6. Modularizarea aplicaţiilor prin utilizarea subprogramelor

Noţiunea de subprogram (procedură sau funcţie) a fost concepută cu scopul de a grupa o mulţime de comenzi *SQL* cu instrucţiuni procedurale, pentru a construi o unitate logică de tratare a unei anumite probleme. În general, procedurile sunt folosite pentru a realiza o acţiune, iar funcţiile pentru a calcula o valoare.

Unităţile de program care pot fi create în *PL/SQL* sunt:

- subprograme locale (definite în partea declarativă a unui bloc *PL/SQL* sau a unui alt subprogram);
- subprograme independente (stocate în baza de date şi considerate obiecte ale acesteia);
- subprograme împachetate (definite într-un pachet care încapsulează proceduri şi funcţii).

Procedurile şi funcţiile stocate sunt unităţi de program *PL/SQL* apelabile (compilate), care există ca obiecte în schema bazei de date *Oracle*. Recuperarea unui subprogram (în cazul unei corecţii) nu cere recuperarea întregii aplicaţii. Subprogramul, încărcat în memorie pentru a fi executat, poate fi partajat între aplicaţiile care îl solicită.

Este important de făcut distincţie între procedurile stocate şi procedurile locale (declarate şi folosite în blocuri anonime).

- Procedurile şi funcţiile stocate, care sunt compilate şi stocate în baza de date, nu mai trebuie să fie compilate la fiecare execuţie, în timp ce procedurile locale sunt compilate de fiecare dată când este executat blocul care le conţine.
- Procedurile declarate şi apelate în blocuri anonime sunt temporare (ele nu mai există după ce blocul anonim a fost executat complet). O procedură stocată (creată cu *CREATE PROCEDURE* sau conţinută într-un pachet) este permanentă, în sensul că ea poate fi invocată de un fişier *SQL*Plus*, un subprogram *PL/SQL* sau un declanşator.
- Procedurile şi funcţiile stocate pot fi apelate din orice bloc, de către utilizatorul care are privilegiul *EXECUTE* asupra acestora, în timp ce procedurile şi funcţiile locale pot fi apelate numai din blocul care le conţine.

Când este creat un subprogram stocat, utilizând comanda *CREATE*, subprogramul este depus în dicţionarul datelor. Este depus atât textul sursă, cât şi forma compilată (*p-code*). Atunci când subprogramul este apelat, *p-code* este citit de pe disc, este depus în *shared pool*, unde poate fi accesat de mai mulţi

utilizatori şi este executat dacă este necesar. El va părăsi *shared pool* conform algoritmului *LRU* (*least recently used*).

Pachetul *DBMS_SHARED_POOL* permite păstrarea de obiecte în *shared pool*. Dacă un obiect este gestionat în această manieră, el va părăsi *shared pool* doar dacă aceasta se cere explicit. Pentru a păstra în *shared pool* subprograme, pachete, declanşatori, cursoare, clase *Java*, tipuri obiect, comenzi *SQL*, secvenţe este utilizată procedura *DBMS_SHARED_POOL.KEEP*. Unica modalitate de a şterge un obiect păstrat în *shared pool*, fără a reporni baza, este cu ajutorul procedurii *DBMS_SHARED_POOL.UNKEEP*.

Subprogramele se pot declara în blocuri *PL/SQL*, în alte subprograme sau în pachete, dar la sfârşitul secţiunii declarative. La fel ca blocurile *PL/SQL* anonime, subprogramele conţin o parte declarativă, o parte executabilă şi, opţional, o parte de tratare a erorilor. Partea declarativă conţine declaraţii de tipuri, cursoare, constante, variabile, excepţii şi subprograme imbricate. Partea executabilă conţine instrucţiuni care asignează valori, controlează execuţia programului şi prelucrează datele. Cea de-a treia parte se ocupă cu tratarea excepţiilor apărute în timpul execuţiei subprogramului.

Crearea subprogramelor stocate

Principalele etape pentru crearea unui subprogram stocat sunt următoarele:

- se editează subprogramul (*CREATE PROCEDURE* sau *CREATE FUNCTION*) şi se salvează într-un *script file SQL*;
- se încarcă şi se execută acest *script file*, se compilează codul sursă, se obţine *p-code* (subprogramul este creat);
- se utilizează comanda *SHOW ERRORS* pentru vizualizarea eventualelor erori la compilare (comanda *CREATE PROCEDURE* sau *CREATE FUNCTION* depune codul sursă în dicţionarul datelor chiar dacă subprogramul conţine erori la compilare);
- se execută subprogramul pentru a realiza acţiunea dorită (de exemplu, procedura poate fi executată fie utilizând comanda *EXECUTE* din *iSQL*Plus*, fie invocând-o dintr-un bloc *PL/SQL*).

Când este apelat subprogramul, motorul *PL/SQL* execută *p-code*.

Dacă există erori la compilare şi se fac corecţiile corespunzătoare, atunci este necesară fie comanda *DROP PROCEDURE* (respectiv *DROP FUNCTION*), fie sintaxa *OR REPLACE* în cadrul comenzii *CREATE*.

Atunci când este apelată o procedură *PL/SQL*, *server*-ul *Oracle* parcurge anumite etape care vor fi detaliate în cele ce urmează.

- Verifică dacă utilizatorul are privilegiul să execute procedura (fie pentru că el a creat procedura, fie pentru că i s-a acordat acest privilegiu).
- Verifică dacă procedura este prezentă în *shared pool*. Dacă este prezentă va fi executată, altfel va fi încărcată de pe disc în *database buffer cache*.
- Verifică dacă starea procedurii este validă (*VALID*) sau invalidă (*INVALID*). Starea unei proceduri *PL/SQL* este invalidă, fie pentru că au fost detectate erori la compilarea procedurii, fie pentru că structura unui obiect s-a schimbat de când procedura a fost executată ultima oară. Dacă starea este invalidă atunci procedura este recompilată automat. Dacă nici o eroare nu a fost detectată, atunci va fi executată noua versiune a procedurii.
- Dacă procedura aparţine unui pachet atunci toate procedurile şi funcţiile pachetului sunt de asemenea încărcate în *database buffer cache* (dacă nu erau deja acolo). Dacă pachetul este activat pentru prima oară într-o sesiune, atunci *server*-ul va executa blocul de iniţializare al pachetului.

Proceduri *PL/SQL*

O procedură *PL/SQL* este un program independent care se află compilat în schema bazei de date *Oracle*. Când procedura este compilată, identificatorul acesteia (stabilit prin comanda *CREATE PROCEDURE*) devine un nume de obiect în dicţionarul datelor. Tipul obiectului este *PROCEDURE*.

Sintaxa generală pentru crearea unei proceduri este următoarea:

[***CREATE*** [***OR REPLACE***]] ***PROCEDURE*** *nume_procedură*
[(*parametru* [*, parametru ...*])]
[***AUTHID*** {***DEFINER*** | ***CURRENT_USER***}]
{***IS*** | ***AS***}
[***PRAGMA AUTONOMOUS_TRANSACTION***;]
[*declaraţii locale*]
BEGIN
partea executabilă
[***EXCEPTION***
partea de tratare a excepţiilor]
END [*nume_procedură*];

Parametrii au următoarea formă sintactică:

nume_parametru [{***IN*** | ***OUT*** [***NOCOPY***] | ***IN OUT*** [***NOCOPY***] }]
tip_de_date [{:= | ***DEFAULT***} *expresie*]

Comanda *CREATE* permite ca procedura să fie stocată în baza de date. Când procedurile sunt create folosind clauza *CREATE OR REPLACE*, ele vor fi stocate în baza de date în formă compilată, formă care permite execuţia mai rapidă a acestora. Dacă procedura există, atunci clauza *OR REPLACE* va avea ca efect ştergerea procedurii şi înlocuirea acesteia cu noua versiune. Dacă procedura există, iar opţiunea *OR REPLACE* nu este prezentă, atunci comanda *CREATE* va returna eroarea „*ORA-00955*: *Name is already used by an existing object*“.

Clauza *AUTHID* specifică faptul că procedura stocată se execută cu drepturile proprietarului (implicit) sau ale utilizatorului curent. De asemenea, această clauză precizează dacă referinţele la obiecte sunt rezolvate în schema proprietarului procedurii sau în cea a utilizatorului curent.

Clauza *PRAGMA_AUTONOMOUS_TRANSACTION* anunţă compilatorul *PL/SQL* că această procedură este autonomă (independentă). Tranzacţiile autonome permit suspendarea tranzacţiei principale, executarea unor instrucţiuni *SQL*, permanentizarea sau anularea acestor operaţii şi continuarea tranzacţiei principale.

Parametrii formali (variabile declarate în lista parametrilor specificaţiei subprogramului) pot să fie de tipul *%TYPE*, *%ROWTYPE* sau de un tip explicit, fără specificarea dimensiunii.

***Exemplu*:**

Să se creeze o procedură stocată care micşorează cu o valoare dată (*cant*) poliţele de asigurare emise de firma *ASIROM*.

```
CREATE OR REPLACE PROCEDURE mic (cant IN NUMBER) AS
BEGIN
  UPDATE  politaasig
  SET     valoare = valoare - cant
  WHERE   firma = 'ASIROM';
END;
/
```

Dacă în subprograme se execută operaţii de reactualizare şi există declanşatori relativ la aceste operaţii care nu trebuie să se execute, atunci înainte de apelarea subprogramului declanşatorii trebuie dezactivaţi, urmând ca ei să fie reactivaţi după ce s-a terminat execuţia subprogramului.

De exemplu, în problema prezentată anterior ar trebui dezactivaţi declanşatorii referitori la tabelul *politaasig*, apelată procedura *mic* şi în final, reactivaţi aceşti declanşatori.

```
ALTER TABLE politaasig DISABLE ALL TRIGGERS;
EXECUTE mic(10000)
```

```
ALTER TABLE politaasig ENABLE ALL TRIGGERS;
```

***Exemplu*:**

Să se creeze o procedură locală prin care se inserează informaţii în tabelul *editata_de*.

```
DECLARE
  PROCEDURE editare
             (v_cod_sursa  editata_de.cod_sursa%TYPE,
              v_cod_autor   editata_de.cod_autor%TYPE)
IS
  BEGIN
    INSERT  INTO editata_de
    VALUES  (v_cod_sursa,v_cod_autor);
  END;
BEGIN
  …
  editare(75643, 13579);
  …
END;
/
```

Procedurile stocate pot fi apelate:

- din corpul altei proceduri sau al unui declanşator;
- interactiv, de către utilizator, folosind un instrument *Oracle* (de exemplu, *iSQL*Plus*);
- explicit dintr-o aplicaţie (de exemplu, *Oracle Forms* sau prin utilizarea de precompilatoare).

Apelarea unei proceduri se poate face în funcţie de mediul care o solicită:

1) în *SQL*Plus*, prin comanda

EXECUTE *nume_procedură* [(*lista_parametri_actuali*)];

2) în *PL/SQL*, prin invocarea numelui procedurii urmat de lista parametrilor actuali.

Parametrii actuali sunt variabile sau expresii referite în lista parametrilor subprogramului apelant. Ei trebuie să fie compatibili ca tip şi număr cu parametrii formali.

Funcţii *PL/SQL*

O funcţie *PL/SQL* este similară unei proceduri cu excepţia că ea trebuie să întoarcă un rezultat. O funcţie fără comanda *RETURN* va genera o eroare la compilare.

Când funcţia este compilată, identificatorul acesteia devine obiect în dicţionarul datelor având tipul *FUNCTION*. Algoritmul din interiorul corpului subprogramului funcţie trebuie să asigure faptul că toate traiectoriile sale conduc la comanda *RETURN*. Dacă o traiectorie a algoritmului trimite în partea de tratare a erorilor, atunci *handler*-ul acesteia trebuie să includă o comandă *RETURN*. Orice funcţie trebuie să conţină clauza *RETURN* în antet şi cel puţin o comandă *RETURN* în partea executabilă.

Sintaxa simplificată pentru scrierea unei funcţii este următoarea:

**[*CREATE* [*OR REPLACE*]] *FUNCTION* *nume_funcţie*
[(*parametru* [*, parametru ...*])]
RETURN *tip_de_date*
[*AUTHID* {*DEFINER* | *CURRENT_USER*}]
[*DETERMINISTIC*]
{*IS* | *AS*}
[*PRAGMA AUTONOMOUS_TRANSACTION*;]
[*declaraţii locale*]
BEGIN
partea executabilă
[*EXCEPTION*
partea de tratare a excepţiilor]
END [*nume_funcţie*];

Opţiunea *tip_de_date* specifică tipul valorii returnate de funcţie, tip care nu poate conţine specificaţii de dimensiune. Dacă totuşi sunt necesare aceste specificaţii, se pot defini subtipuri, iar parametrii şi valoarea returnată vor fi declaraţi de acel subtip.

În interiorul funcţiei trebuie să apară instrucţiunea *RETURN expresie*, unde *expresie* este valoarea rezultatului furnizat de funcţie. Pot să fie mai multe comenzi *RETURN* într-o funcţie, dar numai una din ele va fi executată. Comanda *RETURN* (fără o expresie asociată) poate să apară şi într-o procedură. În acest caz, ea va avea ca efect saltul la comanda ce urmează instrucţiunii apelante.

Opţiunea *DETERMINISTIC* ajută optimizorul *Oracle* în cazul unor apeluri repetate ale aceleaşi funcţii, cu aceleaşi argumente. Ea indică posibilitatea folosirii unui rezultat obţinut anterior.

***Observaţii*:**

- În blocul *PL/SQL* al unei funcţii stocate (cel care defineşte acţiunea

efectuată de funcţie) nu pot fi referite variabile *host* sau variabile *bind*.

- O funcţie poate accepta unul sau mai mulţi parametri, dar trebuie să returneze o singură valoare. Ca şi în cazul procedurilor, lista parametrilor este opţională. Dacă subprogramul nu are parametri, parantezele nu sunt necesare la declarare şi la apelare.
- O procedură care conţine un parametru de tip *OUT* poate fi rescrisă sub forma unei funcţii.

***Exemplu*:**

Să se creeze o funcţie stocată care determină numărul fotografiilor realizate pe hartie lucioasă, ce au fost achiziţionate la o anumită dată.

```
CREATE OR REPLACE FUNCTION  numar_fotografii
          (v_a  IN  fotografie.data_achizitie%TYPE)
  RETURN NUMBER AS
  alfa  NUMBER;
BEGIN
  SELECT  COUNT(ROWID)
  INTO    alfa
  FROM    fotografie
  WHERE   hartie = 'lucioasa'
  AND     data_achizitie = v_a;
  RETURN alfa;
END numar_fotografii;
/
```

Dacă apare o eroare de compilare, utilizatorul o va corecta în fişierul editat şi apoi va trimite compilatorului (cu opţiunea *OR REPLACE*) fişierul modificat.

Sintaxa pentru apelul unei funcţii este:

[[*schema*.]*nume_pachet.*]*nume_funcţie* [**@***dblink*] [(*lista_parametri_actuali*)];

O funcţie stocată poate fi apelată în mai multe moduri. În continuare sunt prezentate trei exemple de apelare.

1) Apelarea funcţiei şi atribuirea valorii acesteia într-o variabilă de legătură *SQL*Plus*:

```
VARIABLE val NUMBER
EXECUTE :val := numar_fotografii(SYSDATE)
PRINT val
```

Când este utilizată declaraţia *VARIABLE* pentru variabilele *host* de tip *NUMBER* nu trebuie specificată dimensiunea, iar pentru cele de tip *CHAR* sau

VARCHAR2 valoarea implicită este 1 sau poate fi specificată o altă valoare între paranteze. *PRINT* şi *VARIABLE* sunt comenzi *SQL*Plus.*

2) Apelarea funcţiei într-o instrucţiune *SQL*:

```
SELECT numar_fotografii(SYSDATE)
FROM   dual;
```

3) Apariţia numelui funcţiei într-o comandă din interiorul unui bloc *PL/SQL* (de exemplu, într-o instrucţiune de atribuire):

```
SET SERVEROUTPUT ON
ACCEPT data PROMPT 'dati data achizitionare'
DECLARE
  num     NUMBER;
  v_data  fotografie.data_achizitie%TYPE := '&data';
BEGIN
  num := numar_fotografii(v_data);
  DBMS_OUTPUT.PUT_LINE('numarul fotografiilor
    achizitionate la data ' || TO_CHAR(v_data) || '
este ' || TO_CHAR(num));
END;
/
SET SERVEROUTPUT OFF
```

***Exemplu*:**

Să se creeze o procedură stocată care pentru un anumit gen de fotografie (dat ca parametru) calculează numărul fotografiilor expuse din genul respectiv, numărul de artişti care au creat aceste fotografii, numărul de expoziţii în care au fost expuse, precum şi valoarea nominală totală a acestora.

```
CREATE OR REPLACE PROCEDURE date_gen_fotografie
                          (v_gen fotografie.tip%TYPE) AS
  FUNCTION nr_fotografii (v_gen fotografie.tip%TYPE)
  RETURN NUMBER IS
    v_numar  NUMBER(3);
  BEGIN
    SELECT COUNT(*)
    INTO   v_numar
    FROM   fotografie
    WHERE  gen = v_gen;
    RETURN v_numar;
  END nr_fotografii;
  FUNCTION valoare_totala
```

9

```
            (v_gen fotografie.gen%TYPE)
  RETURN NUMBER IS
    v_numar  fotografie.valoare%TYPE;
  BEGIN
    SELECT SUM(valoare)
    INTO   v_numar
    FROM   fotografie
    WHERE  gen = v_gen;
    RETURN v_numar;
  END valoare_totala;
  FUNCTION nr_artisti (v_gen fotografie.gen%TYPE)
  RETURN NUMBER IS
    v_numar  NUMBER(3);
  BEGIN
    SELECT COUNT(DISTINCT cod_artist)
    INTO   v_numar
    FROM   fotografie
    WHERE  gen = v_gen;
    RETURN v_numar;
  END nr_artisti;
  FUNCTION nr_expozitii (v_gen fotografie.gen%TYPE)
  RETURN NUMBER IS
    v_numar  NUMBER(3);
  BEGIN
    SELECT COUNT(DISTINCT s.cod_expozitie)
    INTO   v_numar
    FROM   fotografie f, sala s
    WHERE  f.cod_sala = s.cod_sala
    AND    gen = v_gen;
    RETURN v_numar;
  END nr_expozitii;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Numarul   fotografiilor   este
'|| nr_fotografii(v_gen));
  DBMS_OUTPUT.PUT_LINE('Valoarea   fotografiilor   este
'|| valoare_totala(v_gen));
  DBMS_OUTPUT.PUT_LINE('Numarul de artisti este
'|| nr_artisti(v_gen));
  DBMS_OUTPUT.PUT_LINE('Numarul de expozitii este
'|| nr_expozitii(v_gen);
```

```
END date_gen_fotografie;
```

Instrucţiunea *CALL*

O instrucţiune specifică pentru *Oracle9i* este *CALL*, care permite apelarea subprogramelor *PL/SQL* stocate (independente sau incluse în pachete) şi a metodelor *Java*.

CALL este o comandă *SQL* care nu poate să apară de sine stătătoare într-un bloc *PL/SQL*. Ea poate fi utilizată în *PL/SQL* doar dinamic, prin intermediul comenzii *EXECUTE IMMEDIATE*. Pentru executarea acestei comenzi, utilizatorul trebuie să aibă privilegiul *EXECUTE* asupra subprogramului. Instrucţiunea poate fi executată interactiv din *SQL*Plus*.

Comanda *CALL* are sintaxa următoare:

CALL [*schema.*] [{*nume_tip_obiect* | *nume_pachet*}.] *nume_subprogram*
[(*lista_parametri actuali*)] [***@****dblink_nume*] [***INTO*** :*variabila_host*]

Identificatorul *nume_subprogram* este numele unui subprogram sau al unei metode apelate. Clauza *INTO* este folosită numai pentru variabilele de ieşire ale unei funcţii. Dacă lipseşte clauza *@dblink_nume*, atunci apelul se referă la baza de date locală, iar într-un sistem distribuit clauza specifică numele bazei de date ce conţine subprogramul.

Exemplu:

Sunt prezentate două exemple prin care o funcţie *PL/SQL* este apelată din *SQL*Plus*, respectiv o procedură externă *C* este apelată, folosind *SQL* dinamic, dintr-un bloc *PL/SQL*.

```
CREATE OR REPLACE FUNCTION apelfunctie(a IN VARCHAR2)
  RETURN VARCHAR2 AS
BEGIN
  DBMS_OUTPUT.PUT_LINE ('Apel functie cu ' || a);
  RETURN a;
END apelfunctie;
/
```

```
SQL> --apel valid
SQL> VARIABLE v_iesire VARCHAR2(20)
SQL> CALL apelfunctie('Salut!') INTO :v_iesire
Apel functie cu Salut!
Call completed
```

```
SQL> PRINT v_iesire
v_iesire
Salut!
```

11

```
DECLARE
  a  NUMBER(7);
  x  VARCHAR2(10);
BEGIN
  EXECUTE IMMEDIATE 'CALL alfa_extern_procedura (:aa,
:xx)' USING a, x;
END;
/
```

Modificarea şi suprimarea subprogramelor *PL/SQL*

Pentru a lua în considerare modificarea unei proceduri sau funcţii, recompilarea acesteia se face prin comanda:

ALTER {***FUNCTION*** | ***PROCEDURE***} [*schema.*]*nume* ***COMPILE***;

Ca şi în cazul tabelelor, funcţiile şi procedurile pot fi suprimate cu ajutorul comenzii *DROP*. Aceasta presupune eliminarea subprogramelor din dicţionarul datelor. *DROP* este o comandă ce aparţine limbajului de definire a datelor, astfel că se execută un *COMMIT* implicit atât înainte, cât şi după comandă.

Atunci când este şters un subprogram prin comanda *DROP*, automat sunt revocate toate privilegiile acordate referitor la acest subprogram. Dacă este utilizată sintaxa *CREATE OR REPLACE*, privilegiile acordate acestui subprogram rămân aceleaşi.

Comanda *DROP* are următoarea sintaxă:

DROP {***FUNCTION*** | ***PROCEDURE***} [*schema.*]*nume*;

Transferarea valorilor prin parametri

Lista parametrilor unui subprogram este compusă din parametri de intrare (*IN*), de ieşire (*OUT*) sau de intrare/ieşire (*IN OUT*), separaţi prin virgulă.

Dacă nu este specificat tipul parametrului, atunci implicit acesta este considerat de intrare (*IN*). Un parametru formal cu opţiunea *IN* poate primi valori implicite chiar în cadrul comenzii de declarare. Acest parametru este *read-only* şi deci, nu poate fi schimbat în corpul subprogramului. El acţionează ca o constantă. Parametrul actual corespunzător poate fi literal, expresie, constantă sau variabilă iniţializată.

Un parametru formal cu opţiunea *OUT* este neiniţializat şi prin urmare, are automat valoarea *null*. În interiorul subprogramului, parametrilor cu opţiunea *OUT* sau *IN OUT* trebuie să li se asigneze o valoare explicită. Dacă nu se atribuie nici o valoare, atunci parametrul actual corespunzător va avea valoarea

null. Parametrul actual trebuie să fie o variabilă, nu poate fi o constantă sau o expresie.

Dacă în timpul execuţiei procedurii apare o excepţie, atunci valorile parametrilor formali cu opţiunile *IN OUT* sau *OUT* nu sunt copiate în valorile parametrilor actuali.

Implicit, transmiterea parametrilor se face prin referinţă în cazul parametrilor *IN* şi prin valoare în cazul parametrilor *OUT* sau *IN OUT*. Dacă pentru realizarea unor performanţe se doreşte transmiterea prin referinţă şi a parametrilor *IN OUT* sau *OUT*, atunci se poate utiliza opţiunea *NOCOPY*. Dacă opţiunea *NOCOPY* este asociată unui parametru *IN*, atunci se va genera o eroare la compilare, deoarece aceşti parametri se transmit de fiecare dată prin referinţă.

Opţiunea *NOCOPY* va fi ignorată, iar parametrul va fi transmis prin valoare dacă:

- parametrul actual este o componentă a unui tablou indexat (restricţia nu se aplică dacă parametrul este întreg tabelul);
- parametrul actual este constrâns prin specificarea unei precizii, a unei mărimi sau prin opţiunea *NOT NULL*;
- parametrul formal şi cel actual asociat sunt înregistrări care fie au fost declarate implicit ca variabile contor într-un ciclu *LOOP*, fie au fost declarate explicit prin *%ROWTYPE*, dar constrângerile pe câmpurile corespunzătoare diferă;
- transmiterea parametrului actual cere o conversie implicită a tipului;
- subprogramul este o parte a unui apel de tip *RPC* (*remote procedure call*), iar în acest caz, parametrii fiind transmişi în reţea, transferul nu se poate face prin referinţă.

Atunci când este apelată o procedură *PL/SQL*, sistemul *Oracle* furnizează două metode pentru definirea parametrilor actuali: specificarea explicită prin nume şi specificarea prin poziţie.

***Exemplu*:**

Sunt prezentate diferite moduri pentru apelarea procedurii *p1*.

```
CREATE PROCEDURE p1(a IN NUMBER, b IN VARCHAR2,
                    c IN DATE, d OUT NUMBER) AS …;
DECLARE
  var_a  NUMBER;
  var_b  VARCHAR2;
  var_c  DATE;
  var_d  NUMBER;
BEGIN
```

```
  --specificare prin poziţie
  p1(var_a,var_b,var_c,var_d);
  --specificare prin nume
  p1(b=>var_b,c=>var_c,d=>var_d,a=>var_a);
  --specificare prin nume şi poziţie
  p1(var_a,var_b,d=>var_d,c=>var_c);
END;
```

Dacă este utilizată specificaţia prin poziţie, parametrii care au primit o valoare implicită trebuie să fie plasaţi la sfârşitul listei parametrilor actuali.

***Exemplu*:**

Fie *proces_data* o procedură care procesează în mod normal data zilei curente, dar care opţional poate procesa şi alte date. Dacă nu se specifică parametrul actual corespunzător parametrului formal *plan_data,* atunci acesta va lua automat valoarea dată implicit (data curentă a sistemului).

```
PROCEDURE proces_data(data_in    IN NUMBER,
                      plan_data   IN DATE := SYSDATE)
IS …
```

Următoarele comenzi reprezintă apeluri corecte ale procedurii *proces_data*:

```
proces_data(10);
proces_data(10,SYSDATE+1);
proces_data(plan_data=>SYSDATE+1,data_in=>10);
```

O declaraţie de subprogram (procedură sau funcţie) fără parametri este specificată fără paranteze. De exemplu, dacă procedura *react_calc_dur* şi funcţia *obt_date* nu au parametri, atunci:

```
react_calc_dur;       -- apel corect
react_calc_dur();     -- apel incorect
data_mea := obt_date; -- apel corect
```

Module *overload*

Două sau mai multe module pot să aibă aceleaşi nume, dar să difere prin lista parametrilor. Aceste module sunt numite module *overload* (supraîncărcate). Funcţia *TO_CHAR* este un exemplu de modul *overload*. Există o singură funcţie, *TO_CHAR*, pentru a converti date numerice şi calendaristice în date de tip caracter.

În cazul unui apel, compilatorul compară parametrii actuali cu listele parametrilor formali pentru modulele *overload* şi execută modulul corespunzător. Toate programele *overload* trebuie să fie definite în acelaşi bloc

PL/SQL (bloc anonim, modul sau pachet).

Modulele *overload* pot să apară în programele *PL/SQL* fie în secţiunea declarativă a unui bloc, fie în interiorul unui pachet. Supraîncărcarea subprogramelor nu se poate face pentru funcţii sau proceduri stocate, dar este permisă pentru subprograme locale, subprograme care apar în pachete sau pentru metode.

Observaţii:

- Două subprograme *overload* trebuie să difere, cel puţin prin tipul unuia dintre parametri. Două subprograme nu pot fi *overload* dacă parametrii lor formali diferă numai prin tipurile lor şi dacă acestea sunt nişte subtipuri care se bazează pe acelaşi tip de date.
- Nu este suficient ca lista parametrilor subprogramelor *overload* să difere numai prin numele parametrilor formali.
- Nu este suficient ca lista parametrilor subprogramelor *overload* să difere numai prin tipul acestora (*IN*, *OUT*, *IN OUT*). *PL/SQL* nu poate face diferenţa (la apelare) între tipurile *IN* şi *OUT*.
- Nu este suficient ca funcţiile *overload* să difere doar prin tipul de date returnat (tipul de date specificat în clauza *RETURN* a funcţiei).
-

Exemplu:

Următoarele subprograme nu pot fi *overload*.

1)
```
FUNCTION alfa(par IN POSITIVE)…;
FUNCTION alfa(par IN BINARY_INTEGER)…;
```
2)
```
FUNCTION alfa(par IN NUMBER)…;
FUNCTION alfa(parar IN NUMBER)…;
```
3)
```
PROCEDURE beta(par IN VARCHAR2) IS…;
PROCEDURE beta(par OUT VARCHAR2) IS…;
```

***Exemplu*:**

Să se creeze două funcţii (locale) cu acelaşi nume care să calculeze media valorilor fotografiilor dintr-un anumit gen. Prima funcţie va avea un argument reprezentând genul fotografiilor, iar cea de-a doua va avea două argumente, unul reprezentând genul fotografiilor, iar celălalt reprezentând artistul pentru care se calculează valoarea medie (funcţia va calcula media valorilor fotografiilor dintr-un anumit gen şi care sunt create de un artist).

```
DECLARE
  medie1 NUMBER(10,2);
  medie2 NUMBER(10,2);
  FUNCTION valoare_medie (v_gen fotografie.tip%TYPE)
```

```
      RETURN NUMBER IS
    medie NUMBER(10,2);
  BEGIN
    SELECT AVG(valoare)
    INTO   medie
    FROM   fotografie
    WHERE  gen = v_gen;
    RETURN medie;
  END;
  FUNCTION valoare_medie
     (v_gen  fotografie.gen%TYPE,
      v_artist  fotografie.cod_artist%TYPE)
       RETURN NUMBER IS
    medie NUMBER(10,2);
  BEGIN
    SELECT AVG(valoare)
    INTO   medie
    FROM   fotografie
    WHERE  gen = v_gen AND cod_artist = v_artist;
    RETURN medie;
  END;
BEGIN
  medie1 := valoare_medie('natura');
  DBMS_OUTPUT.PUT_LINE('Media valorilor fotografiilor
din natura este ' || medie1);
  medie2 := valoare_medie('natura ', 1);
  DBMS_OUTPUT.PUT_LINE('Media valorilor fotografiilor
din natura create de artistul 1 este ' || medie2);
END;
```

Procedură *versus* funcţie

După cum am mai subliniat, în general, procedura este utilizată pentru realizarea unor acţiuni, iar funcţia este folosită pentru calculul unei valori.

Pot fi marcate câteva deosebiri esenţiale între funcţii şi proceduri.

- Procedura se execută ca o comandă *PL/SQL*, iar funcţia se invocă în cadrul unei expresii.
- Procedura poate returna (sau nu) una sau mai multe valori, iar funcţia trebuie să returneze o singură valoare.
- Procedura nu trebuie să conţină clauza *RETURN expresie*, iar funcţia

trebuie să conţină această opţiune.

De asemenea, pot fi remarcate câteva elemente esenţiale, comune atât funcţiilor cât şi procedurilor. Ambele pot:

- accepta valori implicite;
- conţine secţiuni declarative, executabile şi de tratare a erorilor;
- utiliza specificarea prin nume sau poziţie a parametrilor;
- accepta parametri *NOCOPY*.

Tratarea excepţiilor

Dacă apare o eroare într-un subprogram, atunci este declanşată o excepţie. Dacă excepţia este tratată în subprogram, atunci blocul se termină şi controlul trece la secţiunea de tratare a erorilor din subprogramul respectiv. Dacă nu există o tratare a excepţiei în subprogram, atunci controlul trece la programul apelant de nivel imediat superior (în partea de tratare a erorilor), respectând regulile de propagare a excepţiilor. În acest caz, valorile parametrilor de tip *OUT* sau *IN OUT* nu sunt returnate parametrilor actuali. Aceştia vor avea aceleaşi valori, ca şi cum subprogramul nu ar fi fost apelat.

Recursivitate

Recursivitatea este o tehnică importantă pentru simplificarea modelării algoritmilor. Un subprogram recursiv presupune că acesta se apelează pe el însuşi.

În *Oracle*, o problemă delicată este legată de locul unde se plasează un apel recursiv. De exemplu, dacă apelul este în interiorul unei comenzi *FOR* specifice cursoarelor sau între comenzile *OPEN* şi *CLOSE*, atunci la fiecare apel este deschis alt cursor. În felul acesta, programul poate depăşi limita admisă de cursoare deschise la un moment dat (*OPEN_CURSORS*), setată în parametrul de iniţializare *Oracle*.

***Exemplu*:**

Să se calculeze recursiv al *m*-lea termen din şirul lui Fibonacci.

```
CREATE OR REPLACE FUNCTION fibo(m POSITIVE) RETURN
INTEGER AS
BEGIN
  IF (m = 1) OR (m = 2) THEN
    RETURN 1;
  ELSE
    RETURN fibo(m-1) + fibo(m-2);
  END IF;
END fibo;
```

***Exemplu*:**

Să se calculeze iterativ al *m*-lea termen din şirul lui Fibonacci.

```
CREATE OR REPLACE FUNCTION fibo(m POSITIVE) RETURN
INTEGER AS
  ter1  INTEGER := 1;
  ter2  INTEGER := 0;
  valoare  INTEGER;
BEGIN
  IF (m = 1) OR (m = 2) THEN
    RETURN 1;
  ELSE
    valoare := ter1 + ter2;
    FOR i IN 3..m  LOOP
      ter2 := ter1;
      ter1 := valoare;
      valoare := ter1 + ter2;
    END LOOP;
    RETURN valoare;
  END IF;
END fibo;
```

Declaraţii *forward*

Subprogramele se numesc reciproc recursive dacă se apelează unul pe altul, în mod direct sau indirect. Declaraţiile *forward* permit definirea subprogramelor reciproc recursive.

În *PL/SQL*, un identificator trebuie declarat înainte de a fi folosit. De asemenea, un subprogram trebuie declarat înainte de a fi apelat.

***Exemplu*:**

```
PROCEDURE alfa ( ... ) IS
BEGIN
  beta( ... );         -- apel incorect
  …
END;
PROCEDURE beta ( ... ) IS
BEGIN
  …
END;
```

În acest exemplu, procedura *beta* nu poate fi apelată deoarece nu este încă declarată. Problema se poate rezolva simplu în acest caz, inversând ordinea celor două proceduri. Această soluţie nu este eficientă întotdeauna (de exemplu, dacă

şi procedura *beta* conţine un apel al procedurii *alfa*).

PL/SQL permite un tip special, numit *forward*, de declarare a unui subprogram. El constă dintr-o specificare a antetului unui subprogram, terminată prin caracterul „;“. O declaraţie de tip *forward* pentru procedura *beta* are forma:

```
PROCEDURE beta ( ... );   -- declaraţie forward
  …
PROCEDURE alfa ( ... ) IS
BEGIN
  beta( ... );
  …
END;
PROCEDURE beta ( ... ) IS
BEGIN
  …
END;
```

Declaraţiile *forward* pot fi folosite pentru a defini subprograme într-o anumită ordine logică, pentru a defini subprograme reciproc recursive sau pentru a grupa subprograme într-un pachet.

Lista parametrilor formali din declaraţia *forward* trebuie să fie identică cu cea corespunzătoare corpului subprogramului. Corpul subprogramului poate apărea oriunde după declaraţia sa *forward*, dar să rămână în aceeaşi unitate de program.

Utilizarea în expresii *SQL* a funcţiilor definite de utilizator

O funcţie stocată poate fi referită într-o comandă *SQL* la fel ca orice funcţie standard furnizată de sistem (*built-in function*), dar cu anumite restricţii.

Funcţiile *PL/SQL* definite de utilizator pot fi apelate din orice expresie *SQL* în care se pot folosi funcţii *SQL* standard.

Funcţiile *PL/SQL* pot să apară în:

- lista de câmpuri a comenzii *SELECT*;
- condiţia clauzelor *WHERE* şi *HAVING*;
- clauzele *CONNECT BY*, *START WITH*, *ORDER BY* şi *GROUP BY*;
- clauza *VALUES* a comenzii *INSERT*;
- clauza *SET* a comenzii *UPDATE*.

***Exemplu*:**

Să se afişeze fotografiile (titlu, valoare, stare) a căror valoare este mai mare decât valoarea medie a tuturor fotografiilor expuse.

```
CREATE OR REPLACE FUNCTION valoare_medie
  RETURN NUMBER AS
  v_val_mediu fotografie.valoare%TYPE;
```

```
BEGIN
  SELECT AVG(valoare)
  INTO   v_val_mediu
  FROM   fotografie;
  RETURN v_val_mediu;
END;
```

Referirea acestei funcţii într-o comandă *SQL* se poate face prin secvenţa:

```
SELECT titlu, valoare, stare
FROM   fotografie
WHERE  valoare >= valoare_medie;
```

Există restricţii referitoare la folosirea funcţiilor definite de utilizator într-o comandă *SQL*. Câteva dintre acestea, care s-au păstrat şi pentru *Oracle9i*, vor fi enumerate în continuare:

- funcţia definită de utilizator trebuie să fie o funcţie stocată (procedurile stocate nu pot fi apelate în expresii *SQL*), nu poate fi locală altui bloc;
- funcţia definită de utilizator trebuie să fie o funcţie linie şi nu una grup (restricţia dispare în *Oracle9i*);
- funcţia apelată dintr-o comandă *SELECT* sau din comenzi paralelizate *INSERT, UPDATE* şi *DELETE* nu poate modifica tabelele bazei de date;
- funcţia apelată dintr-o comandă *UPDATE* sau *DELETE* nu poate interoga sau modifica tabele ale bazei reactualizate chiar de aceste comenzi (*table mutating*);
- funcţia apelată din comenzile *SELECT*, *INSERT*, *UPDATE* sau *DELETE* nu poate conţine comenzi *LCD* (*COMMIT*), *ALTER SYSTEM*, *SET ROLE* sau comenzi *LDD* (*CREATE*);
- funcţia nu poate apărea în clauza *CHECK* a unei comenzi *CREATE/ALTER TABLE*;
- funcţia nu poate fi folosită pentru a specifica o valoare implicită pentru o coloană în cadrul unei comenzi *CREATE/ALTER TABLE*;
- funcţia poate fi utilizată într-o comandă *SQL* numai de către proprietarul funcţiei sau de utilizatorul care are privilegiul *EXECUTE* asupra acesteia;
- parametrii unei funcţii *PL/SQL* apelate dintr-o comandă *SQL* trebuie să fie specificaţi prin poziţie (specificarea prin nume nefiind permisă);
- funcţia definită de utilizator, apelabilă dintr-o comandă *SQL*, trebuie să aibă doar parametri de tip *IN*, cei de tip *OUT* şi *IN OUT* nefiind acceptaţi;
- parametrii formali ai unui subprogram funcţie trebuie să fie de tip

specific bazei de date (*NUMBER*, *CHAR*, *VARCHAR2*, *ROWID*, *LONG*, *LONGROW*, *DATE* etc.) şi nu de tipuri *PL/SQL* (*BOOLEAN*, *RECORD* etc.);

- tipul returnat de un subprogram funcţie trebuie să fie un tip intern pentru *server*, nu un tip *PL/SQL*;
- funcţia nu poate apela un subprogram care nu respectă restricţiile anterioare.

***Exemplu*:**

```
CREATE OR REPLACE FUNCTION calcul (p_val NUMBER)
  RETURN NUMBER IS
BEGIN
  INSERT    INTO    fotografie(cod_fotografie,    gen,
data_achizitie, valoare)
  VALUES (1358, 'arhival', SYSDATE, 700000);
  RETURN (p_val*7);
END;
/

UPDATE  fotografie
SET     valoare = calcul (550000)
WHERE   cod_fotografie = 7531;
```

Comanda *UPDATE* va returna o eroare deoarece tabelul *fotografie* este *mutating*. Operaţia de reactualizare este însă permisă asupra oricărui alt tabel diferit de *fotografie*.

Informaţii referitoare la subprograme

Informaţiile referitoare la subprogramele *PL/SQL* şi modul de acces la aceste informaţii sunt următoarele:

- codul sursă, utilizând vizualizarea *USER_SOURCE* din dicţionarul datelor;
- informaţii generale, utilizând vizualizarea *USER_OBJECTS* din dicţionarul datelor;
- tipul parametrilor (*IN*, *OUT*, *IN OUT*), utilizând comanda *DESCRIBE* din *SQL*Plus*;
- *p-code* (nu este accesibil utilizatorilor);
- erorile la compilare, utilizând vizualizarea *USER_ERRORS* din dicţionarul datelor sau comanda *SHOW ERRORS*;
- informaţii de depanare, utilizând pachetul *DBMS_OUTPUT*.

Atunci când este creat un subprogram stocat, informaţiile referitoare la subprogram sunt depuse în dicţionarul datelor.

Vizualizarea *USER_OBJECTS* conţine informaţii generale despre toate obiectele prelucrate în baza de date şi, în particular, despre subprogramele stocate.

Vizualizarea *USER_OBJECTS* are următoarele câmpuri:

- *OBJECT_NAME* – numele obiectului;
- *OBJECT_TYPE* – tipul obiectului (*PROCEDURE*, *FUNCTION* etc.);
- *OBJECT_ID* – identificator intern al obiectului;
- *CREATED* – data la care a fost creat obiectul;
- *LAST_DDL_TIME* – data ultimei modificări a obiectului;
- *TIMESTAMP* – data şi momentul ultimei recompilări;
- *STATUS* – starea de validitate a obiectului.

Pentru a verifica dacă recompilarea explicită (*ALTER*) sau implicită a avut succes se poate verifica starea subprogramelor utilizând coloana *STATUS* din vizualizarea *USER_OBJECTS*.

Orice obiect are o stare (*status*) sesizată în dicţionarul datelor, care poate fi:

- *VALID* (obiectul a fost compilat şi poate fi folosit atunci când este referit);
- *INVALID* (obiectul trebuie compilat înainte de a fi folosit).

***Exemplu*:**

Să se listeze în ordine alfabetică, procedurile şi funcţiile deţinute de utilizatorul curent, precum şi starea acestora.

```
SELECT    OBJECT_NAME, OBJECT_TYPE, STATUS
FROM      USER_OBJECTS
WHERE     OBJECT_TYPE IN ('PROCEDURE','FUNCTION')
ORDER BY  OBJECT_NAME;
```

După ce subprogramul a fost creat, codul sursă al acestuia poate fi obţinut consultând vizualizarea *USER_SOURCE* din dicţionarul datelor. Vizualizarea are următoarele câmpuri: *NAME* (numele obiectului), *TYPE* (tipul obiectului), *LINE* (numărul liniei din codul sursă), *TEXT* (textul liniilor codului sursă).

***Exemplu*:**

Să se afişeze codul complet pentru funcţia *numar_fotografii*.

```
SELECT    TEXT
FROM      USER_SOURCE
WHERE     NAME = 'NUMAR_FOTOGRAFII'
ORDER BY  LINE;
```

***Exemplu*:**

Să se scrie o procedură care recompilează toate obiectele invalide din schema personală.

```
CREATE OR REPLACE PROCEDURE recompileaza IS
  CURSOR obj_curs IS
     SELECT  OBJECT_TYPE, OBJECT_NAME
     FROM    USER_OBJECTS
     WHERE   STATUS = 'INVALID'
     AND     OBJECT_TYPE IN
               ('PROCEDURE', 'FUNCTION', PACKAGE',
                'PACKAGE BODY', 'VIEW');
BEGIN
  FOR obj_rec IN obj_curs LOOP
     DBMS_DDL.ALTER_COMPILE(obj_rec.OBJECT_TYPE,
                      USER, obj_rec.OBJECT_NAME);
  END LOOP;
END recompileaza;
```

Atunci când se recompilează un obiect *PL/SQL*, *server*-ul va recompila orice obiect invalid de care depinde acesta.

Dacă la recompilarea automată implicită a procedurilor locale dependente apar probleme, atunci starea obiectului va rămâne *INVALID* şi *server*-ul *Oracle* va semnala o eroare. Prin urmare:

- este preferabil ca recompilarea să fie manuală, explicită utilizând comanda *ALTER* (*PROCEDURE*, *FUNCTION*, *TRIGGER*, *PACKAGE*) cu opţiunea *COMPILE*;
- este necesar ca după o schimbare referitoare la obiectele bazei, recompilarea să se facă cât mai repede.

Vizualizarea *USER_ERRORS* afişează textul tuturor erorilor de compilare. Câmpurile acesteia (*NAME*, *TYPE*, *SEQUENCE*, *LINE*, *POSITION*, *TEXT*) sunt analizate în capitolul referitor la tratarea excepţiilor.

Pentru a obţine valori (de exemplu, valoarea contorului pentru un *LOOP*, valoarea unei variabile înainte şi după o atribuire etc.) şi mesaje (de exemplu, părăsirea unui subprogram, apariţia unei operaţii etc.) dintr-un bloc *PL/SQL* pot fi utilizate procedurile pachetului *DBMS_OUTPUT*. Aceste informaţii se cumulează într-un *buffer* care poate fi consultat ulterior.

Dependenţa subprogramelor

Atunci când este compilat un subprogram, toate obiectele *Oracle* care sunt referite vor fi înregistrate în dicţionarul datelor. Subprogramul este dependent de aceste obiecte. Un subprogram care are erori la compilare este

marcat ca *INVALID* în dicţionarul datelor. Un subprogram stocat poate deveni, de asemenea, invalid după execuţia unei operaţii *LDD* asupra unui obiect de care depinde.

Obiecte dependente	Obiecte referite
View	*Table*
Procedure	*View*
Function	*Procedure*
Package Specification	*Function*
Package Body	*Synonym*
Database Trigger	*Package Specification*

Modificarea definiţiei unui obiect referit poate să influenţeze (sau nu) funcţionarea normală a obiectului dependent.

Există două tipuri de dependenţe:

- dependenţă directă, în care obiectul dependent (*procedure* sau *function*) face referinţă direct la un obiect de tip *table*, *view*, *sequence*, *procedure*, *function*;
- dependenţă indirectă, în care obiectul dependent (*procedure* sau *function*) face referinţă indirect la un obiect de tip *table*, *view*, *sequence*, *procedure*, *function* prin intermediul unui *view*, *procedure* sau *function*.

În cazul dependenţelor locale, atunci când un obiect referit este modificat, obiectele dependente sunt invalidate. La următorul apel al obiectului invalidat, acesta va fi recompilat automat de către *server*-ul *Oracle*.

În cazul dependenţelor la distanţă, procedurile stocate local şi toate obiectele dependente vor fi invalidate. Ele nu vor fi recompilate automat la următorul apel.

***Exemplu*:**

Se presupune că procedura *filtru* va referi direct tabelul *fotografie* şi că procedura *adaug* va reactualiza tabelul *fotografie* prin intermediul unei vizualizări *nou_fotografie.* Pentru aflarea dependenţelor directe se poate utiliza vizualizarea *USER_DEPENDENCIES* din dicţionarul datelor.

```
SELECT NAME, TYPE, REFENCED_NAME, REFENCED_TYPE
FROM   USER_DEPENDENCIES
WHERE  REFENCED_NAME IN ('fotografie', 'nou_fotografie');
```

NAME	TYPE	REFENCED_NAME	REFENCED_TYPE
filtru	Procedure	fotografie	Table
adaug	Procedure	nou_fotografie	View
nou_fotografie	View	fotografie	Table

Dependenţele indirecte pot fi afişate utilizând vizualizările *DEPTREE* şi

IDEPTREE. Vizualizarea *DEPTREE* afişează o reprezentare a tuturor obiectelor dependente (direct sau indirect). Vizualizarea *IDEPTREE* afişează o reprezentare a aceleaşi informaţii, sub forma unui arbore.

Pentru a utiliza aceste vizualizări furnizate de sistemul *Oracle* trebuie:

- executat scriptul *UTLDTREE*;
- executată procedura *DEPTREE_FILL* (are trei argumente: tipul obiectului referit, schema obiectului referit, numele obiectului referit).

***Exemplu*:**

```
@UTLDTREE
EXECUTE DEPTREE_FILL ('TABLE', 'SCOTT', 'fotografie
')
SELECT NESTED_LEVEL, TYPE, NAME
FROM   DEPTREE
ORDER  BY SEQ#;
```

```
NESTED_LEVEL     TYPE         NAME
            0    Table        fotografie
            1    View         nou_fotografie
            2    Procedure    adaug
            1    Procedure     filtru

SELECT  *
FROM    IDEPTREE;

DEPENDENCIES
          TABLE  nume_schema.fotografie
          VIEW nume_schema.nou_fotografie
          PROCEDURE nume_schema.adaug
          PROCEDURE nume_schema.filtru
```

Dependenţele la distanţă sunt tratate printr-una dintre modalităţile alese de utilizator, modelul *timestamp* (implicit) sau modelul *signature.*

Fiecare unitate *PL/SQL* are un *timestamp* (etichetă de timp) care este setat atunci când unitatea este creată sau recompilată şi care este depus în câmpul *LAST_DDL_TIME* din dicţionarul datelor. Modelul *timestamp* realizează compararea momentelor ultimei modificări a celor două obiecte analizate. Dacă obiectul bazei are momentul ultimei modificări mai recent decât cel al obiectului dependent, atunci obiectul dependent va fi recompilat.

Modelul *signature* (semnătură) determină momentul la care obiectele bazei distante trebuie recompilate. Când este creat un subprogram, o *signature* este depusă în dicţionarul datelor, alături de *p-code*. Aceasta conţine: numele

construcţiei *PL/SQL* (*PROCEDURE*, *FUNCTION*, *PACKAGE*), tipurile parametrilor, ordinea parametrilor, numărul acestora şi modul de transmitere (*IN*, *OUT*, *IN OUT*). Dacă parametrii se schimbă, atunci evident *signature* se schimbă.

Pentru a folosi modelul *signature* este necesară setarea parametrului *REMOTE_DEPENDENCIES_MODE* la *SIGNATURE*. Aceasta se poate realiza prin:

1) comanda *ALTER SESSION*, care va afecta doar sesiunea curentă:

ALTER SESSION SET REMOTE_DEPENDENCIES_MODE = SIGNATURE;

2) comanda *ALTER SYSTEM*, care va afecta întreaga bază de date (toate sesiunile), dar trebuie avut privilegiul sistem pentru a utiliza această instrucţiune:

ALTER SYSTEM SET REMOTE_DEPENDENCIES_MODE = SIGNATURE;

3) adăugarea în fişierul de iniţializare a unei linii de forma:

REMOTE_DEPENDENCIES_MODE = SIGNATURE;

Recompilarea procedurilor şi a funcţiilor dependente este fără succes dacă:

- obiectul referit este suprimat (*DROP*) sau redenumit (*RENAME*);
- tipul coloanei referite este schimbat;
- o vizualizare referită este înlocuită printr-o vizualizare ce conţine alte coloane;
- lista parametrilor unei proceduri referite este modificată.

Recompilarea procedurilor şi funcţiilor dependente este cu succes dacă:

- tabelul referit are noi coloane;
- corpul *PL/SQL* al unei proceduri referite a fost modificat şi recompilat cu succes.

Erorile datorate dependenţelor pot fi minimizate:

- utilizând comenzi *SELECT* cu opţiunea „*“;
- incluzând lista coloanelor în comanda *INSERT*;
- declarând variabile cu atributul *%TYPE*;
- declarând înregistrări cu atributul *%ROWTYPE*;

Concluzii:

- Dacă procedura depinde de un obiect local, atunci se face recompilare automată la prima reexecuţie.
- Dacă procedura depinde de o procedură distantă, atunci se face recompilare automată, dar la a doua reexecuţie. Este preferabilă o recompilare manuală pentru prima reexecuţie sau implementarea unei

strategii de reinvocare a ei (a doua oară).

- Dacă procedura depinde de un obiect distant, dar care nu este procedură, atunci nu se face recompilare automată.

Rutine externe

PL/SQL a fost special conceput pentru *Oracle* şi este specializat pentru procesarea tranzacţiilor *SQL*. Totuşi, într-o aplicaţie complexă pot să apară cerinţe şi funcţionalităţi care sunt mai eficient de implementat în *C*, *Java* sau alt limbaj de programare. De exemplu, *Java* este un limbaj portabil cu un model de securitate bine definit, care lucrează excelent pentru aplicaţii *Internet*.

Dacă aplicaţia trebuie să efectueze anumite acţiuni care nu pot fi implementate optim utilizând *PL/SQL*, atunci este preferabil să fie utilizate alte limbaje care realizează performant acţiunile respective. În acest caz este necesară comunicarea între diferite module ale aplicaţiei care sunt scrise în limbaje diferite.

Până la versiunea *Oracle8*, modalitatea de comunicare între *PL/SQL* şi alte limbaje (de exemplu, limbajul *C*) a fost utilizarea pachetelor *DBMS_PIPE* şi/sau *DBMS_ALERT*.

Începând cu *Oracle8*, comunicarea este simplificată prin utilizarea rutinelor externe. O rutină externă este o procedură sau o funcţie scrisă într-un limbaj diferit de *PL/SQL*, dar apelabilă dintr-un program scris în *PL/SQL*. *PL/SQL* extinde funcţionalitatea *server*-ului *Oracle*, furnizând o interfaţă pentru apelarea rutinelor externe. Orice bloc *PL/SQL* executat pe *server* sau pe *client* poate apela o rutină externă. Singurul limbaj acceptat pentru rutine externe în *Oracle8* este limbajul *C*.

Pentru a marca apelarea unei rutine externe în programul *PL/SQL* este definit un punct de intrare (*wrapper*) care direcţionează spre codul extern (program *PL/SQL* → *wrapper* → cod extern). Pentru crearea unui *wrapper* este utilizată o clauză specială (*AS EXTERNAL*) în cadrul comenzii *CREATE OR REPLACE PROCEDURE*. De fapt, clauza conţine informaţii referitoare la numele bibliotecii în care se găseşte subprogramul extern (clauza *LIBRARY*), numele rutinei externe (clauza *NAME*) şi corespondenţa (*C <–> PL/SQL*) dintre tipurile de date (clauza *PARAMETERS*). În ultimele versiuni s-a renunţat la clauza *AS EXTERNAL*.

Rutinele externe (scrise în *C*) sunt compilate, apoi depuse într-o bibliotecă dinamică (*DLL – dynamic link library*) şi sunt încărcate doar atunci când este necesar. Dacă se invocă o rutină externă scrisă în *C*, trebuie setată conexiunea spre această rutină. Un proces numit *extproc* este declanşat automat de către *server*. La rândul său, procesul *extproc* va încărca biblioteca identificată prin clauza *LIBRARY* şi va apela rutina respectivă.

Oracle8i permite utilizarea de rutine externe scrise în *Java*. De asemenea, un *wrapper* poate include specificaţii de apelare, prin utilizarea clauzei *LANGUAGE*. De fapt, aceste specificaţii permit apelarea rutinelor externe scrise în orice limbaj. De exemplu, o procedură scrisă într-un limbaj diferit de *C* sau *Java* poate fi utilizată în *SQL* sau *PL/SQL* dacă procedura respectivă este apelabilă din *C*. În felul acesta, biblioteci standard scrise în alte limbaje de programare pot fi apelate din programe *PL/SQL*.

Procedura *PL/SQL* executată pe *server*-ul *Oracle* poate apela o rutină externă scrisă în limbajul *C*, depusă într-o bibliotecă partajată. Procedura *C* se execută într-un spaţiu adresă diferit de cel al *server*-ului *Oracle*, în timp ce unităţile *PL/SQL* şi metodele *Java* se execută în spaţiul adresă al *server*-ului. *JVM* (*Java Virtual Machine*) de pe *server* va executa metoda *Java* în mod direct, fără a fi necesar procesul *extproc*.

Maniera de încărcare depinde de limbajul în care este scrisă rutina.

- Pentru a apela rutine externe *C*, *server*-ul trebuie să cunoască poziţionarea bibliotecii dinamice *DLL*. Acest lucru este furnizat de *alias*-ul bibliotecii din clauza *AS LANGUAGE*.
- Pentru apelarea unei rutine externe *Java* se va încărca clasa *Java* în baza de date. Este necesară doar crearea unui *wrappe*r care direcţionează către codul extern. Spre deosebire de rutinele externe *C*, nu este necesară nici biblioteca şi nici setarea conexiunii spre rutina externă.

Clauza *LANGUAGE* din cadrul comenzii de creare a unui subprogram, specifică limbajul în care este scrisă rutina (procedură externă *C* sau metodă *Java*) şi are următoarea formă:

{***IS*** | ***AS***} ***LANGUAGE*** {***C*** | ***JAVA***}

Pentru o procedură *C* sunt date informaţii referitoare la numele acesteia (clauza *NAME*); *alias*-ul bibliotecii în care se găseşte (clauza *LIBRARY*); opţiuni referitoare la tipul, poziţia, lungimea, modul de transmitere (prin valoare sau prin referinţă) al parametrilor (clauza *PARAMETERS*); posibilitatea ca rutina externă să acceseze informaţii despre parametri, excepţii, alocarea memoriei utilizator (clauza *WITH CONTEXT*).

LIBRARY *nume_biblioteca* [***NAME*** *nume_proc_c*] [***WITH CONTEXT***]
[***PARAMETERS*** (*parametru_extern* [, *parametru_extern ...*])]

Pentru o metodă *Java*, în clauză trebuie specificată doar signatura metodei (lista tipurilor parametrilor în ordinea apariţiei).

***Exemplu*:**

```
CREATE OR REPLACE FUNCTION calc (x IN REAL) RETURN
NUMBER
  AS LANGUAGE C
       LIBRARY biblioteca
```

```
NAME "c_calc"
PARAMETERS (x BY REFERENCES);
```

O rutină externă nu este apelată direct, ci se apelează subprogramul *PL/SQL* care referă rutina externă. Apelarea poate să apară în: blocuri anonime, subprograme independente sau care aparţin unui pachet, metode ale unui tip obiect, declanşatori bază de date, comenzi *SQL* care apelează funcţii (în acest caz, trebuie utilizată clauza *PRAGMA RESTRICT_REFERENCES*).

De remarcat că o metodă *Java* poate fi apelată din orice bloc *PL/SQL*, subprogram sau pachet. *JDBC* (*Java Database Connectivity*), care reprezintă interfaţa *Java* standard pentru conectare la baze de date relaţionale, şi *SQLJ* permit apelarea de blocuri *PL/SQL* din programe *Java*. *SQLJ* face posibilă incorporarea operaţiilor *SQL* în codul *Java*. Standardul *SQLJ* acoperă doar operaţii *SQL* statice. *Oracle9i SQLJ* include extensii pentru a suporta direct *SQL* dinamic.

O altă modalitate de a încărca metode *Java* este folosirea interactivă în *SQL*Plus* a comenzii *CREATE JAVA instrucţiune*.

Funcţii tabel

O funcţie tabel (*table function*) returnează drept rezultat un set de linii (de obicei, sub forma unei colecţii). Această funcţie poate fi interogată direct printr-o comandă *SQL*, ca şi cum ar fi un tabel al bazei de date. În felul acesta, funcţia poate fi utilizată în clauza *FROM* a unei cereri.

O funcţie tabel conductă (*pipelined table function*) este similară unei funcţii tabel, dar returnează datele iterativ, pe măsură ce acestea sunt obţinute, nu toate deodată. Aceste funcţii sunt mai eficiente deoarece informaţia este returnată imediat cum este obţinută.

Conceptul de funcţie tabel conductă a fost introdus în versiunea *Oracle9i*. Utilizatorul poate să definească astfel de funcţii. De asemenea, este posibilă execuţia paralelă a funcţiilor tabel (evident şi a celor clasice). În acest caz, funcţia trebuie să conţină în declaraţie opţiunea *PARALLEL_ENABLE*.

Funcţia tabel conductă acceptă orice argument pe care îl poate accepta o funcţie obişnuită şi trebuie să returneze o colecţie (*nested table* sau *varray*). Ea este declarată specificând cuvântul cheie *PIPELINED* în comanda *CREATE OR REPLACE FUNCTION*. Funcţia tabel conductă trebuie să se termine printr-o comandă *RETURN* simplă, care nu întoarce nici o valoare.

Pentru a returna un element individual al colecţiei este folosită comanda *PIPE ROW*, care poate să apară numai în corpul unei funcţii tabel conductă, în caz contrar generându-se o eroare. Comanda poate fi omisă dacă funcţia tabel conductă nu returnează nici o linie.

După ce funcţia a fost creată, ea poate fi apelată dintr-o cerere *SQL*

utilizând operatorul *TABLE*. Cererile referitoare la astfel de funcţii pot să includă cursoare şi referinţe la cursoare, respectându-se semantica de la cursoarele clasice.

Funcţia tabel conductă nu poate să apară în comenzile *INSERT*, *UPDATE*, *DELETE*. Totuşi, pentru a realiza o reactualizare, poate fi creată o vizualizare relativă la funcţia tabel şi folosit un declanşator *INSTEAD OF*.

***Exemplu*:**

Să se obţină o instanţă a unui tabel ce conţine informaţii referitoare la denumirea zilelor săptămânii.

Problema este rezolvată în două variante. Prima reprezintă o soluţie clasică, iar a doua variantă implementează problema cu ajutorul unei funcţii tabel conductă.

Varianta 1:

```
CREATE TYPE t_linie AS OBJECT (
       idl NUMBER, sir VARCHAR2(20));
CREATE TYPE t_tabel AS TABLE OF t_linie;
CREATE OR REPLACE FUNCTION calc1 RETURN t_tabel
AS
  v_tabel  t_tabel;
BEGIN
  v_tabel := t_tabel (t_linie (1, 'luni'));
  FOR j IN 2..7 LOOP
    v_tabel.EXTEND;
    IF j = 2
     THEN v_tabel(j) := t_linie (2, 'marti');
     ELSIF j = 3
      THEN v_tabel(j) := t_linie (3, 'miercuri');
      ELSIF j = 4
       THEN v_tabel(j) := t_linie (4, 'joi');
       ELSIF j = 5
        THEN v_tabel(j) := t_linie (5, 'vineri');
        ELSIF j = 6
         THEN v_tabel(j) := t_linie (6, 'sambata');
         ELSIF j = 7
          THEN v_tabel(j) := t_linie (7, 'duminica');
    END IF;
  END LOOP;
  RETURN v_tabel;
END calc1;
```

Funcţia *calc1* poate fi invocată în clauza *FROM* a unei comenzi *SELECT*:

```
SELECT *
FROM   TABLE (CAST (calc1 AS t_tabel));
```

Varianta 2:

```
CREATE  OR  REPLACE  FUNCTION  calc2  RETURN  t_tabel
PIPELINED
AS
  v_linie  t_linie;
BEGIN
  FOR j IN 1..7 LOOP
    v_linie :=
      CASE j
        WHEN 1 THEN t_linie (1, 'luni')
        WHEN 2 THEN t_linie (2, 'marti')
        WHEN 3 THEN t_linie (3, 'miercuri')
        WHEN 4 THEN t_linie (4, 'joi')
        WHEN 5 THEN t_linie (5, 'vineri')
        WHEN 6 THEN t_linie (6, 'sambata')
        WHEN 7 THEN t_linie (7, 'duminica')
      END;
    PIPE ROW (v_linie);
  END LOOP;
  RETURN;
END calc2;
```

Se observă că tabelul este implicat doar în tipul rezultatului. Pentru apelarea funcţiei *calc2* este folosită sintaxa următoare:

```
SELECT *
FROM   TABLE (calc2);
```

Funcţiile tabel sunt folosite frecvent pentru conversii de tipuri de date. *Oracle9i* introduce posibilitatea de a crea o funcţie tabel care returnează un tip *PL/SQL* (definit într-un bloc). Funcţia tabel care furnizează (la nivel de pachet) drept rezultat un tip de date trebuie să fie de tip conductă. Pentru apelare este utilizată sintaxa simplificată (fără *CAST*).

***Exemplu*:**

```
CREATE OR REPLACE PACKAGE exemplu IS
  TYPE t_linie IS RECORD
(idl NUMBER, sir VARCHAR2(20));
  TYPE t_tabel IS TABLE OF t_linie;
END exemplu;
CREATE OR REPLACE FUNCTION calc3
RETURN exemplu.t_tabel
```

```
  PIPELINED AS
  v_linie  exemplu.t_linie;
BEGIN
  FOR j IN 1..7 LOOP
    CASE j
     WHEN 1 THEN v_linie.idl := 1;
v_linie.sir := 'luni';
     WHEN 2 THEN v_linie.idl := 2;
v_linie.sir := 'marti';
     WHEN 3 THEN v_linie.idl := 3;
v_linie.sir := 'miercuri';
     WHEN 4 THEN v_linie.idl := 4;
v_linie.sir := 'joi';
     WHEN 5 THEN v_linie.idl := 5;
v_linie.sir := 'vineri';
     WHEN 6 THEN v_linie.idl := 6;
v_linie.sir := 'sambata';
     WHEN 7 THEN v_linie.idl := 7;
v_linie.sir := 'duminica';
    END CASE;
    PIPE ROW (v_linie);
  END LOOP;
  RETURN;
END calc3;
```

Procesarea tranzacţiilor autonome

Tranzacţia este o unitate logică de lucru, adică o secvenţă de comenzi care trebuie să se execute ca un întreg pentru a menţine consistenţa bazei de date. În mod uzual, o tranzacţie poate să cuprindă mai multe blocuri, iar într-un bloc pot să fie mai multe tranzacţii.

O tranzacţie autonomă este o tranzacţie independentă lansată de altă tranzacţie, numită tranzacţie principală. Tranzacţia autonomă permite suspendarea tranzacţiei principale, executarea de comenzi *SQL*, permanentizarea (*commit*) şi anularea (*rollback*) acestor operaţii.

Odată începută, tranzacţia autonomă este independentă, în sensul că nu partajează blocări, resurse sau dependenţe cu tranzacţia principală. În felul acesta, o aplicaţie nu trebuie să cunoască operaţiile autonome ale unei proceduri, iar procedura nu trebuie să cunoască nimic despre tranzacţiile aplicaţiei. Tranzacţia autonomă are totuşi toate funcţionalităţile unei tranzacţii obişnuite (permite cereri paralele, procesări distribuite etc.).

Pentru definirea unei tranzacţii autonome se utilizează clauza *PRAGMA AUTONOMOUS_TRANSACTION* care informează compilatorul *PL/SQL* că trebuie să marcheze o rutină ca fiind autonomă. Prin rutină se înţelege: bloc anonim de cel mai înalt nivel (nu imbricat); procedură sau funcţie locală, independentă sau împachetată; metodă a unui tip obiect; declanşator bază de date.

Codul *PRAGMA AUTONOMOUS_TRANSACTION* se specifică în partea declarativă a rutinei.

Codul *PRAGMA AUTONOMOUS_TRANSACTION* marchează numai rutine individuale ca fiind independente. Nu pot fi marcate toate subprogramele unui pachet sau toate metodele unui tip obiect ca autonome. Prin urmare, clauza nu poate să apară în partea de specificaţie a unui pachet.

***Observaţii*:**

- Declanşatorii autonomi, spre deosebire de cei clasici pot conţine comenzi *LCD* (de exemplu, *COMMIT*, *ROLLBACK*).
- Excepţiile declanşate în cadrul tranzacţiilor autonome generează un *rollback* la nivel de tranzacţie şi nu la nivel de instrucţiune.
- Când se intră în secţiunea executabilă a unei tranzacţii autonome, tranzacţia principală este suspendată.

Cu toate că o tranzacţie autonomă este iniţiată de altă tranzacţie, ea nu este o tranzacţie imbricată deoarece:

- nu partajează resurse cu tranzacţia principală;
- nu depinde de tranzacţia principală (de exemplu, dacă tranzacţia principală este anulată, atunci tranzacţiile imbricate sunt de asemenea anulate, în timp ce tranzacţia autonomă nu este anulată);
- schimbările permanentizate din tranzacţii autonome sunt vizibile imediat altor tranzacţii, pe când cele din tranzacţii imbricate sunt vizibile doar după ce tranzacţia principală este permanentizată.