Capitolul 16. Funcții, proceduri şi pachete stocate

Dacă procedura sau funcția reprezintă noțiuni vehiculate de multe decenii de informaticienii cu sau fără acte în regulă, asocierea acestora cu persistența (bazele de date) a fost consacrată de SGBD-urile deceniului al nouălea din precedentul secol. Stocarea se referă la "înăuntrul" bazei, la *dicționarul de date* sau *catalogul sistemului*, cum i se spune la case (de soft) mai mari. Deşi puternic impregnate cu SQL, funcțiile şi procedurile stocate sunt, după cum bine le zice numele, cam… procedurale, depinzând într-o măsură decisivă de extensiile proprietare ale fiecărui SGBD. Şi termenul de procedură nu e tocmai în regulă deoarece partea se regăseşte în întreg, adică *procedurile stocate* sunt, în general, de trei tipuri: funcții, *proceduri* şi declanşatoare. Mai nimerit ar fi termenul *module stocate*, dar dacă nu am fost prea pedanți până în acest capitol, ce rost mai are acum.

În acest capitol ne ocupăm de funcții, proceduri şi pachete (pachete în sens *oraclist*), iar în capitolul următor vom trata declanşatoarele (trigger-ele). Cu acest capitol ieşim mult de sub incidența standardelor SQL şi intrăm pe proprietățile serverelor de baze de date. De peste zece ani, standardul SQL are o componentă specială – *Persistent Stored Modules* – care încearcă să fie un numitor comun al dialectelor procedurale ale serverelor BD. Procesul de armonizare este vizibil, însă diferențele dintre sistemele de gestiune a bazelor de date sunt încă semnificative, uneori uriaşe.

Nu mi-am propus, şi dealtfel nici nu mă pricep, să tratez în extenso subiectul procedularității în cele patru servere. Însă următoarele câteva exemple pot constitui un bun punct de plecare pentru cei care vor să facă performanță. Ca lucrări de referință v-aş recomanda, printre altele:

- pentru DB2 – limbajul SQL PL: [Mullins2004], [Janmohamed s.a. 2004], [Birchall 2007], [Bedoya s.a. 2004];
- pentru Oracle – limbajul PL/SQL: [Lakshman 2003], [Greenwald s.a. 2005], [Price 2008], [Kyte 2005], [Feuerstein 2007], [Urman s.a. 2004], [McLaughlin 2008];
- pentru PostgreSQL – limbajul PL/pgSQL: [Geschwinde & Schönig 2001], [Matthew & Stones 2005], [Douglas & Douglas 2005];
- pentru MS SQL Server – limbajul T-SQL: [Bain s.a. 2003], [Watt 2007], [Šunderi 2006], [Ben-Gan s.a. 2006], [Vieira 2007].

16.1. Funcții stocate

Funcțiile stocate reprezintă un ingredient al tuturor SGBD-urilor semnificative de astăzi. De obicei, în programare, se spune că o funcție este un bloc/program

care, la invocare, returnează (furnizează) o valoare. Valoarea poate fi una simplă, de tip înregistrare sau seturi de înregistrări. Pentru variație, modificăm ordinea de intrare în scenă a celor patru servere BD în abordarea subiectului. Vom începe cu PostgreSQL-ul, vom continua cu Oracle şi SQL Server şi vom încheia cu DB2.

16.1.1. Primele funcții stocate în PostgreSQL

În PostgreSQL funcțiile stocate pot fi create în mai multe limbaje: Perl, C etc. Pe noi ne interesează doar două dintre acestea, SQL şi PLpg/SQL. Cele mai simple sunt cele SQL. De obicei acestea sunt simple interogări parametrizate care returnează o valoare. Spre exemplu, în listing 16.1 apar două comenzi (CREATE FUNCTION) prin care se crează funcții. Prima, denumită *f_proctva*, primeşte, la invocare (apelare) valoarea unui cod de produs şi furnizează procentul său de TVA. A doua, *f_denpr*, primeşte un cod de produs şi furnizează denumirea acestuia.

Listing 16.1. Primele două funcţii stocate PostgreSQL redactate în limbajul SQL

```
-- F_PROCTVA -- pentru un cod de produs returneaza procentul de TVA al produsului respectiv
CREATE OR REPLACE FUNCTION f_proctva (codpr_ NUMERIC)  RETURNS NUMERIC AS $$
     SELECT proctva FROM produse WHERE codpr = $1
$$ LANGUAGE SQL ;

-- F_DENPR - primeste un cod de produs si returneaza denumirea
CREATE OR REPLACE FUNCTION f_denpr (codpr_ produse.codpr%TYPE)
     RETURNS produse.denpr%TYPE AS $$
     SELECT denpr FROM produse WHERE codpr = $1 ;
$$ LANGUAGE SQL ;
```

Ambele funcții folosesc ca parametru de intrare *CodPr_* care este definit direct, *NUMERIC*, în prima funcție şi indirect - *PRODUSE.CodPr%TYPE* -, în a doua. Acest din urmă mod de specificare a tipului unui parametru este poate mai lung, însă mai elegant. Dacă, ulterior, s-ar modifica tipul atributului CodPr din PRODUSE (din NUMERIC în VARCHAR (nu mă întrebați de ce)), ar trebui modificate toate funcțiile stocare care prezintă parametri de genul *CodPr_*, definiți

ca NUMERIC[1], în timp ce funcțiile în care parametrii au fost definiți indirect nu suferă modificări (eventual, ar trebui recompilate). De remarcat că, în corpul funcției, referirea la parametrul *codpr_* se realizează prin *$1* (dacă funcția ar fi avut trei parametri, referirea lor s-ar fi făcut prin *$2* şi *$3*)[2].

Lansarea în execuție a celor două comenzi CREATE FUNCTION poate fi realizată precum în figura 16.1, deşi o variantă mai simplă ar fi lucrul asistat (chiar dacă în *pgAdmin* lucrul asistat nu este întotdeauna o bagatelă).

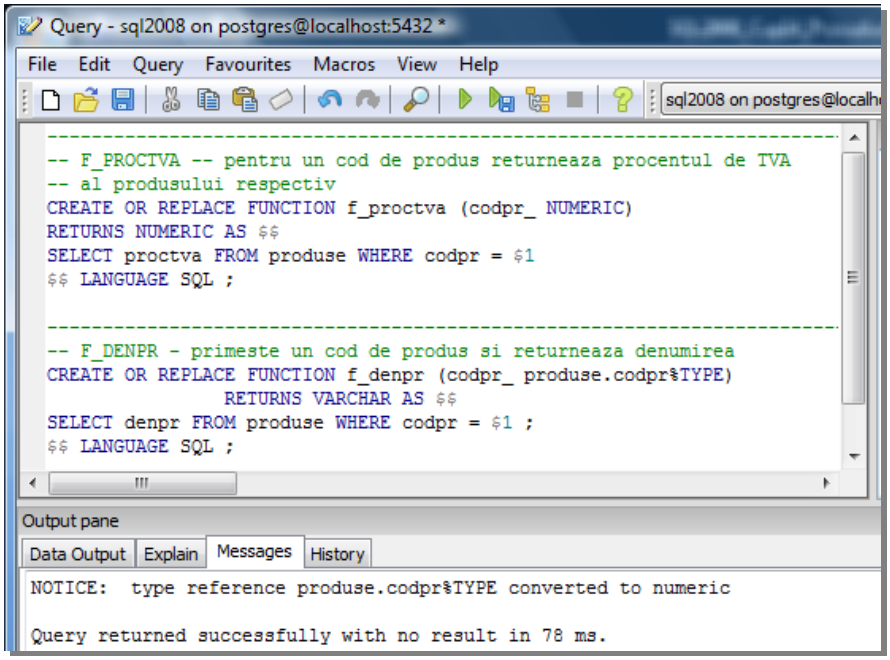


Figura 16.1. Crearea a două funcţii (SQL) simple (de tot) în PostgreSQL

[1] E un fel de mini-problemă a anului 2000 (Y2K) de care cei mai în vârstă dintre dvs. aţi auzit în tinerețe.

[2] La drept vorbind, numele parametrului poate lipsi, doar tipul său fiind important.

După lansarea acestor comenzi, în *pgAdmin*, cele două funcţii pot fi vizualizate ca în figura 16.2 (nu uitaţi de reîmprospătarea – *Refresh* - meniului arborescent). Odată creată, o funcţie poate fi apelată atât din altă funcţie, cât şi dintr-o interogare SQL:

```
SELECT nrfact, f_denpr(codpr) AS denpr, cantitate, pretunit,
       cantitate * pretunit AS Val_fara_tva,
       cantitate * pretunit * f_proctva(codpr) AS tvalinie,
       cantitate * pretunit * (1 + f_proctva(codpr)) AS Val_cu_TVA_linie
FROM liniifact
```

În PostgreSQL o funcţie SQL poate conţine mai multe comenzi SQL, însă valoarea, linia sau setul de linii sunt returnate pe baza ultimei comenzi SQL. De asemenea, dacă valoarea returnată declarată prin clauza RETURNS este una simplă (o valoare sau o înregistrare), iar ultimul SELECT al funcţiei extrage două sau mai multe întregistrări dintr-o tabelă, nu se va declanşa vreo eroare (precum în Oracle PL/SQL), ci se va returna prima linie din setul de înregistrări, ceea ce poate crea confuzie (mai ales în lipsa opţiunii ORDER BY în acestă ultimă interogare a funcţiei).

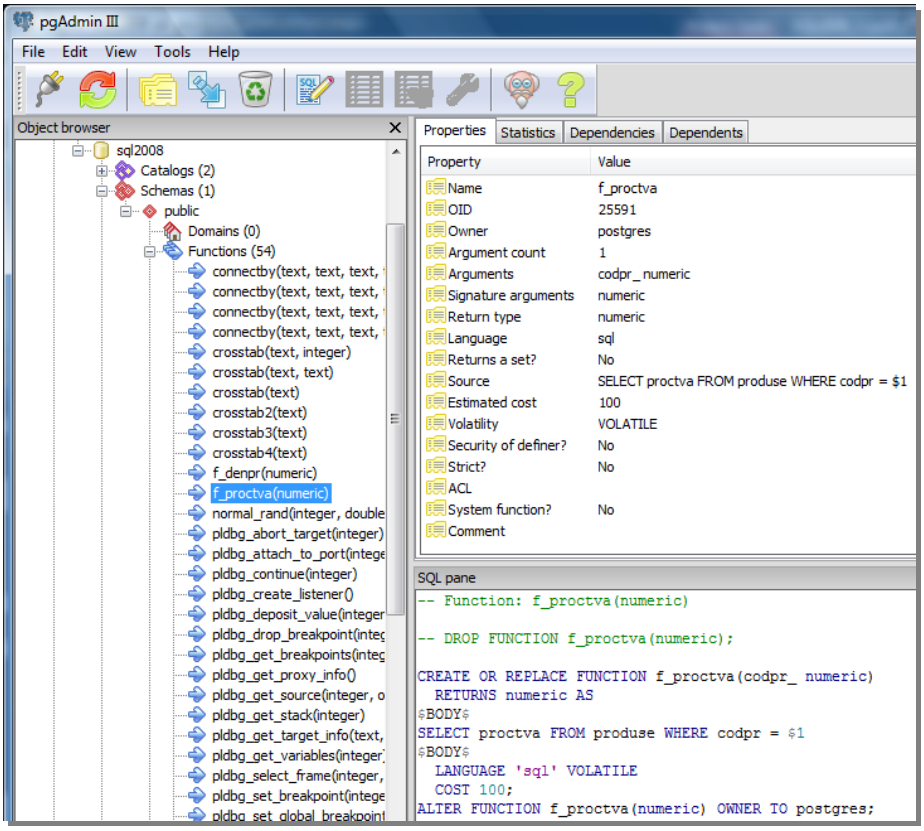


Figura 16.2. Afişarea celor două funcţii în pgAdmin

Să luăm, spre exemplu, funcția *f_test01* (listing 16.2) alcătuită din trei fraze SELECT destul de anapoda, care extrag, pe rând, înregistrările din tabelele CLIENȚI, PERSOANE şi FACTURI (ultima ar extrage numai valorile atributului NrFact din toate liniile tabelei FACTURI):

Listing 16.2. Funcţia SQL (din PostgreSQL) ***f_test01***

```
CREATE OR REPLACE FUNCTION f_test01 () RETURNS NUMERIC AS $$
    SELECT * FROM clienti ;
    SELECT * FROM persoane ;
    SELECT NrFact FROM facturi ;
$$ LANGUAGE SQL ;
```

E oarecum curios că sintaxa aceastei funcții este acceptată în PostgreSQL. Întrucât valoarea declarată în clauza RETURNS este simplă (numerică), apelul funcției nu se soldează cu eroare, ci cu vizualizarea unui număr de factură – vezi figura 16.3.

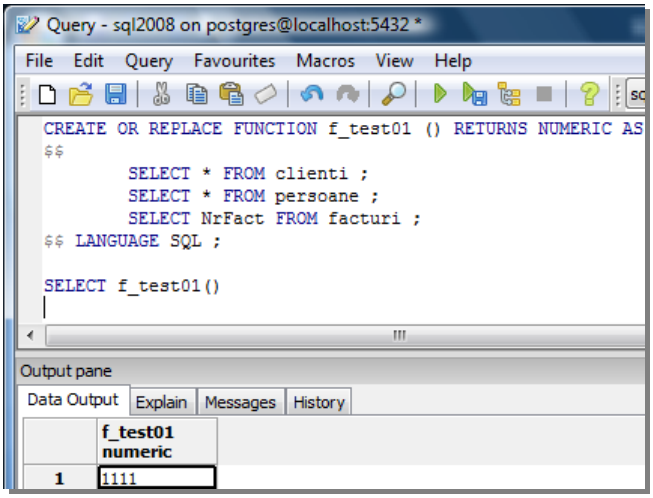


Figura 16.3. O funcţie Postgre(SQL) ciudată şi rezultatul său

Limbajul "clasic" de redactare a funcțiilor stocate complexe este, în PostgreSQL, PL/pgSQL. Sintaxa sa este similară, în multe privințe, limbajului Oracle PL/SQL. Reluăm cele două funcții anterioare şi le rescriem pe sintaxa PL/pgSQL – vezi listing 16.3.

Listing 16.3. Primele două funcţii „trecute” în PLpgSQL

```
-- F_PROCTVA -- pentru un cod de produs returneaza procentul de TVA al produsului
CREATE OR REPLACE FUNCTION f_proctva2 (codpr_ produse.codpr%TYPE)
    RETURNS produse.ProcTVA%TYPE AS $$
DECLARE
    v_proc produse.proctva%TYPE ;
BEGIN
```

```
    SELECT proctva INTO v_proc FROM produse WHERE codpr = codpr_ ;
    RETURN v_proc ;
END ;
$$ LANGUAGE plpgsql ;
-------------------------------------------------------------------------
-- F_DENPR - primeste un cod de produs si returneaza denumirea
CREATE OR REPLACE FUNCTION f_denpr2 (codpr_ produse.codpr%TYPE)
    RETURNS produse.denpr%TYPE AS $$
DECLARE
    v_denpr produse.denpr%TYPE ;
BEGIN
    SELECT denpr INTO v_denpr FROM produse WHERE codpr = codpr_ ;
    RETURN v_denpr ;
END ;
$$ LANGUAGE plpgsql;
```

Un bloc (program) scris în limbajul PL/pgSQL poate avea trei secţiuni; prima (opţională) este declarativă - începe cu DECLARE; a doua este cea „executabilă" - începe cu BEGIN; a treia este tot opţională şi este destinată tratării eventualelor erori ce pot fi declanşate de funcţie, erori denumite *excepţii*. Spre deosebire de funcţiile redactate în „limbajul" SQL, în PL/pgSQL parametrii sunt referiţi prin numele lor.

Ca şi în alte limbaje, putem să redactăm funcţii care să suplinească oferta sistemului. Spre exemplu, în PostgreSQL nu există funcţiile LEFT şi RIGHT (din SQL Server) care furnizeaza primele, respectiv ultimele *n* caractere dintr-un şir. Aşa că putem redacta noi două funcţii pe care le vom numi LEFT_ şi RIGHT_ (am adaugat „un underscore" tocmai pentru a nu intra în conflict cu o eventuală viitoare versiune PostgreSQL care va implementa cele două funcţii. Iată aceste două funcţii în listingul 16.4.

Listing 16.4. Funcţiile ***left_*** şi ***right_*** (în PLpgSQL)

```
-- Functia LEFT_ extrage primele cite_ caractare dintr-un sir_
CREATE OR REPLACE FUNCTION LEFT_ (sir_ TEXT, cite_ INTEGER) RETURNS TEXT
AS $$
DECLARE
    v_sir_returnat TEXT ;
BEGIN
     v_sir_returnat := SUBSTR(sir_, 1, cite_) ;
    RETURN v_sir_returnat ;
END ;
$$ LANGUAGE plpgsql ;

-- Functia RIGHT_ extrage ultimele cite_ caractare dintr-un sir_
CREATE OR REPLACE FUNCTION RIGHT_ (sir_ TEXT, cite_ INTEGER) RETURNS TEXT
AS $$
DECLARE
    v_sir_returnat TEXT ;
BEGIN
     v_sir_returnat := SUBSTR(sir_, LENGTH(sir_) - cite_ + 1, cite_) ;
    RETURN v_sir_returnat ;
END ;
$$ LANGUAGE plpgsql ;
```

Ilustrarea secvenţelor alternative şi iterative în PL/pgSQL ne „împinge” la redactarea unei funcţii care să rezolve o problemă destul de importantă – calcularea scadenţei facturilor peste un număr de zile *lucrătoare*. Funcţia de numeşte *f_data_scadenta* (vezi listing 16.5).

Listing 16.5. Funcţia PLpgSQL ***f_data_scadenta***

```
CREATE OR REPLACE FUNCTION f_data_scadenta (
        datai_ DATE, nrzile_scadenta_ NUMERIC)        RETURNS DATE AS $$
DECLARE
    zi_crt DATE ;
    v_gata BOOLEAN := FALSE ;
    v_nr_zile_lucratoare NUMERIC := 0 ;
BEGIN
    -- scadenta este peste NRZILE_SCADENTA_ (lucratoare)
    zi_crt := datai_  ;
    WHILE NOT v_gata LOOP
        zi_crt := zi_crt + 1 ;
        IF RTRIM(TO_CHAR(zi_crt, 'DAY'))  NOT IN ('SATURDAY','SUNDAY') THEN
                v_nr_zile_lucratoare := v_nr_zile_lucratoare + 1 ;
                IF v_nr_zile_lucratoare >= nrzile_scadenta_ THEN
                        EXIT ;
                END IF ;
        END IF ;
    END LOOP ;
    RETURN zi_crt ;
END ;
$$ LANGUAGE plpgsql ;
```

Dacă presupunem că scadenţa fiecărei facturi este peste 19 zile lucrătoare de la data întocmirii, atunci interogarea devine foarte simplă:

```
SELECT NrFact, DataFact,
f_data_scadenta (DataFact, 19) AS "Scadenta (19 zile lucratoare)"
FROM facturi
```

iar rezultatul este cel din figura 16.4.

nrfact numeric(8,0)	datafact date	Scadenta (19 zile lucratoare) date
1111	2007-08-01	2007-08-28
1112	2007-08-01	2007-08-28
1113	2007-08-01	2007-08-28
1114	2007-08-01	2007-08-28
1115	2007-08-02	2007-08-29
1116	2007-08-02	2007-08-29
1117	2007-08-03	2007-08-30
1118	2007-08-04	2007-08-30
1119	2007-08-07	2007-09-03
1120	2007-08-07	2007-09-03
1121	2007-08-07	2007-09-03
2111	2007-08-14	2007-09-10

Figura 16.4. Scadenţă peste 19 zile lucrătoare

Acum ne propunem să afişăm în dreptul fiecărei facturi, sub formă de şir de caractere lista produselor (denumirile lor) care o alcătuiesc, ca în figura 16.5. O variantă de rezolvare a problemei se bazează pe două funcţii numite *f_codpr_liniefact* şi *f_produse_din_factura_v1* – vezi listingul 16.6. Prima primeşte doi parametri, un număr de factură (*nrfact_*) şi un număr de linie (*linie_*) şi returnează codul produsului de pe linia/factura respective. În cazul în care nu există linia cu numărul *linie_* în factura *nrfact_*, funcţia returnează valoarea 0. A doua - *f_produse_din_factura_v1* – construieşte şirul ce conţine denumirile produselor din factura *nrfact_* printr-o buclă executată de maximum 99 ori, aceasta deoarece presupunem că o factură conţine cel mult 99 de linii (ceea ce e destul de improbabil). Cum liniile din toate facturile sunt dispuse consecutiv, în momentul în care se depistează o linie inexistentă, se iese forţat (EXIT) din buclă, aşa că nu trebuie să ne facem griji pentru mersul în gol al iteraţiei.

Listing 16.6. Funcţiile ***f_codpr_liniefact*** şi ***f_produse_din_factura_v1***

```
CREATE OR REPLACE FUNCTION f_codpr_liniefact
        (nrfact_ liniifact.nrfact%TYPE, linie_ liniifact.linie%TYPE)
    RETURNS liniifact.codpr%TYPE AS $$
DECLARE
    v_codpr liniifact.codpr%TYPE := 0 ;
BEGIN
    SELECT CodPr INTO v_codpr FROM liniifact WHERE NrFact=nrfact_ AND Linie=linie_ ;
    RETURN v_codpr ;
END ;
$$ LANGUAGE plpgsql ;

-----------------------------------------------------------------------------------------
CREATE OR REPLACE FUNCTION f_produse_din_factura_v1
    (nrfact_ facturi.nrfact%TYPE) RETURNS TEXT AS $$
DECLARE
    v_sir TEXT := '' ;
BEGIN
    FOR i IN 1..99 LOOP
        IF f_codpr_liniefact (CAST(nrfact_ AS INTEGER), CAST( i AS SMALLINT)) > 0 THEN
            v_sir := v_sir || ' * ' ||
            f_denpr(f_codpr_liniefact (CAST (nrfact_ AS INTEGER), CAST (i AS SMALLINT)) ) ;
        ELSE
            EXIT ;
        END IF ;
    END LOOP ;
    RETURN v_sir ;
END ;
$$ LANGUAGE plpgsql ;
```

Pentru a preîntâmpina erorile legate de nepotrivirile de tip între parametrii specificaţi la apelarea funcţiei *f_codpr_liniefact* şi cei declaraţi în corpul funcţiei respective, folosim funcţia CAST. Extragerea denumirii unui produs pe baza codului său ne prilejuieşte folosirea funcţiei discutate la începuturile paragrafului, *f_denpr*. Raportul din figura 16.5 este obţinut acum prin interogarea:

```
SELECT facturi.NrFact, facturi.DataFact,
        f_produse_din_factura_v1(facturi.NrFact) AS Lista_produse
```

```
FROM facturi ORDER BY 1
```

nrfact integer	datafact date	lista_produse text
1111	2007-08-01	* Produs 1 * Produs 2 * Produs 5
1112	2007-08-01	* Produs 2 * Produs 3
1113	2007-08-01	* Produs 2
1114	2007-08-01	* Produs 2 * Produs 4 * Produs 5
1115	2007-08-02	* Produs 2
1116	2007-08-02	* Produs 2
1117	2007-08-03	* Produs 2 * Produs 1
1118	2007-08-04	* Produs 2 * Produs 1
1119	2007-08-07	* Produs 2 * Produs 3 * Produs 4 * Produs 5
1120	2007-08-07	* Produs 2
1121	2007-08-07	* Produs 5 * Produs 2
1122	2007-08-07	
2111	2007-08-14	* Produs 1 * Produs 2 * Produs 5
2112	2007-08-14	* Produs 2 * Produs 3

Figura 16.5. Produsele din componenţa fiecărei facturi

16.1.2. Primele funcții stocate în Oracle PL/SQL

De ani buni, există posibilitatea redactării procedurilor stocate Oracle în limbajul Java, însă limbajul predilect de redactare a funcțiilor (procedurilor, pachetelor şi declanşatoarelor) rămâne PL/SQL. Nu vom intra în detalii privind „filosofia limbajului”[3], ci doar vom prezenta câteva dintre opțiunile de lucru. Ca şi PL/pgSQL (pe care l-a inspirat copios), PL/SQL-ul pune la dispoziția programatorilor trei secțiuni, declarativă, executabilă şi de tratare a excepțiilor. Să începem cu redactarea funcțiilor *f_proctva* şi *f_denpr* – vezi listing 16.7.

Listing 16.7. Funcţiile PL/SQL ***f_proctva*** şi ***f_denpr***

```
-- F_PROCTVA -- pentru un cod de produs returneaza procentul de TVA al produsului
CREATE OR REPLACE FUNCTION f_proctva2 (codpr_ produse.codpr%TYPE)
    RETURN produse.ProcTVA%TYPE
AS
```

[3] Pentru o tratare mai detaliată a limbajului, vezi şi [Fotache s.a 2003].

```
    v_proc produse.proctva%TYPE ;
BEGIN
     SELECT proctva INTO v_proc FROM produse WHERE codpr = codpr_ ;
    RETURN v_proc ;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN NULL ;
END ;
/
--------------------------------------------------------------------------
-- F_DENPR - primeste un cod de produs si returneaza denumirea
CREATE OR REPLACE FUNCTION f_denpr2 (codpr_ produse.codpr%TYPE)
    RETURN produse.denpr%TYPE
AS
    v_denpr produse.denpr%TYPE ;
BEGIN
    SELECT denpr INTO v_denpr FROM produse WHERE codpr = codpr_ ;
    RETURN v_denpr ;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN NULL ;
END ;
/
```

Câteva deosebiri față de versiunea PL/pgSQL trebuie subliniate. În procedurile şi funcțiile PL/SQL secțiunea declarativă nu începe cu DECLARE, ci urmează clauzei AS sau IS. Interogarea SELECT trebuie să extragă din tabela/tabelele din clauza FROM nici mai mult nici mai puțin de o linie, altminteri ne vom procopsi cu un mesaj de eroare – NO_DATA_FOUND. Orice eroare (excepție) este tratată în secțiunea excepțiilor. Atunci când valoarea parametrului de intrare nu este în tabela PRODUSE, în locul blocării eventualei aplicații, se returnează NULL.

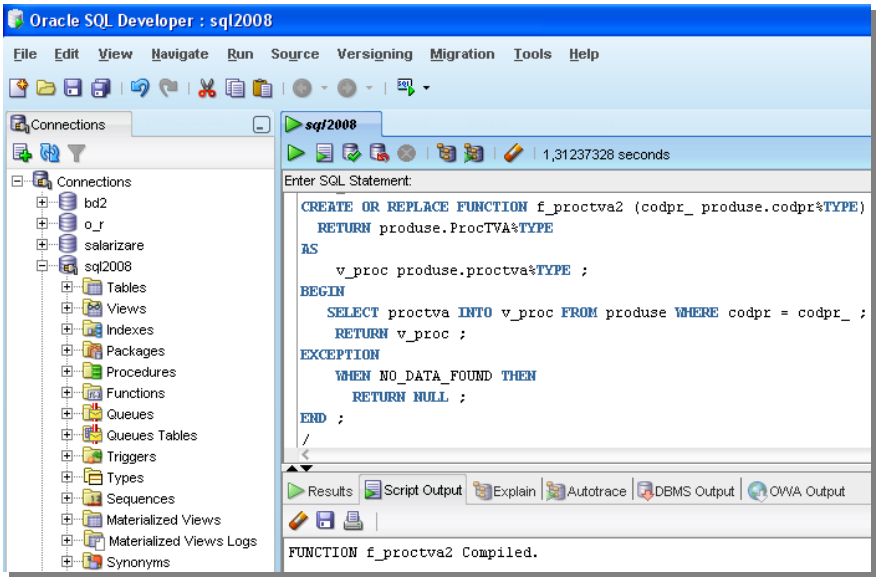


Figura 16.6. Crearea directă a unei funcţii în Oracle SQL Developer (prin comanda CREATE FUNCTION)

Mai trebuie spus că tipurile definite pentru variabile în blocurile PL/SQL sunt mai generoase decât cele discutate în capitolul 3. Astfel, într-o tabelă un atribut nu poate fi declarat de tip BOOLEAN (logic); în schimb, orice variabilă poate fi declarată astfel. La fel stau lucrurile cu tipurile *integer* – BINARY_INTEGER sau PLS_INTEGER -, cu vectorii asociativi etc.

O primă modalitate de creare a funcţiilor în SQL Developer este scriere directă sau copierea dintr-un alt fişier (text), ca în figura 16.6, comanda fiind, fireşte, CREATE OR REPLACE FUNCTION... A doua modalitate este cea asistată – vezi figura 16.7. În meniul arborescent, în cadrul schemei curente ne deplasăm pe nodul *Functions* şi apelăm meniul contextual, care ne va afişa şapte opţiuni (stânga figurii) din care alegem *New Function*. Umează să definim numele funcţiei, parametrul (parametrii) şi tipul datelor returnate, iar apoi sa completăm definiţia funcţiei. În final compilăm funcţia pentru a vedea ce erori s-au strecurat.

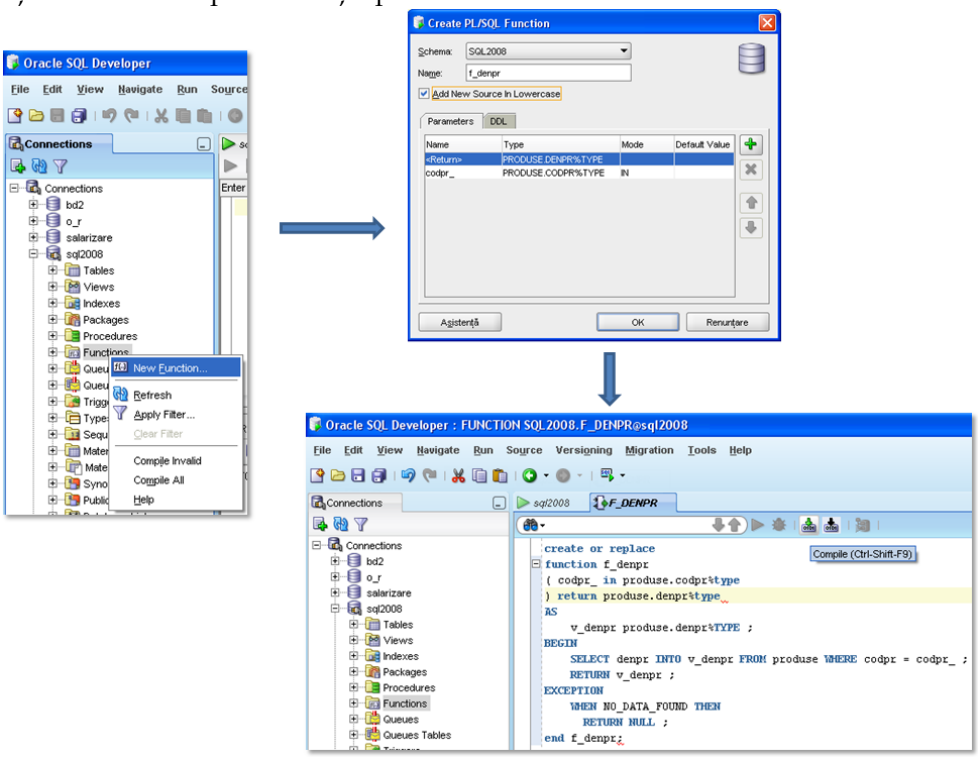


Figura 16.7. Crearea asistată a unei funcţii în Oracle SQL Developer

Urmând exemplele din paragraful anterior, creăm în PL/SQL cele două funcţii care suplinesc lipsa funcţiilor LEFT şi RIGHT (care există în SQL Server), funcţii care se află în listing 16.8.

Listing 16.8. Funcţiile (PL/SQL) ***left_*** şi ***right_***

```
-- Functia LEFT_ extrage primele cite_ caractare dintr-un sir_
CREATE OR REPLACE FUNCTION LEFT_ (sir_ VARCHAR2, cite_ NUMERIC)
    RETURN VARCHAR2
AS
    v_sir_returnat VARCHAR2(5000) ;
BEGIN
     v_sir_returnat := SUBSTR(sir_, 1, cite_) ;
    RETURN v_sir_returnat ;
END LEFT_ ;
/
-- Functia RIGHT_ extrage ultimele cite_ caractare dintr-un sir_
CREATE OR REPLACE FUNCTION RIGHT_ (sir_ VARCHAR2, cite_ NUMERIC)
    RETURN VARCHAR2
AS
    v_sir_returnat VARCHAR2(5000) ;
BEGIN
     v_sir_returnat := SUBSTR(sir_, LENGTH(sir_) - cite_ + 1, cite_) ;
    RETURN v_sir_returnat ;
END ;
/
```

În continuare (listing 16.9) - portăm funcția *f_data_scadenta* (pentru determinarea scadenței facturilor peste un număr de zile lucrătoare) din PL/pgSQL în PL/SQL. După cum se poate observa, deosebirile sunt minore.

Listing 16.9. Funcţia (PL/SQL) ***f_data_scadenta***

```
CREATE OR REPLACE FUNCTION f_data_scadenta (
  datai_ DATE, nrzile_scadenta_ NUMERIC) RETURN DATE
AS
    zi_crt DATE ;
    v_gata BOOLEAN := FALSE ;
    v_nr_zile_lucratoare NUMERIC := 0 ;
BEGIN
    -- scadenta este peste NRZILE_SCADENTA_ (lucratoare)
    zi_crt := datai_ ;
    WHILE NOT v_gata LOOP
        zi_crt := zi_crt + 1 ;
        IF RTRIM(TO_CHAR(zi_crt, 'DAY')) NOT IN ('SATURDAY','SUNDAY') THEN
                v_nr_zile_lucratoare := v_nr_zile_lucratoare + 1 ;
                IF v_nr_zile_lucratoare >= nrzile_scadenta_ THEN
                        EXIT ;
                END IF ;
        END IF ;
    END LOOP ;
    RETURN zi_crt ;
END ;
/
```

Portăm şi cele două funcții PL/pgSQL din listing 16.6, varianta „oraclistă" fiind cea din listingul 16.10. În prima, atunci când nu există linia din factura specificată, funcția va returna NULL.

Listing 16.10. Funcţiile PL/SQL ***f_codpr_liniefact*** şi ***f_produse_din_factura_v1***

```
CREATE OR REPLACE FUNCTION f_codpr_liniefact
        (nrfact_ liniifact.nrfact%TYPE, linie_ liniifact.linie%TYPE)
```

```
    RETURN liniifact.codpr%TYPE AS
    v_codpr liniifact.codpr%TYPE := 0 ;
BEGIN
    SELECT CodPr INTO v_codpr FROM liniifact WHERE NrFact=nrfact_ AND Linie=linie_ ;
    RETURN v_codpr ;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RETURN NULL ;
END f_codpr_liniefact ;
/

-----------------------------------------------------------------------------------------------
CREATE OR REPLACE FUNCTION f_produse_din_factura_v1
    (nrfact_ facturi.nrfact%TYPE) RETURN VARCHAR2
AS
    v_sir VARCHAR2(1000) := '' ;
BEGIN
    FOR i IN 1..99 LOOP
        IF f_codpr_liniefact (nrfact_ , i) > 0 THEN
                v_sir := v_sir || ' * ' || f_denpr(f_codpr_liniefact (nrfact_, i) ) ;
        ELSE
                EXIT ;
        END IF ;
    END LOOP ;
    RETURN v_sir ;
END f_produse_din_factura_v1 ;
```

16.1.3. Primele funcții stocate în SQL Server

Limbajul de programare al SQL Server este Transact-SQL, pe scurt T-SQL. Sintaxa sa diferă serios de cea a PL/pgSQL şi PL/SQL, însă vom putea formula soluții echivalente fără un efort prea mare. Iată in listingul 16.11 comanda de creare a funcției *f_proctva* în schema curentă (dbo). Comanda de creare a funcției este precedată de un IF care şterge funcția din dicționar, în caz că există. IF-ul este util, pentru că T-SQL nu acceptă varianta CREATE OR REPLACE FUNCTION...

Listing 16.11. Funcţia (T-SQL) ***f_proctva***

```
IF OBJECT_ID (N'dbo.f_proctva', N'FN') IS NOT NULL
   DROP FUNCTION dbo.f_proctva;
GO
CREATE FUNCTION f_proctva (@codpr SMALLINT)
    RETURNS NUMERIC(2,2)
WITH EXECUTE AS CALLER
AS
BEGIN
   DECLARE @v_proc NUMERIC(2,2)
     SELECT @v_proc = ProcTVA  FROM produse WHERE CodPr = @codpr
   RETURN(@v_proc)
END;
```

Nu există secțiune declarativă specială, şi nici caracter separator între comenzi. De asemenea, variabilele (ale căror nume este prefixat de @) se declară numai

direct, iar formatul de atribuire a valorii unei variabile pe baza unei interogări este unul particular T-SQL. Funcția *f_denpr* (listing 16.12) este similară.

Listing 16.12. Funcţia (T-SQL) ***f_denpr***

```
IF OBJECT_ID (N'dbo.f_denpr', N'FN') IS NOT NULL
   DROP FUNCTION dbo.f_denpr ;
GO
CREATE FUNCTION f_denpr (@codpr SMALLINT)
     RETURNS VARCHAR(30)
WITH EXECUTE AS CALLER
AS
BEGIN
    DECLARE @v_denpr VARCHAR(30)
      SELECT @v_denpr = DenPr FROM produse WHERE CodPr = @codpr
    RETURN(@v_denpr)
END;
```

Ca şi în precedentele servere, o funcție poate fi creată fie direct, fie interactiv. Varianta directă nu diferă de suratele sale din pgAdmin şi SQL Developer – vezi figura 16.8.

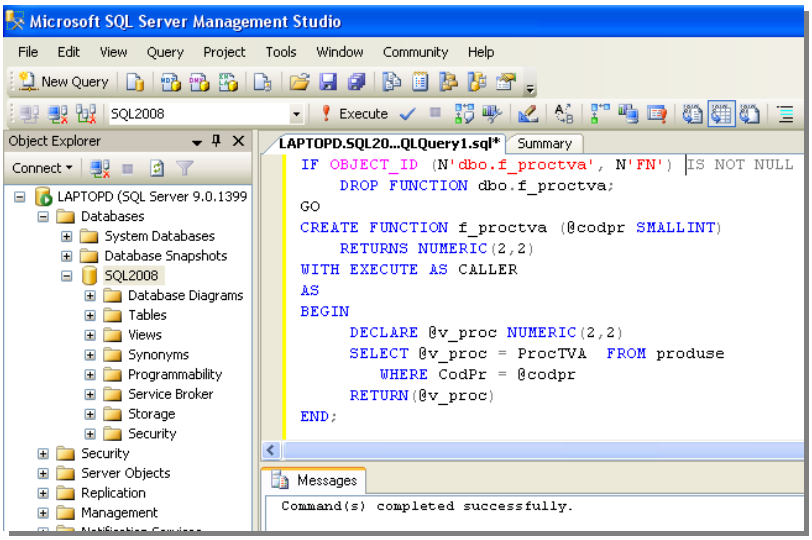


Figura 16.8. Crearea directă a unei funcţii T-SQL

Varianta asistată este mult mai spectaculoasă întrucât *SQL Server Management Studio* este extrem de generos în afişarea unui model (template) în care sunt incluse cele mai frecvente comenzi/instrucțiuni – vezi figura 16.9.

O altă deosebire față de PL/pgSQL şi PL/SQL este că în T-SQL nu există tipul BOOLEAN. Putem folosi un tip la care nu am apelat până acum, şi anume BIT. Tipul BIT permite numai două valori, 0 şi 1, plus NULL. Variabila *@v_gata* din funcția *f_data_scadenta* (listing 16.13) este de acest tip.

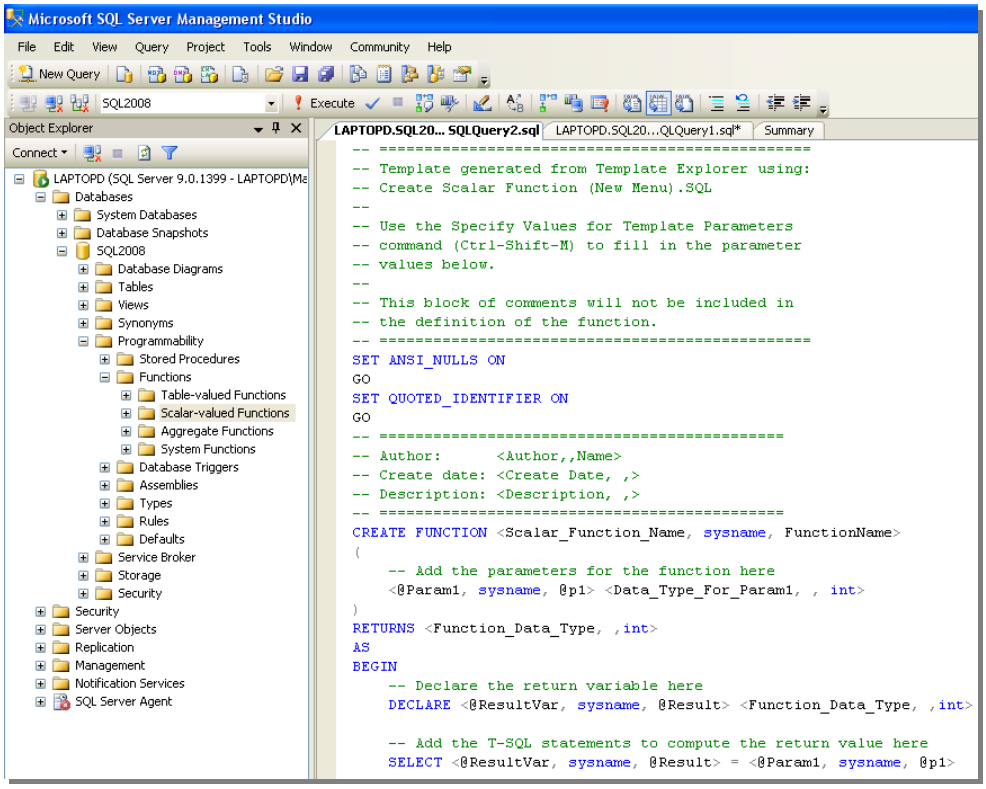


Figura 16.9. Crearea asistată a unei funcţii scalare în T-SQL

Comanda de atribuire este SET, iar pentru un mai bun control al blocurilor executate pe ramurile THEN şi ELSE ale unei structuri IF sau CASE, sau blocurilor din corpul unei bucle (structuri iterative), se folosesc „marcajele" BEGIN şi END. Tot în acest listing observăm că EXIT-ului din PL/(pg)SQL îi corespunde BREAK în T-SQL.

Listing 16.13. Funcţia (T-SQL) ***f_data_scadenta***

```
IF OBJECT_ID (N'dbo.f_data_scadenta', N'FN') IS NOT NULL
   DROP FUNCTION dbo.f_data_scadenta;
GO
CREATE FUNCTION f_data_scadenta (@datai_ SMALLDATETIME, @nrzile_scadenta_ TINYINT)
     RETURNS SMALLDATETIME AS
BEGIN
     DECLARE @zi_crt SMALLDATETIME ;
     DECLARE @v_gata BIT ;
     DECLARE @v_nr_zile_lucratoare TINYINT ;
     SET @v_gata = 0  ;
     SET @v_nr_zile_lucratoare = 0 ;

     -- scadenta este peste NRZILE_SCADENTA_ (lucratoare)
     SET @zi_crt = @datai_ ;
     WHILE @v_gata = 0 BEGIN
          SET @zi_crt = DATEADD(DAY, 1, @zi_crt) ;
```

```
        IF DATEPART(dw,@zi_crt) NOT IN (1,7)
                SET @v_nr_zile_lucratoare = @v_nr_zile_lucratoare + 1 ;
                IF @v_nr_zile_lucratoare >= @nrzile_scadenta_
                        BREAK ;
    END  ;
    RETURN @zi_crt ;
END ;
```

Invocarea funcției într-o interogare nu ridică nicio problemă:

```
SELECT NrFact, DataFact,
dbo.f_data_scadenta (DataFact, 19) AS "Scadenta (19 zile lucratoare)"
FROM facturi
```

În SQL Server există funcțiile LEFT şi RIGHT, aşa că nu mai are rost să creăm funcții echivalente, dar nu există INITCAP, aşa că avem ocupație – vezi listing-ul 16.14. În afara perechii BEGIN-END corespunzătoare blocului principal, mai avem una ce delimitează secvența iterativă (WHILE...) şi o altă pereche ce marchează ramura THEN a testului *IF SUBSTRING...*

Listing 16.14. Funcţia (T-SQL) ***INITCAP_***

```
IF OBJECT_ID (N'dbo.INITCAP_', N'FN') IS NOT NULL
   DROP FUNCTION dbo.INITCAP_;
GO
CREATE FUNCTION dbo.INITCAP_ (@textsursa VARCHAR(100))
    RETURNS VARCHAR (100)
AS
BEGIN
    DECLARE @text_returnat VARCHAR(100) ;
    DECLARE @i_inceput_cuvint TINYINT ;
    DECLARE @i TINYINT ;
    SET @text_returnat = @textsursa ;
    SET @i = 2 ;
    SET @i_inceput_cuvint = 1 ;
    WHILE @i <= LEN(@textsursa)
        BEGIN
        IF SUBSTRING(@textsursa, @i, 1) IN (' ', '-', '.') OR @i=LEN(@textsursa)
                BEGIN
                SET @text_returnat =
                        CASE WHEN @i_inceput_cuvint = 1 THEN '' ELSE
                                SUBSTRING(@text_returnat, 1, @i_inceput_cuvint - 1) END
                                 +
                        UPPER(SUBSTRING(@text_returnat, @i_inceput_cuvint, 1)) +
                        SUBSTRING(@text_returnat, @i_inceput_cuvint + 1, LEN(@textsursa)-
                                @i_inceput_cuvint+1 ) ;
                SET @i_inceput_cuvint = @i + 1 ;
                END ;
        SET @i = @i + 1 ;
        END ;

    RETURN @text_returnat ;
END ;
```

Testarea funcției se poate face atât cu un şir oarecare:

```
SELECT dbo.INITCAP_('ana are mere')
```

cât şi cu un atribut dintr-o tabelă:

```
SELECT c.DenCl, c.Adresa, dbo.INITCAP_(c.Adresa) FROM clienti c
```

Încheiem cu cele două funcţii care furnizează pentru fiecare factură un şir de caractere ce conţine denumirile produselor ce apar în factura respectivă – vezi listing 16.15.

Listing 16.15. Funcţiile (T-SQL) *f_codpr_liniefact* şi *f_produse_din_factura_v1*

```
IF OBJECT_ID (N'dbo.f_codpr_liniefact', N'FN') IS NOT NULL
   DROP FUNCTION dbo.f_codpr_liniefact;
GO
CREATE FUNCTION f_codpr_liniefact (@nrfact INT, @linie TINYINT)
    RETURNS SMALLINT
AS
BEGIN
    DECLARE @v_codpr SMALLINT;
    SET @v_codpr = 0 ;
    SELECT @v_codpr = CodPr FROM liniifact WHERE NrFact=@nrfact AND Linie=@linie ;
    RETURN @v_codpr ;
END  ;
-----------------------------------------------------------------------------------------------------------------
IF OBJECT_ID (N'dbo.f_produse_din_factura_v1', N'FN') IS NOT NULL
   DROP FUNCTION dbo.f_produse_din_factura_v1 ;
GO
CREATE FUNCTION f_produse_din_factura_v1
    (@nrfact INT) RETURNS VARCHAR (1000)
AS
BEGIN
    DECLARE @v_sir VARCHAR(1000) ;
    DECLARE @i TINYINT ;
    SET @v_sir = '' ;
    SET @i = 0 ;
    WHILE @i <= 99 BEGIN
        SET @i = @i + 1
        IF dbo.f_codpr_liniefact (@nrfact , @i) = 0
                BREAK
        ELSE
                SET @v_sir = @v_sir + ' * ' + dbo.f_denpr(dbo.f_codpr_liniefact (@nrfact, @i) )
    END
    RETURN @v_sir
END
```

16.1.4. Primele funcţii stocate în DB2

Limbajul de redactare a procedurilor stocate în DB2 se numeşte SQL PL. Sintaxa sa este elegantă, însă dificultatea noastră principală ţine de incapacitatea *DB2 Control Center* de a asigura un suport adecvat creării procedurilor şi funcţiilor stocate. Documentaţia DB2 recomandă folosirea pachetului software *IBM Data Studio*, aşa că, din acest paragraf, vom apela la serviciile sale.

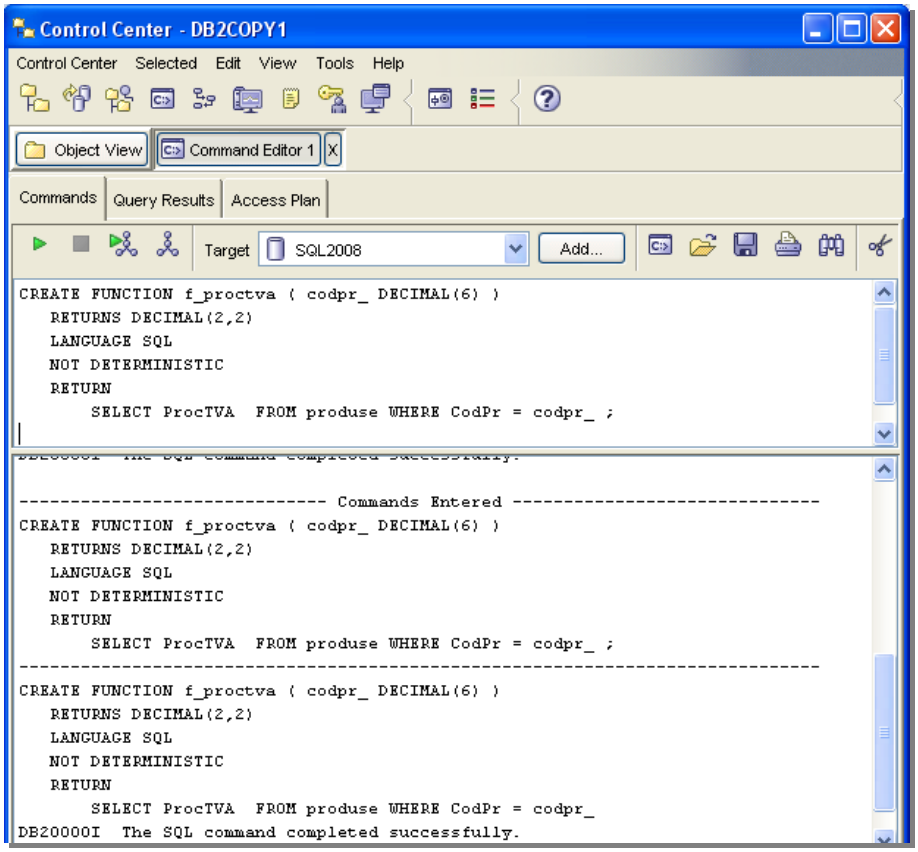


Figura 16.10. Crearea directă a unei funcţii SQL PL în DB2 Control Center

Începem prin a porta în SQL PL primele două funcţii, *f_proctva* şi *f_denpr* (listing 16.16). După specificaţii, singura comandă a funcţiilor este RETURN care are ca argument o interogare.

Listing 16.16. Funcţiile SQL PL ***f_proctva*** şi ***f_denpr***

```
CREATE FUNCTION f_proctva ( codpr_ SMALLINT )
  RETURNS DECIMAL(2,2)
  LANGUAGE SQL
  NOT DETERMINISTIC
  RETURN
    SELECT ProcTVA  FROM produse WHERE CodPr = codpr_ ;

CREATE FUNCTION f_denpr ( codpr_ SMALLINT )
  RETURNS VARCHAR(30)
  LANGUAGE SQL
  NOT DETERMINISTIC
  RETURN
    SELECT DenPr  FROM produse WHERE CodPr = codpr_ ;
```

În DB2 *Control Center* funcțiile trebuie scrise (sau copiate) şi apoi compilate – vezi figura 16.10. Compilarea decurge fără probleme pentru că funcțiile sunt simple. Figura 16.11 ilustrează modul de creare a uneia dintre cele două funcții cu ajutorul *IBM Data Studio*.

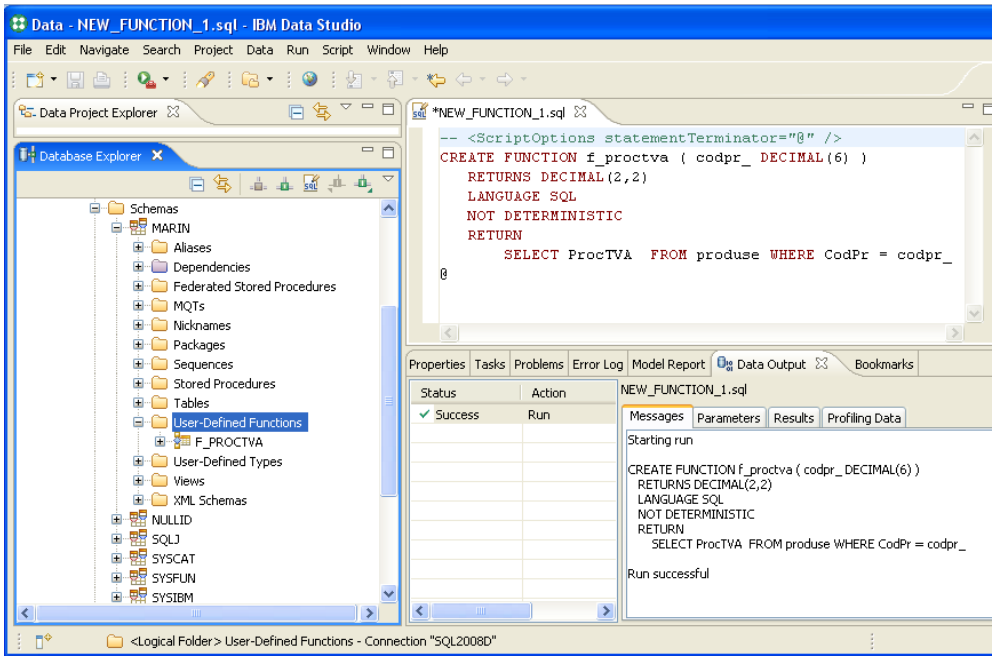


Figura 16.11. Crearea unei funcţii SQL în IBM Data Studio

Nici funcțiile *LEFT_* şi *RIGHT_* (listing 16.17) nu ridică probleme deosebite, şi acestea având o singură comandă – RETURN.

Listing 16.17. Funcţiile SQL PL ***LEFT_*** şi ***RIGHT_***

```
-- Functia LEFT_ extrage primele cite_ caractare dintr-un sir_
CREATE FUNCTION LEFT_ (sir_ VARCHAR(1000), cite_ INTEGER)
    RETURNS VARCHAR(1000)
    LANGUAGE SQL
    NOT DETERMINISTIC
    NO EXTERNAL ACTION
RETURN SUBSTR(sir_, 1, cite_) ;

-- Functia RIGHT_ extrage ultimele cite_ caractare dintr-un sir_
CREATE FUNCTION RIGHT_ (sir_ VARCHAR(1000), cite_ INTEGER)
    RETURNS VARCHAR(1000)
    LANGUAGE SQL
    NOT DETERMINISTIC
    NO EXTERNAL ACTION
RETURN SUBSTR(sir_, LENGTH(sir_) - cite_ + 1, cite_) ;
```

Cu totul altfel stau lucrurile dacă ne apucăm de redactarea funcției INITCAP_ care conține variabile, structuri alternative şi repetitive. Figura 16.12 conține corpul funcției ce respectă întocmai sintaxa SQL PL, dar şi o mică parte din lungul mesaj de eroare afişat.

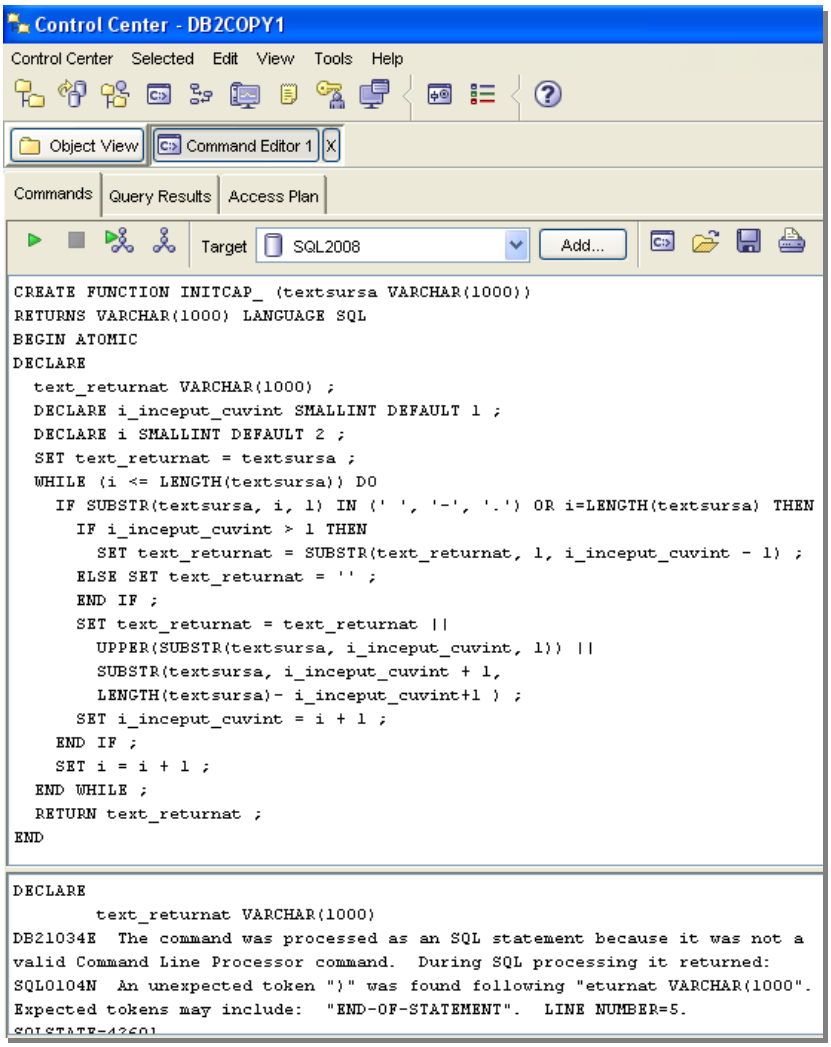


Figura 16.12. Refuzul DB2 Control Center de a recunoaşte funcţia în formatul “structurat”

Înainte de a vă apuca să căutați o eventuală eroare de sintaxă, trebuie să vă spun că aceeaşi funcție, însă dispusă pe câteva rânduri, în care comenzile sunt dispuse pe aceeaşi linie una după alta, ca în listing 16.18 funcționează !

Listing 16.18. Corpul funcţiei SQL PL ***INITCAP_***

```
CREATE FUNCTION INITCAP_ (textsursa VARCHAR(1000))
    RETURNS VARCHAR(1000) LANGUAGE SQL BEGIN ATOMIC
```

```
    DECLARE text_returnat VARCHAR(1000) ; DECLARE i_inceput_cuvint SMALLINT DEFAULT 1 ;
DECLARE i SMALLINT DEFAULT 2 ; SET text_returnat = textsursa ; WHILE (i <= LENGTH(textsursa))
DO IF SUBSTR(textsursa, i, 1) IN (' ', '-', '.') OR i=LENGTH(textsursa) THEN IF i_inceput_cuvint > 1
THEN SET text_returnat = SUBSTR(text_returnat, 1, i_inceput_cuvint - 1) ; ELSE SET text_returnat = '' ;
END IF ; SET text_returnat = text_returnat || UPPER(SUBSTR(textsursa, i_inceput_cuvint, 1)) ||
SUBSTR(textsursa, i_inceput_cuvint + 1, LENGTH(textsursa)- i_inceput_cuvint+1 )
    ; SET i_inceput_cuvint = i + 1 ; END IF ; SET i = i + 1 ; END WHILE ; RETURN text_returnat ; END
```

În fapt, funcţia este dispusă pe doar patru linii – vezi figura 16.13 -, însă listingul conţine mai multe întrucât textul a fost rupt pentru a putea fi vizualizat.

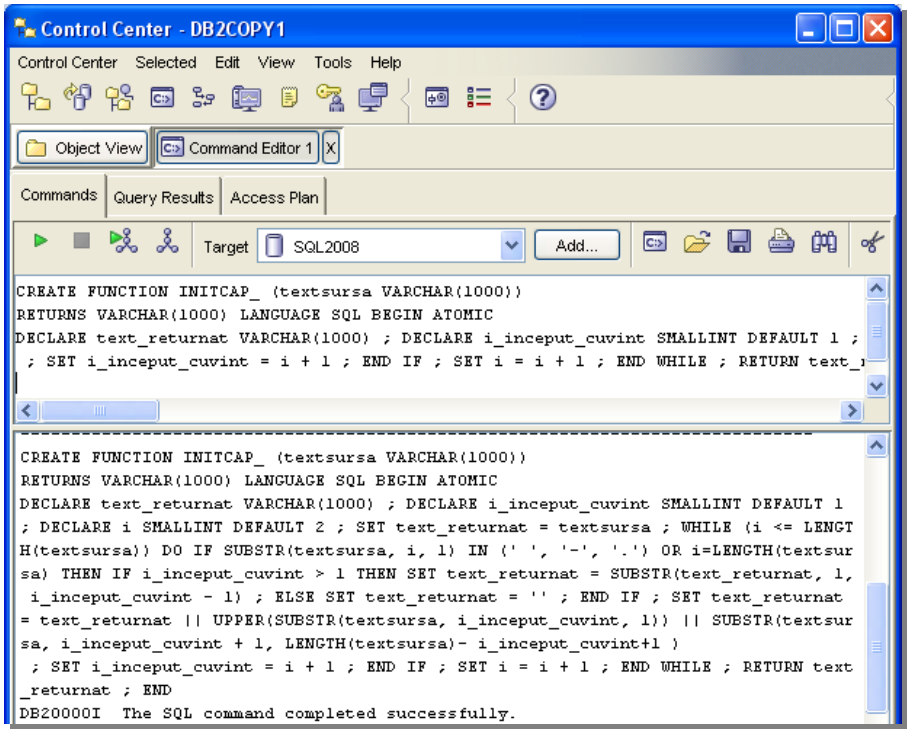


Figura 16.13. Forma funcţiei INITCAP_ acceptată de năzurosul DB2 Control Center

Din acest moment, sătui de fiţele *DB2 Control Center*-ului, vom folosi exclusiv, în materie de funcţii, proceduri şi declanşatoare, *IBM Data Studio*. Cu riscul de a vă obosi, în listing 16.19 includem varianta finală, corectă şi gestionabilă de noua preferinţă IBM.

Listing 16.19. Varianta finală a funcţiei ***INITCAP_*** *editată/compilată în IBM Data Studio*

```
CREATE FUNCTION INITCAP_(TEXTSURSA VARCHAR(1000) )
    RETURNS VARCHAR(1000)
    NO EXTERNAL ACTION
F1: BEGIN ATOMIC
DECLARE TEXT_RETURNAT VARCHAR(1000);
DECLARE I_INCEPUT_CUVINT SMALLINT DEFAULT 1;
DECLARE I SMALLINT DEFAULT 2;
```

```
SET TEXT_RETURNAT = LOWER(TEXTSURSA);
WHILE (I <= LENGTH(TEXTSURSA)) DO
    IF SUBSTR(TEXTSURSA, I, 1) IN (' ', '-', '.') OR I=LENGTH(TEXTSURSA) THEN
        IF I_INCEPUT_CUVINT > 1 THEN
            SET TEXT_RETURNAT = SUBSTR(TEXT_RETURNAT, 1, I_INCEPUT_CUVINT - 1);
        ELSE
            SET TEXT_RETURNAT = '';
        END IF;
        SET TEXT_RETURNAT = TEXT_RETURNAT ||
            UPPER(SUBSTR(TEXTSURSA, I_INCEPUT_CUVINT, 1)) ||
            LOWER(SUBSTR(TEXTSURSA, I_INCEPUT_CUVINT + 1,
                                LENGTH(TEXTSURSA)- I_INCEPUT_CUVINT+1 ));
        SET I_INCEPUT_CUVINT = I + 1;
    END IF;
    SET I = I + 1;
END WHILE;
RETURN TEXT_RETURNAT;
END
```

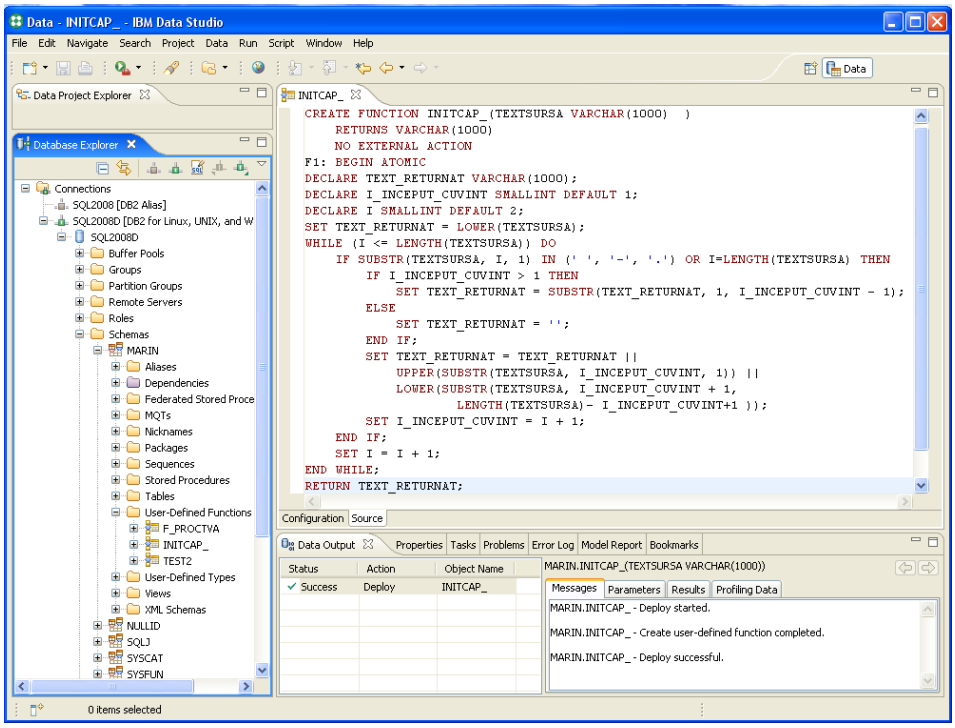


Figura 16.14. Crearea funcţiei INITCAP_ în IBM Data Studio

Faptul că sintaxa este operațională este indicat în figura 16.14. Atenţie, lucrul în Data Studio este mai dificil la început (spre exemplu, pentru a compila şi salva funcția într-o subschemă se foloseşte opțiunea *Deploy*).

Listingul 16.20 conține sintaxa funcției *f_data_scadenta* compilată şi plasată în schema curentă a bazei cu IBM Data Studio. Logica este aceaşi cu versiunile

procedurale din celelalte trei servere, doar că unele comenzi (DECLARE, WHILE) şi funcţii (DAYOFWEEK_ISO) au format uşor diferit.

Listing 16.20. Funcţia ***f_data_scadenta***

```
CREATE FUNCTION F_DATA_SCADENTA (DATAI_ DATE, NRZILE_SCADENTA_ SMALLINT)
    RETURNS DATE
    NO EXTERNAL ACTION
F1: BEGIN ATOMIC
    DECLARE ZI_CRT DATE;
    DECLARE V_GATA SMALLINT DEFAULT 0;
    DECLARE V_NR_ZILE_LUCRATOARE SMALLINT DEFAULT 0;
    SET ZI_CRT = DATAI_;
    WHILE (V_GATA = 0) DO SET ZI_CRT = ZI_CRT + 1 DAY;
        IF DAYOFWEEK_ISO (ZI_CRT) NOT IN (6,7) THEN
                SET V_NR_ZILE_LUCRATOARE = V_NR_ZILE_LUCRATOARE + 1;
                IF V_NR_ZILE_LUCRATOARE >= NRZILE_SCADENTA_ THEN
                        SET V_GATA = 1;
                END IF;
        END IF;
    END WHILE;
    RETURN ZI_CRT;
END
```

La invocarea funcţiei, dorind să aflăm data scadentă a fiecărei facturi în care numărul de zile este 19, DB2-ul mai are o pretenţie: declararea explicită a numărului 19 ca SMALLINT printr-o funcţie CAST:

```
SELECT NrFact, DataFact, marin.f_data_scadenta (facturi.DataFact,
CAST (19 AS SMALLINT)) AS "Scadenta (19 zile lucratoare)"
FROM facturi
```

Cele două funcţii care asigură afişarea denumirii produselor prezente într-o factură sunt conţinute în listing-ul 16.21. Remarcăm formatul de atribuire a rezultatului unei interogări unei variabile.

Listing 16.21. Funcţiile SQL PL f_codpr_liniefact şi f_produse_din_factura_v1

```
CREATE FUNCTION f_codpr_liniefact (nrfact_ INTEGER, linie_ SMALLINT)
    RETURNS SMALLINT
BEGIN ATOMIC
    DECLARE v_codpr SMALLINT DEFAULT 0 ;
    SET v_codpr = (SELECT CodPr FROM liniifact WHERE NrFact=nrfact_ AND Linie=linie_) ;
    RETURN v_codpr ;
END
-- ============================================================
CREATE FUNCTION f_produse_din_factura_v1 (nrfact_ INTEGER)
    RETURNS VARCHAR (1000)
BEGIN ATOMIC
    DECLARE v_sir VARCHAR(1000) DEFAULT '' ;
    DECLARE i SMALLINT DEFAULT 0 ;
    WHILE (i <= 99) DO
        SET i = i + 1 ;
        IF f_codpr_liniefact (nrfact_, i) = 0 THEN
                SET i=100 ;
        ELSE
```

```
                SET v_sir = v_sir || ' * ' || f_denpr( f_codpr_liniefact (nrfact_, i)) ;
        END IF;
    END WHILE ;
    RETURN v_sir;
END
```

16.2. Cursoare

SQL lucrează cu seturi de înregistrări. Noţiunea de înregistare/linie curentă nu există nici în modelul relaţional, nici în SQL. Cu toate acestea, aplicaţiile complexe necesită proceduri şi funcţii în care prelucrarea pe seturi nu este suficientă, oricât de puternice ar fi dialectele SQL în materie de subconsultări, opţiuni OLAP, ierarhii/recursivitate etc. Cursorul reprezintă un instrument ideal pentru prelucrările linie-cu-linie. De asemenea, în aplicaţiile client/server şi web folosind funcţii bazate pe cursoare fluxul datelor în reţea poate fi optimizat sensibil.

```
DECLARE
    -- cursorul se declară printr-o frază SELECT
    CURSOR c_cursor IS
        SELECT .....

    -- se declară variabilă compusă ce va stoca o linie a cursorului
    rec_cursor c_cursor%ROWTYPE ;
...
BEGIN
    ...
    OPEN c_cursor       /* la deschidere se executa fraza SELECT-definiţie
                           şi se rezervă o zonă de memorie pentru liniile extrase */

    FETCH c_cursor INTO rec_cursor      /* se încarcă prima linie din cursor
                                           în variabila rec_cursor */

    WHILE c_cursor%FOUND LOOP           /* se execută bucla atâta timp cât se reuşeşte
                                           încărcarea unei alte linii din cursor */
        ...
        ... -- corpul buclei
        ...

        FETCH c_cursor INTO rec_cursor  /* se încearcă încarcarea următoarei linii din cursor;
                                           dacă nu s-a putut, rezultă că înregistrările cursorului
                                            sunt epuizate, iar c_cursor%FOUND  are
                                           valoarea logică FALSE */
    END LOOP ;
    CLOSE c_cursor              /* de multe ori, un lucru deschis trebuie închis la loc */
    ....
END ;
```

Figura 16.15. Prima variantă de lucru cu un cursor

Un cursor reprezintă un set de înregistrări obţinut printr-o interogare şi stocat undeva în memorie, set ce poate fi parcurs secvenţial, uneori într-o singură direcţie (de la prima înregistrare la ultima), alteori în ambele direcţii, fiecare înregistrare păstrând, în unele cazuri, legătura cu linia tabelei din care provine.

Operaţiunile derulate în lucrul cu un cursor sunt:

- Definirea/declararea;

- Deschiderea;
- Încarcărea uneia sau mai multor înregistrări din cursor;
- Închiderea.

Chiar dacă este particularizată pe sintaxa Oracle PL/SQL, figura 16.15 ilustrează destul de bine prima (şi cea mai complicată) modalitate de folosire a unui cursor. Definirea se face în zona declarativă. Variabila *rec_cursor* este folosită la stocarea înregistrării curente la trecerea prin cursor.

În zona executabilă a blocului mai întâi se deschidere cursorul (OPEN). Deschiderea echivalează cu execuţia interogării de la declararea cursorului şi stocarea setului de linii obţinut prin interogare într-o zonă de memorie. O comandă FETCH încarcă următoarea înregistrare din cursor în variabila *rec_cursor*. Structura repetitivă *WHILE c_cursor%FOUND LOOP ... END LOOP* asigură parcurgerea tuturor înregistrărilor cursorului, iar comanda CLOSE face curăţenie, prin închidere, spaţiul de memorie alocat fiind eliberat.

```
DECLARE
    -- cursorul se declară printr-o frază SELECT
    CURSOR c_cursor IS
        SELECT . . .

    -- nu mai este necesară declararea variabilei compozite REC_CURSOR
...
BEGIN
...
    FOR rec_cursor IN c_cursor  LOOP /* automat se execută şi deschiderea
                                        şi încarcarea înregistrării */
        ...
        ... -- corpul buclei
        ...

        -- citirea următoarei înregistrări se realizează automat
    END LOOP ;
    /* de multe ori, un lucru deschis trebuie închis la loc. NU ŞI DE DATA ACEASTA ! */
    ...
END ;
```

Figura 16.16. A doua variantă de lucru cu un cursor

A doua variantă de lucru, a cărei logică este cea din figura 16.16 este mai simplă întrucât:

- cursorul se declară ad-hoc în zona executabilă;
- variabila care păstrează înregistrarea curentă din cursor nu trebuie definită în zona DECLARE;
- cursorul nu mai trebuie deschis şi închis explicit;
- deplasarea la înregistrarea următoare este automată.

Există şi o treia variantă, derivată din a doua, şi mai simplă decât aceasta, în care declararea cursorului se face direct în structura iterativă de parcurgere a înregistrărilor – vezi figura 16.17.

```
DECLARE
    -- nu mai este necesară declararea variabilei compozite REC_CURSOR
...
BEGIN
...
    FOR rec_cursor IN (SELECT ... )  LOOP  /* cursorul se declară ad-hoc si, simultan,
                                           /* se deschide şi se încarcarcă prima înregistrare */
        ...
        ... -- corpul buclei
        ...

        -- citirea următoarei înregistrări se realizează automat
    END LOOP ;
    ...
END ;
```

Figura 16.17. A treia variantă de lucru cu un cursor

Deşi sunt câteva diferențe de sintaxă, pe care le vom discuta pe scurt în cele ce urmează, logica celor trei variante este implementată în toate cele patru limbaje ale serverelor BD din această lucrare. De asemenea, în paragrafele dedicate procedurilor (16.5.1 - 16.5.4) vom completa discuția referitoare la cursoarele explicite.

16.2.1. Cursoare în PostgreSQL PL/pgSQL

Începem exemplificarea cursoarelor printr-o funcție ce rezolvă o problemă din paragraful 16.1.1 – cea care îşi propunea să obțină raportul din figura 16.5. În locul celor două funcții din listingul 16.7, putem folosi una singură - *f_produse_din_factura_v2* – pe care o prezentăm în listingul 16.22. Ca element de noutate apare şi o variabilă de tip RECORD, *rec_linii*. Această este grozav de versatilă, deoarece poate stoca o linie din orice tabelă, neavând o structură predefinită. În cazul nostru, *rec_linii* va stoca o linie a cursorului ad-hoc, definit prin SELECT-ul din structura *FOR*... Practic, cursorul va conține atâtea înregistrări câte linii sunt în tabela LINIIFACT pentru factura cu numărul *nrfact_*.

Listing 16.22. Funcţia ***f_produse_din_factura_v2***

```
CREATE OR REPLACE FUNCTION f_produse_din_factura_v2 (nrfact_ facturi.nrfact%TYPE)
    RETURNS TEXT
AS $$
DECLARE
    rec_linii RECORD ;
    v_sir TEXT := '' ;
BEGIN
    FOR rec_linii IN (SELECT CodPr FROM liniifact WHERE NrFact = nrfact_ ) LOOP
        v_sir := v_sir || ' * ' || f_denpr(rec_linii.CodPr) ;
    END LOOP ;
    RETURN v_sir ;
END ;
$$ LANGUAGE plpgsql ;
```

Fireşte că apelul funcției nu pune absolut nicio problemă:

```
SELECT facturi.NrFact, facturi.DataFact,
```

```
        f_produse_din_factura_v2(facturi.NrFact) AS Lista_produse
FROM facturi
ORDER BY 1
```

Ne propunem în continuare să creăm funcţiile stocate care să returneze expresiile cheilor primare şi străine, indiferent de numărul atributelor din cheile compuse (în paragraful 15.1.1 am presupus că pot fi maximum patru atribute într-o cheie compusă). Funcţia *f_cheie_primară* din listing 16.23 extrage mai întâi numele restricţiei de tip cheie primară pentru tabela dată. Ştim că o tabelă nu poate avea mai multe chei primare, aşa că nu avem nicio grijă la interogarea tabelei virtuale dicţionar *information_schema.table_constraints*. Cursorul va prelua toate înregistrările extrase din view-ul *information_schema.key_column_usage* pentru restricţia cu numele stocat în variabila *v_nume_restrictie*.

Listing 16.23. Funcţia ***f_cheie_primara***

```
CREATE OR REPLACE FUNCTION f_cheie_primara (tabela_ TEXT)
    RETURNS TEXT AS $$
DECLARE
    v_nume_restrictie TEXT ;
    v_sir VARCHAR(200) := '(' ;
    rec_atribute RECORD ;
BEGIN
    SELECT constraint_name INTO v_nume_restrictie
    FROM information_schema.table_constraints
    WHERE UPPER(table_name)=UPPER(tabela_) AND constraint_type='PRIMARY KEY' ;

    FOR rec_atribute IN (           SELECT column_name, ordinal_position
                                    FROM information_schema.key_column_usage
                                    WHERE constraint_name = v_nume_restrictie
                                    ORDER BY ordinal_position) LOOP
        v_sir := v_sir || CASE WHEN LENGTH(v_sir)=1 THEN '' ELSE ',' END
                || rec_atribute.column_name ;
    END LOOP ;
    RETURN v_sir || ')';
END ;
$$ LANGUAGE plpgsql ;
```

La fiecare înregistrare a cursorului, variabila *v_sir* va fi „incrementată" cu atributul corespunzător liniei curente a cursorului. Testăm funcţia pentru tabela PERSCLIENTI (rezultatul fiind cel din figura 16.18) :

```
SELECT f_cheie_primara ('persclienti')
```

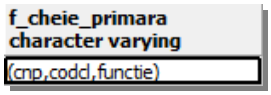


Figura 16.18. Componenţa cheii primare a tabelei PERSCLIENTI

Căutând un pretext pentru folosirea în tandem a două cursoare, ne propunem să construim o funcţie care să afişeze toate expresiile cheilor alternative pentru o tabelă dată. Diferenţa principală faţă de problema anterioară este că o tabelă poate

avea mai multe chei alternative (şi doar o singură cheie primară). Este şi motivul pentru care funcţia (vezi listing 16.24) se numeşte *f_chei_alternative*.

Listing 16.24. Funcţie PL/pgSQL ce foloseşte două cursoare

```
CREATE OR REPLACE FUNCTION f_chei_alternative (tabela_ TEXT)
    RETURNS TEXT AS
$$
DECLARE
    v_sir TEXT := '' ;
    rec_chei_alternative RECORD ;
    rec_atribute RECORD ;
BEGIN
    -- este posibil ca intr-o tabela sa avem mai multe chei alternative, asa ca
    -- folosim un prim cursor
    FOR rec_chei_alternative IN (SELECT constraint_name
                                 FROM information_schema.table_constraints
                                 WHERE UPPER(table_name)=UPPER(tabela_)
                                       AND constraint_type='UNIQUE') LOOP
        v_sir := v_sir || ' UNIQUE (' ;
        -- fiecare cheie alternativa poate fi compusa, asa ca folosim
        --      un al doilea cursor
        FOR rec_atribute IN (       SELECT column_name, ordinal_position
                                    FROM information_schema.key_column_usage
                                    WHERE constraint_name =
                                          rec_chei_alternative.constraint_name
                                    ORDER BY ordinal_position) LOOP
            v_sir := v_sir || CASE WHEN
                     SUBSTR(v_sir, LENGTH(v_sir)-7,9)='UNIQUE ('  THEN '' ELSE ',' END
                     || rec_atribute.column_name ;
        END LOOP ;
    END LOOP ;
    RETURN v_sir || CASE v_sir WHEN '' THEN ''  ELSE ')' END ;
END ;
$$ LANGUAGE plpgsql ;
```

Ar trebui ca explicaţiile inserate în corpul funcţiei să uşureze înţelegerea mecanismului de execuţie. Partea ce mai interesantă este „manevrarea" variabilei *v_sir*. Structura CASE din corpul buclei verifică dacă atributul curent (*rec_atribute.column_name*) este primul din componenţa cheii alternative curente (*rec_chei_alternative.constraint_name*). Dacă nu e primul, trebuie precedat de o virgulă.

La capitolul specificităţi „cursoristice" PL/pgSQL, ar trebui adăugat că sintaxa comenzii FETCH este foarte generoasă. Astfel, putem încărcă direct ultima înregistrare din cursor (prin FETCH LAST) sau reîncărca precedenta înregistrare (prin FETCH RELATIVE -1). Mai mult, există comanda MOVE prin care ne putem deplasa în cursor după bunul plac, fără a încărca noua înregistrare curentă.

16.2.2. Cursoare în Oracle PL/SQL

În Oracle cursoarele se împart în implicite şi explicite. Cele implicite furnizează câteva informaţii despre execuţia comenzilor DML (de exemplu, câte linii au fost şterse în urma execuţii unei comenzi DELETE) şi nu ne interesează în această

lucrare. Cursoarele explicite funcţionează după logica expusă în deschiderea paragrafului 16.2.

Începem cu funcţia corespondentă primeia dintre cele descrise în paragraful anterior (listing 16.25). Tipul returnat de funcţie este VARCHAR2, iar variabila ce stochează înregistrarea curentă din cursor nu trebuie declarată explicit.

Listing 16.25. Funcţia PL/SQL ***f_produse_din_factura_v2***

```
CREATE OR REPLACE FUNCTION f_produse_din_factura_v2 (nrfact_ facturi.nrfact%TYPE)
    RETURN VARCHAR2
AS
    v_sir VARCHAR2(1000) := '' ;
BEGIN
    FOR rec_linii IN (SELECT CodPr FROM liniifact WHERE NrFact = nrfact_ ) LOOP
        v_sir := v_sir || ' * ' || f_denpr(rec_linii.CodPr) ;
    END LOOP ;
    RETURN v_sir ;
END ;
```

Singurele elemente importante ce diferenţiază funcţia *f_cheie_primară* – vezi listing 16.25 - de omoloaga sa din PL/pgSQL (listing 16.23) este denumirea tabelelor virtuale din dicţionarul de date şi atributelor acestora.

Listing 16.25. Funcţia PL/SQL ***f_cheie_primara***

```
CREATE OR REPLACE FUNCTION f_cheie_primara (tabela_ VARCHAR2)
    RETURN VARCHAR2
AS
    v_nume_restrictie VARCHAR2(40) ;
    v_sir VARCHAR(1000) := '(' ;
BEGIN
    SELECT constraint_name INTO v_nume_restrictie
    FROM user_constraints
    WHERE UPPER(table_name)=UPPER(tabela_) AND constraint_type='P' ;

    FOR rec_atribute IN (           SELECT column_name, position
                                    FROM user_cons_columns
                                    WHERE constraint_name = v_nume_restrictie
                                    ORDER BY position) LOOP
        v_sir := v_sir || CASE WHEN LENGTH(v_sir)=1 THEN '' ELSE ',' END
                || rec_atribute.column_name ;
    END LOOP ;
    RETURN v_sir || ')';
END ;
```

Analog stau lucrurile şi pentru funcţia f_chei_alternative din listing-ul 16.27.

Listing 16.27. Funcţia PL/SQL ***f_chei_alternative***

```
CREATE OR REPLACE FUNCTION f_chei_alternative (tabela_ VARCHAR2)
    RETURN VARCHAR2
AS
    v_sir VARCHAR2(1000) := ' ' ;
BEGIN
    -- este posibil ca intr-o tabela sa avem mai multe chei alternative, asa ca
    -- folosim un prim cursor
    FOR rec_chei_alternative IN (SELECT constraint_name
```

```
                FROM user_constraints
                 WHERE UPPER(table_name)=UPPER(tabela_)
                  AND constraint_type='U') LOOP
        v_sir := v_sir || ' UNIQUE (' ;
        -- fiecare cheie alternativa poate fi compusa, asa ca folosim
        --        un al doilea cursor
        FOR rec_atribute IN (        SELECT column_name, position
                                     FROM user_cons_columns
                                     WHERE constraint_name =
                                           rec_chei_alternative.constraint_name
                                     ORDER BY position) LOOP
                v_sir := v_sir || CASE WHEN
                                SUBSTR(v_sir, LENGTH(v_sir)-7,9)='UNIQUE ('  THEN '' ELSE ',' END
                                || rec_atribute.column_name ;
        END LOOP ;
    END LOOP ;
    RETURN v_sir || CASE v_sir WHEN ' ' THEN ''  ELSE ')' END ;
END ;
```

16.2.3. Cursoare în SQL Server Transact-SQL

În Transact-SQL cursorul nu poate fi definit ad-hoc, ci printr-o comandă DECLARE – vezi listing 16.28. Schema de folosire este aceeaşi, singura diferență fiind funcția @@FETCH_STATUS care furnizează informații despre cum a decurs ultima comandă FETCH executată. Astfel, dacă valoarea returnată este 0, atunci încărcarea s-a făcut cu succes, aceasta fiind, de fapt, condiția de repetare a buclei. În Transact-SQL cursorul poate fi parcurs în ambele direcții, dar deocamdată nu avem niciun motiv să apelăm la serviciile acestei opțiuni.

Listing 16.28. Funcţia T-SQL ***f_produse_din_factura_v2***

```
CREATE FUNCTION f_produse_din_factura_v2 (@nrfact INT)
    RETURNS VARCHAR(1000) AS
BEGIN
    DECLARE @v_sir VARCHAR(1000)
    DECLARE @v_codpr SMALLINT
    SET @v_sir = ''
    DECLARE c_linii CURSOR FOR SELECT CodPr FROM liniifact WHERE NrFact = @nrfact
    OPEN c_linii
    FETCH NEXT FROM c_linii INTO @v_codpr
    WHILE @@FETCH_STATUS = 0 BEGIN
        SET @v_sir = @v_sir + ' * ' + dbo.f_denpr(@v_codpr)
        FETCH NEXT FROM c_linii INTO @v_codpr
    END
    CLOSE c_linii
    DEALLOCATE c_linii
    RETURN @v_sir
END
```

Listing-ul 16.29 conține echivalentul T-SQL al funcțiilor PL/pgSQL şi PL/SQL prezentate în paragrafele anterioare prin care se furnizează expresia cheii primare a unei tabele-argument.

Listing 16.29. Funcţia T-SQL ***f_cheie_primara***

```
IF OBJECT_ID (N'dbo.f_cheie_primara', N'FN') IS NOT NULL
   DROP FUNCTION dbo.f_cheie_primara ;
GO
CREATE FUNCTION f_cheie_primara (@tabela VARCHAR(40))
    RETURNS VARCHAR(1000) AS
BEGIN
    DECLARE @v_sir VARCHAR(1000)
    DECLARE @v_nume_restrictie VARCHAR(40)
    SET @v_sir = '(' ;

    SELECT @v_nume_restrictie = constraint_name
    FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS
    WHERE UPPER(table_name)=UPPER(@tabela) AND constraint_type='PRIMARY KEY' ;

    DECLARE @v_atribut VARCHAR(40)
    DECLARE @v_pozitie TINYINT

    DECLARE c_linii CURSOR FOR SELECT column_name, ordinal_position
                                FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
                                WHERE constraint_name = @v_nume_restrictie ORDER BY 2
    OPEN c_linii
    FETCH NEXT FROM c_linii INTO @v_atribut, @v_pozitie
    WHILE @@FETCH_STATUS = 0 BEGIN
        SET @v_sir = @v_sir + CASE WHEN LEN(@v_sir)=1 THEN '' ELSE ',' END +
                @v_atribut ;
        FETCH NEXT FROM c_linii INTO @v_atribut, @v_pozitie
    END
    CLOSE c_linii
    DEALLOCATE c_linii
    RETURN @v_sir + ')'
END
```

Funcţia următoare – *f_chei_alternative* – prezentată în listing 16.30 ilustrează modul în care putem folosi simultan două cursoare, *c_chei_alternative* şi *c_atribute*. Faţă de variantele PL/(pg)SQL, în T-SQL cel de-al doilea cursor este declarat, deschis, parcurs şi închis pentru fiecare înregistrare din primul cursor.

Listing 16.30. Funcţia T-SQL ***f_chei_alternative***

```
IF OBJECT_ID (N'dbo.f_chei_alternative', N'FN') IS NOT NULL
   DROP FUNCTION dbo.f_chei_alternative ;
GO
CREATE FUNCTION f_chei_alternative (@tabela VARCHAR(40))
    RETURNS VARCHAR(1000) AS
BEGIN
    DECLARE @v_sir VARCHAR(1000)
    DECLARE @v_nume_restrictie VARCHAR(100)
    DECLARE @v_atribut VARCHAR(100)
    DECLARE @v_pozitie TINYINT
    SET @v_sir = ' ' ;

    -- primul cursor
    DECLARE c_chei_alternative CURSOR FOR
        SELECT constraint_name
        FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS
        WHERE UPPER(table_name)=UPPER(@tabela) AND constraint_type='UNIQUE'
```

```
    OPEN c_chei_alternative
    FETCH NEXT FROM c_chei_alternative INTO @v_nume_restrictie
    WHILE @@FETCH_STATUS = 0 BEGIN
        SET @v_sir = @v_sir + ' UNIQUE ('

        -- al doilea cursor
        DECLARE c_atribute CURSOR FOR
                        SELECT column_name, ordinal_position
                        FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
                        WHERE constraint_name = @v_nume_restrictie
                        ORDER BY 2
        OPEN c_atribute
        FETCH NEXT FROM c_atribute INTO @v_atribut, @v_pozitie
        WHILE @@FETCH_STATUS = 0 BEGIN
                SET @v_sir = @v_sir + CASE WHEN
                        SUBSTRING(@v_sir, LEN(@v_sir)-7,9)='UNIQUE ('
                                THEN '' ELSE ',' END + @v_atribut
                FETCH NEXT FROM c_atribute INTO @v_atribut, @v_pozitie
        END
        CLOSE c_atribute
        FETCH NEXT FROM c_chei_alternative INTO @v_nume_restrictie
    END

    CLOSE c_chei_alternative
    DEALLOCATE c_atribute
    DEALLOCATE c_chei_alternative

    RETURN @v_sir + CASE @v_sir WHEN ' ' THEN ' '  ELSE ')' END

END
```

16.2.4. Cursoare în DB2 SQL PL

Cea mai simplă variantă a funcției care primeşte un număr de factură şi retunează un şir de caractere ce conține denumirile toturor produselor din factura respectivă este cea din listing 16.31. Definirea cursorului s-a realizat prin secvența *FOR rec1... END FOR*.

Listing 16.31. Funcţia SQL PL ***f_produse_din_factura_v2***

```
CREATE FUNCTION F_PRODUSE_DIN_FACTURA_V2 (nrfact_ INTEGER)
    RETURNS VARCHAR(1000)
    NO EXTERNAL ACTION
F1: BEGIN ATOMIC
    DECLARE v_sir VARCHAR(1000) DEFAULT '';
    FOR rec1 AS SELECT CodPr FROM liniifact WHERE NrFact = nrfact_ DO
        SET v_sir = v_sir || ' * ' ||
            CAST (F_DENPR(CAST (rec1.CodPr AS SMALLINT)) AS VARCHAR(30)) ;
    END FOR;
    RETURN v_sir;
END
```

Continuăm, în tradiția paragrafelor precedente, cu funcția care furnizează expresia cheii primare pentru o tabelă dată (parametru) – vezi listing 16.32. Prima

comandă de atribuire (SET) foloseşte, pe post de expresie, o interogare. Rezultatul interogării va fi atrivuit variabilei *v_nume_restricţie*.

Listing 16.32. Funcţia SQL PL ***f_cheie_primara***

```
CREATE FUNCTION f_cheie_primara(tabela_ VARCHAR(40))
    RETURNS VARCHAR(1000)
    NO EXTERNAL ACTION
F1: BEGIN ATOMIC
    DECLARE v_sir VARCHAR(1000) DEFAULT '(' ;
    DECLARE v_nume_restrictie VARCHAR(40)  ;
    DECLARE v_atribut VARCHAR(40) ;
    DECLARE v_pozitie SMALLINT ;

    SET v_nume_restrictie = (SELECT constraint_name INTO
                              FROM sysibm.table_constraints
                             WHERE table_schema=CURRENT_USER AND
                                    UPPER(table_name) = UPPER(tabela_)
                                    AND  constraint_type=' PRIMARY KEY' ) ;
    FOR rec_linii AS SELECT colname, colseq
                                    FROM SYSCAT.keycoluse
                                    WHERE constname = v_nume_restrictie ORDER BY 2 DO
        SET v_sir = v_sir || CASE WHEN LENGTH(v_sir)=1 THEN '' ELSE ',' END
                || rec_linii.colname ;
    END FOR;
    RETURN v_sir || ')'     ;
END
```

Nici funcţia *f_chei_alternative* (listing 16.33), prin care exemplificăm includerea unei structuri iterative definită pe baza unui cursor în altă structură iterativă corespunzătoare altui cursor, nu aduce noutăţi remarcabile.

Listing 16.33. Funcţia SQL PL ***f_chei_alternative***

```
CREATE FUNCTION f_chei_alternative(tabela_ VARCHAR(40))
    RETURNS VARCHAR(1000)
    NO EXTERNAL ACTION
F1: BEGIN ATOMIC
    DECLARE v_sir VARCHAR(1000) DEFAULT ' ' ;
    DECLARE v_nume_restrictie VARCHAR(40)  ;
    DECLARE v_atribut VARCHAR(40) ;
    DECLARE v_pozitie SMALLINT ;

    FOR rec_chei_altern AS SELECT constraint_name  FROM sysibm.table_constraints
                WHERE table_schema=CURRENT_USER AND
                        UPPER(table_name) = UPPER(tabela_)
                        AND  constraint_type=' UNIQUE' DO
        SET v_sir = v_sir || ' UNIQUE ('  ;
        FOR rec_linii AS SELECT colname, colseq
                        FROM SYSCAT.keycoluse
                        WHERE constname = rec_chei_altern.constraint_name ORDER BY 2
                                DO
                SET v_sir = v_sir || CASE WHEN
                        SUBSTR(v_sir, LENGTH(v_sir)-7,9)='UNIQUE ('
                                THEN '' ELSE ',' END || rec_linii.colname ;
        END FOR;
    END FOR;
    RETURN v_sir || CASE v_sir WHEN ' ' THEN ' '  ELSE ')' END ;
```

```
END
```

16.3. Funcții ce furnizează seturi de înregistrări

Un „debuşeu" clasic al funcțiilor stocate este obținerea de rapoarte dintre cele mai diverse. Specific acestor funcții este că returnează seturi de înregistrări dintr-o tabelă/interogare, set furnizat interfeței sau logiciii aplicației de către serverul BD. Vom vedea că sunt diferențe semnificative, ba chiar, mai mult, presupuşii outsideri – PostgreSQL şi SQL Server – sunt, pe alocuri mai generoşi decât greii DB2 şi Oracle.

16.3.1. Funcții ce returnează seturi de înregistrări în PostgreSQL

Începem cu o funcție simplă – *f_facturi_filtrate_cronologic* - care operează o selecție a liniilor tabelei FACTURI pentru un interval calendaristic – vezi listing 16.34. Eleganța funcției provine din clauza *RETURNS SET OF*... care poate indica structura (atributele) unei tabele, tabele virtuale sau a unui tip-utilizator[4]. Funcția nu este redactată în PL/pgSQL, ci în SQL, având doi parametri de tip DATE care sunt delimitatorii intervalului..

Listing 16.34. Funcţia SQL (din PostgreSQL) ***f_facturi_filtrate_cronologic***

```
CREATE OR REPLACE FUNCTION f_facturi_filtrate_cronologic
        (data_initiala DATE, data_finala DATE)
        RETURNS SETOF facturi AS
$$
    SELECT * FROM facturi WHERE datafact BETWEEN $1 AND $2
$$ LANGUAGE SQL ;
```

[4] Crearea şi utilizarea tipurilor utilizator constituie subiectul capitolului 19.

Odată compilată, funcţia poate fi invocată returnând un rezultat ca orice tabelă sau tabelă virtuală:

```
SELECT *
FROM f_facturi_filtrate_cronologic (DATE'2007-09-14', CURRENT_DATE) t
```

Similar, putem crea o funcţie care returneaza numai facturile unui anumit client (cu un cod specificat) – vezi listing 16.35.

Listing 16.35. Funcţia SQL (din PostgreSQL) ***f_facturile_unui_client***

```
CREATE OR REPLACE FUNCTION f_facturile_unui_client
    (codcl_ clienti.codcl%TYPE) RETURNS SETOF facturi AS
$$
    SELECT * FROM facturi WHERE codcl = $1
$$ LANGUAGE SQL ;
```

Cele două funcţii pot fi folosite în aceeaşi interogare pentru aplicarea a două filtre. Spre exemplu, dorim extragerea tuturor facturilor emise clientului 1001 dupa 5 august 2007:

```
SELECT * FROM f_facturile_unui_client (1001) t
INTERSECT
SELECT *
FROM  f_facturi_filtrate_cronologic (DATE'2007-08-05', CURRENT_DATE) t
```

Funcţia următoare – *f_facturi_filtrate* – redactată tot în SQL, prezintă trei parametri: un cod de client, o dată iniţială şi o dată finală care delimitează un interval calendaristic – vezi listing 16.36. Setul de înregistrări returnate reprezintă facturile destinate clientului dat (parametru) întocmite în intervalul definit de cele două date calendaristice. Dacă la invocarea funcţiei, vreunul dintre parametri este NULL se consideră că nu se mai face filtrarea liniilor după parametrul respectiv. Astfel, când codul clientului este NULL, atunci se vor extrage facturile pentru toţi clienţii. Dacă data iniţială este NULL, atunci se vor extrage facturile de la începuturile bazei de date (să zicem, 1 ian. 2005).

Listing 16.36. Funcţia SQL (din PostgreSQL) ***f_facturi_filtrate***

```
CREATE OR REPLACE FUNCTION f_facturi_filtrate (
codcl_ clienti.codcl%TYPE, data_initiala DATE, data_finala DATE )
    RETURNS SETOF facturi AS
$$
    SELECT * FROM facturi
    WHERE codcl = COALESCE($1, codcl)
                AND datafact BETWEEN COALESCE($2, DATE'2005-01-01')
                AND COALESCE($3, DATE'2010-12-31')
$$ LANGUAGE SQL ;
```

Iată şi câteva modalităţi de folosire a funcţiei pentru a afla informaţii precum:

- facturile clientului 1001 emise între 14 septembrie 2007 şi data curentă:

```
SELECT *
FROM f_facturi_filtrate(1001, DATE'2007-09-14', CURRENT_DATE) t
```

- toate facturile clientului 1001:

```
SELECT *
FROM f_facturi_filtrate(1001, NULL, NULL) t
```

- facturile emise pe 7 august 2007:

```
SELECT *
FROM f_facturi_filtrate(NULL, DATE'2007-08-07', DATE'2007-08-07') t
```

- toate facturile emise:

```
SELECT *
FROM f_facturi_filtrate(NULL, NULL, NULL) t
```

După cum sugeram ceva mai sus, în caz că setul de întregistrări returnat nu are exact structura unei tabele, trebuie creat un tip – *tip_nou* şi folosită clauza RETURNS SET OF *tip_nou*, după cum vom vedea în capitolul 19.

16.3.2. Funcții ce returnează seturi de înregistrări în Oracle PL/SQL

Oracle PL/SQL este cel mai zgârcit dintre cele patru limbaje în materie de obținere a unui set de înregistrări dintr-o funcție-utilizator. Există două soluții principale, plus una ocolitoare. Singura pe care o discutăm acum este cea bazată pe variabile cursor. Cea ocolitoare va fi discutată foarte pe scurt şi în paragrafele 16.6 şi 18.2. Soluția elegantă, care returnează valori-masiv (NESTED TABLE), va fi discutată abia în capitolul 19.

Funcţia *f_facturi_filtrate1* din listing-ul 16.37 returnează o variabilă cursor – *set_of_facturi*. Clauza RETURN indică un tip „cât se poate" de general – SYS_REFCURSOR. Practic, funcția ar putea returna orice set de înregistrări. În zona executabilă variabila cursor este simultan declarată şi populată prin comanda *OPEN... FOR...*

Listing 16.37. Funcţia PL/SQL ***f_facturi_filtrate1***

```
CREATE OR REPLACE FUNCTION f_facturi_filtrate1 (data_initiala DATE, data_finala DATE)
        RETURN SYS_REFCURSOR
IS
    set_of_facturi SYS_REFCURSOR ;
BEGIN
    OPEN set_of_facturi FOR
        SELECT * FROM facturi WHERE datafact BETWEEN data_initiala AND data_finala ;
    RETURN set_of_facturi   ;
END f_facturi_filtrate1 ;
```

Un pic mai dificil este modul de afişare în SQL Developer a rezultatului fucţiei la invocarea acesteia într-o consultare – vezi figura 16.19.

```
SELECT f_facturi_filtrate2(DATE'2007-08-15', CURRENT_DATE)
FROM dual
```

După cum sugerează figura, rezultatul iniţial este dispus pe o singură linie, fiind necesară selectarea liniei respective, activarea meniului contextual, alegerea opţiunii *Single Record View* şi apoi a butonului ... .

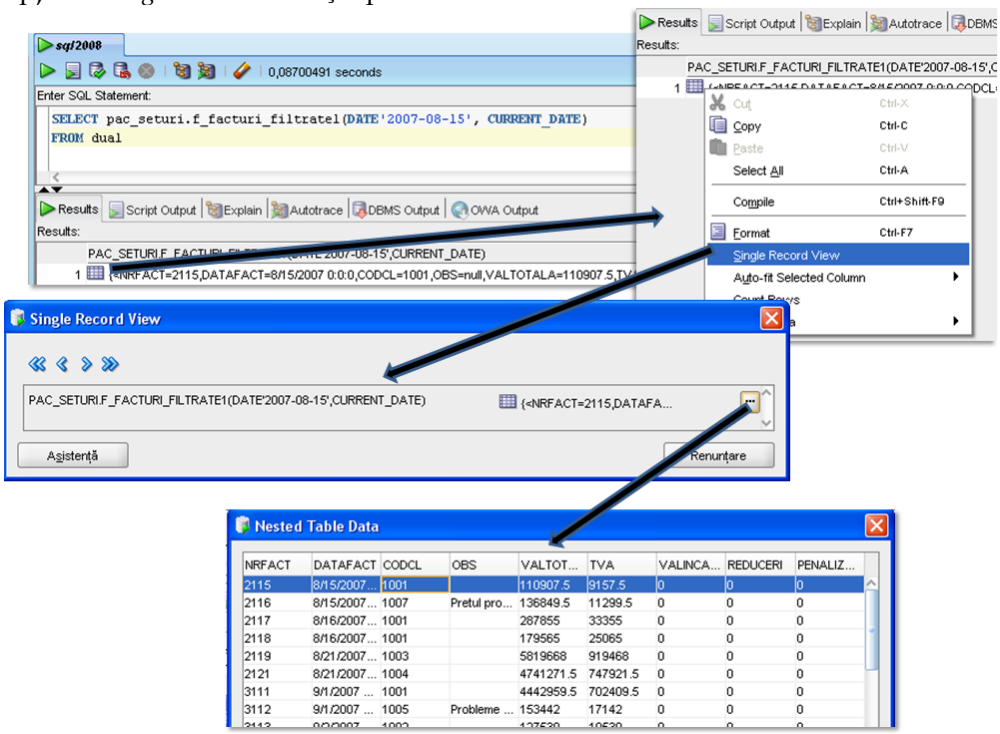
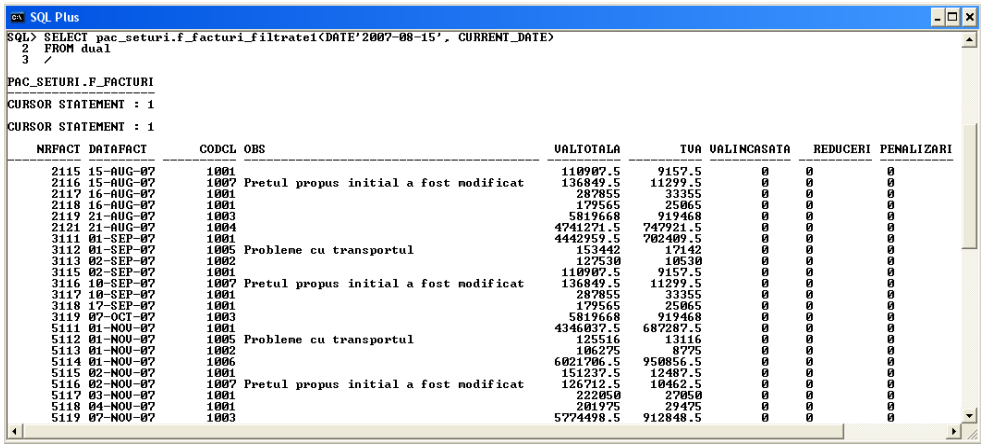


Figura 16.19. Vizualizarea în SQL Developer a setului de înregistrări furnizat de o funcţie ce foloseşte o variabilă cursor

Curios este că, deşi ciudată în SQL Developer, afişarea este corectă „din prima" în clientul clasic şi frugal al Oracle - SQL*Plus – vezi figura 16.20. Trebuie spus, însă, că până la afişarea rezultatului ca în figură, în SQL*Plus trebuie configuraţi o serie întreagă de parametri (mărimea ferestrei, lungimea unei linii, numărul de linii dintr-o pagină etc.), aşa că nu vă lăsaţi înşelaţi de aparenţe.



```
SQL Plus
SQL> SELECT pac_seturi.f_facturi_filtrate1(DATE'2007-08-15', CURRENT_DATE)
  2  FROM dual
  3  /

PAC_SETURI.F_FACTURI
--------------------
CURSOR STATEMENT : 1

CURSOR STATEMENT : 1

    NRFACT DATAFACT       CODCL OBS                                  VALTOTALA        TVA VALINCASATA   REDUCERI PENALIZARI
---------- ---------- ---------- ---------------------------------- ---------- ---------- ---------- ---------- ----------
      2115 15-AUG-07        1001                                    110907.5     9157.5          0          0          0
      2116 15-AUG-07        1007 Pretul propus initial a fost modificat 136849.5   11299.5          0          0          0
      2117 16-AUG-07        1001                                       287855      33355          0          0          0
      2118 16-AUG-07        1001                                       179565      25065          0          0          0
      2119 21-AUG-07        1003                                      5819668     919468          0          0          0
      2121 21-AUG-07        1004                                    4741271.5   747921.5          0          0          0
      3111 01-SEP-07        1001                                    4442959.5   702409.5          0          0          0
      3112 01-SEP-07        1005 Probleme cu transportul               153442      17142          0          0          0
      3113 02-SEP-07        1002                                       127530      10530          0          0          0
      3115 02-SEP-07        1001                                    110907.5     9157.5          0          0          0
      3116 10-SEP-07        1007 Pretul propus initial a fost modificat 136849.5   11299.5          0          0          0
      3117 10-SEP-07        1001                                       287855      33355          0          0          0
      3118 17-SEP-07        1001                                       179565      25065          0          0          0
      3119 07-OCT-07        1003                                      5819668     919468          0          0          0
      5111 01-NOV-07        1001                                    4346037.5   687287.5          0          0          0
      5112 01-NOV-07        1005 Probleme cu transportul               125516      13116          0          0          0
      5113 01-NOV-07        1002                                       106275       8775          0          0          0
      5114 01-NOV-07        1006                                    6021706.5   950856.5          0          0          0
      5115 02-NOV-07        1001                                     151237.5     12487.5          0          0          0
      5116 02-NOV-07        1007 Pretul propus initial a fost modificat 126712.5   10462.5          0          0          0
      5117 03-NOV-07        1001                                       222050       27050          0          0          0
      5118 04-NOV-07        1001                                       201975       29475          0          0          0
      5119 07-NOV-07        1003                                    5774498.5   912848.5          0          0          0
```

Figura 16.20. Vizualizarea în SQL Plus a setului de înregistrări furnizat de o funcţie ce foloseşte o variabilă cursor

16.3.3. Funcții ce returnează seturi de înregistrări în MS SQL Server

Transact-SQL este, probabil, cel mai generos dintre cele patru limbaje discutate în cartea de față în materie de opțiuni pentru funcțiile ce returnează seturi de înregistrări. Clauza *RETURNS TABLE* este cea care simplifică radical lucrurile. Să începem cu funcția *f_filtrate_cronologic* – vezi listing 16.38.

Listing 16.38. Funcţia T-SQL ***f_facturi_filtrate_cronologic***

```
CREATE FUNCTION dbo.f_facturi_filtrate_cronologic
    (@data_initiala SMALLDATETIME, @data_finala SMALLDATETIME)
RETURNS TABLE
AS
RETURN (
     SELECT * FROM facturi WHERE DataFact BETWEEN @data_initiala
                AND @data_finala  )
```

Setul de înregistrări furnizat de funcție la invocare poate avea orice structură (atribute). Pe acelaşi format sunt redactate şi funcțiile *f_facturile_unui_client* (listing 16.39 şi *f_facturi_filtrate* (listing 16.40).

Listing 16.39. Funcţia T-SQL ***f_facturile_unui_client***

```
CREATE FUNCTION dbo.f_facturile_unui_client
    (@codcl_ SMALLINT)
RETURNS TABLE
    AS
RETURN (
     SELECT * FROM facturi WHERE CodCl = @codcl_)
```

Invocarea acestor funcţii este similară celei din PostgreSQL, cu deosebirea că funcţia sistem ce furnizează data curentă a sistemului nu este CURRENT_DATE, ci GETDATE():

```
SELECT * FROM f_facturile_unui_client (1001) t
INTERSECT
SELECT *
FROM  f_facturi_filtrate_cronologic ('2007-08-05', GETDATE()) t
```

Listing 16.40. Funcţia T-SQL ***f_facturi_filtrate***

```
CREATE FUNCTION dbo.f_facturi_filtrate
    (@codcl_ SMALLINT, @data_initiala SMALLDATETIME, @data_finala SMALLDATETIME)
RETURNS TABLE
    AS
RETURN (
     SELECT * FROM facturi WHERE CodCl = COALESCE(@codcl_, CodCl)
                AND DataFact BETWEEN COALESCE(@data_initiala, '2005-01-01')
                          AND COALESCE(@data_finala, '2010-12-31')
       )
```

Pentru a pune în valoare flexibilitatea T-SQL în materie de furnizare prin funcţii-utilizator a unui set de înregistrări redactăm funcţia *f_rute* (listing 16.41) prin care vom afişa traseele posibile dintre două localităţi trecând prin maximum trei localităţi intermediare. De data aceasta, atributele tabelei returnate (numită *@rute*) se declară chiar în clauza RETURNS. N-am vrut să ne întrebuinţăm prea tare, dar am fi putut declara chiar şi restricţii pentru această tabelă. Popularea sa cu înregistrări se face prin comanda *INSERT INTO @rute VALUES...* . Funcţia se temină printr-o comandă RETURN fără argument.

Listing 16.41. Funcţia T-SQL ***f_rute***

```
CREATE FUNCTION dbo.f_rute (@plecare VARCHAR(20), @sosire VARCHAR(20))
RETURNS @rute TABLE
(
-- Atributele tabelei returnate
plecare VARCHAR(20),
sosire VARCHAR(20),
 ruta VARCHAR(500),
 km NUMERIC(5)
)
BEGIN
  DECLARE @Loc1_1 VARCHAR(20)    DECLARE @Loc2_1 VARCHAR(20)
  DECLARE @Loc1_2 VARCHAR(20)    DECLARE @Loc2_2 VARCHAR(20)
  DECLARE @Loc1_3 VARCHAR(20)    DECLARE @Loc2_3 VARCHAR(20)
  DECLARE @Loc1_4 VARCHAR(20)    DECLARE @Loc2_4 VARCHAR(20)

  DECLARE @ruta1 VARCHAR(500)    DECLARE @ruta2 VARCHAR(500)
  DECLARE @ruta3 VARCHAR(500)    DECLARE @ruta4 VARCHAR(500)

  DECLARE @distanta NUMERIC(5)
  DECLARE @km1 NUMERIC(5)        DECLARE @km2 NUMERIC(5)
  DECLARE @km3 NUMERIC(5)        DECLARE @km4 NUMERIC(5)

  DECLARE c_rute1 CURSOR FOR
```

```
SELECT Loc1, Loc2, Distanta FROM distante WHERE Loc1=@plecare UNION
        SELECT Loc2, Loc1, Distanta FROM distante WHERE Loc2=@plecare
 OPEN c_rute1
 FETCH NEXT FROM c_rute1 INTO @Loc1_1, @Loc2_1, @distanta
 WHILE @@FETCH_STATUS = 0 BEGIN
  SET @ruta1 = '*'+ @Loc1_1 + '*' + @Loc2_1 + '*'
  SET @km1 = @distanta
  BEGIN
   IF @Loc2_1 = @sosire
    -- ruta directa
 INSERT INTO @rute VALUES (@Loc1_1, @Loc2_1, @ruta1, @km1)
ELSE
 -- rute printr-o localitate intermediara
BEGIN
 DECLARE c_rute2 CURSOR FOR
    SELECT Loc1, Loc2, Distanta FROM distante
    WHERE Loc1=@Loc2_1 AND @ruta1 NOT LIKE '%*'+Loc2+'*%'
        UNION
    SELECT Loc2, Loc1, Distanta FROM distante
    WHERE Loc2=@Loc2_1 AND @ruta1 NOT LIKE '%*'+Loc1+'*%'
 OPEN c_rute2
 SET @ruta2 = @ruta1
 FETCH NEXT FROM c_rute2 INTO @Loc1_2, @Loc2_2, @distanta
 WHILE @@FETCH_STATUS = 0 BEGIN
  SET @ruta2 = @ruta1 + @Loc2_2 + '*'
  SET @km2 = @km1 + @distanta
  BEGIN
   IF @Loc2_2 = @sosire
    INSERT INTO @rute VALUES (@plecare, @sosire, @ruta2, @km2)
   ELSE
    -- rute prin doua localitati intermediare
    BEGIN
    DECLARE c_rute3 CURSOR FOR
       SELECT Loc1, Loc2, Distanta FROM distante
       WHERE Loc1=@Loc2_2 AND @ruta2 NOT LIKE '%*'+Loc2+'*%'
                   UNION
       SELECT Loc2, Loc1, Distanta FROM distante
       WHERE Loc2=@Loc2_2 AND @ruta2 NOT LIKE '%*'+Loc1+'*%'
    OPEN c_rute3
    SET @ruta3 = @ruta2
    FETCH NEXT FROM c_rute3 INTO @Loc1_3, @Loc2_3, @distanta
    WHILE @@FETCH_STATUS = 0 BEGIN
     SET @ruta3 = @ruta2 + @Loc2_3 + '*'
     SET @km3 = @km2 + @distanta
     BEGIN
      IF @Loc2_3 = @sosire
       INSERT INTO @rute VALUES (@plecare, @sosire, @ruta3, @km3)
      ELSE
       -- rute prin trei localitati intermediare

       BEGIN
        DECLARE c_rute4 CURSOR FOR
         SELECT Loc1, Loc2, Distanta FROM distante
         WHERE Loc1=@Loc2_3 AND @ruta3 NOT LIKE '%*'+Loc2+'*%'
          UNION
         SELECT Loc2, Loc1, Distanta FROM distante
         WHERE Loc2=@Loc2_3 AND @ruta3 NOT LIKE '%*'+Loc1+'*%'
        OPEN c_rute4
        SET @ruta4 = @ruta3
```

```
            FETCH NEXT FROM c_rute4 INTO @Loc1_4, @Loc2_4, @distanta
                WHILE @@FETCH_STATUS = 0 BEGIN
             SET @ruta4 = @ruta3 + @Loc2_4 + '*'
             SET @km4 = @km3 + @distanta

             BEGIN
              IF @Loc2_4 = @sosire

              INSERT INTO @rute VALUES (@plecare, @sosire, @ruta4, @km4)
                  END

                  FETCH NEXT FROM c_rute4 INTO @Loc1_4, @Loc2_4, @distanta
                END
                 CLOSE c_rute4

                 DEALLOCATE c_rute4
                END
          END
          FETCH NEXT FROM c_rute3 INTO @Loc1_3, @Loc2_3, @distanta
        END
        CLOSE c_rute3
        DEALLOCATE c_rute3
      END
    END
   FETCH NEXT FROM c_rute2 INTO @Loc1_2, @Loc2_2, @distanta
     END
     CLOSE c_rute2
     DEALLOCATE c_rute2
    END
   END
   FETCH NEXT FROM c_rute1 INTO @Loc1_1, @Loc2_1, @distanta
  END
  CLOSE c_rute1
  DEALLOCATE c_rute1
  RETURN
END
```

Iată şi un apel al funcției pentru rutele Iaşi-Roman care ne confirmă că funcția se comportă rezonabil (vezi figura 16.21).

```
SELECT *
FROM [SQL2008].[dbo].[f_rute] ( 'Iasi','Roman')
```

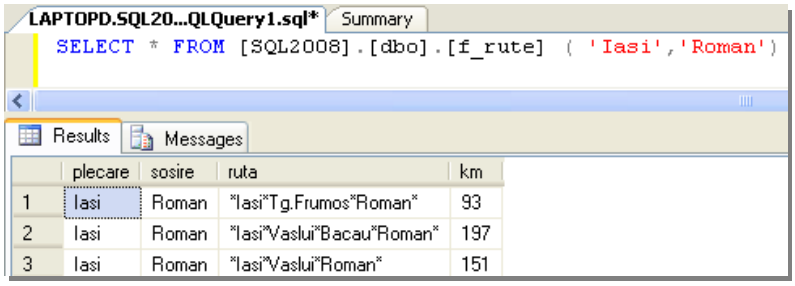


Figura 16.21. Folosirea funcţiei ce returnează o tabelă cu rutele (parţiale) Iaşi-Roman

Acum ne putem întoarce la necazul pricinuit de SQL Server spre finalul paragrafului 12.6 (figura 12.40), când o serie nu putea avea mai mult de 100 de valori returnate de expresia-tabelă. Lucrul acum se simplifică, pentru că, în locul expresiei tabelă, redactăm o funcție ce returnează câte o înregistrare pentru orice valoare dintr-un interval. Funcția se numeşte *f_generare_serii* şi este prezentată în listing 16.42.

Listing 16.42. Funcţia T-SQL ***f_generare_serii***

```
CREATE FUNCTION dbo.f_generare_serii
    (@val_initiala INTEGER, @val_finala INTEGER)
RETURNS @serie TABLE (nr INTEGER )
BEGIN
    DECLARE @i INTEGER
    SET @i = @val_initiala
    WHILE @i <= @val_finala BEGIN
        INSERT INTO @serie VALUES (@i)
        SET @i = @i + 1
    END
    RETURN
END
```

Problema extragerii numerelor de factură nefolosite, rămasă în ceață din paragraful 12.6, presupune execuția interogării următoare care va extrage câte o linie pentru toate numerele cuprinse între valorile minimă şi maximă a numerelor de factură, afişând un mesaj de genul celui din figura 16.22:

nr	NrFact	DataFact	CodCl	Obs	Situatie
1111	1111	2007-08-01 00:00:00	1001	NULL	NULL
1112	1112	2007-08-01 00:00:00	1005	Probleme cu transportul	NULL
1113	1113	2007-08-01 00:00:00	1002	NULL	NULL
1114	1114	2007-08-01 00:00:00	1006	NULL	NULL
1115	1115	2007-08-02 00:00:00	1001	NULL	NULL
1116	1116	2007-08-02 00:00:00	1007	Pretul propus initial a fost modificat	NULL
1117	1117	2007-08-03 00:00:00	1001	NULL	NULL
1118	1118	2007-08-04 00:00:00	1001	NULL	NULL
1119	1119	2007-08-07 00:00:00	1003	NULL	NULL
1120	1120	2007-08-07 00:00:00	1001	NULL	NULL
1121	1121	2007-08-07 00:00:00	1004	NULL	NULL
1122	NULL	NULL	NULL	NULL	Numar nefolosit !
1123	NULL	NULL	NULL	NULL	Numar nefolosit !
1124	NULL	NULL	NULL	NULL	Numar nefolosit !

Figura 16.22. Folosirea funcţiei ce returnează o serie de numere consecutive

```
SELECT nr, NrFact, DataFact, CodCl, Obs,
        CASE WHEN f.NrFact IS NULL THEN 'Numar nefolosit !' ELSE NULL END
                AS Situatie
FROM
        (SELECT nr
```

```
    FROM dbo.f_generare_serii(1, 9999)
    WHERE nr BETWEEN (SELECT MIN(NrFact) FROM facturi)
            AND (SELECT MAX(NrFact) FROM facturi)
) serie LEFT OUTER JOIN facturi f ON nr=NrFact
```

Singura nemulţumire ar putea fi legată de viteza nu prea mare de execuţie a interogării.

16.3.4. Funcţii ce returnează seturi de înregistrări în SQL PL

N-o să intrăm în detalii despre redactarea funcţiilor SQL PL care furnizează seturi de înregistrări. Ne mulţumim cu listingul 16.43 în care este „portată" funcţia *f_facturi_filtrate*. Spre deosebire T-SQL, clauza RETURNS TABLE trebuie să conţină numele atributelor tabelei (setului) returnată.

Listing 16.43. Funcţia SQL PL ***f_facturi_filtrate***

```
CREATE FUNCTION f_facturi_filtrate(codcl_ SMALLINT, data_initiala DATE,
        data_finala DATE  )
    RETURNS TABLE(
        NrFact INTEGER, DataFact DATE, CodCl SMALLINT, Obs VARCHAR(50),
            ValTotala DECIMAL (10,2), TVA DECIMAL(10,2)
                )
    LANGUAGE SQL
    NO EXTERNAL ACTION
F1: BEGIN ATOMIC
    RETURN    SELECT NrFact, DataFact, CodCl, Obs, ValTotala, TVA
              FROM  facturi
              WHERE CodCl = COALESCE(codcl_, CodCl)
                    AND DataFact BETWEEN COALESCE(data_initiala, '2005-01-01')
                    AND COALESCE(data_finala, '2010-12-31') ;
END
```

Apelul funcţiei în interogări necesită funcţiile TABLE şi CAST. În situaţia în care se doreşte setul facturilor clientului 1001, emise între 7 august 2007 şi data curentă, interogarea va avea sintaxa:

```
SELECT *
FROM TABLE (f_facturi_filtrate( CAST (1001 AS SMALLINT),
                CAST ('2007-08-07' AS DATE), CURRENT_DATE))
```

16.4. Funcţii folosite ca valori implicite şi restricţii la nivel de atribut/înregistrare

O aplicabilitate ceva mai exotică a funcţiilor stocate ţine de asocierea lor unor valori implicite (clauze DEFAULT) şi reguli de validare (clauze CHECK). Numai PostgreSQL şi SQL Server permit lucrul acesta, aşa că doar la acestea ne vom referi în cele de urmează.

Să începem printr-o funcție PL/pgSQL ce furnizează cel mai frecvent cod de client din facturi – vezi listing 16.44. Din capitolele anterioare ştim că putem grupa facturile după valorile CodCl şi extrage cea mai mare valoare a numărătorii prin clauza *ORDER BY COUNT(*) DESC LIMIT 1*.

Listing 16.44. Funcţia PL/pgSQL ce urmează a furniza valoarea implicită a FACTURI.CodCl

```
CREATE OR REPLACE FUNCTION f_vi_codcl() RETURNS clienti.codcl%TYPE AS
$$
DECLARE
    v_codcl clienti.codcl%TYPE ;
BEGIN
    SELECT codcl INTO v_codcl FROM facturi GROUP BY codcl
    ORDER BY COUNT(*) DESC LIMIT 1;

    -- daca nu e nicio factura introdusa, functia returneaza cel mai mic cod de client
    IF v_codcl IS NULL THEN
        SELECT MIN(CodCl) INTO v_codcl FROM clienti ;
    END IF ;
    RETURN v_codcl ;
END ;
$$ LANGUAGE plpgsql ;
```

Asocierea acestei funcţii cu valoarea implicită a atributului FACTURI.CodCl se realizează prin comanda ALTER TABLE:

```
ALTER TABLE facturi ALTER COLUMN CodCl SET DEFAULT f_vi_codcl() ;
```

Testăm valabilitatea funcției inserând o factură pentru care nu specificăm codul clientului:

```
INSERT INTO facturi (NrFact, DataFact) VALUES (7050, CURRENT_DATE)
```

La afişarea acestei linii proaspăt inserate – vezi figura 16.23 – observăm că lucrurile au decurs conform aşteptărilor, clientul căruia i-am trimis cele mai multe facturi având codul 1001.

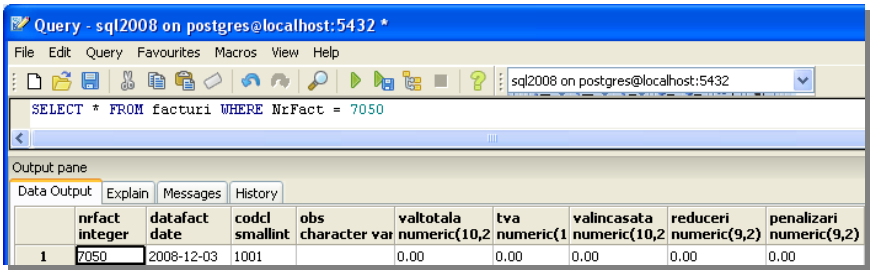


Figura 16.23. Verificarea funcţiei-valoare implicită

Sintaxa Transact-SQL aceleaşi funcții este prezentată în listingul 16.45. Interogarea ce furnizează prima valoare (cea mai frecventă) a atributului *CodCl_*

în tabela FACTURI este inclusă într-o comandă SET. La fel şi cea care extrage cel mai mic cod de client.

Listing 16.45. Funcţia T-SQL ce urmează a furniza valoarea implicită a FACTURI.CodCl

```
CREATE FUNCTION dbo.f_vi_codcl ()
    RETURNS SMALLINT
BEGIN
    DECLARE   @codcl SMALLINT
    SET @codcl = 0

    SET @codcl = (        SELECT TOP 1 CodCl FROM facturi
                          GROUP BY CodCl ORDER BY COUNT(*) DESC
                          )
    -- daca nu e nicio factura introdusa, functia returneaza cel mai mic cod de client
    IF @codcl = 0
        SELECT @codcl = (SELECT TOP 1 CodCl FROM clienti)

    RETURN @codcl
END
```

Şi sintaxa comenzii ALTER TABLE este un pic diferită în T-SQL:

```
ALTER TABLE facturi ADD CONSTRAINT def_facturi_codcl
        DEFAULT dbo.f_vi_codcl () FOR CodCl
```

În schimb testarea funcției se face ca în PostgreSQL, doar că INSERT-ul nu poate folosi funcția CURRENT_DATE, ci GETDATE():

```
INSERT INTO facturi (NrFact, DataFact) VALUES (7050, getdate())
```

Continuăm cu o funcție pe care vrem să o asociem unei clauze CHECK – regulă de validare la nivel de linie. Probabil că în capitolele 3 şi 13 ați fost dezamăgiți de sărăcia clauzei CHECK a comenzilor CREATE/ALTER TABLE. În aplicațiile economice, deseori trebuie gestionate restricții mult mai complexe decât simplele expresii permise de clauza CHECK. Chiar dacă soluția generală de implementare a (aproape) tuturor restricțiilor o reprezintă declanşatoarele (triggerele) discutate în capitolul următor, PostgreSQL şi SQL Server acceptă ca în clauzele CKECK, în locul unui predicat „clasic" să fie specificat numele unei funcții.

Să luăm cazul unei funcții ce primeşte, ca parametru de intrare, valoarea codului unui client (*CodCl_*) şi furnizează TRUE sau FALSE dacă restul de plată al acelui client, adică diferența dintre vânzări şi încasări, este mai mic sau mai mare decât 175000000 RON. Funcția se numeşte *f_ck_facturi_codcl* şi este prezentată în listing 16.46.

Listing 16.46. Funcţia PL/pgSQL ce urmează a fi clauză CHECK în tabela FACTURI

```
CREATE OR REPLACE FUNCTION f_ck_facturi_codcl (codcl_ facturi.CodCl%TYPE)
    RETURNS BOOLEAN AS
$$
BEGIN
    RETURN CASE WHEN
```

```
    COALESCE((
                SELECT SUM(Cantitate * PretUnit * (1 + f_procTVA (lf.CodPr)))
                FROM facturi f INNER JOIN liniifact lf ON f.NrFact=lf.NrFact
                WHERE CodCl=codcl_),0) -
                        COALESCE((
                                SELECT SUM(Transa)
                                FROM facturi f INNER JOIN incasfact i
                                        ON f.NrFact=i.NrFact
                                WHERE CodCl=codcl_
                                ),0) < 175000000 THEN TRUE ELSE FALSE
        END ;
END ;
$$ LANGUAGE plpgsql ;
```

Declararea sa ca regulă de validare pentru tabela FACTURI se realizează, firesc, printr-o comandă ALTER TABLE:

```
ALTER TABLE facturi ADD CONSTRAINT ck_facturi_codcl
CHECK (f_ck_facturi_codcl (codcl)) ;
```

Nu vă rămâne decât să testați funcționalitatea noii reguli, introducând o factură cu o valoare foarte mare. Versiunea Transact-SQL a funcției *f_ck_facturi_codcl* reprezintă subiectul listing-ului 16.47.

Listing 16.47. Funcţia T-SQL ce urmează a fi clauză CHECK în tabela FACTURI

```
CREATE FUNCTION dbo.f_ck_facturi_codcl (@codcl SMALLINT)
    RETURNS BIT
BEGIN
    RETURN CASE WHEN
    COALESCE((
                SELECT SUM(Cantitate * PretUnit * (1 + dbo.f_proctva (lf.CodPr)))
                FROM facturi f INNER JOIN liniifact lf ON f.NrFact=lf.NrFact
                WHERE CodCl = @codcl),0) -
                        COALESCE((
                                SELECT SUM(Transa)
                                FROM facturi f INNER JOIN incasfact i
                                        ON f.NrFact=i.NrFact
                                WHERE CodCl = @codcl
                                ),0) < 175000000 THEN 0 ELSE 1
        END
END
```

Comanda ALTER TABLE pentru asocierea acestei funcții unei clauze CHECK a tabelei FACTURI are sintaxa SQL Server:

```
ALTER TABLE facturi WITH CHECK ADD CONSTRAINT ck_facturi_codcl
        CHECK (dbo.f_ck_facturi_codcl (CodCl) = 0) ;
```

16.5. Proceduri stocate

Mulţi dintre noi ştiu din tinereţe (măcar) că, dacă o funcţie returnează „ceva”, o procedură face „ceva”. Definiţia nu este dintre cele mai pedagogice, dar are calităţile ei. Faţă de funcţii, discuţia legată de proceduri este mai sumară. Dintre cele patru servere BD, PostgreSQL-ul nu are proceduri propriu-zise, ci doar funcţii care returnează VOID.

16.5.1. Proceduri stocate în PostgreSQL

Începem cu o funcţie-procedură care adaugă în observaţiile (Obs) fiecărei facturi mesajul *Nu are linii !* dacă numărul de factură respectiv nu se regăseşte în tabela LINIFACT – vezi listing 16.48.

Listing 16.48. Funcţia-procedură SQL ce „actualizează” valorile atributului Obs

```
CREATE OR REPLACE FUNCTION f_update_facturi_fara_linii()
    RETURNS VOID AS $$
        UPDATE facturi SET obs = COALESCE(obs,' ') || 'Nu are linii !'
        WHERE nrfact NOT IN (SELECT DISTINCT nrfact FROM liniifact)
$$ LANGUAGE SQL ;
```

Apelarea procedurii înseamă, de fapt, apelarea funcţiei, în maniera din paragraful 16.1.1:

```
SELECT f_update_facturi_fara_linii()
```

Se poate reproşa acestei funcţii-proceduri că, la executări repetate, umple observaţiile cu mesaje „Nu are linii !”, aşa că ne putem gândi la o varianta mai acceptabilă – vezi listing 16.49.

Listing 16.49. O variantă mai răsărită a funcţiei-procedură

```
CREATE OR REPLACE FUNCTION f_update_facturi_fara_linii()
    RETURNS VOID AS $$
        UPDATE facturi SET obs = COALESCE(obs,' ') || 'Nu are linii !'
        WHERE nrfact NOT IN (SELECT DISTINCT nrfact FROM liniifact)
                AND COALESCE(obs,' ') NOT LIKE '%Nu are linii !%'
$$ LANGUAGE SQL ;
```

Continuăm cu o funcţie care se substituie clauzei ON DELETE CASCADE din comenzi CREATE/ALTER TABLE. La drept vorbind, aveam nevoie de o asemenea procedură doar în DB2 şi Oracle, însă o vom redacta şi în celelalte două servere BD. Listing-ul 16.50 conţine funcţia *f_on_delete_cascade_factura* dedicată ştergerii unei facturi şi tuturor înregistrărilor copil ale acesteia:

Listing 16.50. O funcţie-procedură de tip ON DELETE CASCADE

```
CREATE OR REPLACE FUNCTION f_on_delete_cascade_factura (nrfact_ facturi.nrfact%TYPE)
        RETURNS VOID AS $$
    DELETE FROM incasfact WHERE nrfact = $1 ;
    DELETE FROM liniifact WHERE nrfact = $1 ;
```

```
    DELETE FROM facturi WHERE nrfact = $1 ;
$$ LANGUAGE SQL ;
```

Continuăm cu o procedură dedicată admiterii la master, admitere discutată în paragraful 13.3.2. Mai întâi creăm cele două tabele, MASTERE şi CANDIDAŢI – vezi listing 16.51.

Listing 16.51. Crearea (PostgreSQL) celor două tabele pentru admiterea la master

```
CREATE TABLE mastere (
    Spec VARCHAR(4) PRIMARY KEY,
    DenSpec VARCHAR(40) NOT NULL,
    NrLocuri NUMERIC(3),
    NrLocuri_Ocupate NUMERIC(3)
     ) ;

CREATE TABLE candidati (
    IdCandidat NUMERIC(6) NOT NULL PRIMARY KEY,
    NumePren VARCHAR(50) NOT NULL,
    Media_AS NUMERIC(4,2) NOT NULL CONSTRAINT ck_media_as
        CHECK (media_as BETWEEN 5 AND 10),
    Media_Lic NUMERIC(4,2) NOT NULL CONSTRAINT ck_media_lic
        CHECK (media_lic BETWEEN 6 AND 10),
    Spec1 VARCHAR(4) NOT NULL REFERENCES mastere(Spec),
    Spec2 VARCHAR(4) REFERENCES mastere(Spec),
    Spec3 VARCHAR(4) REFERENCES mastere(Spec),
    Spec4 VARCHAR(4) REFERENCES mastere(Spec),
    Spec5 VARCHAR(4) REFERENCES mastere(Spec),
    Spec6 VARCHAR(4) REFERENCES mastere(Spec),
    Spec_Repartizat VARCHAR(4) REFERENCES mastere(Spec),
    CONSTRAINT ck_spec CHECK (
        COALESCE(spec2, ' ') = COALESCE(spec2, spec3, ' ') AND
        COALESCE(spec3, ' ') = COALESCE(spec3, spec4, ' ') AND
        COALESCE(spec4, ' ') = COALESCE(spec4, spec5, ' ') AND
        COALESCE(spec5, ' ') = COALESCE(spec5, spec6, ' ')       )
                ) ;
```

Urmează popularea celor două tabele – vezi listing 16.52.

Listing 16.52. Popularea (PostgreSQL) tabelelor MASTERE şi CANDIDAŢI

```
DELETE FROM candidati ;
DELETE FROM mastere ;

INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('SIA',
    'Sisteme informationale pentru afaceri', 4) ;
INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('FAB', 'Finante-Asigurari-Banci', 6) ;
INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('CEA',
    'Contabilitate, expertiza si audit', 6) ;
INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('MARK', 'Marketing', 5) ;
INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('EAI',
    'Economie si afaceri internationale', 5) ;
INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('MRU',
    'Managementul resurselor umane', 4) ;

INSERT INTO candidati VALUES ( 1, 'Popescu I. Irina', 7.5, 9.50, 'FAB', 'MARK', 'EAI', 'MRU',
    NULL, NULL, NULL);
```

```
INSERT INTO candidati VALUES ( 2, 'Babias D. Ecaterina', 8.5, 9.20, 'SIA', 'EAI', NULL,  NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 3, 'Strat P. Iulian', 7.35, 9.50, 'CEA', 'EAI', 'MRU', 'SIA',
    'FAB', 'MARK', NULL);
INSERT INTO candidati VALUES ( 4, 'Georgescu M. Honda', 8.5, 9.00, 'MRU', 'MARK', 'EAI',
    'FAB', 'CEA', 'SIA', NULL);
INSERT INTO candidati VALUES ( 5, 'Munteanu A. Optimista', 9.5, 9.50, 'SIA', 'CEA', 'FAB',
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 6, 'Dumitriu F. Dura', 9.5, 10, 'EAI', NULL, NULL,  NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 7, 'Mesnita G. Plinutul', 10, 10,  'EAI', 'MARK', 'MRU',  NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 8, 'Greavu V. Doru', 9.25, 8.50,  'SIA', 'CEA', 'FAB',  NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 9, 'Baboi P. Iustina', 8.5, 8.50,  'SIA', NULL, NULL,  NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (10, 'Postelnicu I. Irina', 9.5, 8.25,  'MARK', NULL, NULL,
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (11, 'Fotache H. Fanel', 9.75, 9.50,  'MRU', NULL, NULL,
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (12, 'Moscovici J. Cristina', 9.80, 9.30,  'CEA', 'SIA', 'FAB',
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (13, 'Rusu I. Vanda', 7.80, 9.10,  'FAB', 'CEA', 'MARK',  'MRU',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (14, 'Spinu M. Algebra', 7.25, 9.00,  'SIA', 'CEA', 'FAB',  NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (15, 'Sandovici I. Irina', 7.05, 7.50,  'MARK', 'MRU', 'EAI',  'CEA',
     NULL, NULL, NULL);
INSERT INTO candidati VALUES (16, 'Plai V. Picior', 7.5, 7.90,  'EAI', 'SIA', 'CEA', 'FAB',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (17, 'Ambuscada B. Cristina', 8.25, 9.50,  'SIA', 'CEA',
    'FAB', 'EAI',  NULL, NULL, NULL);
INSERT INTO candidati VALUES (18, 'Pinda A. Axinia', 8.75, 9.00,  'SIA', 'FAB', 'CEA',  NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (19, 'Planton V. Grigore', 9.25, 9.50,  'FAB', NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (20, 'Sergentu I. Zece',7.5, 9.50,  'FAB', 'CEA', NULL,  NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (21, 'Ababei T. Marian-Vasile', 7.5, 9.50,  'CEA', 'MRU', NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (22, 'Popistasu J. Maria', 7.5, 9.50,  'FAB', 'EAI', 'CEA', 'MRU',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (23, 'Plop R. Robert', 7.5, 9.50, 'FAB', 'MARK', 'MRU', NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (24, 'Aflorei H. Crina',7.5, 9.50,  'EAI', 'SIA', NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (25, 'Afaunei P. Gina',7.5, 9.50,  'SIA', NULL, NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (26, 'Vatamanu I. Alexandrina', 7.5, 9.50, 'MRU', 'MARK', 'EAI',
     NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (27, 'Lovin P. Marian',7.5, 9.50,  'MARK', 'MRU', NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (28, 'Antiteza W. Florin', 7.5, 9.50, 'EAI', 'MARK', 'MRU',
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (29, 'Prepelita V. Ion', 7.5, 9.50, 'EAI', NULL, NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (30, 'Cioara X. Sanda', 7.5, 9.50, 'CEA', 'FAB', 'EAI', NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (31, 'Metafora Y. Vasile',7.5, 9.50,  'SIA', 'CEA', NULL, NULL,
    NULL, NULL, NULL);
```

```
INSERT INTO candidati VALUES (32, 'Strasina R. Elvis', 7.5, 9.50, 'CEA', NULL, NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (33, 'Durere V. Vasile', 7.5, 9.50, 'FAB', 'CEA', 'EAI', 'MARK',
    'MRU', NULL, NULL);
INSERT INTO candidati VALUES (34, 'Sedentaru L. Marius-Daniel',7.5, 9.50,  'MARK', 'MRU',
    'EAI', NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (35, 'Zgircitu I. Daniel',7.5, 9.50,  'MRU', NULL, NULL, NULL,
    NULL, NULL, NULL);
```

Funcția-procedură *p_admitere* (listing 16.53) aduce câteva noutăți PL/pgSQL. Cursorul *c_mastere* este unul parametrizat, la deschiderea sa făcându-se filtrarea în funcție de valoarea curentă a parametrului *CodCl_*. Ambele cursoare prezintă opțiunea FOR UPDATE ceea ce înseamnă că, pe baza înregistrării curente a cursorului, se va putea face actualizarea liniei din tabela de bază din care provine cursorul. Din păcate, în PL/pgSQL lucrurile funcționează numai pentru cursorul *c_mastere* (nu ştiu de ce pentru *c_candidați* lucrurile nu sunt în regulă la folosirea clauzei FOR UPDATE). Există, de asemenea, trei variabile de tip RECORD care este un tip linie (înregistrare) general.

Listing 16.53. Procedura PL/pgSQL pentru admiterea candidaţilor

```
CREATE OR REPLACE FUNCTION p_admitere () RETURNS VOID AS
$$
DECLARE
    DECLARE c_mastere CURSOR (spec_ mastere.spec%TYPE) IS
        SELECT * FROM mastere WHERE spec = spec_ AND NrLocuri > NrLocuri_Ocupate
        FOR UPDATE ;
    DECLARE c_candidati CURSOR IS
        SELECT * FROM candidati ORDER BY Media_AS * 0.6 + Media_Lic * 0.4 DESC
        FOR UPDATE  ;
    rec_candidati RECORD ;
    rec_optiuni RECORD ;
    rec_mastere RECORD ;

BEGIN
    UPDATE mastere SET NrLocuri_Ocupate = 0 ;
    UPDATE candidati SET spec_repartizat = NULL ;

    OPEN c_candidati ;
    LOOP
        FETCH c_candidati INTO rec_candidati ;
        IF NOT FOUND THEN
                EXIT ;
        END IF ;

        FOR rec_optiuni IN (
                SELECT rec_candidati.IdCandidat, 1 AS OptiuneNr, rec_candidati.Spec1 AS Spec
                        UNION
                SELECT rec_candidati.IdCandidat, 2 AS OptiuneNr, rec_candidati.Spec2 AS Spec
                        UNION
                SELECT rec_candidati.IdCandidat, 3 AS OptiuneNr, rec_candidati.Spec3 AS Spec
                        UNION
                SELECT rec_candidati.IdCandidat, 4 AS OptiuneNr, rec_candidati.Spec4 AS Spec
                        UNION
                SELECT rec_candidati.IdCandidat, 5 AS OptiuneNr, rec_candidati.Spec5 AS Spec
                        UNION
```

```
                SELECT rec_candidati.IdCandidat, 6 AS OptiuneNr, rec_candidati.Spec6 AS Spec
                                        ORDER BY 2 ) LOOP

                IF rec_optiuni.Spec IS NULL THEN
                        EXIT ;
                END IF ;

                OPEN c_mastere (rec_optiuni.Spec) ;
                FETCH c_mastere INTO rec_mastere ;
                IF FOUND THEN
                        UPDATE mastere SET NrLocuri_Ocupate = NrLocuri_Ocupate + 1
                        WHERE CURRENT OF c_mastere ;
                        UPDATE candidati SET spec_repartizat = rec_optiuni.Spec
                        WHERE IdCandidat = rec_candidati.IdCandidat ;
                                -- nu functioneaza CURRENT OF c_candidati !!!!!
                        CLOSE c_mastere ;
                        EXIT;
                ELSE
                        CLOSE c_mastere ;
                END IF ;
        END LOOP  ;
    END LOOP ;
    CLOSE c_candidati ;
END ;
$$ LANGUAGE plpgsql ;
```

Funcția procedură începe prin a face curățenie, NULL-izând atributul *spec_repartizat* pentru toți candidații şi „zerorizând" valorile atributului *NrLocuri_Ocupate* pentru toate specializările de master. Apoi se parcurg candidații în ordinea descrescătoare a mediei de admitere (expresia *Media_AS * 0.6 + Media_Lic * 0.4*). Pentru fiecare candidat se constuieşte un cursor ad-hoc ale cărui înregistrări conțin opțiunile pe care le-a bifat candidatul (ordinea opțiunilor este esențială). Se parcurg toate aceste opțiuni până când, fie candidatului i se găseşte o specializare la care mai sunt locuri neocupate, fie opțiunea următoare este nulă (ceea ce înseamnă că pe candidat nu l-au interesat toate cele şase specializări). Dacă i se găseşte o specializare (dorită) în care există locuri disponibile, atunci pentru acest candidat atributul *spec_repartizat* preia numele specializării, iar aceasteia i se incrementează numărul locurilor ocupate.

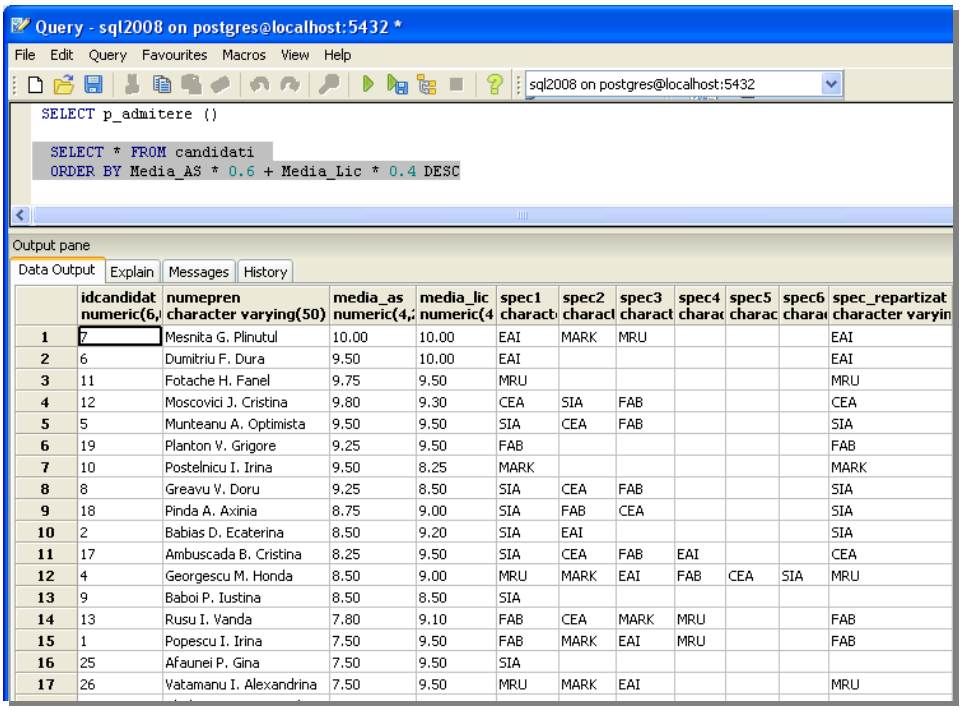
Lansarea în execuție este simplă:

```
SELECT p_admitere ()
```

După lansarea procedurii examinăm o parte din conținutul tabelei CANDIDAȚI ordonat după medie – vezi figura 16.24 - şi observăm că, pe linia 11, candidata *Ambuscadă B. Cristina* a avut ca primă opțiunea specializarea *SIA* (Sisteme informaționale pentru afaceri), însă a fost repartizată la *CEA* (Contabilitate, Expertiză şi Audit). Motivul este că la SIA au existat doar patru locuri care au fost ocupate de alți cadidați cu medie mai mare.

În figură două candidate, Baboi P. Iustina şi Afaunei P. Gina sunt respinse (valoarea atributului *spec_repartizat* este NULL) întrucât acestea au dorit numai

această specializare ale cărei locuri au fost ocupate de candidații cu medii mai mari.



	idcandidat numeric(6,0)	numepren character varying(50)	media_as numeric(4,2)	media_lic numeric(4	spec1 charact	spec2 charact	spec3 charact	spec4 charac	spec5 charac	spec6 charac	spec_repartizat character varyin
1	7	Mesnita G. Plinutul	10.00	10.00	EAI	MARK	MRU				EAI
2	6	Dumitriu F. Dura	9.50	10.00	EAI						EAI
3	11	Fotache H. Fanel	9.75	9.50	MRU						MRU
4	12	Moscovici J. Cristina	9.80	9.30	CEA	SIA	FAB				CEA
5	5	Munteanu A. Optimista	9.50	9.50	SIA	CEA	FAB				SIA
6	19	Planton V. Grigore	9.25	9.50	FAB						FAB
7	10	Postelnicu I. Irina	9.50	8.25	MARK						MARK
8	8	Greavu V. Doru	9.25	8.50	SIA	CEA	FAB				SIA
9	18	Pinda A. Axinia	8.75	9.00	SIA	FAB	CEA				SIA
10	2	Babias D. Ecaterina	8.50	9.20	SIA	EAI					SIA
11	17	Ambuscada B. Cristina	8.25	9.50	SIA	CEA	FAB	EAI			CEA
12	4	Georgescu M. Honda	8.50	9.00	MRU	MARK	EAI	FAB	CEA	SIA	MRU
13	9	Baboi P. Iustina	8.50	8.50	SIA						
14	13	Rusu I. Vanda	7.80	9.10	FAB	CEA	MARK	MRU			FAB
15	1	Popescu I. Irina	7.50	9.50	FAB	MARK	EAI	MRU			FAB
16	25	Afaunei P. Gina	7.50	9.50	SIA						
17	26	Vatamanu I. Alexandrina	7.50	9.50	MRU	MARK	EAI				MRU

Figura 16.24. Verificarea funcţiei-procedură ***p_admitere***

16.5.2. Proceduri stocate în Oracle PL/SQL

Sintaxa şi funcţionalitatea procedurilor stocate redactate în Oracle PL/SQL respectă canoanele „clasice”. Începem cu o procedură care afişează în dreptul fiecărei facturi atât valoarea sa, cât şi valoarea cumulată a facturilor precedente (plus a celei curente) – vezi listing 16.54.

Listing 16.54. Procedura PL/SQL ***p_cumul***

```
-- procedurea de afisare a valorii facturii curente si sumei valorilor facturilor precedente
-- varianta 2 - cursor
CREATE OR REPLACE PROCEDURE p_cumul2
IS
    v_suma NUMBER(16,2) := 0 ;
BEGIN
    FOR rec_fact IN (
                SELECT nrfact, datafact, f_valtot(nrfact) AS valfact
                FROM facturi ) LOOP
        v_suma := v_suma + NVL(rec_fact.valfact,0) ;
        DBMS_OUTPUT.PUT_LINE (rec_fact.NrFact || ' - ' || rec_fact.datafact ||
                ' - ' || TO_CHAR(NVL(rec_fact.valfact,0), '999999999999.99') || ' - ' ||
                TO_CHAR(v_suma, '999999999999.99') ) ;
```

```
    END LOOP ;
END ;
```

Procedura care adaugă la observaţiile unei facturi un eventual mesaj care semnalizează că factura nu are nicio linie (echivalenta celei din listing 16.49) este subiectul listing-ului 16.55.

Listing 16.55. Procedura PL/SQL ***p_update_facturi_fara_linii***

```
CREATE OR REPLACE PROCEDURE p_update_facturi_fara_linii
IS
BEGIN
    UPDATE facturi SET obs = COALESCE(obs,' ') || 'Nu are linii !'
    WHERE nrfact NOT IN (SELECT DISTINCT nrfact FROM liniifact)
        AND COALESCE(obs,' ') NOT LIKE '%Nu are linii !%' ;
END ;
/
```

Lansarea procedurii se poate face dintr-un bloc anonim astfel:

```
BEGIN
  p_update_facturi_fara_linii;
END ;
```

Continuăm cu funcţia substitut al clauzei ON DELETE CASCADE (vezi listing 16.56), mult mai utilă în Oracle decât în PostgreSQL şi SQL Server.

Listing 16.56. Procedura PL/SQL ***p_on_delete_cascade_factura***

```
CREATE OR REPLACE PROCEDURE
    p_on_delete_cascade_factura (nrfact_ facturi.nrfact%TYPE)
IS
BEGIN
    DELETE FROM incasfact WHERE nrfact = nrfact_ ;
    DELETE FROM liniifact WHERE nrfact = nrfact_ ;
    DELETE FROM facturi WHERE nrfact = nrfact_ ;
END ;
```

Ajungem la procedura dedicată admiterii la master. În Oracle avem deja create tabelele (din paragraful 13.3.2), însă la structura iniţială trebuie să mai facem câteva adăugiri:

```
ALTER TABLE mastere ADD (NrLocuri_Ocupate NUMERIC(3) DEFAULT 0) ;
ALTER TABLE candidati ADD (Spec_Repartizat VARCHAR2(4)
REFERENCES mastere(Spec)) ;
```

Procedura seamănă leit cu cea din PL/pgSQL (din listing 16.53) – vezi listing 16.57 – cu observaţia că ambele cursoare pot actualiza tabelele prin comanda *UPDATE... FOR CURRENT...* .

Listing 16.57. Procedura PL/SQL ***p_admitere***

```
CREATE OR REPLACE PROCEDURE p_admitere
AS
    CURSOR c_mastere (spec_ mastere.spec%TYPE) IS
```

```
        SELECT * FROM mastere WHERE spec = spec_ AND NrLocuri > NrLocuri_Ocupate
                FOR UPDATE  ;
    CURSOR c_candidati IS
        SELECT * FROM candidati ORDER BY (Media_AS * 0.6 + Media_Lic * 0.4) DESC
                FOR UPDATE  ;
BEGIN
    UPDATE mastere SET NrLocuri_Ocupate = 0 ;
    UPDATE candidati SET spec_repartizat = NULL ;

    FOR rec_candidati IN c_candidati LOOP
        FOR rec_optiuni IN (
                SELECT rec_candidati.IdCandidat, 1 AS OptiuneNr, rec_candidati.Spec1 AS Spec
                        FROM dual UNION
                SELECT rec_candidati.IdCandidat, 2 AS OptiuneNr, rec_candidati.Spec2 AS Spec
                        FROM dual UNION
                SELECT rec_candidati.IdCandidat, 3 AS OptiuneNr, rec_candidati.Spec3 AS Spec
                        FROM dual UNION
                SELECT rec_candidati.IdCandidat, 4 AS OptiuneNr, rec_candidati.Spec4 AS Spec
                        FROM dual UNION
                SELECT rec_candidati.IdCandidat, 5 AS OptiuneNr, rec_candidati.Spec5 AS Spec
                        FROM dual UNION
                SELECT rec_candidati.IdCandidat, 6 AS OptiuneNr, rec_candidati.Spec6 AS Spec
                        FROM dual ORDER BY 2 ) LOOP
                IF rec_optiuni.Spec IS NULL THEN
                        EXIT ;
                END IF ;

                FOR rec_mastere IN c_mastere (rec_optiuni.Spec) LOOP
                        UPDATE mastere SET NrLocuri_Ocupate = NrLocuri_Ocupate + 1
                                WHERE CURRENT OF c_mastere ;
                        UPDATE candidati SET spec_repartizat = rec_optiuni.Spec
                                WHERE CURRENT OF c_candidati ;
                        EXIT;
                END LOOP  ;
        END LOOP ;
    END LOOP ;
END ;
```

Încheiem paragraful cu o procedură destinată actualizării tabelei LINIIFACT – astfel încât liniile din fiecare factură să fie consecutive. Cu alte cuvinte, dacă dintr-o factură se şterge o înregistrare (să zicem, linia 2) care nu este ultima, ar apărea un interval (gaură) în numerotarea liniilor. Ori, procedura *p_renumerotare_linii* (din listing 16.58) „astupă găurile" din toată tabela.

Listing 16.58. Procedura PL/SQL ***p_renumerotare_linii***

```
CREATE OR REPLACE PROCEDURE p_renumerotare_linii
AS
 i NUMBER(2) ;
BEGIN
  FOR rec_facturi IN (SELECT NrFact FROM liniifact
    GROUP BY NrFact HAVING COUNT(*) <> MAX(Linie)
            ) LOOP
    i := 1 ;
    FOR rec_liniifact IN (SELECT * FROM liniifact WHERE NrFact=rec_facturi.NrFact
                                ORDER BY Linie) LOOP
      IF rec_liniifact.linie <> i THEN
```

```
            UPDATE liniifact SET linie = 1
                    WHERE NrFact=rec_facturi.NrFact AND linie=rec_liniifact.linie ;
          END IF ;
          i := i + 1 ;
        END LOOP ;
      END LOOP ;
    END ;
```

16.5.3. Proceduri stocate în Transact-SQL

Procedurile stocate în T-SQL au câteva note de exotism. Mai întâi, ca şi în PostgreSQL (deşi în PostgreSQL nu avem porceduri), într-o procedură T-SQL putem înşirui oricâte comenzi SQL, inclusiv SELECT-uri. Prin urmare, procedurile pot semăna pe alocuri cu scripturile. În altă ordine de idei, analog tabelelor, procedurile pot fi declarate ca fiind temporare, locale sau globale. Noi ne vom feri de exotisme, păstrând exemplele din paragrafele anterioare. Începem cu procedura *p_update_facturi_fara_linii* din listing-ul 16.59.

Listing 16.59. Procedura T-SQL ***p_update_facturi_fara_linii()***

```
CREATE PROCEDURE p_update_facturi_fara_linii
AS
BEGIN
    UPDATE facturi SET Obs = COALESCE(Obs,' ') + 'Nu are linii !'
    WHERE NrFact NOT IN (SELECT DISTINCT NrFact FROM liniifact)
                AND COALESCE(Obs,' ') NOT LIKE '%Nu are linii !%'
END
```

Apelul procedurii se face prin comanda EXECUTE:

```
EXECUTE p_update_facturi_fara_linii
```

Continuăm cu procedura dedicată admiterii la master. Listing-ul 16.60 conţine comenzile de creare şi populare a celor două tabele.

Listing 16. 60. Crearea/ popularea în T-SQL a celor două tabele pentru admiterea la master

```
CREATE TABLE mastere (
    Spec VARCHAR(4) NOT NULL PRIMARY KEY,
    DenSpec VARCHAR(40) NOT NULL,
    NrLocuri NUMERIC(3),
    NrLocuri_Ocupate NUMERIC(3)
     ) ;

CREATE TABLE candidati (
    IdCandidat NUMERIC(6) NOT NULL PRIMARY KEY,
    NumePren VARCHAR(50) NOT NULL,
    Media_AS NUMERIC(4,2) NOT NULL CONSTRAINT ck_media_as
        CHECK (Media_AS BETWEEN 5 AND 10),
    Media_Lic NUMERIC(4,2) NOT NULL CONSTRAINT ck_media_lic
        CHECK (Media_Lic BETWEEN 6 AND 10),
    Spec1 VARCHAR(4) NOT NULL REFERENCES mastere(Spec),
    Spec2 VARCHAR(4) REFERENCES mastere(Spec),
```

```
        Spec3 VARCHAR(4) REFERENCES mastere(Spec),
        Spec4 VARCHAR(4) REFERENCES mastere(Spec),
        Spec5 VARCHAR(4) REFERENCES mastere(Spec),
        Spec6 VARCHAR(4) REFERENCES mastere(Spec),
        Spec_Repartizat VARCHAR(4) REFERENCES mastere(Spec),
        CONSTRAINT ck_spec CHECK (
            COALESCE(Spec2, ' ') = COALESCE(Spec2, Spec3, ' ') AND
            COALESCE(Spec3, ' ') = COALESCE(Spec3, Spec4, ' ') AND
            COALESCE(Spec4, ' ') = COALESCE(Spec4, Spec5, ' ') AND
            COALESCE(Spec5, ' ') = COALESCE(Spec5, Spec6, ' ')      )
                    ) ;

INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('SIA',
    'Sisteme informationale pentru afaceri', 4) ;
INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('FAB', 'Finante-Asigurari-Banci', 6) ;
INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('CEA',
    'Contabilitate, expertiza si audit', 6) ;
INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('MARK', 'Marketing', 5) ;
INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('EAI',
    'Economie si afaceri internationale', 5) ;
INSERT INTO mastere (Spec, DenSpec, NrLocuri) VALUES ('MRU',
    'Managementul resurselor umane', 4) ;

INSERT INTO candidati VALUES ( 1, 'Popescu I. Irina', 7.5, 9.50, 'FAB', 'MARK', 'EAI', 'MRU',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 2, 'Babias D. Ecaterina', 8.5, 9.20, 'SIA', 'EAI', NULL,  NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 3, 'Strat P. Iulian', 7.35, 9.50, 'CEA', 'EAI', 'MRU', 'SIA',
    'FAB', 'MARK', NULL);
INSERT INTO candidati VALUES ( 4, 'Georgescu M. Honda', 8.5, 9.00, 'MRU', 'MARK', 'EAI',
    'FAB', 'CEA', 'SIA', NULL);
INSERT INTO candidati VALUES ( 5, 'Munteanu A. Optimista', 9.5, 9.50, 'SIA', 'CEA', 'FAB',
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 6, 'Dumitriu F. Dura', 9.5, 10, 'EAI', NULL, NULL,  NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 7, 'Mesnita G. Plinutul', 10, 10,  'EAI', 'MARK', 'MRU',  NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 8, 'Greavu V. Doru', 9.25, 8.50,  'SIA', 'CEA', 'FAB',  NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES ( 9, 'Baboi P. Iustina', 8.5, 8.50,  'SIA', NULL, NULL,  NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (10, 'Postelnicu I. Irina', 9.5, 8.25,  'MARK', NULL, NULL,
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (11, 'Fotache H. Fanel', 9.75, 9.50,  'MRU', NULL, NULL,
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (12, 'Moscovici J. Cristina', 9.80, 9.30,  'CEA', 'SIA', 'FAB',
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (13, 'Rusu I. Vanda', 7.80, 9.10,  'FAB', 'CEA', 'MARK',  'MRU',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (14, 'Spinu M. Algebra', 7.25, 9.00,  'SIA', 'CEA', 'FAB',  NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (15, 'Sandovici I. Irina', 7.05, 7.50,  'MARK', 'MRU', 'EAI',  'CEA',
     NULL, NULL, NULL);
INSERT INTO candidati VALUES (16, 'Plai V. Picior', 7.5, 7.90,  'EAI', 'SIA', 'CEA', 'FAB',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (17, 'Ambuscada B. Cristina', 8.25, 9.50,  'SIA', 'CEA',
    'FAB', 'EAI',  NULL, NULL, NULL);
INSERT INTO candidati VALUES (18, 'Pinda A. Axinia', 8.75, 9.00,  'SIA', 'FAB', 'CEA',  NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (19, 'Planton V. Grigore', 9.25, 9.50,  'FAB', NULL, NULL, NULL,
```

```
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (20, 'Sergentu I. Zece',7.5, 9.50,  'FAB', 'CEA', NULL,  NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (21, 'Ababei T. Marian-Vasile', 7.5, 9.50,  'CEA', 'MRU',
    NULL, NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (22, 'Popistasu J. Maria', 7.5, 9.50,  'FAB', 'EAI', 'CEA', 'MRU',
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (23, 'Plop R. Robert', 7.5, 9.50, 'FAB', 'MARK', 'MRU', NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (24, 'Aflorei H. Crina',7.5, 9.50,  'EAI', 'SIA', NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (25, 'Afaunei P. Gina',7.5, 9.50,  'SIA', NULL, NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (26, 'Vatamanu I. Alexandrina', 7.5, 9.50, 'MRU', 'MARK', 'EAI',
     NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (27, 'Lovin P. Marian',7.5, 9.50,  'MARK', 'MRU', NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (28, 'Antiteza W. Florin', 7.5, 9.50, 'EAI', 'MARK', 'MRU',
    NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (29, 'Prepelita V. Ion', 7.5, 9.50, 'EAI', NULL, NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (30, 'Cioara X. Sanda', 7.5, 9.50, 'CEA', 'FAB', 'EAI', NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (31, 'Metafora Y. Vasile',7.5, 9.50,  'SIA', 'CEA', NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (32, 'Strasina R. Elvis', 7.5, 9.50, 'CEA', NULL, NULL, NULL,
    NULL, NULL, NULL);
INSERT INTO candidati VALUES (33, 'Durere V. Vasile', 7.5, 9.50, 'FAB', 'CEA', 'EAI', 'MARK',
    'MRU', NULL, NULL);
INSERT INTO candidati VALUES (34, 'Sedentaru L. Marius-Daniel',7.5, 9.50,  'MARK', 'MRU',
    'EAI', NULL, NULL, NULL, NULL);
INSERT INTO candidati VALUES (35, 'Zgircitu I. Daniel',7.5, 9.50,  'MRU', NULL, NULL, NULL,
    NULL, NULL, NULL);
```

Procedura *p_admitere* (listing 16.61) foloseşte, ca şi suratele ei din PL/pgSQL şi PL/SQL două cursoare, *c_candidaţi* şi *c_opţiuni*. Pentru ambele cursoare se foloseşte (şi funcţionează) clauzele *FOR UPDATE ...* şi *FOR CURRENT OF... .*

Listing 16.61. Procedura T-SQL pentru admiterea candidaţilor

```
CREATE PROCEDURE p_admitere  AS
BEGIN
    DECLARE @spec VARCHAR(4)
    DECLARE @gata BIT
    DECLARE @nr TINYINT

    DECLARE @idcandidat NUMERIC(6)
    DECLARE @spec1 VARCHAR(4)          DECLARE @spec2 VARCHAR(4)
    DECLARE @spec3 VARCHAR(4)          DECLARE @spec4 VARCHAR(4)
    DECLARE @spec5 VARCHAR(4)          DECLARE @spec6 VARCHAR(4)

    DECLARE @spec_m VARCHAR(4) DECLARE @denspec VARCHAR(40)
    DECLARE @nrlocuri NUMERIC(3) DECLARE @nrlocuri_ocupate NUMERIC(3)

    UPDATE mastere SET NrLocuri_Ocupate = 0
    UPDATE candidati SET Spec_Repartizat = NULL

    DECLARE c_candidati CURSOR FOR
```

```
        SELECT IdCandidat, Spec1, Spec2, Spec3, Spec4, Spec5, Spec6
        FROM candidati ORDER BY Media_AS * 0.6 + Media_Lic * 0.4 DESC FOR UPDATE

    OPEN c_candidati ;
    FETCH NEXT FROM c_candidati INTO @idcandidat, @spec1, @spec2, @spec3,
        @spec4, @spec5, @spec6
    WHILE @@FETCH_STATUS = 0 BEGIN

        DECLARE c_optiuni CURSOR FOR
                SELECT 1 AS Nr, @spec1 AS Spec UNION
                SELECT 2 AS Nr, @spec2 AS Spec UNION
                SELECT 3 AS Nr, @spec3 AS Spec UNION
                SELECT 4 AS Nr, @spec4 AS Spec UNION
                SELECT 5 AS Nr, @spec5 AS Spec UNION
                SELECT 6 AS Nr, @spec6 AS Spec ORDER BY 1
        OPEN c_optiuni
        FETCH NEXT FROM c_optiuni INTO @nr, @spec
        SET @gata = 0
        WHILE @@FETCH_STATUS = 0 AND @gata=0 BEGIN
                IF @spec IS NULL SET @gata=1
                ELSE
                BEGIN
                        DECLARE c_mastere CURSOR FOR
                                SELECT * FROM mastere WHERE Spec = @spec
                                        AND NrLocuri > NrLocuri_Ocupate
                                FOR UPDATE

                        OPEN c_mastere  ;
                        FETCH NEXT FROM c_mastere INTO @spec_m, @denspec,
                                @nrlocuri, @nrlocuri_ocupate
                        IF @@FETCH_STATUS = 0
                                BEGIN
                                UPDATE mastere SET NrLocuri_Ocupate =
                                        NrLocuri_Ocupate + 1
                                        WHERE CURRENT OF c_mastere ;
                                UPDATE candidati SET Spec_Repartizat = @spec
                                        WHERE CURRENT OF c_candidati
                                CLOSE c_mastere
                                DEALLOCATE c_mastere
                                SET @gata = 1
                                END
                        ELSE
                                BEGIN
                                CLOSE c_mastere
                                DEALLOCATE c_mastere
                                END
                END
                FETCH NEXT FROM c_optiuni INTO @nr, @spec
        END
        CLOSE c_optiuni
        DEALLOCATE c_optiuni
        FETCH NEXT FROM c_candidati INTO @idcandidat, @spec1, @spec2, @spec3,
                @spec4, @spec5, @spec6
    END
    CLOSE c_candidati
    DEALLOCATE c_candidati
END
```

16.5.1. Proceduri stocate în SQL PL

Nici sintaxa procedurilor redactate în IBM SQL Programming Language nu ridică probleme deosebite, odată ce am trecut prin experiența funcțiilor. Iată în listing 16.62 procedura pentru actualizarea observațiilor pentru facturile care nu au nicio linie (în LINIIFACT).

Listing 16.62. Procedura SQL PL ***p_update_facturi_fara_linii***

```
CREATE PROCEDURE p_update_facturi_fara_linii ( )
P1: BEGIN
    UPDATE facturi SET obs = COALESCE(obs,' ') || 'Nu are linii !'
    WHERE nrfact NOT IN (SELECT DISTINCT nrfact FROM liniifact)
        AND COALESCE(obs,' ') NOT LIKE '%Nu are linii !%' ;
END P1
```

Deosebirea cea mai importantă este, probabil, cea legată de apelul procedurilor care se realizează cu ajutorul comenzii CALL:

```
CALL p_update_facturi_fara_linii
```

Cea mai simplă rămâne procedura de ştergere în cascadă a unei facturi – listing 16.63.

Listing 16.63. Procedura SQL PL ***p_on_delete_cascade_factura***

```
CREATE PROCEDURE p_on_delete_cascade_factura (nrfact_ INTEGER)
    DYNAMIC RESULT SETS 1
P1: BEGIN
    DELETE FROM incasfact WHERE nrfact = nrfact_ ;
    DELETE FROM liniifact WHERE nrfact = nrfact_ ;
    DELETE FROM facturi WHERE nrfact = nrfact_ ;
END P1
```

La funcția ce returnează seturi de înregistrări din paragraful 16.3.4 (listing 16.43) adăugăm procedura *f_facturi_filtrate2* din listing 16.64[5]. Procedura declară un cursor creat pe baza unei interogări în care se face filtrarea după valoarea

[5] Idee preluată din [IBM 2007], p.100 şi [Janmohamed s.a. 2004]

parametrilor de intrare şi lasă acest cursor deschis pentru aplicaţia client. De remarcat şi clauza WITH RETURN folosită la declararea cursorului.

Listing 16.64. Procedură SQL PL ce „returnează” un cursor - set de înregistrări

```
CREATE PROCEDURE f_facturi_filtrate2 ( codcl_ SMALLINT, data_initiala 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 = COALESCE(codcl_, CodCl)
                        AND DataFact BETWEEN COALESCE(data_initiala, '2005-01-01')
                        AND COALESCE(data_finala, '2010-12-31') ;
    -- Cursorul rămâne deschis pentru aplicaţia client
    OPEN cursor1;
END P1
```

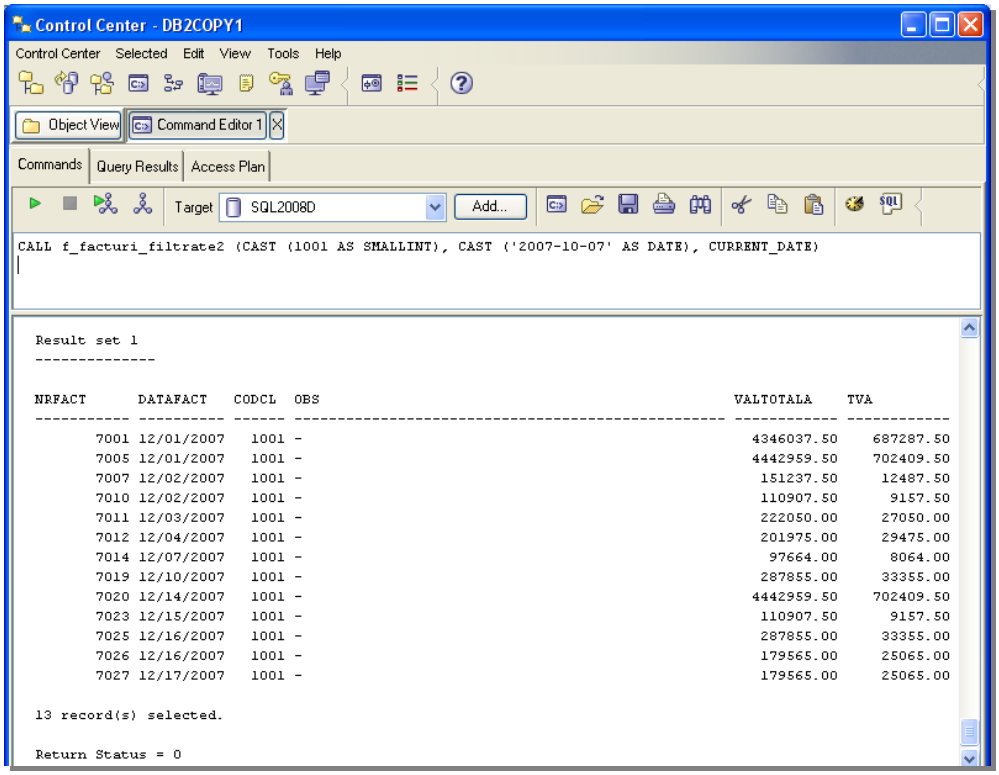


Figura 16.25. Apelul procedurii ***f_facturi_filtrate2***

Procedura nu poate fi apelată din interogări, ci doar din alte blocuri/module interne (SQL PL) sau externe (scrise în alte limbaje). Cu toate acestea, putem testa

corectitudinea procedurii prin comanda CALL lansată în *DB2 Control Center* (vezi figura 16.25):

```
CALL f_facturi_filtrate2 (CAST (1001 AS SMALLINT),
        CAST ('2007-10-07' AS DATE), CURRENT_DATE)
```

. Încheiem, cum e şi normal, cu procedura de admitere. Crearea şi popularea celor două tabele – MASTERE şi CANDIDATI - se realizează în DB2 cu acelaşi script din SQL Server (listing 16.60). Listingul 16.65 conține corpul procedurii p_admitere realizată în sintaxă SQL PL.

Listing 16.65. Procedură SQL PL de admitere la master

```
CREATE PROCEDURE p_admitere (  )
P1: BEGIN
    DECLARE v_idcandidat DECIMAL(6) ;
    DECLARE v_spec1 VARCHAR(4) ;
    DECLARE v_spec2 VARCHAR(4) ;
    DECLARE v_spec3 VARCHAR(4) ;
    DECLARE v_spec4 VARCHAR(4) ;
    DECLARE v_spec5 VARCHAR(4) ;
    DECLARE v_spec6 VARCHAR(4) ;
    DECLARE v_spec_repartizat VARCHAR(4) ;
    DECLARE v_spec VARCHAR(4) ;
    DECLARE v_spec_x VARCHAR(4) ;
    DECLARE v_nlo DECIMAL(3) ;
    DECLARE SQLSTATE CHAR(5);
    DECLARE v_gata SMALLINT  ;

    DECLARE c_candidati CURSOR FOR
        SELECT idcandidat, spec1, spec2, spec3, spec4, spec5, spec6, spec_repartizat
        FROM candidati ORDER BY (Media_AS * 0.6 + Media_Lic * 0.4) DESC
            FOR UPDATE OF spec_repartizat ;

    DECLARE  c_mastere CURSOR FOR
        SELECT spec, NrLocuri_Ocupate
        FROM mastere WHERE spec = v_spec_x AND NrLocuri > NrLocuri_Ocupate
            FOR UPDATE OF NrLocuri_Ocupate ;


    UPDATE mastere SET NrLocuri_Ocupate = 0 ;
    UPDATE candidati SET spec_repartizat = NULL ;

    OPEN c_candidati ;
    FETCH FROM c_candidati INTO v_idcandidat, v_spec1, v_spec2, v_spec3, v_spec4,
        v_spec5, v_spec6, v_spec_repartizat  ;
    WHILE ( SQLSTATE = '00000' ) DO
        FOR rec_optiuni AS
            SELECT v_IdCandidat, 1 AS OptiuneNr, v_Spec1 AS Spec FROM sysibm.dual UNION
            SELECT v_IdCandidat, 2 AS OptiuneNr, v_Spec2 AS Spec FROM sysibm.dual
                                WHERE v_spec2 IS NOT NULL UNION
            SELECT v_IdCandidat, 3 AS OptiuneNr, v_Spec3 AS Spec FROM sysibm.dual
                    WHERE v_spec3 IS NOT NULL UNION
            SELECT v_IdCandidat, 4 AS OptiuneNr, v_Spec4 AS Spec FROM sysibm.dual
                    WHERE v_spec4 IS NOT NULL UNION
            SELECT v_IdCandidat, 5 AS OptiuneNr, v_Spec5 AS Spec FROM sysibm.dual
                    WHERE v_spec5 IS NOT NULL UNION
            SELECT v_IdCandidat, 6 AS OptiuneNr, v_Spec6 AS Spec FROM sysibm.dual
```

```
                        WHERE v_spec6 IS NOT NULL          ORDER BY 2  DO

        SET v_spec_x =  rec_optiuni.spec ;

        SET v_gata = 0 ;
        OPEN c_mastere ;
        FETCH FROM c_mastere INTO v_spec, v_nlo ;
        WHILE ( SQLSTATE = '00000' ) AND v_gata = 0 DO
                UPDATE mastere SET NrLocuri_Ocupate = NrLocuri_Ocupate + 1
                        WHERE CURRENT OF c_mastere ;
                UPDATE candidati SET spec_repartizat = rec_optiuni.Spec
                        WHERE CURRENT OF c_candidati ;
                SET v_gata = 1 ;
        END WHILE  ;
        CLOSE c_mastere ;
    END FOR ;
        FETCH FROM c_candidati INTO v_idcandidat, v_spec1, v_spec2, v_spec3, v_spec4,
        v_spec5, v_spec6, v_spec_repartizat  ;
    END WHILE ;
END P1
```

16.6. Pachete

Există mai multe accepțiuni ale pachetelor. Noi ne vom concentra asupra elementelor procedurale aşa cum se găsesc în Oracle PL/SQL. Fiind oarecum specifice, nici la acest subiect nu vom insista cu prea multe detalii. Folosirea pachetelor în PL/SQL are cel puțin două avantaje. Mai întâi, permite gruparea tuturor procedurilor şi funcțiilor destinate unui grup de probleme, precum un director pe disc în care se grupează o serie de fişiere. În al doilea rând, toate variabilele declarate în specificațiile unui pachet sunt publice.

Începem cu un pachet care are aceeaşi funcționalitate cu funcția *f_facturi_filtrate_1* din paragraful 16.3.2 (listing 16.37) ce returnează un set de înregistrări. Orice pachet are două părți. Prima – *specificațiile* - este publică şi obligatorie, conținând variabile, cursoare şi excepții, precum şi elementele de identificare (nume, parametri şi tipuri returnate, în cazul funcțiilor) ale unor eventuale proceduri şi/sau funcții. A doua – *corpul* -este „privată" (ascunsă), fiind cea în care este descris corpul procedurilor şi funcțiilor declarate în specificații. Cele două părți ale pachetului *pac_seturi* din listing 16.66 nu aduc neapărat un plus de productivitate, ci constituie doar o alternativă la varianta din paragraful 16.3.2.

Listing 16.66. Primul pachet PL/SQL

```
    ---------- specificatiile (antetul) pachetului
    CREATE OR REPLACE PACKAGE pac_seturi AS
      TYPE t_ref_facturi IS REF CURSOR ;
      FUNCTION f_facturi_filtrate1
            (data_initiala DATE, data_finala DATE)
            RETURN t_ref_facturi ;
    END ;
```

```
/
---------- corpul pachetului
CREATE OR REPLACE PACKAGE BODY pac_seturi AS
--------------------------------------------------------------------------------
 FUNCTION f_facturi_filtrate1 (data_initiala DATE,
   data_finala DATE) RETURN t_ref_facturi
AS
  set_of_facturi t_ref_facturi ;
BEGIN
  OPEN set_of_facturi FOR SELECT * FROM facturi WHERE datafact
         BETWEEN data_initiala AND data_finala ;
  RETURN set_of_facturi    ;
END f_facturi_filtrate1;

END ; -- aici se termina corpul pachetului
/
```

Tipul *t_ref_facturi* este declarat în specificaţii, fiind public. Devine astfel posibilă folosirea sa la definirea (în clauza RETURN) funcţiei *f_falcturi_filtrate1*. Apelul funcţiei aflate în pachet presupune prefixarea numelui ei cu numele pachetului:

```
SELECT pac_seturi.f_facturi_filtrate1(DATE'2007-08-15', CURRENT_DATE)
FROM dual
```

În continuare, funcţii şi proceduri descrise în paragrafe anterioare – *f_proctva*, *f_valtot*, *f_val_cumul*, *p_cumul*, *p_cumul2* - sunt incluse într-un pachetul *pac_vinzari* – vezi listing 16.67. Funcţia *f_proctva* are două semnături. Dacă la invocarea sa, parametrul trimis este numeric, PL/SQL „înţelege" că acesta este un cod de produs şi se execută prima „instanţă a", funcţiei, cea care returnează procentul de tva al produsului al cărui *cod* este specificat. Dacă, la invocare, parametrul este de tip şir de caractere, atunci se va executa cea doua instanţă a funcţiei, cea care returnează procentul de tva al unui produs a cărei *denumire* este specificată. Fireşte, această a doua instanţă a funcţiei funcţionează corect doar dacă nu există în tabela PRODUSE cel puţin două sortimente cu denumire identică. În caz contrar, SELECT-ul va identifica mai multe linii în tabelă, iar funcţia va genera o eroare.

Listing 16.67. Antetul pachetului ***pac_vinzari***

```
-- in pachetul PAC_VINZARI declaram patru variabile publice
CREATE OR REPLACE PACKAGE pac_vinzari AS
   -- acestea sunt cele patru variabile publice
   v_trg_liniifact BOOLEAN := FALSE ;
   v_trg_incasfact BOOLEAN := FALSE ;
   v_nrfact liniifact.nrfact%TYPE;
   v_linie liniifact.linie%TYPE;

     FUNCTION f_proctva (codpr_ produse.codpr%TYPE) RETURN produse.proctva%TYPE  ;
     FUNCTION f_proctva (denpr_ produse.denpr%TYPE) RETURN produse.proctva%TYPE  ;
     FUNCTION f_valtot (nrfact_ facturi.nrfact%TYPE)      RETURN NUMERIC ;
     FUNCTION f_val_cumul (nrfact_ facturi.nrfact%TYPE) RETURN NUMERIC ;
     PROCEDURE p_cumul ;
     PROCEDURE p_cumul2 ;
```

```
END ; -- pac_vinzari
/
-----------------------------------------------------------------
CREATE OR REPLACE PACKAGE BODY pac_vinzari AS
-----------------------------------------------------------------------------------
-- functie ce returneaza procentul TVA pentru un cod produs dat
FUNCTION f_proctva (codpr_ produse.codpr%TYPE)
        RETURN produse.proctva%TYPE
IS
    v_proc produse.proctva%TYPE ;
BEGIN
    SELECT proctva INTO v_proc FROM produse WHERE codpr = codpr_ ;
    RETURN v_proc ;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN 0;
    WHEN OTHERS THEN
        RETURN NULL ;
END f_proctva ;

-----------------------------------------------------------------------------------------------------
-- aceeasi functie ce returneaza procentul TVA pentru o denumire de produs data
FUNCTION f_proctva (denpr_ produse.denpr%TYPE)
        RETURN produse.proctva%TYPE
IS
    v_proc produse.proctva%TYPE ;
BEGIN
    SELECT proctva INTO v_proc FROM produse WHERE denpr = denpr_ ;
    RETURN v_proc ;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN 0;
    WHEN OTHERS THEN
        RETURN NULL ;
END f_proctva ;

-----------------------------------------------
-- functie ce returneaza valoarea cu TVA a unei facturi
FUNCTION f_valtot (nrfact_ facturi.nrfact%TYPE)
        RETURN NUMERIC
IS
    v_valfact NUMERIC(14,2) ;
BEGIN
    SELECT SUM(cantitate * pretunit * (1 + f_proctva(codpr)))
        INTO v_valfact FROM liniifact WHERE nrfact = nrfact_ ;
    RETURN v_valfact ;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN 0;
    WHEN OTHERS THEN
        RETURN NULL ;
END f_valtot ;

-----------------------------------------------
-- functie ce returneaza suma valorilor facturilor precedente (unei facturi)
FUNCTION f_val_cumul (nrfact_ facturi.nrfact%TYPE)
    RETURN NUMERIC
IS
    v_valtot NUMERIC(16,2) ;
BEGIN
```

```
        SELECT SUM(cantitate * pretunit * (1 + f_proctva(codpr)))
            INTO v_valtot
        FROM liniifact
        WHERE nrfact <= nrfact_ ;
        RETURN v_valtot ;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RETURN 0;
        WHEN OTHERS THEN
            RETURN NULL ;
    END f_val_cumul ;

    ------------------------------------------------
    -- procedurea de afisare a valorii facturii curente si sumei valorilor facturilor precedente
    -- varianta 2 - cursor
    PROCEDURE p_cumul
    IS
        CURSOR c_fact IS
            SELECT nrfact, datafact, f_valtot(nrfact) AS valfact
            FROM facturi ;
        rec_fact c_fact%ROWTYPE ;
        v_suma NUMBER(16,2) := 0 ;
    BEGIN
        OPEN c_fact ;
        FETCH c_fact INTO rec_fact ;
        LOOP
            EXIT WHEN c_fact%NOTFOUND ;
            v_suma := v_suma + NVL(rec_fact.valfact,0) ;
            DBMS_OUTPUT.PUT_LINE (rec_fact.NrFact || ' - ' || rec_fact.datafact ||
                    ' - ' || TO_CHAR(NVL(rec_fact.valfact,0), '999999999999.99') || ' - ' ||
                    TO_CHAR(v_suma, '999999999999.99') ) ;
            FETCH c_fact INTO rec_fact ;
        END LOOP ;
        CLOSE c_fact ;
    END p_cumul ;

    ------------------------------------------------
    -- procedurea de afisare a valorii facturii curente si sumei valorilor facturilor precedente
    -- varianta 3 - (TOT) cursor
    PROCEDURE p_cumul2
    IS
        v_suma NUMBER(16,2) := 0 ;
    BEGIN
        FOR rec_fact IN (
                    SELECT nrfact, datafact, f_valtot(nrfact) AS valfact
                    FROM facturi ) LOOP
            v_suma := v_suma + NVL(rec_fact.valfact,0) ;
            DBMS_OUTPUT.PUT_LINE (rec_fact.NrFact || ' - ' || rec_fact.datafact ||
                    ' - ' || TO_CHAR(NVL(rec_fact.valfact,0), '999999999999.99') || ' - ' ||
                    TO_CHAR(v_suma, '999999999999.99') ) ;
        END LOOP ;
    END p_cumul2 ;

    END ; -- sfirsit corp pachet pac_vinzari
```

Chiar dacă ne putem declara mulţumiţi pentru că procedurile şi funcţiile sunt mai adunate, plus că avem la dispoziţie câteva variabile publice pe care le vom pune la lucru în capitolul următor, trebuie să observăm şi un dezavantaj al unor

pachete. Atunci când lungimea lor creşte, sunt mai greu de urmărit, şi orice eroare strecurată într-o funcţie sau procedură atrage după sine invalidarea întregului pachet.