECT *
FROM TABLE (pac_arhivare2.f_fuziune_facturi2 (DATE'2007-09-05', DATE'2007-11-10'))
ORDER BY 1

```

	NRFACT	DATAFACT	CODCL	OBS	VALTOTALA	TVA	VA
1	3116	2007-09-10	1007	Pretul propus initial a fost modificat	136849,5	11299,5	
2	3117	2007-09-10	1001	(null)	287855	33355	
3	3118	2007-09-17	1001	(null)	179565	25065	
4	3119	2007-10-07	1003	(null)	5819668	919468	
5	5111	2007-11-01	1001	(null)	4346037,5	687287,5	
6	5112	2007-11-01	1005	Probleme cu transportul	125516	13116	
7	5113	2007-11-01	1002	(null)	106275	8775	
8	5114	2007-11-01	1006	(null)	6021706,5	950856,5	
9	5115	2007-11-02	1001	(null)	151237,5	12487,5	
10	5116	2007-11-02	1007	Pretul propus initial a fost modificat	126712,5	10462,5	
11	5117	2007-11-03	1001	(null)	222050	27050	
12	5118	2007-11-04	1001	(null)	201975	29475	
13	5119	2007-11-07	1003	(null)	5774498,5	912848,5	
14	5120	2007-11-07	1001	(null)	97664	8064	
15	5121	2007-11-07	1004	(null)	4737838	747638	

Figura 19.6. Testarea unei dintre funcțiile ce returnează o colecție

Acum putem redacta funcția care să primească un parametru de tip *nested table* sau *varray* (parametru ce conține oricâte coduri de client) și care să returneze facturile corespunzătoare clienților-parametru. Rămâne să o realizați dvs. în contul temei pentru acasă.

Despre colecții și modul lor de folosire în tabele vom cheltui câteva pagini în ultimul paragraf.

19.2.3. Colecții și SQL Server

Un titlu mai adecvat pentru acest paragraf ar fi fost *Nuca-n perete*, întrucât, așa cum spuneam în deschiderea paragrafului 19.2, SQL Server este cel mai nevoieaș în materie de colecții. Noroc cu variabilele de tip *TABLE* cu ajutorul cărora putem formula una dintre soluțiile la problema realizării unei funcții care să primească, drept parametru, mai multe coduri de client și care să returneze *recordset*-ul. Cum n-am văzut prin documentație să existe colecții, vom redacta, în deschidere, o funcție care primește un șir de caractere ce conține coduri de clienți separate prin virgulă și va returna o tabelă în care fiecare linie reprezintă un cod – vezi listing 19.14. Figura 19.7 demonstrează că funcția este operațională.

Listing 19.14. Funcție ce transformă un șir de caractere în înregistrările unei tabele

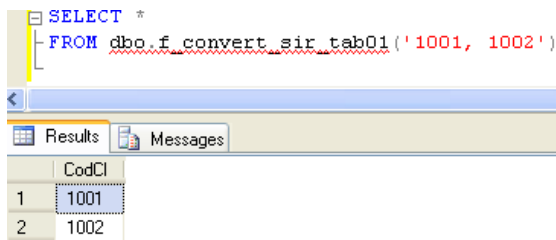
```

CREATE FUNCTION dbo.f_convert_sir_tab01 (
    @sir_coduri_cl VARCHAR(1000) )
RETURNS @tabela TABLE (CodCI SMALLINT)
AS
BEGIN
    DECLARE @sir_sursa VARCHAR(1000);
    DECLARE @i_inceput_cod SMALLINT;
    DECLARE @un_cod VARCHAR(10);
    DECLARE @i SMALLINT;
    SET @sir_sursa = LTRIM(RTRIM(@sir_coduri_cl))
    SET @i = 2;
    SET @i_inceput_cod = 1;

    WHILE @i <= LEN(@sir_sursa)
    BEGIN
        IF SUBSTRING(@sir_sursa, @i, 1) IN (',', '.') OR @i=LEN(@sir_sursa)
            BEGIN
                SET @un_cod = LTRIM(RTRIM(SUBSTRING(@sir_sursa, @i_inceput_cod,
                    @i - 1)))
                IF @un_cod <> ''
                    INSERT INTO @tabela
                        SELECT CAST (@un_cod AS SMALLINT)
                        WHERE CAST (@un_cod AS SMALLINT) NOT IN
                            (SELECT CodCI FROM @tabela)

                SET @i_inceput_cod = @i + 1;
            END;
        SET @i = @i + 1;
    END;
    RETURN
END

```



	CodCI
1	1001
2	1002

Figura 19.7. Testarea funcției ce convertește un șir de caractere în tabelă

Am avut probleme cu redactarea funcției care să primească tabela returnată de *f_convert_sir_tab01* drept parametru de intrare; de fapt, nu cu redactarea funcției, ci cu apelul său, întrucât, la apelare, argumentul său era funcția de mai sus. Până la urmă, sintaxa funcțională este cea din listing 19.15.

Listing 19.15. Funcție ce filtrează facturilor după mai multe coduri de client

```

CREATE FUNCTION dbo.f_facturi_clienti (
    @sir_coduri_cl VARCHAR(1000) )
RETURNS @facturi TABLE
(
    NrFact INTEGER,
    DataFact DATE,
    CodCI SMALLINT,
    Obs VARCHAR(50) ,

```

```

ValTotala NUMERIC(10, 2),
TVA NUMERIC(10, 2),
ValIncasata NUMERIC(10, 2),
Reduceri NUMERIC(9, 2),
Penalizari NUMERIC(9, 2)
)
AS
BEGIN
-- DECLARE @coduri_cl t_coduri_cl -- nu functioneaza !!!!
DECLARE @coduri_cl TABLE (CodCl SMALLINT)

INSERT INTO @coduri_cl
    SELECT * FROM dbo.f_convert_sir_tab01 (@sir_coduri_cl_)

INSERT INTO @facturi
    SELECT * FROM facturi
    WHERE CodCl IN (SELECT DISTINCT CodCl FROM @coduri_cl)
RETURN
END

```

Rezultatul execuției funcției pentru codurile de client 1002 și 1003 se prezintă în figura 19.8:

```

SELECT *
FROM dbo.f_facturi_clienti ('1002,1003')
ORDER BY 3, 1

```

	NrFact	DataFact	CodCl	Obs	ValTotala	TVA	V
1	2113	2007-08-14	1002	NULL	127530.00	10530.00	0
2	3113	2007-09-02	1002	NULL	127530.00	10530.00	0
3	5113	2007-11-01	1002	NULL	106275.00	8775.00	0
4	6113	2007-11-14	1002	NULL	127530.00	10530.00	0
5	7003	2007-12-01	1002	NULL	106275.00	8775.00	0
6	7009	2007-12-02	1002	NULL	127530.00	10530.00	0
7	7022	2007-12-14	1002	NULL	127530.00	10530.00	0
8	1119	2007-08-07	1003	NULL	5774498.50	912848.50	0
9	2119	2007-08-21	1003	NULL	5819668.00	919468.00	0
10	2119	2007-10-07	1003	NULL	5819668.00	919468.00	0

Figura 19.8. Testarea funcției ce filtrează facturile după mai multe coduri-client

Concluzia este că în SQL Server nu avem colecții, dar putem improviza câteva ceva.

19.2.4. Colecții în DB2

SQL PL gestionează destul de elegant colecțiile de tip ARRAY, de fapt, singurul tip disponibil de colecții. Declararea unui masiv care va conține coduri de client se realizează astfel:

```
CREATE TYPE t_CodClArray as SMALLINT ARRAY[100]
```

În continuare redactăm procedura de filtrare a facturilor după mai multe coduri de client (și două date calendaristice). Inițial, am fi vrut să realizăm o funcție, însă opțiunile de lucru cu masive sunt disponibile doar în proceduri. Ne reamintim procedura din listing 16.64 pe care o re-denumim și modificăm tocmai pentru a primi un parametru de tip ARRAY și pentru a-l folosi în interogarea de definire a cursorului – vezi listing 19.16. Funcția ce transformă colecția în seturi de înregistrări este UNNEST.

Listing 19.16. Procedură SQL PL ce filtrează facturilor după mai multe coduri de client

```
CREATE PROCEDURE p_fact_filtrate_array ( coduriclienti_ t_CodClArray,
data_iniciala DATE, data_finala DATE )
DYNAMIC RESULT SETS 1
P1: BEGIN
-- Declararea cursorului
DECLARE cursor1 CURSOR WITH RETURN FOR
SELECT NrFact, DataFact, CodCl, Obs, ValTotala, TVA
FROM facturi
WHERE CodCl IN (SELECT tCl.CodCl FROM UNNEST (coduriclienti_) AS tCl (CodCl) )
AND DataFact BETWEEN COALESCE(data_iniciala, '2005-01-01')
AND COALESCE(data_finala, '2010-12-31');
-- Cursorul rămâne deschis pentru aplicația client
OPEN cursor1;
END P1
```

La apelul procedurii este necesar constructorul ARRAY:

```
CALL p_fact_filtrate_array ( ARRAY[1002,1003],
CAST ('2007-10-07' AS DATE), CURRENT_DATE)
```

rezultatul fiind cel din figura 19.9.

Din păcate, în SQL PL elementele unui masiv nu pot fi tipuri structurate.

```
CREATE TYPE tip_email AS (email VARCHAR(100) ) MODE DB2SQL FINAL
```

funcționează, nu însă și

```
CREATE TYPE tip_telefon AS (numar CHAR(10)) MODE DB2SQL NOT FINAL ;
CREATE TYPE tip_telefoane AS (tip_telefon) ARRAY[10] ;
```

Așa că singura variantă este:

```
CREATE TYPE tip_telefoane AS CHAR(10) ARRAY[10] ;
```

```
CALL p_fact_filttrate_array ( ARRAY[1002,1003], CAST ('2007-10-07' AS DATE), CURRENT_DATE)
```

```
-----
```

NRFACT	DATAFACT	CODCL	OBS	VALTOTALA	TVA
3119	10/07/2007	1003	-	5819668.00	919468.00
4000	01/21/2009	1002	-	21800.00	1800.00
4001	01/22/2009	1002	-	33700.00	3700.00
4002	01/22/2009	1002	-	33700.00	3700.00
5113	11/01/2007	1002	-	106275.00	8775.00
5119	11/07/2007	1003	-	5774498.50	912848.50
6113	11/14/2007	1002	-	127530.00	10530.00
6119	11/21/2007	1003	-	5819668.00	919468.00
7003	12/01/2007	1002	-	106275.00	8775.00
7009	12/02/2007	1002	-	127530.00	10530.00
7013	12/07/2007	1003	-	5774498.50	912848.50
7017	12/07/2007	1003	-	5819668.00	919468.00
7022	12/14/2007	1002	-	127530.00	10530.00
7028	12/21/2007	1003	-	5819668.00	919468.00

```
14 record(s) selected.
```

```
Return Status = 0
```

Figura 19.9. Testarea procedurii SQL PL ce filtrează facturile după mai multe coduri-client

O altă limită a DB2 este că tipurile colecții nu pot fi folosite la declararea atributelor unui alt tip. Nici comanda următoare nu este acceptată:

```
CREATE TYPE tip_persoana AS (
    cnp tip_cnp,
    nume VARCHAR(30),
    prenume VARCHAR(30),
    adresa tip_adresa_generala,
    telefoane tip_telefoane,
    faxuri tip_telefoane,
    emailuri tip_emailuri
) MODE DB2SQL NOT FINAL
```

19.3. Stocarea tipurilor utilizator în tabele

Toată discuția legată de tipurile utilizator, inclusiv de ierarhie, polimorfism, ar avea doar un caracter pitoresc dacă nu ar exista posibilitatea stocării lor în tabele. Putem vorbi de două situații: (1) cea în care într-o tabelă, unul sau mai multe attribute sunt definite pe tipuri structurate și (2) cea în care o linie (înregistrare) a unei tabele reprezintă o instanță a unui tip structurat de date, aceasta din urmă fiind o tabelă *tipată*.

19.3.1. Tipuri și tabele în PostgreSQL

În PostgreSQL nu putem implementa de ierarhii de tipuri, crea de metode redactate PL/pgSQL și nici tabele-tipate. În afara ierarhiilor de tabele (de care nu suflăm o vorbă), singura posibilitate este de a crea tabele în care attributele pot fi tipuri structurate sau colecții:

```
CREATE TABLE persoane_ (
    cnp CHAR(13) PRIMARY KEY,
    nume VARCHAR(30),
    prenume VARCHAR(30),
    adresa tip_adresa,
    telefoane CHAR(13) ARRAY[5],
    faxuri CHAR(13) ARRAY[5],
    emailuri VARCHAR(35) ARRAY[5]
);
```

Atribute Adresa din tabela PERSOANE_ a fost definit pe *tip_adresa*, iar attributele Telefoane, Faxuri și Emailuri sunt colecții. Scriptul din listing 19.17 ilustrează modul de specificare a valorilor atributelor în comanda INSERT.

Listing 19.17. Popularea cu înregistrări a unei tabele care are attribute definite pe tipuri structurate

```
DELETE FROM persoane_ ;

INSERT INTO persoane_ VALUES ('1760210390803', 'Popescu', 'Ioneliu',
    ROW ('123457', 'Jorasti', 'Vinatori', ROW('VN', 'Vrancea', 'Moldova'), NULL, NULL, NULL,
    NULL, NULL, NULL),
    ARRAY ['0744123454', '0745123456', '0232222222'], -- telefoane
    NULL, -- n-are fax !
    ARRAY ['popescu_i@d.ro'] -- o adresa de email
);
INSERT INTO persoane_ VALUES ('1660210390802', 'Ionescu', 'Vasile',
    ROW ('123456', 'Vinatori', 'Vinatori', ROW('VN', 'Vrancea', 'Moldova'), NULL, NULL, NULL,
    NULL, NULL, NULL),
    ARRAY ['0744123455', '0788123456', '0232222225'],
    ARRAY ['0232222225'], -- fax
    ARRAY ['ionescu_v@d.ro', 'io201@yahoo.com']
);
INSERT INTO persoane_ VALUES ('2821211390804', 'Iatu', 'Vasilica',
    ROW ('123458', 'Jaristea', 'Jaristea', ROW('VN', 'Vrancea', 'Moldova'), NULL, NULL, NULL,
    NULL, NULL, NULL),
    NULL, -- n-are tel.
    NULL, -- n-are fax
    NULL -- n-are adrese de e-mail
);
INSERT INTO persoane_ VALUES ('1781005390810', 'Cretu', 'Livi',
    ROW ('700111', 'Iasi', NULL, ROW('IS', 'Iasi', 'Moldova'), 'Independentei',
    '11bis', 'G4', 'B', 'parter', 3),
    ARRAY ['0744293455', '0788123456', '0723128956', '0232277925'],
    ARRAY ['023227725'],
    ARRAY ['cretu_liviu45@d.ro', 'cl6788@yahoo.com']
);
INSERT INTO persoane_ VALUES ('2850419390812', 'Cretu', 'Livia',
```



```

ROW ('700112', 'Iasi', NULL, ROW('IS', 'Iasi', 'Moldova'), 'Dependentei', '144', 'H14', 'D', '8', 45),
ARRAY ['0744298855', '0788555456', '0728888956', '0235577925'],
ARRAY['0232557725'],
ARRAY ['cretu_livia@doi.ro', 'livia23@yahoo.com']
);
INSERT INTO persoane_ VALUES ('2831119390814', 'Bratu', 'Ioana',
ROW ('700112', 'Iasi', NULL, ROW('IS', 'Iasi', 'Moldova'), 'Dependentei', '145', NULL, NULL,
NULL, NULL),
ARRAY ['0744298855', '0788777456', '0728999956', '0235577929'],
NULL, -- fax
NULL
);

```

La interogare, afișarea valorilor atributelor definite pe tipuri structurate și colecții este, în *pgAdmin*, cea din figura 19.10.

	cnp character(13)	nume character character	prenume character	adresa tip_adresa	telefoane character(13)[]	faxuri character(13)[]	emailuri character
1	1760210390803	Popescu	Ioneliu	(123457,Jorasti,Vinatori,"(VN,Vrancea,Moldova)",,,,,,,)	{'0744123454 ','0745123456 ','0232222222 '}	{'0232222225 '}	{popescu_
2	1660210390802	Ionescu	Vasile	(123456,Vinatori,Vinatori,"(VN,Vrancea,Moldova)",,,,,,,)	{'0744123455 ','0788123456 ','0232222225 '}	{'0232222225 '}	{ionescu_
3	2821211390804	Iatu	Vasilica	(123458,Jaristea,Jaristea,"(VN,Vrancea,Moldova)",,,,,,,)			
4	1781005390810	Cretu	Liviu	(700111,Iasi,"(IS,Iasi,Moldova)",Independentei,11bis,G4,B,parter,3)	{'0744293455 ','0788123456 ','0723128956 ','0232277925 '}	{'0232227725 '}	{cretu_l
5	2850419390812	Cretu	Livia	(700112,Iasi,"(IS,Iasi,Moldova)",Dependentei,144,H14,D,8,45)	{'0744298855 ','0788555456 ','0728888956 ','0235577925 '}	{'0232557725 '}	{cretu_l
6	2831119390814	Bratu	Ioana	(700112,Iasi,"(IS,Iasi,Moldova)",Dependentei,145,,,))	{'0744298855 ','0788777456 ','0728999956 ','0235577929 '}		

Figura 19.10. Afișarea conținutului tabeli PERSOANE_

Extragerea poate fi limitată la doar componente ale colecțiilor. Astfel, ne interesează prima adresă de e-mail a fiecărei persoane (figura 19.11). Interogarea necesară este:

```

SELECT p.nume, p.prenume, p.emailuri[1] AS prima_adr_em
FROM persoane_ p
ORDER BY 1,2

```

	nume character varying	prenume character varying	prima_adr_em character varying(35)
1	Bratu	Ioana	
2	Cretu	Livia	cretu_livia@doi.ro
3	Cretu	Liviu	cretu_liviu45@d.ro
4	Iatu	Vasilica	
5	Ionescu	Vasile	ionescu_v@d.ro
6	Popescu	Ioneliu	popescu_i@d.ro

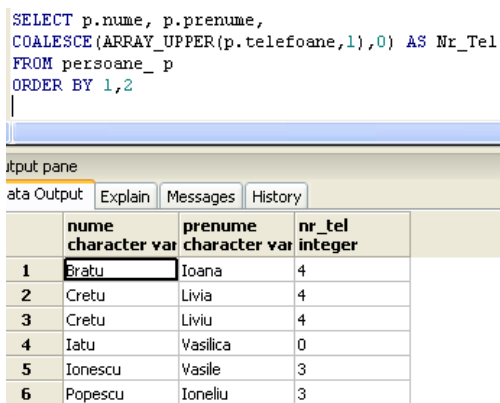
Figura 19.11. Prima adresă de e-mail a fiecărei persoane

Pentru căutarea în masiv, după cum am văzut în paragraful 19.2.1, se folosește operatorul ANY. Astfel, fraza SELECT ce răspunde la întrebarea *Cine are telefonul '0788555456'?* este:

```
SELECT p.num, p.prenume, p.telefoane FROM persoane_ p
WHERE '0788555456' = ANY (p.telefoane) ORDER BY 1,2
```

În afara lui ANY, mai pot fi folosiți și alți operatori, cum ar fi ARRAY_UPPER care indică poziția ultimei componente a unui masiv. Astfel, pentru *întrebarea Câte numerele de telefon are fiecare persoană?* putem formula o soluție de genul (vezi figura 19.12):

```
SELECT p.num, p.prenume,
       COALESCE(ARRAY_UPPER(p.telefoane,1),0) AS Nr_Tel
FROM persoane_ p ORDER BY 1,2
```



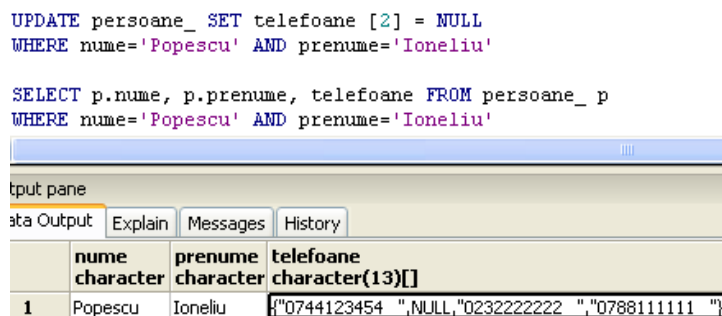
	num	prenume	nr_tel
	character var	character var	integer
1	Bratu	Ioana	4
2	Cretu	Livia	4
3	Cretu	Liviu	4
4	Iatu	Vasilica	0
5	Ionescu	Vasile	3
6	Popescu	Ioneliu	3

Figura 19.12. Numărul...numerelor de telefon al fiecărei persoane

Acum ne propunem să exemplificăm actualizarea atributelor de tip colecție. Astfel, adăugăm un număr de ZAPP lui Popescu Ioneliu:

```
UPDATE persoane_ SET telefoane [4] = '0788111111'
WHERE num='Popescu' AND prenume='Ioneliu'
```

Situația telefoanelor sale după această comandă este cea din figura 19.13.



```
UPDATE persoane_ SET telefoane [2] = NULL
WHERE num='Popescu' AND prenume='Ioneliu'
```

```
SELECT p.num, p.prenume, telefoane FROM persoane_ p
WHERE num='Popescu' AND prenume='Ioneliu'
```

	num	prenume	telefoane
	character	character	character(13)[]
1	Popescu	Ioneliu	{"0744123454 ", "NULL", "0232222222 ", "0788111111 "}

Figura 19.13. Numerele de telefon ale lui Ioneliu Popescu după adăugarea numărului din rețeaua ZAPP

19.3.2. Tipuri și tabele în Oracle

Cum avem deja un exemplu de tabelă în care câteva atribute sunt definite pe tipuri structurate și pe tipuri colecție, în Oracle ne grăbim să creăm o tabelă „tipată”:

CREATE TABLE persoane1 OF tip_persoana1

Fiecare linie a acestei tabele va stoca o instanță a clasei (tipului) *tip_persoana1* definit în paragraful 19.2.2. Listingul 19.19 conține câteva comenzi INSERT prin care tabela PERSOANE1 este populată cu șase înregistrări.

Listing 19.19. Popularea unei tabele „tipate”

```
INSERT INTO persoane1 VALUES (NEW tip_persoana1 (
    tip_cnp (1660210390802),
    'Ionescu', 'Vasile',
    tip_adresa_sat(123456, 'Vinatori', 'Vrancea', 'Vinatori')
))
/
INSERT INTO persoane1 VALUES (NEW tip_persoana1 (
    tip_cnp (1760210390803),
    'Popescu', 'Ioneliu',
    tip_adresa_sat(123457, 'Jorasti', 'Vrancea', 'Vinatori')
))
/
INSERT INTO persoane1 VALUES (NEW tip_persoana1 (
    tip_cnp (2821211390804),
    'Iatu', 'Vasilica',
    tip_adresa_sat(123458, 'Jaristea', 'Vrancea', 'Jaristea')
))
/
INSERT INTO persoane1 VALUES (NEW tip_persoana1 (
    tip_cnp (1781005390810),
    'Cretu', 'Liviu',
    tip_adresa_bloc(700111, 'Iasi', 'Iasi', 'Independentei', '11bis',
    'G4', 'B', 'parter', 3)
))
/
INSERT INTO persoane1 VALUES (NEW tip_persoana1 (
    tip_cnp (2850419390812),
    'Cretu', 'Livia',
    tip_adresa_bloc(700112, 'Iasi', 'Iasi', 'Dependentei', '144',
    'H14', 'D', '8', 45)
))
/
INSERT INTO persoane1 VALUES (NEW tip_persoana1 (
    tip_cnp (2831119390814),
    'Bratu', 'Ioana',
    tip_adresa_oras(700112, 'Iasi', 'Iasi', 'Dependentei', '144')
))
/
```

Actualizarea atributului Adresa definit pe tipul structurat *tip_adresa_generala* se realizează folosind constructorul sub-tipului care se referă la categoria adresei persoanei respective (în cazul nostru este vorba de o adresă urbană pentru casă/vilă):

```
UPDATE persoane1 p
SET adresa = tip_adresa_oras(700112, 'Iasi', 'Iasi', 'Florilor', '7')
WHERE p.cnp.cnp =2831119390814
```

SQL Developerul afișează conținutului tabeli „tipate” de maniera celei din figura 19.14.

	CNP	NUME	PRENUME	ADRESA
1	SQL2008.TIP_CNP(1660210390802)	Ionescu	Vasile	SQL2008.TIP_ADRESA_SAT(123456,Vinatori,Vrancea,Vinatori)
2	SQL2008.TIP_CNP(1760210390803)	Popescu	Ioneliu	SQL2008.TIP_ADRESA_SAT(123457,Jorasti,Vrancea,Vinatori)
3	SQL2008.TIP_CNP(2821211390804)	Iatu	Vasilica	SQL2008.TIP_ADRESA_SAT(123458,Jaristea,Vrancea,Jaristea)
4	SQL2008.TIP_CNP(1781005390810)	Cretu	Liviu	SQL2008.TIP_ADRESA_BLOC(700111,Iasi,Iasi,Independentei,11bis,G4,B,parter,3)
5	SQL2008.TIP_CNP(2850419390812)	Cretu	Livia	SQL2008.TIP_ADRESA_BLOC(700112,Iasi,Iasi,Dependentei,144,H14,D,8,45)
6	SQL2008.TIP_CNP(2831119390814)	Bratu	Ioana	SQL2008.TIP_ADRESA_ORAS(700112,Iasi,Iasi,Florilor,7)

Figura 19.14. Afișarea conținutului tabeli PERSOANE1

Pentru afișarea valorii unor atribute ale tipurilor structurate folosite la definirea *tip_persoane1*, acestea pot fi calificate astfel:

```
SELECT p.num, p.prenume, p.cnp.cnp, p.adresa.Localitate, p.adresa.Judet
FROM persoane1 p
```

Rezultatul e cel din figura 19.15. A doua manieră va fi prezentată în paragraful 19.5 și presupune invocarea metodelor definite la nivel de tip.

	NUME	PRENUME	CNP.CNP	ADRESA.LOCALITATE	ADRESA.JUDET
1	Ionescu	Vasile	1660210390802	Vinatori	Vrancea
2	Popescu	Ioneliu	1760210390803	Jorasti	Vrancea
3	Iatu	Vasilica	2821211390804	Jaristea	Vrancea
4	Cretu	Liviu	1781005390810	Iasi	Iasi
5	Cretu	Livia	2850419390812	Iasi	Iasi
6	Bratu	Ioana	2831119390814	Iasi	Iasi

Figura 19.14. Calificarea atributelor din tipurile folosite la crearea tabeli PERSOANE1

Atributul Localitate face parte din tipul rădăcină al ierarhiei adreselor – *tip_adresa_generala*. La invocarea unui atribut de pe un sub-tip al rădăcinii, cum ar cazul comunei:

```
SELECT p.num, p.prenume, p.adresa.judet AS judet,
       p.adresa.comuna
FROM persoane1 p
```

Recepționăm un binemeritat mesaj de eroare – vezi figura 19.15.

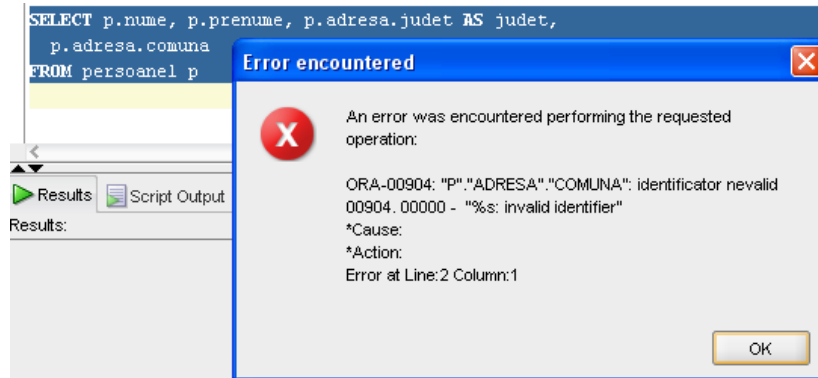


Figura 19.15. Eroare datorată accesării unui atribut aflat pe o ramură a ierarhiei de adrese

Soluția ține de folosirea clauzei TREAT:

```
SELECT p.num, p.prenume, p.adresa.judet AS judet,
       TREAT (p.adresa AS tip_adresa_sat).comuna
FROM persoane1 p
```

Filtrarea înregistrărilor din PERSOANE1, astfel încât să fie selectate numai persoanele care domiciliază la sate, este posibilă cu suportul operatorului IS OF:

```
SELECT p.num, p.prenume, p.adresa.getJudet() AS Judet,
       TREAT (p.adresa AS tip_adresa_sat).getComuna() AS Comuna
FROM persoane2 p
WHERE p.adresa IS OF (tip_adresa_sat)
```

Înlocuirea lui *tip_adresa_sat* cu *tip_adresa_oras* conduce la extragerea persoanelor cu adrese urbane, atât a celor de la casă/vilă, cât și a celor de la bloc:

```
SELECT p.num, p.prenume, p.adresa.Localitate,
       TREAT (p.adresa AS tip_adresa_oras).Strada
FROM persoane1 p
WHERE p.adresa IS OF (tip_adresa_oras)
```

Limitarea la cei care domiciliază la casă/vilă presupune combinarea operatorului IS OF cu clauza ONLY:

```
SELECT p.num, p.prenume, p.adresa.Localitate,
```

```
TREAT (p.adresa AS tip_adresa_oras).Strada
FROM persoane1 p
WHERE p.adresa IS OF (ONLY tip_adresa_oras)
```

19.3.3. Tipuri și tabele în SQL Server

Acesta este unul dintre paragrafele mele favorite. Nu avem ce discuta, întrucât SQL Server nu acceptă definirea atributelor din tabele pe tipuri utilizator. Singurul motiv pentru care am inclus această frază ține de respectarea numerotării paragrafelor, relativ la cele patru servere BD.

19.3.4. Tipuri și tabele în DB2

Pe baza *tip_personal1* creat în paragraful 19.1.4, creăm o tabelă „tipată” după modelul celei din Oracle:

```
CREATE TABLE persoane1 OF tip_persoana1
(REF IS OID USER GENERATED);
```

Popularea tabelii are câteva particularități ce decurg din obligativitatea folosirii identificatorului de obiecte (OID) – vezi listing 19.20.

Listing 19.20. Popularea tabelii „tipate” în DB2

```
INSERT INTO persoane1 (oid, cnp, nume, prenume, adresa) VALUES ( tip_persoana1(1),
    tip_cnp()..cnp(1660210390802) , 'Ionescu', 'Vasile',
    tip_adresa_sat() ..codpostal('123456')
    ..localitate('Jorasti') ..judet('Vrancea') ..comuna('Vinatori')
);
INSERT INTO persoane1 (oid, cnp, nume, prenume, adresa) VALUES ( tip_persoana1(2),
    tip_cnp()..cnp(1760210390803) , 'Popescu', 'Ioneliu',
    tip_adresa_sat() ..codpostal('123457')
    ..localitate('Jorasti') ..judet('Vrancea') ..comuna('Vinatori')
);
INSERT INTO persoane1 (oid, cnp, nume, prenume, adresa) VALUES ( tip_persoana1(3),
    tip_cnp()..cnp(2821211390804) , 'Iatu', 'Vasilica',
    tip_adresa_sat() ..codpostal('123458')
    ..localitate('Jaristea') ..judet('Vrancea') ..comuna('Jaristea')
);
INSERT INTO persoane1 (oid, cnp, nume, prenume, adresa) VALUES ( tip_persoana1(4),
    tip_cnp()..cnp(1781005390810) , 'Cretu', 'Liviu',
    tip_adresa_bloc() ..codpostal('700111') ..localitate('Iasi') ..judet('Iasi') ..strada('Independentei')
    ..nr('11bis') ..bloc('G4') ..scara('b') ..etaj('parter') ..apartament(3)
);
INSERT INTO persoane1 (oid, cnp, nume, prenume, adresa) VALUES ( tip_persoana1(5),
    tip_cnp()..cnp(2850419390812) , 'Cretu', 'Livia',
    tip_adresa_bloc() ..codpostal('700112') ..localitate('Iasi') ..judet('Iasi') ..strada('Dependentei')
    ..nr('144') ..bloc('H14') ..scara('D') ..etaj('8') ..apartament(45)
);
INSERT INTO persoane1 (oid, cnp, nume, prenume, adresa) VALUES ( tip_persoana1(6),
    tip_cnp()..cnp(2831119390814) , 'Bratu', 'Ioana',
    tip_adresa_oras() ..codpostal('700112') ..localitate('Iasi') ..judet('Iasi') ..strada('Dependentei')
    ..nr('144')
);
```

Surprinzător este faptul că *DB2 Control Center* nu execută cea mai simplă formă de interogare a tabeli „tipate” – vezi figura 19.16.

```
SELECT p.* FROM persoanel p
|
|
|----- Commands Entered -----|
SELECT p.* FROM persoanel p;
|-----|
SELECT p.* FROM persoanel p
SQL2001SN  A transform group "DB2_PROGRAM" is not defined for data type
"MARIN.TIP_ADRESA_GENERALA".  SQLSTATE=42741
SQL2001SN  A transform group "DB2_PROGRAM" is not defined for data type "MARIN.TIP_ADRESA_GENERALA"
```

Figura 19.16. Imposibilitatea DB2 CC de a afișa înregistrările tabeli PERSOANE1

Putem, totuși, afișa orice atribut folosind modalitatea din funcțiile realizate în paragraful 19.1.4:

```
SELECT p.num, p.prenume, p.cnp..cnp, p.adresa..localitate, p.adresa..judet
FROM persoanel p
```

Similar celor discutate în Oracle, specificarea unui atribut sau metodă de pe o anumită ramură a ierarhiei poate fi realizată prin opțiunile *TREAT* și *IS OF*. Astfel, pentru a afișa (și) comunele persoanele cu adrese rurale, sintaxa este:

```
SELECT p.num, p.prenume, p.cnp..cnp, p.adresa..localitate, p.adresa..judet,
       TREAT (p.adresa AS tip_adresa_sat)..comuna
FROM persoanel p
WHERE p.adresa IS OF (tip_adresa_sat)
```

19.4. Metode asociate tipurilor structurate în Oracle

Am recurs la această formă deșănțată de favoritism nu numai din comoditate. În PostgreSQL și SQL Server nu există *CREATE METHOD*, iar pentru DB2 sursele bibliografice mi-au fost greu accesibile⁶. Așa că voi ilustra tema propusă doar în Oracle, re folosind tipuri create în paragraful 19.3.2 și adăugând, după nevoi, noi opțiuni și operatori.

Primul tip pe care-l înzestrăm cu metode este *tip_cnp* – vezi listing 19.21. Similar pachetelor PL/SQL, un tip are o parte de specificații, publică, și un corp (privat) care conține implementarea metodelor declarate în specificații.

⁶ Afirmația este ușor exagerată. Cea mai bună documentare a lucrului cu tipuri și tabele „tipate” am găsit-o în [IBM 2007-2], dar pe site-ul IBM sunt câteva articole interesante pe această temă, cum ar fi cel de la pagina: <http://www.ibm.com/developerworks/data/library/techarticle/dm-0506melnyk/>

Listing 19.21. *Tip_cnp*, într-o nouă prezentare (cu metode)

```

DROP TYPE tip_cnp FORCE
/
CREATE OR REPLACE TYPE tip_cnp AS OBJECT (
    cnp NUMBER(13),
    MEMBER FUNCTION getCNP RETURN NUMBER,
    MEMBER FUNCTION getSex RETURN CHAR,
    MEMBER FUNCTION getDataNasterii RETURN DATE,
    MEMBER FUNCTION getVirstaCurentaAni RETURN NUMBER,
    MEMBER FUNCTION getVirstaCurentaAniLuni RETURN NUMBER,
    MEMBER FUNCTION getVirstaCurentaAniLuni2 RETURN INTERVAL YEAR TO MONTH,
    MEMBER FUNCTION getVirstaCurentaSir RETURN VARCHAR2,
    MEMBER FUNCTION getVirstaRelativaAni (data_calcul_ DATE) RETURN NUMBER,
    MEMBER FUNCTION getVirstaAni RETURN NUMBER,
    MEMBER FUNCTION getVirstaAni (data_calcul_ DATE) RETURN NUMBER -- supraincarcare
) FINAL
/

CREATE OR REPLACE TYPE BODY tip_cnp AS
    -----
    MEMBER FUNCTION getCNP RETURN NUMBER IS
    BEGIN
        RETURN SELF.cnp ;
    END getCNP ;
    -----
    MEMBER FUNCTION getSex RETURN CHAR IS
    BEGIN
        -- se analizeaza primul caracter din CNP
        RETURN
            CASE SUBSTR(SELF.cnp,1,1)
                WHEN '1' THEN 'M'
                WHEN '2' THEN 'F'
                WHEN '5' THEN 'M'
                WHEN '6' THEN 'F'
            WHEN '7' THEN 'M'
                WHEN '8' THEN 'F'
                ELSE NULL
            END ;
    END getSex ;
    -----
    MEMBER FUNCTION getDataNasterii RETURN DATE IS
        v_data DATE ;
    BEGIN
        -- se decupeaza si interpreteaza ca data caracterele 2-7 din CNP
        v_data := TO_DATE(SUBSTR(SELF.cnp,2,6),'YYMMDD') ;

        IF SUBSTR(SELF.cnp,1,1) IN (1,2) THEN
            v_data := ADD_MONTHS(v_data, - 100 * 12) ;
        END IF ;
        RETURN v_data ;
    END getDataNasterii ;
    -----
    MEMBER FUNCTION getVirstaCurentaAni RETURN NUMBER IS
    BEGIN
        -- facem apel la metoda de mai sus
        RETURN EXTRACT (YEAR FROM
            (CURRENT_DATE - SELF.getDataNasterii()) YEAR TO MONTH
        ) ;

```



```

END getVirstaCurentaAni ;
-----
MEMBER FUNCTION getVirstaCurentaAniLuni RETURN NUMBER IS
BEGIN
  -- folosim doua metode anterioare
  RETURN SELF.getVirstaCurentaAni() +
    TRUNC ( (MONTHS_BETWEEN(CURRENT_DATE, SELF.getDataNasterii)
      - SELF.getVirstaCurentaAni() * 12) * 0.01, 2) ;
END getVirstaCurentaAniLuni ;
-----
MEMBER FUNCTION getVirstaCurentaAniLuni2 RETURN INTERVAL YEAR TO MONTH IS
BEGIN
  RETURN (CURRENT_DATE - SELF.getDataNasterii()) YEAR TO MONTH ;
END getVirstaCurentaAniLuni2 ;
-----
MEMBER FUNCTION getVirstaCurentaSir RETURN VARCHAR2 IS
BEGIN
  RETURN SELF.getVirstaCurentaAni() || ' ani si ' ||
    (SELF.getVirstaCurentaAniLuni() - SELF.getVirstaCurentaAni()) * 100
    || ' luni' ;
END getVirstaCurentaSir ;
-----
MEMBER FUNCTION getVirstaRelativaAni (data_calcul_ DATE) RETURN NUMBER IS
BEGIN
  RETURN TRUNC(MONTHS_BETWEEN(data_calcul_, SELF.getDataNasterii()) / 12, 0) ;
END getVirstaRelativaAni ;
-----
MEMBER FUNCTION getVirstaAni RETURN NUMBER IS
BEGIN
  RETURN SELF.getVirstaCurentaAni() ;
END getVirstaAni ;
-----
MEMBER FUNCTION getVirstaAni (data_calcul_ DATE) RETURN NUMBER
-- metoda difera de precedenta prin semnatura (are un parametru)
IS
BEGIN
  RETURN SELF.getVirstaRelativaAni(data_calcul_) ;
END getVirstaAni ;
END ;
/

```

Metodele asociate tipului valorifică încărcătura semantică a Codului Numeric Persoanal, extrăgând informații precum sexul și data nașterii și, pe baza lor, calculând vârsta curentă, exprimată în ani sau ani și luni, și vârsta relativă (vârsta pe care a avut-o sau o va avea „posesorul” CNP-ului la o anumită dată specificată).

Blocul anonim din listing 19.22 și figura 19.17 sunt destinate a ilustra o modalitate de folosirea a metodelor acestui tip de date.

Listing 19.22. Apelarea metodelor *tip_cnp*

```

-- un prima instantiere intr-un bloc PL/SQL a unui obiect de tip TIP_CNP
DECLARE
  a tip_cnp ;
  b tip_cnp := tip_cnp(2871010370709) ;
BEGIN
  -- apelam constructorul implicit

```

```

a := tip_cnp (1660210390802) ;

-- apelam metode ale tipului TIP_CNP
DBMS_OUTPUT.PUT_LINE('Sex: ' || a.getSex());
DBMS_OUTPUT.PUT_LINE('DataNasterii: ' || a.getDataNasterii());
DBMS_OUTPUT.PUT_LINE('Virsta - ANI - la data curenta: ' || a.getVirstaCurentaAni());
DBMS_OUTPUT.PUT_LINE('Virsta - ANI.LUNI - la data curenta: ' || a.getVirstaCurentaAniLuni());
DBMS_OUTPUT.PUT_LINE('Virsta - ANI-LUNI2 - la data curenta: ' || a.getVirstaCurentaAniLuni2());
DBMS_OUTPUT.PUT_LINE('Virsta (sir) - la data curenta: ' || a.getVirstaCurentaSir());
DBMS_OUTPUT.PUT_LINE('Virsta (ani impliniti) - la 1 ian 2005: ' ||
    a.getVirstaRelativaAni(DATE'2005-01-01'));
-- testam supraincarcarea functiei getVirstaAni
DBMS_OUTPUT.PUT_LINE('Virsta (ani impliniti) curenta: ' || a.getVirstaAni());
DBMS_OUTPUT.PUT_LINE('Virsta (ani impliniti) - la 1 ian 2005: ' ||
    a.getVirstaAni(DATE'2005-01-01'));
END ;

```

```

DECLARE
a tip_cnp ;
b tip_cnp := tip_cnp(2871010370709) ;
BEGIN
a := tip_cnp (1660210390802) ;
-- apelam metode ale tipului TIP_CNP
DBMS_OUTPUT.PUT_LINE('Sex: ' || a.getSex());
DBMS_OUTPUT.PUT_LINE('DataNasterii: ' || a.getDataNasterii());
DBMS_OUTPUT.PUT_LINE('Virsta - ANI - la data curenta: ' || a.getVirstaCurentaAni());
DBMS_OUTPUT.PUT_LINE('Virsta - ANI.LUNI - la data curenta: ' || a.getVirstaCurentaAniLuni());
DBMS_OUTPUT.PUT_LINE('Virsta - ANI-LUNI2 - la data curenta: ' || a.getVirstaCurentaAniLuni2());
DBMS_OUTPUT.PUT_LINE('Virsta (sir) - la data curenta: ' || a.getVirstaCurentaSir());
DBMS_OUTPUT.PUT_LINE('Virsta (ani impliniti) - la 1 ian 2005: ' ||
    a.getVirstaRelativaAni(DATE'2005-01-01'));
-- testam supraincarcarea functiei getVirstaAni
DBMS_OUTPUT.PUT_LINE('Virsta (ani impliniti) curenta: ' || a.getVirstaAni());
DBMS_OUTPUT.PUT_LINE('Virsta (ani impliniti) - la 1 ian 2005: ' ||
    a.getVirstaAni(DATE'2005-01-01'));
END ;

```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Buffer Size: 20000 Poll

```

Sex: M
DataNasterii: 10-02-1966
Virsta - ANI - la data curenta: 43
Virsta - ANI.LUNI - la data curenta: 43
Virsta - ANI-LUNI2 - la data curenta: +43-00
Virsta (sir) - la data curenta: 43 ani si 0 luni
Virsta (ani impliniti) - la 1 ian 2005: 38
Virsta (ani impliniti) curenta: 43
Virsta (ani impliniti) - la 1 ian 2005: 38

```

Figura 19.17. Apelul unui bloc anonim pentru verificarea metodelor *tip_cnp*

Continuăm cu declararea câtorva metode pentru tipul rădăcină destinat adreselor – *tip_adresa_generala*. Fiind neinstantiabil, declararea de metode-membru ar părea de prisos. Nu este chiar așa, întrucât toate subtipurile sale vor moșteni atât atributele (proprietățile), cât și metodele. Dintre noutăți, amintim:

- metodă de tip procedură (până aici, și de aici încolo, metodele sunt/vor fi preponderent funcții), *afisare*;

- metodă de mapare, denumită *ordonare*, utilă la compararea a două instanțe a tipului (clasei) *tip_adresa_generala* (care, între noi fie vorba, n-o să existe niciodată, pentru că tipul este neinstantiabil.

Listing 19.22. Noile „dotări” ale *tip_adresa_generala*

```
-- 2. un tip de data compozit cu subtipuri - ierarhie (probabil, cel mai celebru) - ADRESA
=====
DROP TYPE tip_adresa_generala FORCE
/
CREATE OR REPLACE TYPE tip_adresa_generala AS OBJECT (
    codpostal CHAR(6),
    localitate VARCHAR2(25),
    judet VARCHAR2(25),
    MEMBER FUNCTION getCodPostal RETURN CHAR,
    MEMBER FUNCTION getLocalitate RETURN VARCHAR2,
    MEMBER FUNCTION getJudet RETURN VARCHAR2,
    MEMBER PROCEDURE afisare (SELF IN OUT NOCOPY tip_adresa_generala),
    MEMBER FUNCTION afisare_sir RETURN VARCHAR2,
    MAP MEMBER FUNCTION ordonare RETURN VARCHAR2
) NOT FINAL NOT INSTANTIABLE
/

=====
CREATE OR REPLACE TYPE BODY tip_adresa_generala AS

    MEMBER FUNCTION getCodPostal RETURN CHAR IS
    BEGIN
        RETURN SELF.CodPostal ;
    END getCodPostal ;

    MEMBER FUNCTION getLocalitate RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Localitate ;
    END getLocalitate ;

    MEMBER FUNCTION getJudet RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Judet ;
    END getJudet ;

    MEMBER PROCEDURE afisare (SELF IN OUT NOCOPY tip_adresa_generala) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Cod postal:' || SELF.CodPostal ||
            ', Localitate:' || SELF.localitate || ', judet:' || SELF.Judet );
    END afisare ;

    MEMBER FUNCTION afisare_sir RETURN VARCHAR2 IS
    BEGIN
        RETURN NULL ;
    END afisare_sir ;

    MAP MEMBER FUNCTION ordonare RETURN VARCHAR2 IS
    BEGIN
        RETURN judet || localitate || codpostal ;
    END ordonare ;

END ; -- TYPE BODY tip_adresa_generala
/
```

Continuăm cu *tip_adresa_sat*. În afara proprietății Comuna, noutățile acestui (sub)tip sunt metoda funcție ce returnează valoarea atributului Comuna (*getComuna*) și suprascrierea procedurii *afisare* și a funcțiilor *afisare_sir* și *ordonare* (vezi listing 19.23).

Listing 19.23. „Re-formatarea” *tip_adresa_sat*

```

===== subtipuri ale TIP_ADRESA_GENERALA =====
DROP TYPE tip_adresa_sat FORCE
/
-- pentru TIP_ADRESA_SAT introducem atributul COMUNA si suprascriem procedura de afisare
CREATE OR REPLACE TYPE tip_adresa_sat UNDER tip_adresa_generala (
    comuna VARCHAR2(30),
    MEMBER FUNCTION getComuna RETURN VARCHAR2,
    OVERRIDING MEMBER PROCEDURE afisare (SELF IN OUT NOCOPY tip_adresa_sat),
    OVERRIDING MEMBER FUNCTION afisare_sir RETURN VARCHAR2,
    OVERRIDING MAP MEMBER FUNCTION ordonare RETURN VARCHAR2
)
FINAL
/

=====
CREATE OR REPLACE TYPE BODY tip_adresa_sat AS
    MEMBER FUNCTION getComuna RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Comuna ;
    END getComuna ;

    OVERRIDING MEMBER PROCEDURE afisare (SELF IN OUT NOCOPY tip_adresa_sat) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Cod postal:' || SELF.CodPostal ||
        ', comuna:' || SELF.comuna || ', sat:' || SELF.localitate ||
        ', judet:' || SELF.Judet );
    END afisare ;

    OVERRIDING MEMBER FUNCTION afisare_sir RETURN VARCHAR2 IS
    BEGIN
        RETURN 'Cod postal:' || SELF.CodPostal ||
        ', comuna:' || SELF.comuna || ', sat:' || SELF.localitate ||
        ', judet:' || SELF.Judet;
    END afisare_sir ;

    OVERRIDING MAP MEMBER FUNCTION ordonare RETURN VARCHAR2 IS
    BEGIN
        RETURN judet || comuna || localitate ;
    END ordonare ;

    END ; -- TYPE BODY tip_adresa_sat
/

```

Fiind un tip instanțiable, putem testa modul în care funcționează metodele de tip procedură și funcție ale *tip_adresa_sat* – vezi listing 19.24 și figura 19.18.

Listing 19.24. Apelarea metodelor *tip_adresa_sat*

```

DECLARE
    a tip_adresa_sat := tip_adresa_sat ('600515', 'Jorasti', 'Vrancea', 'Vinatori') ;
    b tip_adresa_sat := tip_adresa_sat ('600514', 'Radulesti', 'Vrancea', 'Vinatori') ;
BEGIN

```

```

DBMS_OUTPUT.PUT_LINE('Metoda 1 de afisare - procedura');
-- apel metoda-procedura
a.afisare ;
-- apel metoda-functie
DBMS_OUTPUT.PUT_LINE('Metoda 2 de afisare - functie');
DBMS_OUTPUT.PUT_LINE(a.afisare_sir);

-- folosirea metodei de tip MAP pentru comparatie
DBMS_OUTPUT.PUT_LINE('=====');
DBMS_OUTPUT.PUT_LINE('a=' || a.afisare_sir);
DBMS_OUTPUT.PUT_LINE('b=' || b.afisare_sir);
IF a > b THEN
    DBMS_OUTPUT.PUT_LINE('a > b');
ELSE
    DBMS_OUTPUT.PUT_LINE('a <= b');
END IF ;
END ;
/

```

```

Metoda 1 de afisare - procedura
Cod postal:600515, comuna:Vinatori, sat:Jorasti, judet:Vrancea
Metoda 2 de afisare - functie
Cod postal:600515, comuna:Vinatori, sat:Jorasti, judet:Vrancea
=====
a=Cod postal:600515, comuna:Vinatori, sat:Jorasti, judet:Vrancea
b=Cod postal:600514, comuna:Vinatori, sat:Radulesti, judet:Vrancea
a <= b

```

Figura 19.18. Apelul unui bloc anonim pentru verificarea metodelor *tip_adresa_sat*

Fără comentarii suplimentare, în listing-urile 19.25 și 19.26 sunt re-create *tip_adresa_oras* și *tip_adresa_bloc*, cu metodele de rigoare.

Listing 19.25. Noul *tip_adresa_oras*

```

DROP TYPE tip_adresa_oras FORCE
/
CREATE OR REPLACE TYPE tip_adresa_oras UNDER tip_adresa_generala (
    strada VARCHAR2(35),
    nr VARCHAR2(10),
    MEMBER FUNCTION getStrada RETURN VARCHAR2,
    MEMBER FUNCTION getNr RETURN VARCHAR2,
    OVERRIDING MEMBER PROCEDURE afisare (SELF IN OUT NOCOPY tip_adresa_oras),
    OVERRIDING MEMBER FUNCTION afisare_sir RETURN VARCHAR2,
    OVERRIDING MAP MEMBER FUNCTION ordonare RETURN VARCHAR2
)
NOT FINAL
/

-----
CREATE OR REPLACE TYPE BODY tip_adresa_oras AS
    -----
    MEMBER FUNCTION getStrada RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Strada ;
    END getStrada ;

    -----
    MEMBER FUNCTION getNr RETURN VARCHAR2 IS
    BEGIN

```

```

        RETURN SELF.Nr ;
    END getNr ;
-----
    OVERRIDING MEMBER PROCEDURE afisare (SELF IN OUT NOCOPY tip_adresa_oras) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Str:' || SELF.Strada || ', nr:' || SELF.nr ||
            ', oras:' || SELF.localitate || ', judet:' || SELF.Judet ||
            ', cod postal:' || SELF.CodPostal );
    END afisare ;
-----
    OVERRIDING MEMBER FUNCTION afisare_sir RETURN VARCHAR2      IS
    BEGIN
        RETURN 'Str:' || SELF.Strada || ', nr:' || SELF.nr ||
            ', oras:' || SELF.localitate || ', judet:' || SELF.Judet ||
            ', cod postal:' || SELF.CodPostal ;
    END afisare_sir ;
-----
    OVERRIDING MAP MEMBER FUNCTION ordonare RETURN VARCHAR2 IS
    BEGIN
        RETURN judet || localitate || strada || nr ;
    END ordonare ;
-----
END ; -- TYPE BODY tip_adresa_oras
/

```

Listing 19.26. *Tip_adresa_bloc* într-o nouă prezentare

```

DROP TYPE tip_adresa_bloc FORCE
/
CREATE OR REPLACE TYPE tip_adresa_bloc UNDER tip_adresa_oras (
    bloc VARCHAR2(20),
    scara VARCHAR2(10),
    etaj VARCHAR2(10),
    apartament VARCHAR2(10),
    MEMBER FUNCTION getBloc RETURN VARCHAR2,
    MEMBER FUNCTION getScara RETURN VARCHAR2,
    MEMBER FUNCTION getEtaj RETURN VARCHAR2,
    MEMBER FUNCTION getApartament RETURN VARCHAR2,
    OVERRIDING MEMBER PROCEDURE afisare (SELF IN OUT NOCOPY tip_adresa_bloc),
    OVERRIDING MEMBER FUNCTION afisare_sir RETURN VARCHAR2,
    OVERRIDING MAP MEMBER FUNCTION ordonare RETURN VARCHAR2
)
FINAL
/
-----
CREATE OR REPLACE TYPE BODY tip_adresa_bloc AS

    MEMBER FUNCTION getBloc RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Bloc ;
    END getBloc ;
-----
    MEMBER FUNCTION getScara RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Scara ;
    END getScara ;
-----
    MEMBER FUNCTION getEtaj RETURN VARCHAR2 IS
    BEGIN

```

```

        RETURN SELF.Etaj ;
    END getEtaj ;

-----
    MEMBER FUNCTION getApartament RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Apartament ;
    END getApartament ;

-----
    OVERRIDING MEMBER PROCEDURE afisare (SELF IN OUT NOCOPY tip_adresa_bloc) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Str:' || SELF.Strada || ', nr:' || SELF.nr);
        DBMS_OUTPUT.PUT_LINE('    Bloc:' || SELF.bloc || ', scara:' || SELF.scara ||
            '    etaj:' || SELF.etaj || ', apart:' || SELF.apartament );
        DBMS_OUTPUT.PUT_LINE('    Oras:' || SELF.localitate || ', judet:' || SELF.Judet ||
            ', cod postal:' || SELF.CodPostal );
    END afisare ;

-----
    OVERRIDING MEMBER FUNCTION afisare_sir RETURN VARCHAR2      IS
    BEGIN
        RETURN 'Str.' || SELF.Strada || ', nr.' || SELF.nr || ', bl.' || SELF.bloc ||
            ', sc.' || SELF.scara || ', etaj ' || SELF.etaj || ', ap.' || SELF.apartament ||
            ', loc.' || SELF.localitate || ', jud.' || SELF.Judet ||
            ', cod ' || SELF.CodPostal ;
    END afisare_sir ;

-----
    OVERRIDING MAP MEMBER FUNCTION ordonare RETURN VARCHAR2 IS
    BEGIN
        RETURN judet || localitate || strada || nr || bloc || scara || apartament ;
    END ordonare ;

-----
END ; -- TYPE BODY tip_adresa_bloc
/

```

Nu puteam lăsa de izbeliște tipurile colecții, așa că, pornind de la un tip „simplu”, *tip_telefon*, vor crea un tip colecție, *tip_telefoane*. Specificațiile și corpul primului sunt incluse în listing 19.27. Metoda *getRetea* este, astăzi, mai puțin relevantă, întrucât clienții pot schimba rețeaua de telefonie fără a modifica numărul.

Listing 19.27. Specificațiile și corpul *tip_telefon*

```

DROP TYPE tip_telefon FORCE
/
CREATE OR REPLACE TYPE tip_telefon AS OBJECT (
    numar CHAR(10),
    MEMBER FUNCTION getNumar RETURN CHAR,
    MEMBER FUNCTION getRetea RETURN VARCHAR2,
    MEMBER FUNCTION getPrefixJud RETURN CHAR
) FINAL
/
CREATE OR REPLACE TYPE BODY tip_telefon AS
-----
    MEMBER FUNCTION getNumar RETURN CHAR IS
    BEGIN
        RETURN SELF.Numar ;
    END getNumar ;

-----
    MEMBER FUNCTION getRetea RETURN VARCHAR2 IS

```

```

v_prefix CHAR(4);
BEGIN
v_prefix := SUBSTR(SELF.Numar,1,4);
CASE
WHEN SUBSTR(v_prefix,1,3) = '023' THEN -- ROMTELECOM
RETURN 'ROMTELECOM';
WHEN SUBSTR(v_prefix,1,3) IN ('074', '075') THEN -- ORANGE
RETURN 'ORANGE';
WHEN SUBSTR(v_prefix,1,3) IN ('072', '073') THEN -- VODAFONE
RETURN 'VODAFONE';
WHEN SUBSTR(v_prefix,1,3) IN ('078') THEN -- Zapp
RETURN 'ZAPP';
END CASE;
END getRetea;
-----
MEMBER FUNCTION getPrefixJud RETURN CHAR IS
BEGIN
IF SUBSTR(SELF.Numar,1,2) = '02' THEN
RETURN SUBSTR(SELF.Numar,1,4);
ELSE
RETURN NULL;
END IF;
END getPrefixJud;
-----
END; -- TYPE BODY tip_telefon
/

```

O persoană poate avea oricâte telefoane și faxuri (numai să dispună de bani și buzunare suficiente), așa că apelăm la un tip de colecție din categoria NESTED TABLE:

```
DROP TYPE tip_telefoane FORCE;
```

```
CREATE OR REPLACE TYPE tip_telefoane IS TABLE OF tip_telefon;
```

Analog procedăm pentru adresele de poștă electronică:

```
DROP TYPE tip_email FORCE;
```

```
CREATE OR REPLACE TYPE tip_email AS OBJECT ( email VARCHAR2(100));
```

```
DROP TYPE tip_emailuri FORCE;
```

```
CREATE OR REPLACE TYPE tip_emailuri IS TABLE OF tip_email;
```

Și acum ajungem la tipul-concluzie al discuției noastre – *tip_persoana_complet* definit ca un sub-tip al *tip_persoana1* din paragraful 19.1.2. Partea cea mai interesantă este prezența a două metode statice, una (*getNrPersoane*) care furnizează numărul de înregistrări ale tabelului PERSONANE3 și a doua (*getCineAreTelefonul*) „cotrobăie” tot în PERSONAL3 după persoana ce are un număr de telefon parametru. Mai ciudat este că aceste două metode interoghează o tabelă care nu există și pe care dorim să o creăm prin clauza *OF tip_persoana_complet*. Pentru acest gen de probleme, Oracle permite declararea incompletă a tipului. Mai întâi, declarăm numai specificațiile – vezi listing 19.28.

Listing 19.28. (Doar) Specificațiile *tip_persoana_complet*

```

DROP TYPE tip_persoana_complet FORCE
/

```



```

CREATE OR REPLACE TYPE tip_persoana_complet UNDER tip_persoana1 (
    telefoane tip_telefoane,
    faxuri tip_telefoane,
    emailuri tip_emailuri,
    MEMBER FUNCTION getNume RETURN VARCHAR2,
    MEMBER FUNCTION getPrenume RETURN VARCHAR2,
    MEMBER FUNCTION getNumePren RETURN VARCHAR2,
    MEMBER FUNCTION getToataAdresa RETURN tip_adresa_generala,
    MEMBER FUNCTION afisare_sir RETURN VARCHAR2,
    MEMBER FUNCTION getTelefoane RETURN tip_telefoane,
    MEMBER FUNCTION getFaxuri RETURN tip_telefoane,
    MEMBER FUNCTION getEmailuri RETURN tip_emailuri,
    STATIC FUNCTION getNrPersoane RETURN NUMBER,
    STATIC FUNCTION getCineAreTelefonul (tel_ tip_telefon) RETURN tip_persoana_complet
) NOT FINAL
/

```

Acum putem crea tabela „tipată”:

```

CREATE TABLE persoane3 OF tip_persoana_complet
    (PRIMARY KEY (cnp.cnp))
    NESTED TABLE telefoane STORE AS telefoane_nt
    NESTED TABLE faxuri STORE AS faxuri_nt
    NESTED TABLE emailuri STORE AS emailuri_nt

```

după care se compilează corpul tipului (listing 19.29).

Listing 19.28. Corpul *tip_persoana_complet*

```

-- corpul tipului TIP_PERSOANA_COMPLET
CREATE OR REPLACE TYPE BODY tip_persoana_complet AS
    -----
    MEMBER FUNCTION getNume RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Nume ;
    END getNume ;
    -----
    MEMBER FUNCTION getPrenume RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Prenume ;
    END getPrenume ;
    -----
    MEMBER FUNCTION getNumePren RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.Nume || ' ' || SELF.Prenume ;
    END getNumePren ;
    -----
    MEMBER FUNCTION getToataAdresa RETURN tip_adresa_generala IS
    BEGIN
        RETURN SELF.Adresa ;
    END getToataAdresa ;
    -----
    MEMBER FUNCTION afisare_sir RETURN VARCHAR2 IS
    BEGIN
        RETURN SELF.prenume || ' ' || SELF.numa || ', sex:' || SELF.cnp.getSex()
            || ', virsta:' || SELF.cnp.getVirstaAni () || ' ani, ' || SELF.adresa.afisare_sir ;
    END afisare_sir ;
    -----

```

```

MEMBER FUNCTION getTelefoane RETURN tip_telefoane IS
BEGIN
    RETURN SELF.telefoane ;
END getTelefoane ;

-----

MEMBER FUNCTION getFaxuri RETURN tip_telefoane IS
BEGIN
    RETURN SELF.faxuri ;
END getFaxuri ;

-----

MEMBER FUNCTION getEmailuri RETURN tip_emailuri IS
BEGIN
    RETURN SELF.emailuri ;
END getEmailuri ;

-----

STATIC FUNCTION getNrPersoane RETURN NUMBER IS
    v_nr NUMBER(4) := 0 ;
BEGIN
    SELECT COUNT(*) INTO v_nr FROM persoane3 ;
    RETURN v_nr;
END getNrPersoane ;

-----

STATIC FUNCTION getCineAreTelefonul (tel_ tip_telefon) RETURN tip_persoana_complet
IS
    v_pers tip_persoana_complet ;
BEGIN
    SELECT object_value INTO v_pers FROM persoane3 p
    WHERE p.cnp.cnp IN
        (SELECT p.cnp.cnp
         FROM persoane3 p, TABLE (p.telefoane) t
         WHERE RTRIM(t.numar)= RTRIM(tel_.getNumar()) ) ;
    RETURN v_pers ;

END getCineAreTelefonul ;

END ; -- TYPE BODY tip_persoana_complet
/

```

Inserarea înregistrărilor în PERSOANE3 se realizează de maniera prezentată în listing 19.29.

Listing 19.29. Insererea de înregistrări în PERSOANE3

```

DELETE FROM persoane3 ;
INSERT INTO persoane3 VALUES (NEW tip_persoana_complet (
    tip_cnp (1760210390803),
    'Popescu', 'Ioneliu',
    tip_adresa_sat(123457, 'Jorasti', 'Vrancea', 'Vinatori'),
    tip_telefoane (tip_telefon('0744123454'),
                  tip_telefon('0745123456'),
                  tip_telefon('0232222222')
    ),
    NULL, -- n-are fax !
    tip_emailuri(tip_email('popescu_i@d.ro'))
));
INSERT INTO persoane3 VALUES (NEW tip_persoana_complet (
    tip_cnp (1660210390802),
    'Ionescu', 'Vasile',
    tip_adresa_sat(123456, 'Vinatori', 'Vrancea', 'Vinatori'),

```

```

        tip_telefoane (tip_telefon('0744123455'), -- tel.
                        tip_telefon('0788123456'),
                        tip_telefon('0232222225')
                    ),
        tip_telefoane (tip_telefon('0232222225') -- fax
                    ),
        tip_emailuri(tip_email('ionescu_v@d.ro'), tip_email('io201@yahoo.com'))
    ));
INSERT INTO persoane3 VALUES (NEW tip_persoana_complet (
    tip_cnp (2821211390804),
    'Iatu', 'Vasilica',
    tip_adresa_sat(123458, 'Jaristea', 'Vrancea', 'Jaristea'),
    NULL, -- n-are tel.
    NULL, -- n-are fax
    tip_emailuri() -- n-are adrese de e-mail
));
INSERT INTO persoane3 VALUES (NEW tip_persoana_complet (
    tip_cnp (1781005390810),
    'Cretu', 'Liviu',
    tip_adresa_bloc(700111, 'Iasi', 'Iasi', 'Independentei', '11bis',
    'G4', 'B', 'parter', 3),
    tip_telefoane (tip_telefon('0744293455'), -- tel.
                    tip_telefon('0788123456'),
                    tip_telefon('0723128956'),
                    tip_telefon('0232277925')
                ),
    tip_telefoane (tip_telefon('0232277925') -- fax
                ),
    tip_emailuri(tip_email('cretu_liviu45@d.ro'), tip_email('cl6788@yahoo.com'))
));
INSERT INTO persoane3 VALUES (NEW tip_persoana_complet (
    tip_cnp (2850419390812),
    'Cretu', 'Livia',
    tip_adresa_bloc(700112, 'Iasi', 'Iasi', 'Dependentei', '144',
    'H14', 'D', '8', 45),
    tip_telefoane (tip_telefon('0744298855'), -- tel.
                    tip_telefon('0788555456'),
                    tip_telefon('0728888956'),
                    tip_telefon('0235577925')
                ),
    tip_telefoane (tip_telefon('0232557725') -- fax
                ),
    tip_emailuri(tip_email('cretu_livia@doi.ro'), tip_email('livia23@yahoo.com'))
));
INSERT INTO persoane3 VALUES (NEW tip_persoana_complet (
    tip_cnp (2831119390814),
    'Bratu', 'Ioana',
    tip_adresa_oras(700112, 'Iasi', 'Iasi', 'Dependentei', '144'),
    tip_telefoane (tip_telefon('0744298855'), -- tel.
                    tip_telefon('0788777456'),
                    tip_telefon('0728999956'),
                    tip_telefon('0235577929')
                ),
    tip_telefoane (), -- fax
    tip_emailuri()
));

```

Dacă în paragraful 19.3.2 afișarea valorilor atributelor dintr-o tip structurat era realizată într-o manieră SQL-istă, prin „calificarea” numele atribulelor, acum putem redacta și variante mai aproape de filosofia OO în care valorile atributelor se obțin prin invocarea metodelor:

```
SELECT p.num, p.prenume, p.cnp.getSex() AS Sex,
       p.cnp.getDataNasterii() AS DataNasterii,
       p.cnp.getVirstaCurentaSir() AS Virsta
FROM persoane3 p
```

În privința metodelor statice (cele care nu se aplică la nivel de instanță, ci la nivel de clasă/tip), acestea pot fi invocate atât din blocuri anonime:

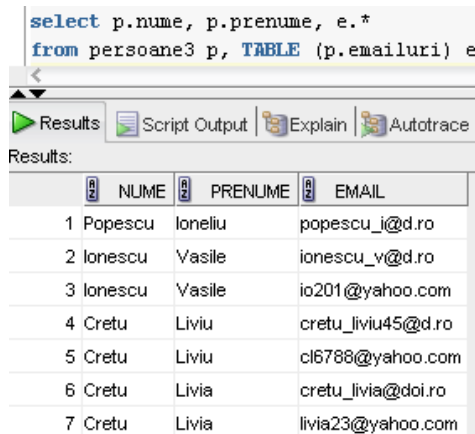
```
BEGIN
  DBMS_OUTPUT.PUT_LINE('Nr inregistrarilor din tabela PERSOANE este '
    || tip_persoana_complet.getNrPersoane() );
END ;
```

cât și dintr-o interogare:

```
SELECT tip_persoana_complet.getNrPersoane() FROM dual ;
```

După cum am discutat în paragraful 19.2.2, conversia masivelor de tip NESTED TABLES și VARRAYS în tabele presupune folosirea funcției TABLE. Astfel, pentru afișarea sub forma de tabelă a adreselor de e-mail ale tuturor persoanelor (figura 19.19) interogarea are forma:

```
SELECT p.num, p.prenume, e.*
FROM persoane3 p, TABLE (p.emailuri) e
```



	NUM	PRENUME	EMAIL
1	Popescu	Ioneliu	popescu_i@d.ro
2	Ionescu	Vasile	ionescu_v@d.ro
3	Ionescu	Vasile	io201@yahoo.com
4	Cretu	Liviu	cretu_liviu45@d.ro
5	Cretu	Liviu	cl6788@yahoo.com
6	Cretu	Livia	cretu_livia@doi.ro
7	Cretu	Livia	livia23@yahoo.com

Figura 19.19. Afișarea sub formă

Pentru extragerea numai a adreselor Liviei Cretu sunt disponibile două variante:

```
SELECT e.*
FROM persoane3 p, TABLE (p.emailuri) e
WHERE num='Cretu' AND prenume='Livia'
```

sau

```
SELECT *
FROM TABLE (
    SELECT p.emailuri
    FROM persoane3 p
    WHERE nume='Cretu' AND prenume='Livia'
)
```

Pentru a afla persoanele care au numere de telefon în rețeaua ORANGE și numerele respective, o sintaxă funcțională este:

```
SELECT p.nume, p.prenume, t.getNumar() AS Numar
FROM persoane3 p, TABLE (p.telefoane) t
WHERE t.getRetea() = 'ORANGE'
```

Pentru a pune la treabă cea de-a doua metodă statică, ne interesează numele de familie a persoanei care folosește telefonul cu numărul 0235577929, lucrul posibil, după cum am văzut deja, atât printr-un bloc anonim:

```
DECLARE
    a tip_persoana_complet ;
BEGIN
    a := tip_persoana_complet.getCineAreTelefonul(tip_telefon('0235577929'));
    dbms_output.put_line (a.nume) ;
END ;
```

Cât și dintr-o interogare

```
SELECT tip_persoana_complet.getCineAreTelefonul(tip_telefon('0235577929')).nume
FROM dual
```

Încheiem cu un exemplu de actualizarea colecțiilor. Adăugăm un număr de telefon din rețeaua ZAPP lui Popescu Ioneliu:

```
INSERT INTO TABLE (
    SELECT p.telefoane
    FROM persoane3 p
    WHERE p.nume='Popescu' AND p.prenume='Ioneliu'
)
VALUES (tip_telefon('0788111111'));
```

și, pentru compensare, îi ștergem lui Popescu Ioneliu al doilea număr Orange:

```
DELETE FROM TABLE (
    SELECT p.telefoane
    FROM persoane3 p
    WHERE p.nume='Popescu' AND p.prenume='Ioneliu'
)
WHERE numar = '0745123456' ;
```

În fine, pentru a modifica prima adresă de e-mail a Liviei Cretu din 'cretu_livia@doi.ro' în 'cretu_livia@trei.com' sintaxa este:

```
UPDATE TABLE (SELECT p.emailuri FROM persoane3 p
                WHERE p.num='Cretu' AND p.prenume='Livia'
                ) e
SET VALUE(e) = tip_email('cretu_livia@trei.com')
WHERE email= 'cretu_livia@doi.ro'
```

Au rămas multe alte ingrediente O-R nediscutate: tipuri REF, constructori expliți, tabele virtuale de obiecte, modificarea tipurilor etc. Din acest motiv, vă puteți declara nemulțumiți. Dacă, însă, luăm în considerare că niciuna din cărțile faimoase de SQL pe care le-am amintit în prefață nu oferă nici măcar o fracțiune din ceea ce am discutat în acest capitol, atunci avem motive să fim optimiști.