Controlul execuţiei unui bloc *PL/SQL*

PL/SQL este un limbaj cu structură de bloc, adică programele sunt compuse din blocuri care pot fi complet separate sau imbricate. Structura unui bloc poate fi obţinută combinând subprograme, pachete, blocuri imbricate. Blocurile pot fi folosite în utilitarele *Oracle*.

Pentru modularizarea unui program este necesară:

- gruparea logică a instrucţiunilor în blocuri;
- imbricarea de subblocuri în blocuri mai mari;
- descompunerea unei probleme complexe într-o mulţime de module logice şi implementarea acestora cu ajutorul blocurilor;
- plasarea în biblioteci a codului *PL/SQL* reutilizabil, de unde poate fi folosit de aplicaţii;
- depunerea codului într-un *server Oracle*, de unde este accesibil oricărei aplicaţii care interacţionează cu baza de date *Oracle*.

Un program *PL/SQL* poate cuprinde unul sau mai multe blocuri. Un bloc poate fi anonim sau neanonim.

Blocurile **anonime** sunt blocuri *PL/SQL* fără nume, care sunt construite dinamic şi sunt executate o singură dată. Acest tip de bloc nu are argumente şi nu returnează un rezultat. Ele sunt declarate într-un punct al aplicaţiei, unde vor fi executate (trimise motorului *PL/SQL*). În blocurile anonime pot fi declarate proceduri şi funcţii *PL/SQL*.

Blocurile anonime pot să apară într-un program ce lucrează cu precompilator sau în *SQL*Plus*. De obicei, blocurile anonime sunt plasate într-un fişier, iar apoi fişierul este executat din *SQL*Plus*. De asemenea, declanşatorii din componentele *Developer Suite* constau din astfel de blocuri.

Blocurile **neanonime** sunt fie blocuri cu nume (etichetate) construite static sau dinamic şi executate o singură dată, fie subprograme, pachete sau declanşatori.

Subprogramele sunt proceduri sau funcţii depuse în baza de date. Aceste blocuri sunt executate de mai multe ori şi, în general, nu mai sunt modificate după ce au fost construite. Procedurile şi funcţiile stocate sunt depuse pe *server*-ul *Oracle*, acceptă parametri şi pot fi apelate prin nume. Procedurile şi funcţiile aplicaţie sunt depuse într-o aplicaţie *Developer Suite* sau într-o bibliotecă.

Pachetele (stocate sau aplicaţie) sunt blocuri neanonime care grupează proceduri, funcţii, cursoare, tipuri, constante, variabile într-o unitate logică, în baza de date.

Declanşatorii sunt blocuri *PL/SQL* neanonime depuse în baza de date, care pot fi asociaţi bazei, iar în acest caz sunt executaţi implicit ori de câte ori apare un anumit eveniment declanşator (de exemplu, instrucţiuni *INSERT*, *UPDATE* sau *DELETE* ce se execută asupra unui tabel al bazei de date) sau pot fi asociaţi unei aplicaţii (de exemplu, declanşator *SQL*Forms*), ceea ce presupune că se execută automat, în funcţie de anumite condiţii sistem.

Structura unui bloc *PL/SQL*

Un bloc *PL/SQL* este compus din trei secţiuni distincte.

- Secţiunea declarativă (opţională) conţine declaraţii pentru toate variabilele, constantele, cursoarele şi erorile definite de utilizator la care se face referinţă în secţiunea executabilă sau chiar în cea declarativă. De asemenea, pot fi declarate subprograme locale care sunt vizibile doar în blocul respectiv.
- Secţiunea executabilă conţine instrucţiuni neprocedurale *SQL* pentru prelucrarea datelor din baza de date şi instrucţiuni *PL/SQL* pentru prelucrarea datelor în cadrul blocului.
- Secţiunea pentru tratarea erorilor (opţională) specifică acţiunile ce vor fi efectuate atunci când în execuţia blocului apar erori sau condiţii anormale.

Blocul *PL/SQL* are următoarea structură generală:

[<<*nume_bloc*>>]
[***DECLARE***
instrucţiuni de declarare]
BEGIN
instrucţiuni executabile (*SQL sau PL/SQL*)
[***EXCEPTION***
tratarea erorilor]
END [*nume_bloc*];

Dacă blocul *PL/SQL* este executat fără erori, invariant va apărea mesajul:

PL/SQL procedure successfully completed

Compatibilitate *SQL*

Din punct de vedere al compatibilităţii dintre *PL/SQL* şi *SQL*, se remarcă următoarele reguli de bază:

- *PL/SQL* furnizează toate comenzile *LMD* ale lui *SQL*, comanda *SELECT* cu clauza *INTO*, comenzile *LCD*, funcţiile, pseudocoloanele şi operatorii *SQL*;
- *PL/SQL* nu furnizează comenzile *LDD*.

Totuşi, în ultimele sale versiuni, *Oracle* permite folosirea dinamică a comenzilor *SQL*, utilizând tehnica oferită de *SQL* dinamic. În felul acesta, orice comandă *SQL* (inclusiv comandă *LDD*) poate să fie utilizată în *PL/SQL*.

Majoritatea funcţiilor *SQL* sunt disponibile în *PL/SQL*. Există însă funcţii specifice *PL/SQL*, cum sunt funcţiile *SQLCODE* şi *SQLERRM*. De asemenea, există funcţii *SQL* care nu sunt disponibile în instrucţiuni procedurale (*DECODE*, funcţiile grup), dar care sunt disponibile în instrucţiunile *SQL* dintr-un bloc *PL/SQL*. *SQL* nu poate folosi funcţii sau atribute specifice *PL/SQL*.

Funcţiile grup trebuie folosite cu atenţie, deoarece clauza *GROUP BY* nu are sens să apară în instrucţiunea *SELECT ... INTO*. *Oracle9i* introduce clauza *OVER*, care permite ca funcţia grup căreia îi este asociată să fie considerată o funcţie analitică (poate returna mai multe linii pentru fiecare grup).

Următoarele funcţii *SQL* nu sunt permise în *PL/SQL*: *WIDTH_BUCKET*, *BIN_TO_NUM*, *COMPOSE*, *DECOMPOSE*, *TO_LOB*, *DECODE*, *DUMP*, *EXISTSNODE*, *TREAT*, *NULLIF*, *SYS_CONNECT_BY_PATH*, *SYS_DBURIGEN*, *EXTRACT*.

Instrucţiuni *PL/SQL*

Orice program poate fi scris utilizând structuri de control de bază care sunt combinate în diferite moduri pentru rezolvarea problemei propuse. *PL/SQL* dispune de comenzi ce permit controlul execuţiei unui bloc. Instrucţiunile limbajului pot fi: iterative (*LOOP*, *WHILE*, *FOR*), de atribuire (:=), condiţionale (*IF*, *CASE*), de salt (*GOTO*, *EXIT*) şi instrucţiunea vidă (*NULL*).

***Observaţii*:**

- Comentariile sunt ignorate de compilatorul *PL/SQL*. Există comentarii pe o singură linie, prefixate de simbolurile „--“, care încep în orice punct al liniei şi se termină la sfârşitul acesteia. De asemenea, există comentarii pe mai multe linii, care sunt delimitate de simbolurile „/*“ şi „*/“. Nu se admit comentarii imbricate.
- Caracterul „;“ este separator pentru instrucţiuni.
- Atât operatorii din *PL/SQL*, cât şi ordinea de execuţie a acestora, sunt identici cu cei din *SQL*. În *PL/SQL* este introdus un nou operator („**“) pentru ridicare la putere.
- Un identificator este vizibil în blocul în care este declarat şi în toate subblocurile, procedurile şi funcţiile imbricate în acesta. Dacă blocul nu găseşte identificatorul declarat local, atunci îl caută în secţiunea declarativă a blocurilor care includ blocul respectiv şi niciodată nu caută în blocurile încuibărite în acesta.
- Comenzile *SQL*Plus* nu pot să apară într-un bloc *PL/SQL*.

- În comanda *SELECT* trebuie specificate variabilele care recuperează rezultatul acţiunii acestei comenzi. În clauza *INTO*, care este obligatorie, pot fi folosite variabile *PL/SQL* sau variabile de legătură.
- Referirea la o variabilă de legătură se face prin prefixarea acesteia cu simbolul „:“.
- Cererea dintr-o comandă *SELECT* trebuie să returneze o singură linie drept rezultat. Atunci când comanda *SELECT* întoarce mai multe linii, apare eroarea *TOO_MANY_ROWS*, iar în cazul în care comanda nu găseşte date se generează eroarea *NO_DATA_FOUND*.
- Un bloc *PL/SQL* nu este o unitate tranzacţională. Într-un bloc pot fi mai multe tranzacţii sau blocul poate face parte dintr-o tranzacţie. Acţiunile *COMMIT*, *SAVEPOINT* şi *ROLLBACK* sunt independente de blocuri, dar instrucţiunile asociate acestor acţiuni pot fi folosite într-un bloc.
- *PL/SQL* nu suportă comenzile *GRANT* şi *REVOKE*, utilizarea lor fiind posibilă doar prin *SQL* dinamic.

Fluxul secvenţial de execuţie a comenzilor unui program *PL/SQL* poate fi modificat cu ajutorul structurilor de control: *IF*, *CASE*, *LOOP*, *FOR*, *WHILE*, *GOTO*, *EXIT*.

Instrucţiunea de atribuire

Instrucţiunea de atribuire se realizează cu ajutorul operatorului de asignare (:=) şi are forma generală clasică (*variabila* := *expresie*). Comanda respectă proprietăţile instrucţiunii de atribuire din clasa *LG3*. De remarcat că nu poate fi asignată valoarea *null* unei variabile care a fost declarată *NOT NULL*.

***Exemplu*:**

Următorul exemplu prezintă modul în care acţionează instrucţiunea de atribuire în cazul unor tipuri de date particulare.

```
DECLARE
  alfa  INTERVAL YEAR TO MONTH;
BEGIN
  alfa := INTERVAL '200-7' YEAR TO MONTH;
  -- alfa ia valoarea 200 de ani si 7 luni
  alfa := INTERVAL '200' YEAR;
  -- pot fi specificati numai anii
  alfa := INTERVAL '7' MONTH;
  -- pot fi specificate numai lunile
  alfa := '200-7';
  -- conversie implicita din caracter
END;
```

```
DECLARE
  beta  opera%ROWTYPE;
  gama  opera%ROWTYPE;
  cursor epsilon IS SELECT * FROM opera;
  delta  epsilon%ROWTYPE;
BEGIN
  beta := gama;  -- corect
  gama := delta;  -- incorect???-testati!
END;
```

Instrucţiunea *IF*

Un program *PL/SQL* poate executa diferite porţiuni de cod, în funcţie de rezultatul unui test (predicat). Instrucţiunile care realizează acest lucru sunt cele condiţionale (*IF, CASE*).

Structura instrucţiunii *IF* în *PL/SQL* este similară instrucţiunii *IF* din alte limbaje procedurale, permiţând efectuarea unor acţiuni în mod selectiv, în funcţie de anumite condiţii. Instrucţiunea *IF-THEN-ELSIF* are următoarea formă sintactică:

IF *condiţie1* ***THEN***
  secvenţa_de_comenzi_1
[*ELSIF* *condiţie2* ***THEN***
  secvenţa_de_comenzi_2]
...
[***ELSE***
  secvenţa_de_comenzi_n]
END IF;

O secvenţă de comenzi din *IF* este executată numai în cazul în care condiţia asociată este *TRUE*. Atunci când condiţia este *FALSE* sau *NULL*, secvenţa nu este executată. Dacă pe ramura *THEN* se doreşte verificarea unei alternative, se foloseşte ramura *ELSIF* (atenţie, nu *ELSEIF*) cu o nouă condiţie. Este permis un număr arbitrar de opţiuni *ELSIF*, dar poate apărea cel mult o clauză *ELSE*. Aceasta se referă la ultimul *ELSIF*.

***Exemplu*:**

Să se specifice dacă o galerie este *mare*, *medie* sau *mica* după cum numărul operelor de artă expuse în galeria respectivă este mai mare decât 200, cuprins între 100 şi 200 sau mai mic decât 100.

```
SET SERVEROUTPUT ON
DEFINE p_cod_gal = 753
DECLARE
  v_cod_galerie  opera.cod_galerie%TYPE := &p_cod_gal;
  v_numar        NUMBER(3) := 0;
  v_comentariu   VARCHAR2(10);
BEGIN
  SELECT  COUNT(*)
  INTO    v_numar
  FROM    opera
  WHERE   cod_galerie = v_cod_galerie;
  IF v_numar < 100 THEN
    v_comentariu := 'mica';
  ELSIF v_numar BETWEEN 100 AND 200 THEN
    v_comentariu := 'medie';
  ELSE
    v_comentariu := 'mare';
  END IF;
  DBMS_OUTPUT.PUT_LINE('Galeria avand codul '||
    v_cod_galerie ||' este de tip '|| v_comentariu);
END;
/
SET SERVEROUTPUT OFF
```

Instrucţiunea *CASE*

Oracle9i furnizează o nouă comandă (*CASE*) care permite implementarea unor condiţii multiple. Instrucţiunea are următoarea formă sintactică:

[<<*eticheta*>>]
CASE *test_var*
  WHEN *valoare_1* ***THEN*** *secvenţa_de_comenzi_1*;
  WHEN *valoare_2* ***THEN*** *secvenţa_de_comenzi_2*;
  ...
  WHEN *valoare_k* ***THEN*** *secvenţa_de_comenzi_k*;
  [***ELSE*** *altă_secvenţă*;]
END CASE [*eticheta*];

Se va executa *secvenţa_de_comenzi_p*, dacă valoarea selectorului *test_var* este *valoare_p*. După ce este executată secvenţa de comenzi, controlul va trece la următoarea instrucţiune după *CASE*. Selectorul *test_var* poate fi o variabilă sau o expresie complexă care poate conţine chiar şi apeluri de funcţii.

Clauza *ELSE* este opţională. Dacă această clauză este necesară în implementarea unei probleme, dar totuşi lipseşte, iar *test_var* nu ia nici una dintre valorile ce apar în clauzele *WHEN*, atunci se declanşează eroarea predefinită *CASE_NOT_FOUND* (*ORA - 06592*).

Comanda *CASE* poate fi etichetată şi, în acest caz, eticheta poate să apară la sfârşitul clauzei *END CASE*. De remarcat că eticheta după *END CASE* este permisă numai în cazul în care comanda *CASE* este etichetată.

Selectorul *test_var* poate să lipsească din structura comenzii *CASE*, care în acest caz va avea următoarea formă sintactică:

[<<*eticheta*>>]
CASE
  WHEN *condiţie_1* ***THEN*** *secvenţa_de_comenzi_1*;
  WHEN *condiţie_2* ***THEN*** *secvenţa_de_comenzi_2*;
  ...
  WHEN *condiţie_k* ***THEN*** *secvenţa_de_comenzi_k*;
  [***ELSE*** *altă_secvenţă*;]
END CASE [*eticheta*];

Fiecare clauză *WHEN* conţine o expresie booleană. Dacă valoarea lui *condiţie_p* este *TRUE*, atunci este executată *secvenţa_de_comenzi_p*.

***Exemplu*:**

În funcţie de o valoare introdusă de utilizator, care reprezintă abrevierea zilelor unei săptămâni, să se afişeze (în cele două variante) un mesaj prin care este specificată ziua săptămânii corespunzătoare abrevierii respective.

Varianta 1:

```
SET SERVEROUTPUT ON
DEFINE p_zi = x
DECLARE
  v_zi  CHAR(2) := UPPER('&p_zi');
BEGIN
  CASE v_zi
    WHEN 'L'  THEN DBMS_OUTPUT.PUT_LINE('Luni');
    WHEN 'M'  THEN DBMS_OUTPUT.PUT_LINE('Marti');
    WHEN 'MI' THEN DBMS_OUTPUT.PUT_LINE('Miercuri');
    WHEN 'J'  THEN DBMS_OUTPUT.PUT_LINE('Joi');
    WHEN 'V'  THEN DBMS_OUTPUT.PUT_LINE('Vineri');
    WHEN 'S'  THEN DBMS_OUTPUT.PUT_LINE('Sambata');
    WHEN 'D'  THEN DBMS_OUTPUT.PUT_LINE('Duminica');
    ELSE DBMS_OUTPUT.PUT_LINE('este o eroare!');
  END CASE;
END;
/
SET SERVEROUTPUT OFF
```

Varianta 2:

```
SET SERVEROUTPUT ON
DEFINE p_zi = x
DECLARE
  v_zi  CHAR(2) := UPPER('&p_zi');
BEGIN
  CASE
    WHEN v_zi = 'L'  THEN
              DBMS_OUTPUT.PUT_LINE('Luni');
    WHEN v_zi = 'M'  THEN
              DBMS_OUTPUT.PUT_LINE('Marti');
    WHEN v_zi = 'MI' THEN
              DBMS_OUTPUT.PUT_LINE('Miercuri');
    WHEN v_zi = 'J'  THEN
              DBMS_OUTPUT.PUT_LINE('Joi');
    WHEN v_zi = 'V'  THEN
              DBMS_OUTPUT.PUT_LINE('Vineri');
    WHEN v_zi = 'S'  THEN
              DBMS_OUTPUT.PUT_LINE('Sambata');
    WHEN v_zi = 'D'  THEN
              DBMS_OUTPUT.PUT_LINE('Duminica');
    ELSE DBMS_OUTPUT.PUT_LINE('Este o eroare!');
  END CASE;
END;
/
SET SERVEROUTPUT OFF
```

Oracle8i a implementat suportul pentru expresii *CASE* care sunt permise numai în comenzi *SQL*. În *Oracle9i* poate fi utilizată o construcţie *CASE* într-o comandă *SQL* a unui bloc *PL/SQL*.

Expresia *CASE* are sintaxa similară comenzii *CASE*, dar clauzele *WHEN* nu se termină prin caracterul „;“, clauza *END* nu include cuvântul cheie *CASE* şi nu se fac atribuiri în clauza *WHEN*.

***Exemplu*:**

```
BEGIN
  FOR j IN (SELECT
              CASE valoare
                WHEN 1000 THEN 1100
                WHEN 10000 THEN 11000
                WHEN 100000 THEN 110000
                ELSE valoare
              END
             FROM opera)
  …
  END LOOP;
END;
```

Instrucţiuni iterative

Există trei tipuri de comenzi iterative: ciclarea simplă *LOOP*, ciclarea *WHILE* şi ciclarea *FOR*.

Acestea permit repetarea (condiţionată sau necondiţionată) execuţiei uneia sau mai multor instrucţiuni. Ciclurile pot fi imbricate pe mai multe niveluri. Ele pot fi etichetate, iar ieşirea din ciclu se poate realiza cu ajutorul comenzii *EXIT*.

Se utilizează:

- comanda *LOOP*, dacă instrucţiunile din cadrul ciclului trebuie să se execute cel puţin o dată;
- comanda *WHILE*, în cazul în care condiţia trebuie evaluată la începutul fiecărei iteraţii;
- comanda *FOR*, dacă numărul de iteraţii este cunoscut.

Instrucţiunea *LOOP* are următoarea formă sintactică:

LOOP
secvenţa_de_comenzi;
END LOOP;

Ciclarea simplă cuprinde o mulţime de comenzi incluse între cuvintele cheie *LOOP* şi *END LOOP*. Aceste comenzi se execută cel puţin o dată. Dacă nu este utilizată comanda *EXIT*, ciclarea poate continua la infinit.

***Exemplu*:**

Se presupune că a fost creată structura tabelului *org_tab*, constând din două coloane: *cod_tab* de tip *INTEGER*, ce conţine un contor al înregistrărilor şi *text_tab* de tip *VARCHAR2*, ce conţine un text asociat fiecărei înregistrări. Să se introducă 70 de înregistrări în tabelul *org_tab*.

```
DECLARE
  v_contor BINARY_INTEGER := 1;
BEGIN
  LOOP
    INSERT INTO org_tab
    VALUES (v_contor, 'indicele ciclului');
    v_contor := v_contor + 1;
    EXIT WHEN v_contor > 70;
  END LOOP; COMMIT;
END;
```

Instrucţiunea repetitivă *WHILE* permite repetarea unei secvenţe de instrucţiuni, atâta timp cât o anumită condiţie specificată este adevărată.

Comanda *WHILE* are următoarea sintaxă:

WHILE *condiţie* ***LOOP***
  secvenţa_de_comenzi;
END LOOP;

Dacă variabilele care apar în condiţie nu se schimbă în interiorul ciclului, atunci condiţia rămâne adevărată şi ciclul nu se termină.

Când condiţia este evaluată ca fiind *FALSE* sau *NULL*, atunci secvenţa de comenzi nu este executată şi controlul trece la prima instrucţiune după *END LOOP*.

***Exemplu*:**

```
DECLARE
  v_contor BINARY_INTEGER := 1;
BEGIN
  WHILE v_contor <= 70 LOOP
    INSERT INTO org_tab
    VALUES (v_contor, 'indicele ciclului');
    v_contor := v_contor + 1;
  END LOOP;
END;
```

Instrucţiunea repetitivă *FOR* (ciclare cu pas) permite executarea unei secvenţe de instrucţiuni pentru valori ale variabilei *contor* cuprinse între două limite, *lim_inf* şi *lim_sup*. Dacă este prezentă opţiunea *REVERSE*, iteraţia se face (în sens invers) de la *lim_sup* la *lim_inf*.

Comanda *FOR* are sintaxa:

FOR *contor_ciclu* ***IN*** [***REVERSE***] *lim_inf..lim_sup* ***LOOP***
  secvenţa_de_comenzi;
END LOOP;

Variabila *contor_ciclu* nu trebuie declarată. Ea este neidentificată în afara ciclului şi implicit de tip *BINARY_INTEGER*. Pasul are implicit valoarea 1 şi nu poate fi modificat. Limitele domeniului pot fi variabile sau expresii, care să poată fi convertite la întreg.

***Exemplu*:**

În structura tabelului *opera* se va introduce un nou câmp (*stea*). Să se creeze un bloc *PL/SQL* care va reactualiza acest câmp, introducând o steluţă pentru fiecare 10000$ din valoarea unei opere de artă al cărei cod este specificat.

```
ALTER TABLE  opera
ADD stea     VARCHAR2(20);

DEFINE p_cod_opera = 7777
DECLARE
  v_cod_opera  opera.cod_opera%TYPE := &p_cod_opera;
  v_valoare    opera.valoare%TYPE;
  v_stea       opera.stea%TYPE := NULL;
BEGIN
  SELECT  NVL(ROUND(valoare/10000),0)
  INTO    v_valoare
  FROM    opera
  WHERE   cod_opera = v_cod_opera;
  IF v_valoare > 0 THEN
    FOR i IN 1..v_valoare LOOP
      v_stea := v_stea || '*';
    END LOOP;
  END IF;
  UPDATE  opera
  SET     stea = v_stea
  WHERE   cod_opera = v_cod_opera;
  COMMIT;
END;
```

Instrucţiuni de salt

Instrucţiunea *EXIT* permite ieşirea dintr-un ciclu. Ea are o formă necondiţională (ieşire fără condiţii) şi una condiţională. Controlul trece fie la prima instrucţiune situată după clauza *END LOOP* corespunzătoare, fie după instrucţiunea *LOOP* având eticheta *nume_eticheta*.

EXIT [*nume_eticheta*] [***WHEN*** *condiţie*];

Numele etichetelor urmează aceleaşi reguli ca cele definite pentru identificatori. Eticheta se plasează înaintea comenzii, fie pe aceeaşi linie, fie pe o linie separată. În *PL/SQL* etichetele se definesc prin intercalarea numelui etichetei între caracterele „<<“ şi „>>“ (<<*eticheta*>>).

***Exemplu*:**

```
DECLARE
  v_contor    BINARY_INTEGER := 1;
  raspuns     VARCHAR2(10);
  alt_raspuns VARCHAR2(10);
BEGIN
  …
  <<exterior>>
  LOOP
    v_contor := v_contor + 1;
    EXIT WHEN v_contor > 70;
    <<interior>>
    LOOP
      …
      EXIT exterior WHEN raspuns = 'DA';
      -- se parasesc ambele cicluri
      EXIT WHEN alt_raspuns = 'DA';
      -- se paraseste ciclul interior
      …
    END LOOP interior;
    …
  END LOOP exterior;
END;
```

Instrucţiunea *GOTO* determină un salt necondiţionat la o instrucţiune executabilă sau la începutul unui bloc care are eticheta specificată în comandă. Instrucţiunea are următoarea formă sintactică:

GOTO *nume_eticheta*;

Nu este permis saltul:

- în interiorul unui bloc (subbloc);
- în interiorul unei comenzi *IF*, *CASE* sau *LOOP*;
- de la o clauză a comenzii *CASE*, la altă clauză a aceleaşi comenzi;
- de la tratarea unei excepţii, în blocul curent;
- în exteriorul unui subprogram.

Instrucţiunea vidă

Instrucţiunea vidă (*NULL*) este folosită pentru o mai bună lizibilitate a programului. *NULL* este instrucţiunea care nu are nici un efect, marcând faptul că nu trebuie întreprinsă nici o acţiune. Nu trebuie confundată instrucţiunea *NULL* cu valoarea *null*!

Uneori instrucţiunea *NULL* este folosită într-o comandă *IF*, indicând faptul că pentru o anumită clauză *ELSIF* nu se execută nici o acţiune.

Tipuri de date în *PL/SQL*

Fiecare variabilă sau constantă utilizată într-un bloc *PL/SQL* este de un anumit tip prin care se specifică:

- formatul său de stocare,
- constrângerile care trebuie să le verifice
- domeniul valorilor sale.

Variabilele folosite în *Oracle9i* pot fi:

- specifice *PL/SQL;*
- nespecifice *PL/SQL.*

Variabile specifice *PL/SQL* se clasifică în variabile:

- de tip scalar,
- compuse,
- referinţă,
- *LOB* (*large objects*),
- tipuri obiect.

Variabile nespecifice *PL/SQL* pot fi:

- variabile de legătură (*bind variables*),
- variabile gazdă (*host variables*),
- variabile indicator.

Variabile specifice *PL/SQL*

Tipurile de date scalare

Nu au componente interne (conţin valori atomice). Se împart în 5 clase.

- Tipurile de date ce stochează **valori numerice** cuprind tipul *NUMBER* cu subtipurile *DEC*, *DECIMAL*, *DOUBLE PRECISION*, *FLOAT*, *INTEGER*, *INT*, *NUMERIC*, *REAL*, *SMALLINT*; tipul *BINARY_INTEGER* cu subtipurile *NATURAL*, *NATURALN*, *POSITIVE*, *POSITIVEN*, *SIGNTYPE*; tipul *PLS_INTEGER*.
- Tipurile de date ce stochează **caractere** cuprind tipul *VARCHAR2* cu subtipurile *STRING*, *VARCHAR*; tipul de date *CHAR* cu subtipul *CHARACTER*; tipurile *LONG*, *RAW*, *LONG RAW*, *ROWID*.
- Tipurile de date ce stochează **data calendaristică şi ora** cuprind tipurile *DATE*, *TIMESTAMP*, *TIMESTAMP WITH TIME ZONE*, *TIMESTAMP WITH LOCAL TIME ZONE*, *INTERVAL YEAR TO MONTH*, *INTERVAL DAY TO SECOND*.
- Tipurile de date **globalizare** ce stochează date *unicode* includ tipurile *NCHAR* şi *NVARCHAR2*.
- Tipul de date *BOOLEAN* stochează **valori logice** (*true*, *false* sau *null*).

Tipurile de date compuse

Au componente interne care pot fi manipulate individual. *Oracle* oferă programatorului două tipuri de date compuse:

- înregistrare (*RECORD*);
- colecţie (*INDEX-BY TABLE*, *NESTED TABLE*, *VARRAY*).

Tipurile de date referinţă

REF CURSOR si *REF obiect* sunt tipuri de date ale căror valori, numite *pointeri*, fac referinţă către obiecte din program. Pointerii conţin locaţia de memorie (adresa) unui element şi nu elementul în sine. Tipul *REF CURSOR* este folosit pentru a face referinţă la un cursor explicit. Tipul *REF obiect* face referinţă la adresa unui obiect.

Tipurile de date *LOB*

Large object sunt acele tipuri de date ale căror valori, numite locatori (*locators*) specifică localizarea unor obiecte de dimensiuni mari, adică blocuri de date nestructurate, cum ar fi texte, imagini grafice, clipuri video şi sunete. Tipurile *LOB* sunt manipulate cu ajutorul pachetului *DBMS_LOB*.

Aceste tipuri sunt:

- *CLOB* (*character large object*),
- *BLOB* (*binary large object*),
- *BFILE* (*binary file*),
- *NCLOB* (*national language character large object*).

Tipurile obiect

Sunt tipuri compuse, definite de utilizator, care încapsulează structuri de date (atribute) împreună cu subprograme pentru manipularea datelor (metode).

Dintre tipurile scalare *PL/SQL*, următoarele sunt şi tipuri *SQL* (adică pot fi folosite pentru coloanele tabelelor *Oracle*): *NUMBER*, *VARCHAR2*, *CHAR*, *LONG*, *RAW*, *LONG RAW*, *ROWID*, *NCHAR*, *NVARCHAR2*, *DATE*. În unele cazuri, tipurile de date *PL/SQL* diferă de corespondentele lor *SQL* prin dimensiunea maximă permisă.

Tipul *NUMBER* memorează numerele în virgulă fixă şi virgulă mobilă. El are forma generală *NUMBER* (*m*, *n*), unde *m* reprezintă numărul total de cifre, iar *n* numărul de zecimale. Valoarea unei variabile de tip *NUMBER* este cuprinsă între 1.0E-129 şi 9.99E125. Numărul de zecimale determină poziţia în care apare rotunjirea. Valoarea sa este cuprinsă între -84 şi 127, iar implicit este 0.

Tipul *NUMBER* are următoarele subtipuri, care au aceleaşi intervale de valori: *NUMERIC*, *REAL*, *DEC*, *DECIMAL* şi *DOUBLE PRECISION* (pentru memorarea datelor numerice în virgulă fixă), *FLOAT* (pentru memorarea datelor numerice în virgulă mobilă), *SMALLINT*, *INTEGER* şi *INT* (pentru memorarea numerelor întregi). Aceste subtipuri se pot utiliza pentru compatibilitate *ANSI/ISO*, *IBM SQL/DS* sau *IBM DB2*.

Tipul *BINARY_INTEGER* memorează numere întregi cu semn având valori cuprinse între $-2^{31} - 1$ şi $2^{31} - 1$. Acest tip de date este utilizat frecvent pentru indecşii tabelelor, nu necesită conversii şi admite mai multe subtipuri. De exemplu, pentru a restricţiona domeniul variabilelor la valori întregi nenegative se utilizează tipurile *NATURAL* ($0 .. 2^{31} - 1$) şi *POSITIVE* ($1 .. 2^{31} - 1$).

Tipul *PLS_INTEGER* este utilizat pentru stocarea numerelor întregi cu semn şi are acelaşi interval de definire ca şi tipul *BINARY_INTEGER*. Operaţiile cu acest tip sunt efectuate mai rapid (folosesc aritmetica maşinii), decât cele cu tipurile *NUMBER* sau *BINARY_INTEGER* (folosesc librării aritmetice). Prin urmare, pentru o mai bună performanţă, este preferabil să se utilizeze tipul *PLS_INTEGER*.

Variabilele alfanumerice pot fi de tip *CHAR*, *VARCHAR2*, *LONG*, *RAW* şi *LONGRAW*. Reprezentarea internă depinde de setul de caractere ales (*ASCII* sau *EBCDIC*).

Tipurile *CHAR*, *VARCHAR2* şi *RAW* pot avea un parametru pentru a preciza lungimea maximă. Dacă aceasta nu este precizată atunci, implicit, se consideră 1. Lungimea este exprimată în octeţi (nu în caractere). Subtipurile acestor tipuri se pot utiliza pentru compatibilitate *ANSI/ISO*, *IBM SQL/DS* sau *IBM DB2*.

În *Oracle9i* a fost extinsă sintaxa pentru *CHAR* şi *VARCHAR2*, permiţând ca variabila ce precizează lungimea maximă să fie de tip *CHAR* sau *BYTE*.

Variabilele de tip *LONG* pot memora texte, tabele de caractere sau documente, prin urmare şiruri de caractere de lungime variabilă de până la 32760 octeţi. Este similar tipului *VARCHAR2*.

Tipul *RAW* permite memorarea datelor binare (biţi) sau a şirurilor de octeţi. De exemplu, o variabilă *RAW* poate memora o secvenţă de caractere grafice sau o imagine digitizată. Tipul *RAW* este similar tipului alfanumeric, cu excepţia faptului că *PL/SQL* nu interpretează datele de tip *RAW*. *Oracle* nu face conversia datelor de acest tip, atunci când se transmit de la un sistem la altul. Chiar dacă lungimea maximă a unei variabile *RAW* poate fi 32767 octeţi, într-o coloană *RAW* a bazei de date nu se pot introduce decât 2000 octeţi. Pentru a insera valori mai mari se foloseşte o coloană de tip *LONG RAW*, care are lungimea maximă 2^{31} octeţi. *LONG RAW* este similar tipului *LONG*, dar datele nu mai sunt interpretate de *PL/SQL*.

Tipurile *TIMESTAMP*, *TIMESTAMP WITH TIME ZONE*, *TIMESTAMP WITH LOCAL TIME ZONE*, *INTERVAL YEAR TO MONTH*, *INTERVAL DAY TO SECOND* au fost introduse în *Oracle9i* şi permit rafinări ale tipului *DATE*. De exemplu, *TIMESTAMP* poate lua în considerare şi fracţiuni de secundă.

PL/SQL suportă două seturi de caractere: una specifică bazei de date care este utilizată pentru definirea identificatorilor şi a codului sursă (*database character set* - *DCS*) şi o mulţime de caractere naţionale care este folosită pentru reprezentarea informaţiei cu caracter naţional (*national character set* - *NCS*).

Tipurile de date *NCHAR* şi *NVARCHAR2* sunt utilizate pentru stocarea în baza de date a şirurilor de caractere ce folosesc *NCS*. Ele oferă suport pentru globalizarea datelor, astfel încât utilizatorii din toată lumea pot interacţiona cu *Oracle* în limba lor naţională. Aceste tipuri de date suportă numai date *Unicode*.

Unicode este o mulţime de caractere globale care permite stocarea de informaţie în orice limbă, folosind o mulţime unică de caractere.

Prin urmare, *unicode* furnizează o valoare cod unică pentru fiecare caracter, indiferent de platformă, program sau limbă.

Variabile nespecifice *PL/SQL*

Variabila de legătură (*bind*) se declară într-un mediu gazdă şi este folosită pentru transferul (la execuţie) valorilor numerice sau de tip caracter în/din unul sau mai multe programe *PL/SQL*. Variabilele declarate în mediul gazdă sau în cel apelant pot fi referite în instrucţiuni *PL/SQL*, dacă acestea nu sunt în cadrul unei proceduri, funcţii sau pachet.

În *SQL*Plus*, variabilele de legătură se declară folosind comanda *VARIABLE*, iar pentru afişarea valorilor acestora se utilizează comanda *PRINT*. Ele sunt referite prin prefixarea cu simbolul „:“, pentru a putea fi deosebite de variabilele declarate în *PL/SQL*.

Deoarece instrucţiunile *SQL* pot fi integrate în programe *C*, este necesar un mecanism pentru a transfera valori între mediul de programare *C* şi instrucţiunile *SQL* care comunică cu *server*-ul bazei de date *Oracle*. În acest scop, în programul încapsulat sunt definite variabilele gazdă (*host*). Acestea sunt declarate între directivele *BEGIN DECLARE SECTION* şi *END DECLARE SECTION* ale preprocesorului.

O valoare *null* în baza de date nu are o valoare corespunzătoare în mediul limbajului gazdă (de exemplu, limbajul *C*). Pentru a rezolva problema comunicării valorilor *null* între programul scris în limbaj gazdă şi sistemul *Oracle*, au fost definite variabilele indicator. Acestea sunt variabile speciale de tip întreg, folosite pentru a indica dacă o valoare *null* este recuperată (extrasă) din baza de date sau stocată în aceasta. Ele au următoarea formă:

:*nume_extern* [: *indicator*]

De exemplu, dacă atribuirea este făcută de limbajul gazdă, valoarea –1 a indicatorului specifică faptul că *PL/SQL* trebuie să înlocuiască valoarea variabilei prin *null*, iar o valoare a indicatorului mai mare ca zero precizează că *PL/SQL* trebuie să considere chiar valoarea variabilei.

Declararea variabilelor

Identificatorii *PL/SQL* trebuie declaraţi înainte de a fi referiţi în blocul *PL/SQL*. Dacă în declaraţia unei variabile apar referiri la alte variabile, acestea trebuie să fi fost declarate anterior. Orice variabilă declarată într-un bloc este accesibilă blocurilor conţinute sintactic în acesta.

Tipurile scalare sunt predefinite în pachetul *STANDARD*. Pentru a folosi un astfel de tip într-un program este suficient să fie declarată o variabilă de tipul respectiv.

Tipurile compuse sunt definite de utilizator. Prin urmare, în acest caz trebuie definit efectiv tipul şi apoi declarată variabila de tipul respectiv.

În declararea variabilelor pot fi utilizate atributele *%TYPE* şi *%ROWTYPE*, care reprezintă tipuri de date implicite. Aceste tipuri permit declararea unei variabile în concordanţă cu declaraţii de variabile făcute anterior.

Atributul *%TYPE* permite definirea unei variabile având tipul unei variabile declarate anterior sau tipul unei coloane dintr-un tabel.

Atributul *%ROWTYPE* permite definirea unei variabile având tipul unei înregistrări dintr-un tabel. Avantajul utilizării acestui atribut constă în faptul că nu este necesar să se cunoască numărul şi tipurile coloanelor tabelului. Elementele individuale ale acestei structuri de tip înregistrare sunt referite în maniera clasică, prefixând numele coloanei cu numele variabilei declarate.

Calitatea atributelor *%TYPE* şi *%ROWTYPE* constă în faptul că simplifică întreţinerea codului *PL/SQL*. De exemplu, poate fi modificată dimensiunea unei coloane, fără să fie necesară modificarea declaraţiei variabilelor al căror tip s-a definit făcând referinţă la tipul coloanei respective.

Sintaxa declarării unei variabile este următoarea:

identificator [***CONSTANT***] {*tip_de_date* | *identificator****%TYPE*** |
*identificator****%ROWTYPE***} [***NOT NULL***]
[{:= | ***DEFAULT***} *expresie_PL/SQL*];

Se pot defini constante (valoarea stocată nu poate fi modificată) prin specificarea la declarare a cuvântului cheie *CONSTANT*.

***Exemplu*:**

```
v_valoare        NUMBER(15) NOT NULL := 0;
v_data_achizitie DATE DEFAULT SYSDATE;
v_material       VARCHAR2(15) := 'Matase';
c_valoare        CONSTANT NUMBER := 100000;
v_stare          VARCHAR2(20) DEFAULT 'Buna';
v_clasificare    BOOLEAN  DEFAULT FALSE;
v_cod_opera      opera.cod_opera%TYPE;
v_opera          opera%ROWTYPE;
int_an_luna      INTERVAL YEAR TO MONTH :=
                   INTERVAL '3-2' YEAR TO MONTH;
```

***Observaţii*:**

1. Pentru a denumi o variabilă este utilizată frecvent (pentru uşurinţa referirii) prefixarea cu litera *v* (*v_identificator*), iar pentru o constantă este folosită prefixarea cu litera *c* (*c_identificator*).

2. Variabilele pot fi iniţializate, iar dacă o variabilă nu este iniţializată, valoarea implicită a acesteia este *null*. Dacă o variabilă este declarată *NOT NULL*, atunci ea va fi obligatoriu iniţializată.
3. Pentru a iniţializa o variabilă sau o constantă poate fi utilizată o expresie *PL/SQL* compatibilă ca tip cu variabila sau constanta respectivă.
4. Constantele trebuie iniţializate când sunt declarate, altfel apare o eroare la compilare.
5. În secţiunea declarativă, pe fiecare linie, există o singură declaraţie de variabilă.
6. Două obiecte (variabile) pot avea acelaşi nume cu condiţia să fie definite în blocuri diferite. Dacă ele coexistă, poate fi folosit doar obiectul declarat în blocul curent.
7. Atributul *%ROWTYPE* nu poate include clauze de iniţializare.

Definirea subtipurilor

Subtipurile derivă dintr-un tip de bază, la care se adaugă anumite restricţii. De exemplu, *NATURAL* este un subtip predefinit *PL/SQL*, derivat din tipul de bază *BINARY_INTEGER*, cu restricţia că permite prelucrarea valorilor întregi nenegative.

Prin urmare, un subtip nu reprezintă un nou tip de date, ci un tip existent asupra căruia se aplică anumite constrângeri. Subtipurile presupun acelaşi set de operaţii ca şi tipul de bază, dar aplicate unui subset de valori al acestui tip.

Sistemul *Oracle* permite ca utilizatorul să-şi definească propriile sale tipuri şi subtipuri de date în partea declarativă a unui bloc *PL/SQL*, subprogram sau pachet utilizând sintaxa:

SUBTYPE *nume_subtip* ***IS*** *tip_de_baza* [***NOT NULL***];

În dicţionarul datelor există vizualizări care furnizează informaţii despre tipurile de date create de utilizator (*USER_TYPES*, *USER_TYPE_ATTRS*).

Conversii între tipuri de date

Există două tipuri de conversii:

implicite;

explicite.

PL/SQL face automat conversii implicite între caractere şi numere sau între caractere şi date calendaristice. Chiar dacă sistemul realizează automat aceste conversii, în practică se utilizează frecvent funcţii de conversie explicită.

Funcţiile de conversie explicită din *SQL* sunt utilizabile şi în *PL/SQL*. *A*cestea sunt: *TO_NUMBER, TO_CHAR, TO_DATE, TO_MULTI_BYTE, TO_SINGLE_BYTE, CHARTOROWID, ROWIDTOCHAR, RAWTOHEX, HEXTORAW, TO_CLOB, TO_LOB.*

În *Oracle9i* se pot folosi următoarele funcţii de conversie: *ASCIISTR*, *BIN_TO_NUM*, *NUMTODSINTERVAL*, *TO_TIMESTAMP*, *TO_YMINTERVAL*, *TO_NCHAR*, *TO_NCLOB*, *TO_TIMESTAMP_TZ*, *NUMTOYMINTERVAL*, *TO_DSINTERVAL*, *REFTOHEX*, *RAWTOHEX*, *RAWTONHEX*, *FROM_TZ*, *ROWIDTONCHAR*, *COMPOSE*, *DECOMPOSE*.

Denumirile acestor funcţii reflectă posibilităţile pe care le oferă. De exemplu, *TO_YMINTERVAL* converteşte argumentele sale la tipul *INTERVAL YEAR TO MONTH* conform unui format specificat. Funcţia *COMPOSE* converteşte un şir de caractere la un şir *unicode* (asociază o valoare cod unică pentru fiecare simbol din şir).

Înregistrări

Tipul *RECORD* oferă un mecanism pentru prelucrarea înregistrărilor. Înregistrările au mai multe câmpuri ce pot fi de tipuri diferite, dar care sunt legate din punct de vedere logic.

Inregistrările trebuie definite în doi paşi:

- se defineşte tipul *RECORD*;
- se declară înregistrările de acest tip.

Declararea tipului *RECORD* se face conform următoarei sintaxe:

TYPE *nume_tip* ***IS RECORD***
(*nume_câmp1* {*tip_câmp* | *variabilă****%TYPE*** |
*nume_tabel.coloană****%TYPE*** | *nume_tabel****%ROWTYPE***}
[[***NOT NULL***] {:= | ***DEFAULT***} *expresie1*],
(*nume_câmp2* {*tip_câmp* | *variabilă****%TYPE*** |
*nume_tabel.coloană****%TYPE*** | *nume_tabel****%ROWTYPE***}
[[***NOT NULL***] {:= | ***DEFAULT***} *expresie2*],…);

Identificatorul *nume_tip* reprezintă numele tipului *RECORD* care se va specifica în declararea înregistrărilor, *nume_câmp* este numele unui câmp al înregistrării, iar *tip_câmp* este tipul de date al câmpului.

***Observaţii*:**

- Dacă un câmp nu este iniţializat atunci implicit se consideră că are valoarea *NULL*. Dacă s-a specificat constrângerea *NOT NULL*, atunci obligatoriu câmpul trebuie iniţializat cu o valoare diferită de *NULL*.
- Pentru referirea câmpurilor individuale din înregistrare se prefixează numele câmpului cu numele înregistrării.
- Pot fi asignate valori unei înregistrări utilizând comenzile *SELECT*, *FETCH* sau instrucţiunea clasică de atribuire. De asemenea, o înregistrare poate fi asignată altei înregistrari de acelaşi tip.

- Componentele unei înregistrări pot fi de tip scalar, *RECORD, TABLE,* obiect, colecţie (dar, nu tipul *REF CURSOR*).
- *PL/SQL* permite declararea şi referirea înregistrărilor imbricate.
- Numărul de câmpuri ale unei înregistrări nu este limitat.
- Înregistrările nu pot fi comparate (egalitate, inegalitate sau *null*).
- Înregistrările pot fi parametri în subprograme şi pot să apară în clauza *RETURN* a unei funcţii.

Diferenţa dintre atributul *%ROWTYPE* şi tipul de date compus *RECORD:*

- tipul *RECORD* permite specificarea tipului de date pentru câmpuri şi permite declararea câmpurilor sale;
- atributul *%ROWTYPE* nu cere cunoaşterea numărului şi tipurilor coloanelor tabloului.

Oracle9i introduce câteva facilităţi legate de acest tip de date.

- Se poate insera (*INSERT*) o linie într-un tabel utilizând o înregistrare. Nu mai este necesară listarea câmpurilor individuale, ci este suficientă utilizarea numelui înregistrării.
- Se poate reactualiza (*UPDATE*) o linie a unui tabel utilizând o înregistrare. Sintaxa *SET ROW* permite să se reactualizeze întreaga linie folosind conţinutul unei înregistrări.
- Într-o înregistrare se poate regăsi şi returna sau şterge informaţia din clauza *RETURNING* a comenzilor *UPDATE* sau *DELETE*.
- Dacă în comenzile *UPDATE* sau *DELETE* se modifică mai multe linii, atunci pot fi utilizate în sintaxa *BULK COLLECT INTO*, colecţii de înregistrări.

***Exemplu*:**

Exemplul următor arată modul în care poate să fie utilizată o înregistrare în clauza *RETURNING* asociată comenzii *DELETE*.

```
DECLARE
  TYPE val_opera IS RECORD (
    cheie  NUMBER,
    val    NUMBER);
  v_info_valoare  val_opera;
BEGIN
  DELETE FROM opera
      WHERE cod_opera = 753
    RETURNING cod_opera, valoare
      INTO v_info_valoare;
  …
END;
```

Colecţii

Uneori este preferabil să fie prelucrate simultan mai multe variabile de acelaşi tip. Tipurile de date care permit acest lucru sunt colecţiile. Fiecare element are un indice unic, care determină poziţia sa în colecţie.

Oracle7 a furnizat tipul *index-by table*, iniţial numit *PL/SQL table* datorită asemănării sale cu structura tabelelor relaţionale.

Oracle8 a introdus două tipuri colecţie, *nested table* şi *varray*. *Oracle9i* permite crearea de colecţii pe mai multe niveluri, adică colecţii de colecţii.

În *PL/SQL* există trei tipuri de colecţii:

- tablouri indexate (*index-by tables*);
- tablouri imbricate (*nested tables*);
- vectori (*varrays* sau *varying arrays*).

Tipul *index-by table* poate fi utilizat **numai** în declaraţii *PL/SQL*. Tipurile *varray* şi *nested table* pot fi utilizate atât în declaraţii *PL/SQL*, cât şi în declaraţii la nivelul schemei (de exemplu, pentru definirea tipului unei coloane a unui tabel relaţional).

***Exemplu*:**

În exemplul care urmează sunt ilustrate cele trei tipuri de colecţii.

```
DECLARE
  TYPE  tab_index IS TABLE OF NUMBER
    INDEX BY BINARY_INTEGER;
  TYPE  tab_imbri IS TABLE OF NUMBER;
  TYPE  vector IS VARRAY(15) OF NUMBER;
  v_tab_index  tab_index;
  v_tab_imbri  tab_imbri;
  v_vector     vector;
BEGIN
  v_tab_index(1) := 72;
  v_tab_index(2) := 23;
  v_tab_imbri := tab_imbri(5, 3, 2, 8, 7);
  v_vector := vector(1, 2);
END;
```

***Observaţii*:**

- Deoarece colecţiile nu pot fi comparate (egalitate sau inegalitate), ele nu pot să apară în clauzele *DISTINCT*, *GROUP BY*, *ORDER BY*.
- Tipul colecţie poate fi definit într-un pachet.
- Tipul colecţie poate să apară în clauza *RETURN* a unei funcţii.
- Colecţiile pot fi parametri formali într-un subprogram.
- Accesul la elementele individuale ale unei colecţii se face prin utilizarea unui indice.

Tablouri indexate

Tipul de date *index-by table* oferă un mecanism pentru prelucrarea tablourilor. Tabloul indexat *PL/SQL* are două componente: o coloană ce cuprinde cheia primară pentru acces la liniile tabloului şi o coloană care include valoarea efectivă a elementelor tabloului.

Oracle7 asigură definirea tablourilor de înregistrări care pot fi declarate şi utilizate numai în programe *PL/SQL*, *Oracle8* realizează definirea tablourilor de tipuri obiect, iar *Oracle9i* permite definirea tablourilor de colecţii.

În *Oracle9i* tipul *index-by table* este redenumit *associative array* pentru compatibilitate (de limbaj) cu termenul folosit în alte limbaje de programare (*C++*, *JavaScript*, *Perl*) pentru a defini această structură de date.

Tablourile indexate *PL/SQL* trebuie definite în doi paşi: se defineşte tipul *TABLE*; se declară tabloul indexat *PL/SQL* de acest tip.

Declararea tipului *TABLE* se face respectând următoarea sintaxă:

TYPE *nume_tip* ***IS TABLE OF***
{*tip_coloană* | *variabilă****%TYPE*** |
*nume_tabel.coloană****%TYPE*** [***NOT NULL***] |
*nume_tabel****%ROWTYPE***}
INDEX BY *tip_indexare*;

Identificatorul *nume_tip* este numele noului tip definit care va fi specificat în declararea tabloului *PL/SQL*, iar *tip_coloană* este un tip scalar simplu (de exemplu, *VARCHAR2*, *CHAR*, *DATE* sau *NUMBER*).

Până la versiunea *Oracle9i* unicul tip de indexare acceptat era *INDEX BY BINARY_INTEGER*. *Oracle9i* permite următoarele opţiuni pentru *tip_indexare*: *PLS_INTEGER*, *NATURAL*, *POSITIVE*, *VARCHAR2*(*n*) sau chiar indexarea după un tip declarat cu *%TYPE*. Nu sunt permise indexările *INDEX BY NUMBER*, *INDEX BY INTEGER*, *INDEX BY DATE*, *INDEX BY VARCHAR2*, *INDEX BY CHAR*(*n*) sau indexarea după un tip declarat cu *%TYPE* în care intervine unul dintre tipurile enumerate anterior.

***Observaţii*:**

- Elementele unui tablou indexat nu sunt într-o ordine particulară şi pot fi inserate cu chei arbitrare.
- Deoarece nu există constrângeri de dimensiune, dimensiunea tabloului se modifică dinamic.
- Tabloul indexat *PL/SQL* nu poate fi iniţializat în declararea sa.
- Un tablou indexat neiniţializat este vid (nu conţine nici valori, nici chei).

- Un element al tabloului este nedefinit atâta timp cât nu are atribuită o valoare efectivă.
- Iniţial, un tablou indexat este nedens. După declararea unui tablou se poate face referire la liniile lui prin precizarea valorii cheii primare.
- Dacă se face referire la o linie care nu există, atunci se produce excepţia *NO_DATA_FOUND*.
- Dacă se doreşte contorizarea numărului de linii, trebuie declarată o variabilă în acest scop sau poate fi utilizată o metodă asociată tabloului.
- Deoarece numărul de linii nu este limitat, operaţia de adăugare de linii este restricţionată doar de dimensiunea memoriei alocate.
- Tablourile pot să apară ca argumente într-o procedură.

Pentru inserarea unor valori din tablourile *PL/SQL* într-o coloană a unui tabel de date se utilizează instrucţiunea *INSERT* în cadrul unei secvenţe repetitive *LOOP*.

Asemănător, pentru regăsirea unor valori dintr-o coloană a unei baze de date într-un tablou *PL/SQL* se utilizează instrucţiunea *FETCH* (cursoare) sau instrucţiunea de atribuire în cadrul unei secvenţe repetitive *LOOP*.

Pentru a şterge liniile unui tablou fie se asignează elementelor tabloului valoarea *null*, fie se declară un alt tablou *PL/SQL* (de acelaşi tip) care nu este iniţializat şi acest tablou vid se asignează tabloului *PL/SQL* care trebuie şters. În *PL/SQL* 2.3 ştergerea liniilor unui tabel se poate face utilizând metoda *DELETE*.

***Exemplu*:**

Să se definească un tablou indexat *PL/SQL* având elemente de tipul *NUMBER*. Să se introducă 20 de elemente în acest tablou. Să se şteargă tabloul.

```
DECLARE
  TYPE  tablou_numar IS TABLE OF NUMBER
    INDEX BY PLS_INTEGER;
  v_tablou  tablou_numar;
BEGIN
  FOR i IN 1..20 LOOP
    v_tablou(i) := i*i;
    DBMS_OUTPUT.PUT_LINE(v_tablou(i));
  END LOOP;
  --v_tablou := NULL;
  --aceasta atribuire da eroarea PLS-00382
  FOR i IN v_tablou.FIRST..v_tablou.LAST LOOP
    v_tablou(i) := NULL;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('tabloul are ' || v_tablou.COUNT ||
                       ' elemente');
END;
```

În *PL/SQL* este folosit frecvent tipul tablou de înregistrări. Referirea la un element al tabloului se face prin forma clasică: *tabel*(*index*).*câmp*.

***Exemplu*:**

Să se definească un tablou de înregistrări având tipul celor din tabelul *organizator*. Să se iniţializeze un element al tabloului şi să se introducă în tabelul *organizator*. Să se şteargă elementele tabloului.

```
DECLARE
  TYPE org_table_type IS TABLE OF organizator%ROWTYPE
    INDEX BY BINARY INTEGER;
  org_table  org_table_type;
  i          NUMBER;
BEGIN
  IF org_table.COUNT <>0 THEN
    i := org_table.LAST+1;
    ELSE i:=1;
  END IF;
  org_table(i).cod_org := 752;
  org_table(i).nume := 'Grigore Ion';
  org_table(i).adresa := 'Calea Plevnei 18 Sibiu';
  org_table(i).tip := 'persoana fizica';
  INSERT INTO organizator
  VALUES (org_table(i).cod_org, org_table(i).nume,
          org_table(i).adresa, org_table(i).tip);
  -- sau folosind noua facilitate Oracle9i
  -- INSERT INTO organizator
  -- VALUES (org_table(i));
  org_table.DELETE; -- sterge toate elementele
  DBMS_OUTPUT.PUT_LINE('Dupa aplicarea metodei DELETE
      sunt '||TO_CHAR(org_table.COUNT)||' elemente');
END;
```

Vectori

Vectorii (*varray*) sunt structuri asemănătoare vectorilor din limbajele *C* sau *Java*. Spre deosebire de tablourile indexate, vectorii au o dimensiune maximă (constantă) stabilită la declarare. În special, se utilizează pentru modelarea relaţiilor *one-to-many*, atunci când numărul maxim de elemente din partea „*many*“ este cunoscut şi ordinea elementelor este importantă.

Vectorii reprezintă **structuri dense**. Fiecare element are un index care dă poziţia sa în vector şi care este folosit pentru accesarea elementelor particulare. Limita inferioară a indicelui este 1. Vectorul poate conţine un număr variabil de elemente, de la 0 (vid) la numărul maxim specificat obligatoriu în definiţia sa.

Tipul de date vector este declarat utilizând sintaxa:

TYPE *nume_tip* ***IS***
{***VARRAY*** | ***VARYING ARRAY***} (*lungime_maximă*)
OF *tip_elemente* [***NOT NULL***];

Identificatorul *nume_tip* este numele tipului de date vector, iar *lungime_maximă* reprezintă numărul maxim de elemente din vector. *Tip_elemente* este un tip scalar *PL/SQL*, tip înregistrare sau tip obiect. De asemenea, acest tip poate fi definit utilizând atributele *%TYPE* sau *%ROWTYPE*.

În *Oracle9i* sunt permise (pentru *tip_elemente*) tipurile *TABLE* sau alt tip *VARRAY*. Există restricţii referitoare la tipul elementelor, în sensul că acesta nu poate să fie *BOOLEAN*, *NCHAR*, *NCLOB*, *NVARCHAR2*, *REF CURSOR*, *PLS_INTEGER*, *LONG*, *LONG RAW*, *NATURAL*, *NATURALN*, *POSITIVE*, *POSITIVEN*, *BINARY_INTEGER*, *SIGNTYPE*, *STRING*, tip obiect cu atribute *TABLE* sau *VARRAY*, *BLOB*, *CLOB*, tip obiect cu atribute *BLOB* sau *CLOB.*

***Exemplu*:**

```
DECLARE
  TYPE secventa IS VARRAY(5) OF VARCHAR2(10);
  v_sec  secventa := secventa ('alb', 'negru', 'rosu',
                               'verde');
BEGIN
  v_sec (3) := 'rosu';
  v_sec.EXTEND; -- adauga un element null
  v_sec(5) := 'albastru';
  -- extinderea la 6 elemente va genera eroarea ORA-06532
  v_sec.EXTEND;
END;
```

Tablouri imbricate

Tablourile imbricate (*nested table*) sunt tablouri indexate a căror dimensiune nu este stabilită. Numărul maxim de linii ale unui tablou imbricat este dat de capacitatea maximă 2 *GB.*

Un tablou imbricat este o mulţime neordonată de elemente de acelaşi tip. Valorile de acest tip:

-- pot fi stocate în baza de date,

-- pot fi prelucrate direct în instrucţiuni *SQL*

-- au excepţii predefinite proprii.

Sistemul *Oracle* nu stochează liniile unui tablou imbricat într-o ordine particulară. Dar, când se regăseşte tabloul în variabile *PL/SQL*, liniile vor avea indici consecutivi începând cu valoarea 1. Iniţial, aceste tablouri sunt structuri dense, dar se poate ca în urma prelucrării să nu mai aibă indici consecutivi.

Comanda de declarare a tipului de date tablou imbricat are sintaxa:

TYPE *nume_tip* ***IS TABLE OF*** *tip_ elemente* [***NOT NULL***];

Identificatorul *nume_tip* reprezintă numele noului tip de date tablou imbricat, iar *tip_elemente* este tipul fiecărui element din tabloul imbricat, care poate fi un tip definit de utilizator sau o expresie cu *%TYPE*, respectiv *%ROWTYPE*.

În *Oracle9i* sunt permise (pentru *tip_elemente*) tipurile *TABLE* sau alt tip *VARRAY*. Există restricţii referitoare la tipul elementelor, în sensul că acesta nu poate să fie *BOOLEAN*, *STRING*, *NCHAR*, *NCLOB*, *NVARCHAR2*, *REF CURSOR*, *BINARY_INTEGER*, *PLS_INTEGER*, *LONG*, *LONG RAW*, *NATURAL*, *NATURALN*, *POSITIVE*, *POSITIVEN*, *SIGNTYPE*, tip obiect cu atributele *TABLE* sau *VARRAY*.

Tabloul imbricat are o singură coloană, iar dacă aceasta este de tip obiect, tabloul poate fi vizualizat ca un tabel multicoloană, având câte o coloană pentru fiecare atribut al tipului obiect.

***Exemplu*:**

```
DECLARE
  TYPE numartab IS TABLE OF NUMBER;
  -- se creeaza un tablou cu un singur element
  v_tab_1  numartab := numartab(-7);
  -- se creeaza un tablou cu 4 elemente
  v_tab_2  numartab := numartab(7,9,4,5);
  -- se creeaza un tablou fara nici un element
  v_tab_3  numartab := numartab();
BEGIN
  v_tab_1(1) := 57;
  FOR j IN 1..4 LOOP
    DBMS_OUTPUT.PUT_LINE (v_tab_2(j) || ' ');
  END LOOP;
END;
```

Se observă că singura diferenţă sintactică între tablourile indexate şi cele imbricate este absenţa clauzei *INDEX BY BINARY_INTEGER*. Mai exact, dacă această clauză lipseşte, tipul este tablou imbricat.

***Observaţii*:**

- Spre deosebire de tablourile indexate, vectorii şi tablourile imbricate pot să apară în definirea tabelelor bazei de date.
- Tablourile indexate pot avea indice negativ, domeniul permis pentru index fiind –2147483647..2147483647, iar pentru tabele imbricate domeniul indexului este 1..2147483647.
- Tablourile imbricate, spre deosebire de tablourile indexate, pot fi prelucrate prin comenzi *SQL*.

- Tablourile imbricate trebuie iniţializate şi/sau extinse pentru a li se adăuga elemente.

Când este creat un tablou indexat care nu are încă elemente, el este vid. Dacă un tablou imbricat (sau un vector) este declarat, dar nu are încă nici un element (nu este iniţializat), el este automat iniţializat (atomic) *null*. Adică, colecţia este *null*, nu elementele sale. Prin urmare, pentru tablouri imbricate poate fi utilizat operatorul *IS NULL*. Dacă se încearcă **să se adauge** un element la un tablou imbricat *null*, se va genera eroarea „*ORA - 06531: reference to uninitialized collection*“ care corespunde excepţiei predefinite *COLLECTION_IS_NULL*.

Prin urmare, cum poate fi iniţializat un tablou imbricat? Ca şi obiectele, vectorii şi tablourile imbricate sunt iniţializate cu ajutorul **constructorului**. Acesta are acelaşi nume ca şi tipul colecţiei referite. *PL/SQL* apelează un constructor numai în mod explicit. Tabelele indexate nu au constructori.

Constructorul primeşte ca argumente o listă de valori de tip *tip_elemente*. Elementele sunt numerotate în ordine, de la 1 la numărul de valori date ca parametrii constructorului. Dimensiunea iniţială a colecţiei este egală cu numărul de argumente date în constructor, când aceasta este iniţializată. Pentru vectori nu poate fi depăşită dimensiunea maximă precizată la declarare. Atunci când constructorul este fără argumente, va crea o colectie fără nici un element (vida), dar care are valoarea *not null*. Exemplul următor este concludent în acest sens.

***Exemplu*:**

```
DECLARE
  TYPE  alfa IS TABLE OF VARCHAR2(50);
  -- creeaza un tablou (atomic) null
  tab1  alfa ;
  /* creeaza un tablou cu un element care este null, dar
     tabloul nu este null, el este initializat, poate
     primi elemente */
  tab2  alfa := alfa() ;
BEGIN

  IF tab1 IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('tab1 este NULL');
  ELSE
    DBMS_OUTPUT.PUT_LINE('tab1 este NOT NULL');
  END IF;
  IF tab2 IS NULL THEN
```

```
    DBMS_OUTPUT.PUT_LINE('tab2 este NULL');
  ELSE
    DBMS_OUTPUT.PUT_LINE('tab2 este NOT NULL');
  END IF;
END;
```

În urma execuţiei acestui bloc se obţine următorul rezultat:

```
tab1 este NULL
tab2 este NOT NULL
```

Excepţiile semnificative care apar în cazul utilizării incorecte a colecţiilor:

Exemplu:

```
DECLARE
  TYPE  numar IS TABLE OF INTEGER;
  alfa  numar;
BEGIN
  alfa(1) := 77;
  -- declanseaza exceptia COLLECTION_IS_NULL
  alfa := numar(15, 26, 37);
  alfa(1) := ASCII('X');
  alfa(2) := 10*alfa(1);
  alfa('P') := 77;
  /* declanseaza exceptia VALUE_ERROR deoarece indicele
     nu este convertibil la intreg */
  alfa(4) := 47;
  /* declanseaza exceptia SUBSCRIPT_BEYOND_COUNT deoarece
     indicele se refera la un element neinitializat */
  alfa(null) := 7; -- declanseaza exceptia VALUE_ERROR
  alfa(0) := 7; -- exceptia SUBSCRIPT_OUTSIDE_LIMIT
  alfa.DELETE(1);
  IF alfa(1) = 1 THEN … -- exceptia NO_DATA_FOUND
  …
END;
```

Tablourile imbricate şi vectorii pot fi utilizaţi drept câmpuri în tabelele bazei. Aceasta presupune că fiecare înregistrare din tabelul respectiv conţine un obiect de tip colecţie. Înainte de utilizare, tipul trebuie stocat în dicţionarul datelor, deci trebuie declarat prin comanda:

CREATE TYPE *nume_tip* ***AS*** {***TABLE*** | ***VARRAY***} ***OF*** *tip_elemente*;

După crearea tabelului (prin comanda *CREATE TABLE*), pentru fiecare câmp de tip tablou imbricat din tabel este necesară clauza de stocare:

NESTED TABLE *nume_câmp* ***STORE AS*** *nume_tabel*;

Colecţii pe mai multe niveluri

În *Oracle9i* se pot construi colecţii pe mai multe niveluri (*multilevel collections*), prin urmare colecţii ale căror elemente sunt, în mod direct sau indirect, colecţii. În felul acesta pot fi definite structuri complexe: vectori de vectori, vectori de tablouri imbricate, tablou imbricat de vectori, tablou imbricat de tablouri imbricate, tablou imbricat sau vector de un tip definit de utilizator care are un atribut de tip tablou imbricat sau vector.

Aceste structuri complexe pot fi utilizate ca tipuri de date pentru definirea:

- coloanelor unui tabel relaţional,
- atributelor unui obiect într-un tabel obiect,
- variabilelor *PL/SQL*.

Observaţii:

- Numărul nivelurilor de imbricare este limitat doar de capacitatea de stocare a sistemului.
- Pentru a accesa un element al colecţiei incluse sunt utilizate două seturi de paranteze.
- Obiectele de tipul colecţie pe mai multe niveluri nu pot fi comparate.

***Exemplu*:**

În exemplele care urmează sunt definite trei structuri complexe şi sunt prezentate câteva modalităţi de utilizare ale acestora. Exemplele se referă la vectori pe mai multe niveluri, tablouri imbricate pe mai multe niveluri şi tablouri indexate pe mai multe niveluri.

```
DECLARE
  TYPE  alfa IS VARRAY(10) OF INTEGER;
  TYPE  beta IS VARRAY(10) OF alfa;
  valf  alfa := alfa(12,31,5); --initializare
  vbet  beta := beta(valf,alfa(55,6,77),alfa(2,4),valf);
  i     integer;
  var1  alfa;
BEGIN
  i := vbet(2)(3); -- i va lua valoarea 77
  vbet.EXTEND; -- se adauga un element de tip vector la vbet
  vbet(5) := alfa(56,33);
  vbet(4) := alfa(44,66,77,4321);
  vbet(4)(4) := 7; -- 4321 este inlocuit cu 7
  vbet(4).EXTEND; -- se adauga un element la al 4-lea element
  vbet(4)(5) := 777; -- acest nou element adaugat va fi 777
END;
```

```
/
DECLARE
  TYPE  gama IS TABLE OF VARCHAR2(20);
  TYPE  delta IS TABLE OF gama;
  TYPE  teta IS VARRAY(10) OF INTEGER;
  TYPE  epsi IS TABLE OF teta;
  var1  gama := gama('alb','negru');
  var2  delta := delta(var1);
  var3  epsi := epsi(teta(31,15),teta(1,3,5));
BEGIN
  var2.EXTEND;
  var2(2) := var2(1);
  var2.DELETE(1); -- sterge primul element din var2
  /* sterge primul sir de caractere din al doilea
     tablou al tabloului imbricat */
  var2(2).DELETE(1);
END;
/
DECLARE
  TYPE  alfa IS TABLE OF INTEGER INDEX BY BINARY_INTEGER;
  TYPE  beta IS TABLE OF alfa INDEX BY BINARY_INTEGER;
  TYPE  gama IS VARRAY(10) OF VARCHAR2(30);
  TYPE  delt IS TABLE OF gama INDEX BY BINARY_INTEGER;
  var1  gama := gama('alb','negru');
  var2  beta;
  var3  delt;
  var4  alfa;
  var5  alfa; -- tablou vid
BEGIN
  var4(1) := 324;
  var4(2) := 222;
  var4(42) := 333;
  var2(27) := var4;
  var3(39) := gama(77,76,89,908);
  -- var2(40)(3) := 55; eroare nu exista element 40 in var2
  var2(40) := var5; -- asignez un tablou null
  var2(40)(3) := 55; -- corect
END;
/
```

Prelucrarea colecţiilor

O colecţie poate fi exploatată fie în întregime (atomic) utilizând comenzi *LMD*, fie pot fi prelucrate elemente individuale dintr-o colecţie (*piecewise updates*) utilizând operatori *SQL* sau anumite facilităţi oferite de *PL/SQL*.

- Comanda *INSERT* permite inserarea unei colecţii într-o linie a unui tabel. Colecţia trebuie să fie creată şi iniţializată anterior.
- Comanda *UPDATE* este folosită pentru modificarea unei colecţii stocate.
- Comanda *DELETE* poate şterge o linie ce conţine o colecţie.
- Colecţiile din baza de date pot fi regăsite în variabile *PL/SQL*, utilizând comanda *SELECT*.

***Exemplu*:**

```
CREATE OR REPLACE TYPE operalist AS VARRAY(10) OF
                                      NUMBER(4);
CREATE TABLE gal_ope (
  cod_galerie   NUMBER(10),
  nume_galerie  VARCHAR2(20),
  info          operalist);
DECLARE
  v_opera    operalist := operalist (777, 888, 999);
  v_info_op  operalist := operalist (7007);
  v_info     gal_ope.info%TYPE;
  v_cod      gal_ope.cod_galerie%TYPE := 2345;
BEGIN
  INSERT INTO gal_ope
  VALUES (4567, 'Impresionisti', operalist(4567,4987));
  INSERT INTO gal_ope
  VALUES (2345, 'Cubism', v_opera);
  INSERT INTO gal_ope
  VALUES (123, 'Alfa', v_info_op);
  SELECT info
  INTO   v_info
  FROM   gal_ope
  WHERE  cod_galerie = v_cod;
END;
```

Un vector stocat într-un tabel este prelucrat ca un întreg (nu pot fi modificate elemente individuale). Prin urmare, elementele individuale ale unui vector nu pot fi referite în comenzile *INSERT*, *UPDATE* sau *DELETE*. Pentru referirea acestora trebuie utilizate comenzi procedurale *PL/SQL*. Pentru a modifica un vector, el trebuie selectat într-o variabilă *PL/SQL* a cărei valoare poate fi modificată şi apoi reinserată în tabel.

Tablourile imbricate depuse în baza de date sunt mai flexibile, deoarece pot fi prelucrate fie în întregime, fie ca elemente individuale. În fiecare caz pot fi utilizate **numai** comenzi *SQL*.

Se pot face reactualizări sau inserări asupra tablourilor imbricate care dau o valoare nouă pentru întreaga colecţie sau se pot face inserări, ştergeri, reactualizări de elemente particulare din colecţie.

O colecţie poate fi asignată altei colecţii prin comenzile *INSERT*, *UPDATE*, *FETCH*, *SELECT*, instrucţiunea de atribuire sau prin apelul unui subprogram, dar colecţiile trebuie să fie de acelaşi tip. Dacă unei colecţii i se asignează o colecţie atomic *null*, aceasta devine atomic *null* şi trebuie reiniţializată.

În *Oracle8i* a fost introdus operatorul *TABLE*, ce permite **prelucrarea elementelor unui tablou imbricat care este stocat într-un tabel**. Operatorul permite interogarea unei colecţii în clauza *FROM* (la fel ca un tabel).

Operandul lui *TABLE* este:

- fie numele unei colecţii şi atunci rezultatul operatorului este tot o colecţie,
- fie este o subinterogare referitoare la o colecţie, iar în acest caz, operatorul *TABLE* returnează o singură valoare (coloană) care este un tablou imbricat sau un vector. Prin urmare, lista din clauza *SELECT* a subcererii trebuie să aibă un singur articol.

***Exemplu*:**

Se presupune că tabelul *opera* are o coloană *info* de tip tablou imbricat. Acest tablou are două componente în care pentru fiecare operă de artă sunt depuse numele articolului referitor la opera respectivă şi revista în care a apărut. Să se insereze o linie în tabelul imbricat.

```
INSERT INTO TABLE (SELECT info
                   FROM   opera
                   WHERE  titlu = 'Primavara')
VALUES ('Pictura moderna', 'Orizonturi');
```

Listarea codului fiecărei opere de artă şi a colecţiei articolelor referitoare la aceste opere de artă se face prin comanda:

```
SELECT  a.cod_opera, b.*
FROM    opera a, TABLE (a.info) b;
```

Pentru tablouri imbricate pe mai multe niveluri, operaţiile *LMD* pot fi făcute atomic sau pe elemente individuale, iar pentru vectori pe mai multe niveluri, operaţiile pot fi făcute numai atomic.

Pentru prelucrarea unei colecţii locale se poate folosi şi operatorul *CAST*. *CAST* are forma sintactică:

CAST (*nume_colecţie* ***AS*** *tip_colecţie*)

Operanzii lui *CAST* sunt o colecţie declarată local (de exemplu, într-un bloc *PL/SQL* anonim) şi un tip colecţie *SQL*. *CAST* converteşte colecţia locală la tipul specificat. În felul acesta, o colecţie poate fi prelucrată ca şi cum ar fi un tabel *SQL* al bazei de date.

Metodele unei colecţii

PL/SQL oferă subprograme numite metode (*methods*), care operează asupra unei colecţii. Acestea pot fi apelate **numai din comenzi procedurale**, şi nu din *SQL*.

Metodele sunt apelate prin expresia:

nume_colecţie.nume_metodă [(*parametri*)]

Metodele care se pot aplica colecţiilor *PL/SQL* sunt următoarele:

COUNT returnează numărul curent de elemente ale unei colecţii *PL/SQL*;

***DELETE*(*n*)** şterge elementul *n* dintr-o colecţie *PL/SQL*; *DELETE*(*m, n*) şterge toate elementele având indecşii între *m* şi *n*; *DELETE* şterge toate elementele unei colecţii *PL/SQL* (nu este validă pentru tipul *varrays*);

***EXISTS*(*n*)** returnează *TRUE* dacă există al *n*-lea element al unei colecţii *PL/SQL* (altfel, returnează *FALSE*, chiar dacă elementul este *null*);

FIRST*, *LAST returnează indicele primului, respectiv ultimului element din colecţie;

***NEXT*(*n*)**, ***PRIOR*(*n*)** returnează indicele elementului următor, respectiv precedent celui de rang *n* din colecţie, iar dacă nu există un astfel de element returnează valoarea *null*;

EXTEND adaugă elemente la sfârşitul unei colecţii: *EXTEND* adaugă un element *null* la sfârşitul colecţiei, *EXTEND*(*n*) adaugă *n* elemente *null*, *EXTEND*(*n, i*) adaugă *n* copii ale elementului de rang *i* (nu este validă pentru tipul *index-by tables*); nu poate fi utilizată pentru a iniţializa o colecţie atomic *null*;

LIMIT returnează numărul maxim de elemente ale unei colecţii (cel de la declarare) pentru tipul vector şi *null* pentru tablouri imbricate (nu este validă pentru tipul *index-by tables*);

TRIM şterge elementele de la sfârşitul unei colecţii: *TRIM* şterge ultimul element, *TRIM*(*n*) şterge ultimele *n* elemente (nu este validă pentru tipul *index-by tables*). Similar metodei *EXTEND*, metoda *TRIM* operează asupra dimensiunii interne a tabloului imbricat.

EXISTS este singura metodă care poate fi aplicată unei colecţii atomice *null*. Orice altă metodă declanşează excepţia *COLLECTION_IS_NULL*.

Bulk bind

În exemplul care urmează, comanda *DELETE* este trimisă motorului *SQL* pentru fiecare iteraţie a comenzii *FOR*.

Exemplu:

```
DECLARE
  TYPE nume IS VARRAY(20) OF NUMBER;
  alfa nume := nume(10,20,70); -- coduri ale galeriilor
BEGIN
  FOR j IN alfa.FIRST..alfa.LAST
    DELETE FROM opera
    WHERE  cod_galerie = alfa (j);
  END LOOP;
END;
```

Pentru a realiza mai rapid această operaţie, ar trebui să existe posibilitatea de a şterge (prelucra) întreaga colecţie şi nu elemente individuale. Tehnica care permite acest lucru este cunoscută sub numele ***bulk bind***.

În timpul compilării, compilatorul *PL/SQL* asociază identificatorii cu o adresă, un tip de date şi o valoare. Acest proces este numit *binding*.

Comenzile *SQL* din blocurile *PL/SQL* sunt trimise motorului *SQL* pentru a fi executate. Motorul *SQL* poate trimite înapoi date motorului *PL/SQL* (de exemplu, ca rezultat al unei interogări). De multe ori, datele care trebuie manipulate aparţin unei colecţii, iar colecţia este iterată printr-un ciclu *FOR*. Prin urmare, transferul (în ambele sensuri) între *SQL* şi *PL/SQL* are loc pentru fiecare linie a colecţiei.

Începând cu *Oracle8i* există posibilitatea ca toate liniile unei colecţii să fie transferate simultan printr-o singură operaţie. Procedeul este numit *bulk bind* şi este realizat cu ajutorul comenzii *FORALL*, ce poate fi folosită cu orice tip de colecţie.

Comanda *FORALL* are sintaxa:

FORALL *index* ***IN*** *lim_inf..lim_sup*
comanda_sql;

Motorul *SQL* execută *comanda_sql* o singură dată pentru toate valorile indexului. *Comanda_sql* este una din comenzile *INSERT*, *UPDATE*, *DELETE* care referă elementele uneia sau mai multor colecţii. Variabila *index* poate fi referită numai în comanda *FORALL* şi numai ca indice de colecţie.

În exemplul care urmează este optimizată problema anterioară, în sensul că instrucţiunea *DELETE* este trimisă motorului *SQL* o singură dată, pentru toate liniile colecţiei.

Exemplu:

```
DECLARE
  TYPE nume IS VARRAY(20) OF NUMBER;
  alfa nume := nume(10,20,70); -- coduri ale galeriilor
BEGIN
  …
  FORALL j IN alfa.FIRST..alfa.LAST
    DELETE FROM opera
    WHERE  cod_galerie = alfa (j);
END;
```

Pentru utilizarea comenzii *FORALL* este necesară respectarea următoarelor restricţii:

- comanda poate fi folosită numai în programe *server-side*, altfel apare eroarea "*this feature is not supported in client-side programs*";
- comenziile *INSERT*, *UPDATE*, *DELETE* trebuie să refere cel puţin o colecţie;
- toate elementele colecţiei din domeniul precizat trebuie să existe (dacă, de exemplu, un element a fost şters, atunci este semnalată o eroare);
- indicii colecţiilor nu pot să fie expresii şi trebuie să aibă valori continue.

Exemplu:

```
CREATE TABLE exemplu (x NUMBER, y NUMBER);
DECLARE
  TYPE  nume IS TABLE OF NUMBER;
  ttt   nume:= nume(1,2,3);
BEGIN
  FORALL i IN ttt.FIRST..ttt.LAST
    INSERT INTO exemplu
    VALUES(ttt(i), 10);  -- corect
  FORALL i IN 1..3
    INSERT INTO exemplu
    VALUES(7, 9);        -- exceptie
END;


  FORALL i IN gama.FIRST..gama.LAST
    DELETE FROM carte
    WHERE codel = gama(i+1);
  -- eroare dubla (expresie si >LAST)
```

```
DECLARE
  TYPE alfa IS TABLE OF NUMBER;
  xx   alfa := alfa(10,20,30,40);
BEGIN
  xx.DELETE(3);
  FORALL i IN xx.FIRST..xx.LAST
    DELETE FROM carte
    WHERE codel = xx(i);  -- eroare
END;
```

Dacă există o eroare în procesarea unei linii printr-o operaţie *LMD* de tip *bulk*, numai acea linie va fi *rollback*.

Cursorul *SQL* are un atribut compus *%BULK_ROWCOUNT* care numără liniile afectate de iteraţiile comenzii *FORALL*. *%BULK_ROWCOUNT*(*i*) reprezintă numărul de linii procesate de a *i*-a execuţie a comenzii *SQL*. Dacă nu este afectată nici o linie, valoarea atributului este 0. *%BULK_ROWCOUNT* nu poate să fie parametru în subprograme şi nu poate fi asignat altor colecţii.

Începând cu *Oracle9i* este utilizabilă o nouă clauză în comanda *FORALL*. Clauza, numită *SAVE EXCEPTIONS*, permite ca toate excepţiile care apar în timpul execuţiei comenzii *FORALL* să fie salvate şi astfel procesarea poate să continue. În acest context, poate fi utilizat atributul cursor *%BULK_EXCEPTIONS* pentru a vizualiza informaţii despre aceste excepţii.

Atributul acţionează ca un tablou *PL/SQL* şi are două câmpuri:

- *%BULK_EXCEPTIONS(i).ERROR_INDEX*, reprezentând iteraţia în timpul căreia s-a declanşat excepţia;
- *%BULK_EXCEPTIONS(i).ERROR_CODE*, reprezentând codul *Oracle* al erorii respective.

Regăsirea rezultatului unei interogări în colecţii (înainte de a fi trimisă motorului *PL/SQL*) se poate obţine cu ajutorul clauzei *BULK COLLECT*.

Clauza poate să apară în:

- comenzile *SELECT INTO* (cursoare implicite),
- comenzile *FETCH INTO* (cursoare explicite),
- clauza *RETURNING INTO* a comenzilor *INSERT*, *UPDATE*, *DELETE*.

Clauza are următoarea sintaxă:

*...**BULK COLLECT INTO** nume_colecţie* [,*nume_colecţie*]...

```
DECLARE
  TYPE tip1 IS TABLE OF opera.cod_opera%TYPE;
  TYPE tip2 IS TABLE OF opera.titlu%TYPE;
  alfa  tip1;
  beta  tip2;
BEGIN
  …
  /* motorul SQL incarca in intregime coloanele
     cod_opera si titlu in tabelele imbricate,
     inainte de a returna
     tabelele motorului PL/SQL */
  SELECT cod_opera, titlu BULK COLLECT INTO alfa,beta
  FROM   opera;
  …
  /* daca exista n opere de arta in stare buna,
     atunci alfa va contine codurile celor n opere */
  DELETE FROM opera WHERE stare = 'buna'
  RETURNING cod_opera BULK COLLECT INTO alfa;
  …
END;
```

Comanda *FORALL* se poate combina cu clauza *BULK COLLECT*. Totuşi, trebuie subliniat că ele nu pot fi folosite simultan în comanda *SELECT*.

Motorul SQL incarca toate liniile unei coloane. Cum se poate limita numarul de linii procesate? Exemplelele urmatoare dau 2 variante de rezolvare.

```
DECLARE
  TYPE alfa IS TABLE OF carte.pret%TYPE;
  xx   alfa;
BEGIN
  SELECT pret BULK COLLECT INTO xx
  FROM   carte WHERE  ROWNUM <= 500;
END;

DECLARE
  …
  zz NATURAL := 10;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 BULK COLLECT INTO xx, yy LIMIT zz;
    EXIT WHEN c1%NOTFOUND;
    …
END;
```

Gestiunea cursoarelor în *PL/SQL*

Sistemul *Oracle* foloseşte, pentru a procesa o comandă *SQL*, o zonă de memorie cunoscută sub numele de zonă context (*context area*). Când este procesată o instrucţiune *SQL*, *server*-ul *Oracle* deschide această zonă de memorie în care comanda este analizată sintactic şi este executată.

Zona conţine informaţii necesare procesării comenzii, cum ar fi:

- numărul de rânduri procesate de instrucţiune;
- un *pointer* către reprezentarea internă a comenzii;
- în cazul unei cereri, mulţimea rândurilor rezultate în urma execuţiei acestei comenzi (*active set*).

Un cursor este un *pointer* la această zonă context. Prin intermediul cursoarelor, un program *PL/SQL* poate controla zona context şi transformările petrecute în urma procesării comenzii.

Există două tipuri de cursoare:

- implicite, generate de *server*-ul *Oracle* când în partea executabilă a unui bloc *PL/SQL* apare o instrucţiune *SQL*;
- explicite, declarate şi definite de către utilizator atunci când o cerere (*SELECT*), care apare într-un bloc *PL/SQL*, întoarce mai multe linii ca rezultat.

Atât cursoarele implicite cât şi cele explicite au o serie de atribute ale căror valori pot fi folosite în expresii. Lista atributelor este următoarea:

- %*ROWCOUNT*, care este de tip întreg şi reprezintă numărul liniilor încărcate de cursor;
- %*FOUND*, care este de tip boolean şi ia valoarea *TRUE* dacă ultima operaţie de încărcare (*FETCH*) dintr-un cursor a avut succes (în cazul cursoarelor explicite) sau dacă instrucţiunea *SQL* a întors cel puţin o linie (în cazul cursoarelor implicite);
- %*NOTFOUND*, care este de tip boolean şi are semnificaţie opusă faţă de cea a atributului %*FOUND*;
- %*ISOPEN*, care este de tip boolean şi indică dacă un cursor este deschis (în cazul cursoarelor implicite, acest atribut are întotdeauna valoarea *FALSE*, deoarece un cursor implicit este închis de sistem imediat după executarea instrucţiunii *SQL* asociate).

Atributele pot fi referite prin expresia *SQL%nume_atribut*, în cazul cursoarelor implicite, sau prin *nume_cursor%nume_atribut*, în cazul unui cursor explicit. Ele pot să apară în comenzi *PL/SQL*, în funcţii, în secţiunea de tratare a erorilor, dar nu pot fi utilizate în comenzi *SQL*.

Cursoare implicite

Când se procesează o comandă *LMD*, motorul *SQL* deschide un cursor implicit. Atributele scalare ale cursorului implicit (*SQL%ROWCOUNT*, *SQL%FOUND*, *SQL%NOTFOUND*, *SQL%ISOPEN*) furnizează informaţii referitoare la ultima comandă *INSERT*, *UPDATE*, *DELETE* sau *SELECT INTO* executată. Înainte ca *Oracle* să deschidă cursorul *SQL* implicit, atributele acestuia au valoarea *null*.

În *Oracle9i*, pentru cursoare implicite a fost introdus atributul compus *%BULK_ROWCOUNT*, care este asociat comenzii *FORALL*. Atributul are semantica unui tablou indexat. Componenta *%BULK_ROWCOUNT*(*j*) conţine numărul de linii procesate de a *j*-a execuţie a unei comenzi *INSERT*, *DELETE* sau *UPDATE*. Dacă a *j*-a execuţie nu afectează nici o linie, atunci atributul returnează valoarea 0. Comanda *FORALL* şi atributul *%BULK_ROWCOUNT* au aceiaşi indici, deci folosesc acelaşi domeniu. Dacă *%BULK_ROWCOUNT*(*j*) este zero, atributul *%FOUND* este *FALSE*.

***Exemplu*:**

În exemplul care urmează, comanda *FORALL* inserează un număr arbitrar de linii la fiecare iteraţie, iar după fiecare iteraţie atributul *%BULK_ROWCOUNT* returnează numărul acestor linii inserate.

```
SET SERVEROUTPUT ON
DECLARE
  TYPE alfa IS TABLE OF NUMBER;
  beta alfa;
BEGIN
  SELECT cod_artist BULK COLLECT INTO beta FROM artist;
  FORALL j IN 1..beta.COUNT
    INSERT INTO tab_art
      SELECT cod_artist,cod_opera
      FROM   opera
      WHERE  cod_artist = beta(j);
  FOR j IN 1..beta.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE ('Pentru artistul avand codul ' ||
      beta(j) || ' au fost inserate ' ||
      SQL%BULK_ROWCOUNT(j)
      || inregistrari (opere de arta)');
  END LOOP;
  DBMS_OUTPUT.PUT_LINE ('Numarul total de inregistrari
      inserate este '||SQL%ROWCOUNT);
END;
/
SET SERVEROUTPUT OFF
```

Cursoare explicite

Pentru gestiunea cursoarelor explicite sunt necesare următoarele etape:

- declararea cursorului (atribuirea unui nume şi asocierea cu o comandă *SELECT*);
- deschiderea cursorului pentru cerere (executarea interogării asociate şi determinarea mulţimii rezultat);
- recuperarea liniilor rezultatului în variabile *PL/SQL*;
- închiderea cursorului (eliberarea resurselor relative la cursor).

Prin urmare, pentru a utiliza un cursor, el trebuie declarat în secţiunea declarativă a programului, trebuie deschis în partea executabilă, urmând să fie utilizat apoi pentru extragerea datelor. Dacă nu mai este necesar în restul programului, cursorul trebuie să fie închis.

DECLARE
    declarare cursor
BEGIN
    deschidere cursor (***OPEN***)
    WHILE *rămân linii de recuperat* ***LOOP***
        recuperare linie rezultat (***FETCH***)
        ...
    END LOOP
    închidere cursor (***CLOSE***)
    ...
END;

Pentru a controla activitatea unui cursor sunt utilizate comenzile *DECLARE*, *OPEN*, *FETCH* şi *CLOSE*.

Declararea unui cursor explicit

Prin declaraţia *CURSOR* în cadrul comenzii *DECLARE* este definit un cursor explicit şi este precizată structura cererii care va fi asociată acestuia.

Declaraţia *CURSOR* are următoarea formă sintactică:

CURSOR *nume_cursor* ***IS*** *comanda_select*

Identificatorul *nume_cursor* este numele cursorului, iar *comanda_select* este cererea *SELECT* care va fi procesată.

Observaţii:

- Comanda *SELECT* care apare în declararea cursorului, nu trebuie să includă clauza *INTO*.
- Dacă se cere procesarea liniilor într-o anumită ordine, atunci în cerere este utilizată clauza *ORDER BY*.

- Variabilele care sunt referite în comanda de selectare trebuie declarate înaintea comenzii *CURSOR*. Ele sunt considerate variabile de legătură.
- Dacă în lista comenzii *SELECT* apare o expresie, atunci pentru expresia respectivă trebuie utilizat un *alias*, iar câmpul expresie se va referi prin acest *alias*.
- Numele cursorului este un identificator unic în cadrul blocului, care nu poate să apară într-o expresie şi căruia nu i se poate atribui o valoare.

Deschiderea unui cursor explicit

Comanda *OPEN* execută cererea asociată cursorului, identifică mulţimea liniilor rezultat şi poziţionează cursorul înaintea primei linii.

Deschiderea unui cursor se face prin comanda:

OPEN *nume_cursor*;

Identificatorul *nume_cursor* reprezintă numele cursorului ce va fi deschis.

La deschiderea unui cursor se realizează următoarele operaţii:

- se evaluează cererea asociată (sunt examinate valorile variabilelor de legătură ce apar în declaraţia cursorului);
- este determinată mulţimea rezultat (*active set*) prin executarea cererii *SELECT*, având în vedere valorile de la pasul anterior;
- *pointer*-ul este poziţionat la prima linie din mulţimea activă.

Încărcarea datelor dintr-un cursor explicit

Comanda *FETCH* regăseşte liniile rezultatului din mulţimea activă. *FETCH* realizează următoarele operaţii:

- avansează *pointer*-ul la următoarea linie în mulţimea activă (*pointer*-ul poate avea doar un sens de deplasare de la prima spre ultima înregistrare);
- citeşte datele liniei curente în variabile *PL/SQL*;
- dacă *pointer*-ul este poziţionat la sfârşitul mulţimii active atunci se iese din bucla cursorului.

Comanda *FETCH* are următoarea sintaxă:

FETCH *nume_cursor*
INTO {*nume_variabilă* [, *nume_variabilă*] … | *nume_înregistrare*};

Identificatorul *nume_cursor* reprezintă numele unui cursor declarat şi deschis anterior. Variabila sau lista de variabile din clauza *INTO* trebuie să fie compatibilă (ca ordine şi tip) cu lista selectată din cererea asociată cursorului.

La un moment dat, comanda *FETCH* regăseşte o singură linie. Totuşi, în ultimele versiuni *Oracle* pot fi încărcate mai multe linii (la un moment dat) într-o colecţie, utilizând clauza *BULK COLLECT*.

***Exemplu*:**

În exemplul care urmează se încarcă date dintr-un cursor în două colecţii.

```
DECLARE
  TYPE   ccopera IS TABLE OF opera.cod_opera%TYPE;
  TYPE   ctopera IS TABLE OF opera.titlu%TYPE;
  cod1   ccopera;
  titlu1 ctopera;
  CURSOR alfa IS SELECT cod_opera, titlu
                 FROM   opera
                 WHERE  stil = 'impresionism';
BEGIN
  OPEN alfa;
  FETCH alfa BULK COLLECT INTO cod1, titlu1;
  …
  CLOSE alfa;
END;
```

Închiderea unui cursor explicit

După ce a fost procesată mulţimea activă, cursorul trebuie închis. Prin această operaţie, *PL/SQL* este informat că programul a terminat folosirea cursorului şi resursele asociate acestuia pot fi eliberate. Aceste resurse includ spaţiul utilizat pentru memorarea mulţimii active şi spaţiul temporar folosit pentru determinarea mulţimii active.

Cursorul va fi închis prin comanda *CLOSE*, care are următoarea sintaxă:

CLOSE *nume_cursor*;

Identificatorul *nume_cursor* este numele unui cursor deschis anterior.

Pentru a reutiliza cursorul este suficient ca acesta să fie redeschis. Dacă se încearcă încărcarea datelor dintr-un cursor închis, atunci apare excepţia *INVALID_CURSOR*. Un bloc *PL/SQL* poate să se termine fără a închide cursoarele, dar acest lucru nu este indicat, deoarece este bine ca resursele să fie eliberate.

***Exemplu*:**

Pentru toţi artiştii care au opere de artă expuse în muzeu să se insereze în tabelul *temp* informaţii referitoare la numele acestora şi anul naşterii.

```
DECLARE
  v_nume     artist.nume%TYPE;
  v_an_nas   artist.an_nastere%TYPE;
  CURSOR info IS
    SELECT  DISTINCT nume, an_nastere
    FROM    artist;
BEGIN
```

```
  OPEN info;
  LOOP
    FETCH info INTO v_nume, v_an_nas;
    EXIT WHEN info%NOTFOUND;
    INSERT INTO temp
    VALUES (v_nume || TO_CHAR(v_an_nas));
  END LOOP;
  CLOSE info;
  COMMIT;
END;
```

Valorile atributelor unui cursor explicit sunt prezentate în următorul tabel:

		%FOUND	***%ISOPEN***	***%NOTFOUND***	***%ROWCOUNT***
OPEN	Înainte	Excepţie	*False*	Excepţie	Excepţie
	După	*Null*	*True*	*Null*	0
Prima Încărcare	Înainte	*Null*	*True*	*Null*	0
	După	*True*	*True*	*False*	1
Următoarea încărcare	Înainte	*True*	*True*	*False*	1
	După	*True*	*True*	*False*	Depinde de date
Ultima încărcare	Înainte	*True*	*True*	*False*	Depinde de date
	După	*False*	*True*	*True*	Depinde de date
CLOSE	Înainte	*False*	*True*	*True*	Depinde de date
	După	Excepţie	*False*	Excepţie	Excepţie

După prima încărcare, dacă mulţimea rezultat este vidă, *%FOUND* va fi *FALSE*, *%NOTFOUND* va fi *TRUE*, iar *%ROWCOUNT* este 0.

Într-un pachet poate fi separată specificarea unui cursor de corpul acestuia. Cursorul va fi declarat în specificaţia pachetului prin comanda:

CURSOR *nume_cursor* [(*parametru* [, *parametru*]…)]
RETURN *tip_returnat*;

În felul acesta va creşte flexibilitatea programului, putând fi modificat doar corpul cursorului, fără a schimba specificaţia.

***Exemplu*:**

```
CREATE PACKAGE exemplu AS
  CURSOR alfa (p_valoare_min  NUMBER) RETURN opera%ROWTYPE;
    -- declaratie specificatie cursor
  …
END exemplu;
```

```
CREATE PACKAGE BODY exemplu AS
  CURSOR alfa (p_valoare_min  NUMBER) RETURN opera%ROWTYPE
IS
    SELECT * FROM opera WHERE valoare > p_valoare_min;
    -- definire corp cursor
  …
END exemplu;
```

Procesarea liniilor unui cursor explicit

Pentru procesarea diferitelor linii ale unui cursor explicit se foloseşte operaţia de ciclare (*LOOP*, *WHILE*, *FOR*), prin care la fiecare iteraţie se va încărca o nouă linie. Comanda *EXIT* poate fi utilizată pentru ieşirea din ciclu, iar valoarea atributului *%ROWCOUNT* pentru terminarea ciclului.

Procesarea liniilor unui cursor explicit se poate realiza şi cu ajutorul unui ciclu *FOR* special, numit ciclu cursor. Pentru acest ciclu este necesară doar declararea cursorului, operaţiile de deschidere, încărcare şi închidere ale acestuia fiind implicite.

Comanda are următoarea sintaxă:

FOR** nume_înregistrare **IN** nume_cursor **LOOP
secvenţă_de_instrucţiuni;
END LOOP;

Variabila *nume_înregistrare* (care controlează ciclul) nu trebuie declarată. Domeniul ei este doar ciclul respectiv.

Pot fi utilizate cicluri cursor speciale care folosesc subcereri, iar în acest caz nu mai este necesară nici declararea cursorului. Exemplul care urmează este concludent în acest sens.

Exemplu:

Să se calculeze, utilizând un ciclu cursor cu subcereri, valoarea operelor de artă expuse într-o galerie al cărei cod este introdus de la tastatură. De asemenea, să se obţină media valorilor operelor de artă expuse în galeria respectivă.

```
SET SERVEROUTPUT ON
ACCEPT p_galerie PROMPT 'Dati codul galeriei:'
DECLARE
  v_cod_galerie  galerie.cod_galerie%TYPE:=&p_galerie;
  val    NUMBER;
  media  NUMBER;
  i      INTEGER;
BEGIN
  val:=0;
  i:=0;
  FOR numar_opera IN
```

```
            (SELECT  cod_opera, valoare
             FROM    opera
             WHERE   cod_galerie = v_cod_galerie) LOOP
        val := val + numar_opera.valoare;
        i := i+1;
  END LOOP;--închidere implicită
  DBMS_OUTPUT.PUT_LINE('Valoarea operelor de arta din
     galeria cu numarul ' || TO_CHAR(v_cod_galerie) || '
     este ' || TO_CHAR(val));
  IF i=0 THEN
     DBMS_OUTPUT.PUT_LINE('Galeria nu are opere de arta');
       ELSE
     media := val/i;
     DBMS_OUTPUT.PUT_LINE('Media valorilor operelor de arta
        din galeria cu numarul ' || TO_CHAR(v_cod_galerie)
        || ' este ' || TO_CHAR(media));
  END IF;
END;
/
SET SERVEROUTPUT OFF
```

Cursoare parametrizate

Unei variabile de tip cursor îi corespunde o comandă *SELECT*, care nu poate fi schimbată pe parcursul programului. Pentru a putea lucra cu nişte cursoare ale căror comenzi *SELECT* ataşate depind de parametri ce pot fi modificaţi la momentul execuţiei, în *PL/SQL* s-a introdus noţiunea de cursor parametrizat. Prin urmare, un cursor parametrizat este un cursor în care comanda *SELECT* ataşată depinde de unul sau mai mulţi parametri.

Transmiterea de parametri unui cursor parametrizat se face în mod similar procedurilor stocate. Un astfel de cursor este mult mai uşor de interpretat şi de întreţinut, oferind şi posibilitatea reutilizării sale în blocul *PL/SQL*.

Declararea unui astfel de cursor se face respectând următoarea sintaxă:

CURSOR *nume_cursor* [(*nume_parametru*[*, nume_parametru ...*])]
[***RETURN*** *tip_returnat*]
IS *comanda_select*;

Identificatorul *comanda_select* este o instrucţiune *SELECT* fără clauza *INTO*, *tip_returnat* reprezintă un tip înregistrare sau linie de tabel, iar *nume_parametru* are sintaxa:

nume_parametru [***IN***] *tip_parametru* [{:= | ***DEFAULT***} *expresie*]

În această declaraţie, atributul *tip_parametru* reprezintă tipul parametrului,

care este un tip scalar. Parametrii formali sunt de tip *IN* şi, prin urmare, nu pot returna valori parametrilor actuali. Ei nu suportă constrângerea *NOT NULL*.

Deschiderea unui astfel de cursor se face asemănător apelului unei funcţii, specificând lista parametrilor actuali ai cursorului. În determinarea mulţimii active se vor folosi valorile actuale ale acestor parametri.

Sintaxa pentru deschiderea unui cursor parametrizat este:

OPEN *nume_cursor* [(*valoare_parametru* [, *valoare_parametru*] ...)];

Parametrii sunt specificaţi similar celor de la subprograme. Asocierea dintre parametrii formali şi cei actuali se face prin:

- poziţie – parametrii formali şi actuali sunt separaţi prin virgulă;
- nume – parametrii actuali sunt aranjaţi într-o ordine arbitrară, dar cu o corespondenţă de forma *parametru formal => parametru actual*.

Dacă în definiţia cursorului, toţi parametrii au valori implicite (*DEFAULT*), cursorul poate fi deschis fără a specifica vreun parametru.

***Exemplu*:**

Utilizând un cursor parametrizat să se obţină codurile operelor de artă din fiecare sală, identificatorul sălii şi al galeriei. Rezultatele să fie inserate în tabelul *mesaje*.

```
DECLARE
  v_cod_sala     sala.cod_sala%TYPE;
  v_cod_galerie  galerie.cod_galerie%TYPE;
  v_car          VARCHAR2(75);
  CURSOR sala_cursor IS
    SELECT  cod_sala,cod_galerie
    FROM    sala;
  CURSOR ope_cursor (v_id_sala NUMBER,v_id_galerie NUMBER) IS
    SELECT  cod_opera || cod_sala || cod_galerie
    FROM    opera
    WHERE   cod_sala = v_id_sala
    AND     cod_galerie = v_id_galerie;
BEGIN
  OPEN sala_cursor;
  LOOP
    FETCH sala_cursor INTO v_cod_sala,v_cod_galerie;
    EXIT WHEN sala_cursor%NOTFOUND;
    IF ope_cursor%ISOPEN THEN
      CLOSE ope_cursor;
    END IF;
    OPEN ope_cursor (v_cod_sala, v_cod_galerie);
    LOOP
      FETCH ope_cursor INTO v_car;
      EXIT WHEN ope_cursor%NOTFOUND;
      INSERT INTO mesaje (rezultat)
```

```
      VALUES (v_car);
    END LOOP;
    CLOSE ope_cursor;
  END LOOP;
  CLOSE sala_cursor;
  COMMIT;
END;
```

Cursoare *SELECT FOR UPDATE*

Uneori este necesară blocarea liniilor înainte ca acestea să fie şterse sau reactualizate. Blocarea se poate realiza (atunci când cursorul este deschis) cu ajutorul comenzii *SELECT* care conţine clauza *FOR UPDATE*.

Declararea unui astfel de cursor se face conform sintaxei:

CURSOR *nume_cursor* ***IS***
comanda_select
FOR UPDATE [***OF*** *lista_câmpuri*] [***NOWAIT***];

Identificatorul *lista_câmpuri* este o listă ce include câmpurile tabelului care vor fi modificate. Atributul *NOWAIT* returnează o eroare dacă liniile sunt deja blocate de altă sesiune. Liniile unui tabel sunt blocate doar dacă clauza *FOR UPDATE* se referă la coloane ale tabelului respectiv.

În momentul deschiderii unui astfel de cursor, liniile corespunzătoare mulţimii active, determinate de clauza *SELECT*, sunt blocate pentru operaţii de scriere (reactualizare sau ştergere). În felul acesta este realizată consistenţa la citire a sistemului. De exemplu, această situaţie este utilă când se reactualizează o valoare a unei linii şi trebuie avută siguranţa că linia nu este schimbată de alt utilizator înaintea reactualizării. Prin urmare, alte sesiuni nu pot schimba liniile din mulţimea activă până când tranzacţia nu este permanentizată sau anulată. Dacă altă sesiune a blocat deja liniile din mulţimea activă, atunci comanda *SELECT ... FOR UPDATE* va aştepta (sau nu) ca aceste blocări să fie eliberate. Pentru a trata această situaţie se utilizează clauza *WAIT*, respectiv *NOWAIT*.

În *Oracle9i* este utilizată sintaxa:

SELECT *...* ***FROM*** *...* ***FOR UPDATE*** [***OF*** *lista_campuri*]
[{***WAIT*** *n* | ***NOWAIT***}];

Valoarea lui *n* reprezintă numărul de secunde de aşteptare. Dacă liniile nu sunt deblocate în *n* secunde, atunci se declanşează eroarea *ORA-30006*, respectiv eroarea *ORA-00054*, după cum este specificată clauza *WAIT*, respectiv *NOWAIT*. Dacă nu este specificată nici una din clauzele *WAIT* sau *NOWAIT*, sistemul aşteaptă până ce linia este deblocată şi atunci returnează rezultatul comenzii *SELECT*.

Dacă un cursor este declarat cu clauza *FOR UPDATE*, atunci comenzile

DELETE şi *UPDATE* corespunzătoare trebuie să conţină clauza *WHERE CURRENT OF nume_cursor*.

Această clauză referă linia curentă care a fost găsită de cursor, permiţând ca reactualizările şi ştergerile să se efectueze asupra acestei linii, fără referirea explicită a cheii primare sau pseudocoloanei *ROWID*. De subliniat că instrucţiunile *UPDATE* şi *DELETE* vor reactualiza numai coloanele listate în clauza *FOR UPDATE*.

Pseudocoloana *ROWID* poate fi utilizată dacă tabelul referit în interogare nu are o cheie primară specificată. *ROWID*-ul fiecărei linii poate fi încărcat într-o variabilă *PL/SQL* (declarată de tipul *ROWID* sau *UROWID*), iar această variabilă poate fi utilizată în clauza *WHERE* (*WHERE ROWID = v_rowid*).

După închiderea cursorului este necesară comanda *COMMIT* pentru a realiza scrierea efectivă a modificărilor, deoarece cursorul lucrează doar cu nişte copii ale liniilor reale existente în tabele.

Deoarece blocările implicate de clauza *FOR UPDATE* vor fi eliberate de comanda *COMMIT*, nu este recomandată utilizarea comenzii *COMMIT* în interiorul ciclului în care se fac încărcări de date. Orice *FETCH* executat după *COMMIT* va eşua. În cazul în care cursorul nu este definit prin *SELECT…FOR UPDATE*, nu sunt probleme în acest sens şi, prin urmare, în interiorul ciclului unde se fac schimbări ale datelor poate fi utilizat un *COMMIT*.

***Exemplu*:**

Să se dubleze valoarea operelor de artă pictate pe pânză care au fost achiziţionate înainte de 1 ianuarie 1956.

```
DECLARE
  CURSOR calc IS
    SELECT *
    FROM   opera
    WHERE  material = 'panza'
    AND     data_achizitie <= TO_DATE('01-JAN-56','DD-MON-
YY')
    FOR UPDATE OF valoare NOWAIT;
BEGIN
  FOR x IN calc LOOP
    UPDATE  opera
    SET     valoare = valoare*2
    WHERE CURRENT OF calc;
  END LOOP;
  -- se permanentizeaza actiunea si se elibereaza blocarea
  COMMIT;
END;
```

Cursoare dinamice

Toate exemplele considerate anterior se referă la cursoare statice. Unui cursor static i se asociază o comandă *SQL* care este cunoscută în momentul în care blocul este compilat.

În *PL/SQL* a fost introdusă variabila cursor, care este de tip referinţă. Variabilele cursor sunt similare tipului *pointer* din limbajele *C* sau *Pascal*. Prin urmare, un cursor este un obiect static, iar un cursor dinamic este un *pointer* la un cursor.

În momentul declarării, variabilele cursor nu solicită o comandă *SQL* asociată. În acest fel, diferite comenzi *SQL* pot fi asociate variabilelor cursor, la diferite momente de timp. Acest tip de variabilă trebuie declarată, deschisă, încărcată şi închisă în mod similar unui cursor static.

Variabilele cursor sunt dinamice deoarece li se pot asocia diferite interogări atâta timp cât coloanele returnate de fiecare interogare corespund declaraţiei variabilei cursor.

Aceste variabile sunt utile în transmiterea seturilor de rezultate între subprograme *PL/SQL* stocate şi diferiţi clienţi. De exemplu, un *client OCI*, o aplicaţie *Oracle Forms* şi *server*-ul *Oracle* pot referi aceeaşi zonă de lucru (care conţine mulţimea rezultat). Pentru a reduce traficul în reţea, o variabilă cursor poate fi declarată pe staţia *client*, deschisă şi se pot încărca date din ea pe *server*, apoi poate continua încărcarea, dar de pe staţia *client* etc.

Pentru a crea o variabilă cursor este necesară definirea unui tip *REF CURSOR*, urmând apoi declararea unei variabile de tipul respectiv. După ce variabila cursor a fost declarată, ea poate fi deschisă pentru orice cerere *SQL* care returnează date de tipul declarat.

Sintaxa pentru declararea variabilei cursor este următoarea:

TYPE *tip_ref_cursor* ***IS REF CURSOR*** [***RETURN*** *tip_returnat*];
var_cursor tip_ref_cursor;

Identificatorul *var_cursor* este numele variabilei cursor, *tip_ref_cursor* este un nou tip de dată ce poate fi utilizat în declaraţiile următoare ale variabilelor cursor, iar *tip_returnat* este un tip înregistrare sau tipul unei linii dintr-un tabel al bazei. Acest tip corespunde coloanelor returnate de către orice cursor asociat variabilelor cursor de tipul definit. Dacă lipseşte clauza *RETURN*, cursorul poate fi deschis pentru orice cerere *SELECT*.

Dacă variabila cursor apare ca parametru într-un subprogram, atunci trebuie specificat tipul parametrului (tipul *REF CURSOR*) şi forma acestuia (*IN* sau *IN OUT*).

Există anumite restricţii referitoare la utilizarea variabilelor cursor:

- nu pot fi declarate într-un pachet;

- cererea asociată variabilei cursor nu poate include clauza *FOR UPDATE* (restricţia dispare în *Oracle9i*);
- nu poate fi asignată valoarea *null* unei variabile cursor;
- nu poate fi utilizat tipul *REF CURSOR* pentru a specifica tipul unei coloane în comanda *CREATE TABLE*;
- nu pot fi utilizaţi operatorii de comparare pentru a testa egalitatea, inegalitatea sau valoarea *null* a variabilelor cursor;
- nu poate fi utilizat tipul *REF CURSOR* pentru a specifica tipul elementelor unei colecţii (*varray*, *nested table*);
- nu pot fi folosite cu *SQL* dinamic în *Pro*C/C++*.

În cazul variabilelor cursor, instrucţiunile de deschidere (*OPEN*), încărcare (*FETCH*), închidere (*CLOSE*) vor avea o sintaxă similară celor comentate anterior.

Comanda *OPEN...FOR* asociază o variabilă cursor cu o cerere multilinie, execută cererea, identifică mulţimea rezultat şi poziţionează cursorul la prima linie din mulţimea rezultat. Sintaxa comenzii este:

OPEN {*variabila_cursor* | :*variabila_cursor_host*}
FOR {*cerere_select* |
şir_dinamic [***USING*** *argument_bind* [*, argument_bind ...*]] };

Identificatorul *variabila_cursor* specifică o variabilă cursor declarată anterior, dar fără opţiunea *RETURN tip*, *cerere_select* este interogarea pentru care este deschisă variabila cursor, iar *şir_dinamic* este o secvenţă de caractere care reprezintă cererea multilinie.

Opţiunea *şir_dinamic* este specifică prelucrării dinamice a comenzilor, iar posibilităţile oferite de *SQL* dinamic vor fi analizate într-un capitol separat. Identificatorul :*variabila_cursor_host* reprezintă o variabilă cursor declarată într-un mediu gazdă *PL/SQL* (de exemplu, un program *OCI*).

Comanda *OPEN - FOR* poate deschide acelaşi cursor pentru diferite cereri. Nu este necesară închiderea variabilei cursor înainte de a o redeschide. Dacă se redeschide variabila cursor pentru o nouă cerere, cererea anterioară este pierdută.

Exemplu:

```
CREATE OR REPLACE PACKAGE alfa AS
  TYPE  ope_tip IS REF CURSOR RETURN opera%ROWTYPE;
  PROCEDURE deschis_ope (ope_var IN OUT ope_tip,
                         alege IN NUMBER);
END alfa;

CREATE OR REPLACE PACKAGE BODY alfa AS
  PROCEDURE deschis_ope (ope_var IN OUT ope_tip,
```

```
                              alege IN NUMBER) IS
  BEGIN
   IF alege = 1 THEN
    OPEN ope_var FOR SELECT * FROM opera;
   ELSIF alege = 2 THEN
    OPEN ope_var FOR SELECT * FROM opera WHERE valoare>2000;
   ELSIF alege = 3 THEN
    OPEN ope_var FOR SELECT * FROM opera WHERE valoare=7777;
   END IF;
  END deschis_ope;
END alfa;
```

Exemplu:

În următorul exemplu se declară o variabilă cursor care se asociază unei comenzi *SELECT* (*SQL* dinamic) ce returnează anumite linii din tabelul *opera*.

```
DECLARE
  TYPE  operaref IS REF CURSOR;
  opera_var  operaref;
  mm_val  INTEGER := 100000;
BEGIN
  OPEN opera_var FOR
    'SELECT cod_opera,valoare FROM opera WHERE valoare> :vv'
     USING mm_val;
  …
END;
```

Comanda *FETCH* returnează o linie din mulţimea rezultat a cererii multi-linie, atribuie valori componentelor din lista cererii prin clauza *INTO*, avansează cursorul la următoarea linie. Sintaxa comenzii este:

***FETCH* {*variabila_cursor* | *:variabila_cursor_host*}**
***INTO* {*variabila* [, *variabila*]… | *înregistrare*}**
[*BULK COLLECT INTO* {*nume_colecţie* [, *nume_colecţie*]…} |
{*nume_array_host* [, *nume_array_host*]…}
[*LIMIT expresie_numerica*]];

Clauza *BULK COLLECT* permite încărcarea tuturor liniilor simultan în una sau mai multe colecţii. Atributul *nume_colecţie* indică o colecţie declarată anterior, în care sunt depuse valorile respective, iar *nume_array_host* identifică un vector declarat într-un mediu gazdă *PL/SQL* şi trimis lui *PL/SQL* ca variabilă de legătură. Prin clauza *LIMIT* se limitează numărul liniilor încărcate din baza de date.

Exemplu:

```
DECLARE
  TYPE alfa IS REF CURSOR RETURN opera%ROWTYPE;
  TYPE beta IS TABLE OF opera.titlu%TYPE;
  TYPE gama IS TABLE OF opera.valoare%TYPE;
  var1 alfa;
  var2 beta;
  var3 gama;
BEGIN
  OPEN alfa FOR SELECT titlu, valoare FROM opera;
  FETCH var1 BULK COLLECT INTO var2, var3;
  …
  CLOSE var1;
END;
```

Comanda *CLOSE* dezactivează variabila cursor precizată. Ea are sintaxa:

CLOSE {variabila_cursor | :variabila_cursor_host}

Cursoarele şi variabilele cursor nu sunt interoperabile. Nu poate fi folosită una din ele, când este aşteptată cealaltă. Următoarea secvenţă este incorectă.

```
DECLARE
  TYPE beta IS REF CURSOR RETURN opera%ROWTYPE;
  gama beta;
BEGIN
  FOR k IN gama LOOP --nu este corect!
  …
END;
```

Expresie cursor

În *Oracle9i* a fost introdus conceptul de expresie cursor (*cursor expression*), care returnează un cursor imbricat (*nested cursor*).

Expresia cursor are următoarea sintaxă:

CURSOR (*subcerere*)

Fiecare linie din mulţimea rezultat poate conţine valori uzuale şi cursoare generate de subcereri. *PL/SQL* acceptă cereri care au expresii cursor în cadrul unei declaraţii cursor, declaraţii *REF CURSOR* şi a variabilelor cursor.

Prin urmare, expresia cursor poate să apară într-o comandă *SELECT* ce este utilizată pentru deschiderea unui cursor dinamic. De asemenea, expresiile cursor pot fi folosite în cereri *SQL* dinamice sau ca parametri actuali într-un subprogram.

Un cursor imbricat este încărcat automat atunci când liniile care îl conţin sunt încărcate din cursorul „părinte“. El este închis dacă:

- este închis explicit de către utilizator;
- cursorul „părinte“ este reexecutat, închis sau anulat;
- apare o eroare în timpul unei încărcări din cursorul „părinte“.

Există câteva restricţii asupra folosirii unei expresii cursor:

- nu poate fi utilizată cu un cursor implicit;
- poate să apară numai într-o comandă *SELECT* care nu este imbricată în altă cerere (exceptând cazul în care este o subcerere chiar a expresiei cursor) sau ca argument pentru funcţii tabel, în clauza *FROM* a lui *SELECT*;
- nu poate să apară în interogarea ce defineşte o vizualizare;
- nu se pot efectua operaţii *BIND* sau *EXECUTE* cu aceste expresii.

***Exemplu*:**

Să se definească un cursor care furnizează codurile operelor expuse în cadrul unei expoziţii având un cod specificat (*val_cod*) şi care se desfăşoară într-o localitate precizată (*val_oras*). Să se afişeze data când a avut loc vernisajul acestei expoziţii.

În acest caz cursorul returnează două coloane, cea de-a doua coloană fiind un cursor imbricat.

```
CURSOR alfa (val_cod NUMBER, val_oras VARCHAR2(20)) IS
  SELECT  l.datai,
       CURSOR (SELECT d.cod_expo,
                   CURSOR (SELECT f.cod_opera
                           FROM   figureaza_in f
                           WHERE  f.cod_expo=d.cod_expo) AS xx
                FROM   expozitie d
                WHERE  l.cod_expo = d.cod_expo) AS yy
  FROM    locped l
  WHERE   cod_expo = val_cod AND nume_oras= val_oras;
```

***Exemplu*:**

Să se listeze numele galeriilor din muzeu şi pentru fiecare galerie să se afişeze numele sălilor din galeria respectivă.

Sunt prezentate două variante de rezolvare. Prima variantă reprezintă o implementare simplă utilizând programarea secvenţială clasică, iar a doua utilizează expresii cursor pentru rezolvarea acestei probleme.

Varianta 1:

```
BEGIN
  FOR gal IN (SELECT cod_galerie, nume_galerie
              FROM   galerie)
  LOOP
```

```
    DBMS_OUTPUT.PUT_LINE (gal.nume_galerie);
    FOR sal IN (SELECT cod_sala, nume_sala
                FROM   sala
                WHERE  cod_galerie = gal.cod.galerie)
    LOOP
      DBMS_OUTPUT.PUT_LINE (sal.nume_sala);
    END LOOP;
  END LOOP;
END;
```

Varianta 2:

```
DECLARE
  CURSOR c_gal IS
    SELECT nume_galerie,
           CURSOR (SELECT nume_sala
                   FROM   sala s
                   WHERE  s.cod_galerie = g.cod_galerie)
    FROM   galerie g;
  v_nume_gal   galerie.nume_galerie%TYPE;
  v_sala       SYS_REFCURSOR;
  TYPE sala_nume IS TABLE OF sala.nume_sala%TYPE
                    INDEX BY BINARY_INTEGER;
  v_nume_sala  sala_nume;
BEGIN
  OPEN c_gal;
  LOOP
    FETCH c_gal INTO v_nume_gal, v_sala;
    EXIT WHEN c_gal%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (v_nume_gal);
    FETCH v_sala BULK COLLECT INTO v_nume_sala;
    FOR ind IN v_nume_sala.FIRST..v_nume_sala.LAST
      LOOP
        DBMS_OUTPUT.PUT_LINE (v_nume_sala (ind));
      END LOOP;
  END LOOP;
  CLOSE c_gal;
END;
```

Subprograme în *PL/SQL*

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 tratament.

Unităţile de program ce 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 drept 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, 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 obiectele (aplicaţii) care îl solicită.

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

- Procedurile care sunt declarate şi apelate în blocuri anonime sunt temporare. O procedură stocată (creată cu *CREATE PROCEDURE* sau conţinută într-un pachet) este permanentă în sensul că ea poate fi invocată printr-un script *iSQL**Plus, un subprogram *PL/SQL* sau un declanşator.
- Procedurile şi funcţiile stocate, care sunt compilate şi stocate în baza de date, nu mai trebuie să fie compilate a doua oară pentru a fi executate, în timp ce procedurile locale sunt compilate de fiecare dată când este executat blocul care conţine procedurile şi funcţiile respective.
- Procedurile şi funcţiile stocate pot fi apelate din orice bloc de către utilizatorul care are privilegiul *EXECUTE* asupra subprogramului, 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 OR REPLACE*, subprogramul este depus în dicţionarul datelor. Este depus atât textul sursă, cât şi forma compilată (*p-code*). 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*).

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.

Dezvoltare de subprograme utilizând *iSQL*Plus*

Oracle9i dispune de o interfaţă *Internet* pentru *SQL*Plus*, care este *iSQL*Plus*. Se poate utiliza un *Web browser* pentru conectare la o bază de date *Oracle* şi pentru a executa prin *iSQL*Plus* orice acţiune fezabilă cu *SQL*Plus*.

Algoritmul de lucru este următorul:

1) se utilizează un editor de texte pentru a crea un *script file SQL* (extensia este *.sql*);

2) se utilizează opţiunea *Browse* din meniu pentru a localiza acest fişier;

3) se utilizează opţiunea *Load Script* din meniul afişat pentru a încărca acest *script* în *buffer*-ul *iSQL*Plus;*

4) se utilizează opţiunea *Execute* din meniu pentru executarea codului (implicit, rezultatul codului este afişat pe ecran).

Crearea subprogramelor stocate

1) se editează subprogramul (*CREATE PROCEDURE* sau *CREATE FUNCTION*) şi se salvează într-un *script file SQL*;

2) se încarcă şi se execută acest *script file*, este compilat codul sursă, se obţine *p-code* (subprogramul este creat);

3) se utilizează comanda *SHOW ERRORS* (în *iSQL*Plus* sau in *SQL*Plus*) pentru vizualizarea eventualelor erori la compilare ale procedurii care a fost cel mai recent compilata sau *SHOW ERRORS PROCEDURE nume* pentru orice procedura compilata anterior (nu poate fi invocata o procedura care contine erori de compilare);

4) se execută subprogramul pentru a realiza acţiunea dorită (de exemplu, procedura poate fi executată de câte ori este necesar, utilizând comanda *EXECUTE* din *iSQL*Plus*) sau se invocă funcţia 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*.

Când este apelată o procedură *PL/SQL*, *server*-ul *Oracle* parcurge etapele:

1) Verifică dacă utilizatorul are privilegiul să execute procedura (fie pentru că el a creat procedura, fie pentru că i s-a dat acest privilegiu).

2) 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*.

3) Verifică dacă starea procedurii este *validă* sau *invalidă*. 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 procedurii este *invalidă* atunci este recompilată automat. Dacă nici o eroare nu a fost detectată, atunci va fi executată noua versiune a procedurii.

4) Dacă procedura aparţine unui pachet atunci toate procedurile şi funcţiile pachetului sunt de asemenea încărcate în *database cache* (dacă ele nu erau deja acolo). Dacă pachetul este activat pentru prima oară într-o sesiune, atunci *server*-ul va executa blocul de iniţializare al pachetului.

Dezvoltare de subprograme utilizând *Oracle Procedure Builder*

Oracle Procedure Builder permite crearea şi depanarea diferitelor unităţi de program (aplicaţie sau unităţi stocate). Subprogramele pot fi editate, create, compilate, salvate, depanate, apelate utilizând facilităţile oferite de acest instrument *Oracle*, prin intermediul unui editor grafic. *Procedure Builder* permite dezvoltarea de subprograme *PL/SQL* care pot fi folosite de aplicaţii *client* şi aplicaţii *server*.

Procedure Builder conţine cinci componente.

- *Object Navigator* furnizează o interfaţă *outline* pentru a vizualiza obiecte, relaţii între obiecte, pentru a edita proprietăţi ale obiectelor. Cu ajutorul acestei componente poate fi listată o ierarhie a tuturor obiectelor ce pot fi accesate în timpul sesiunii utilizatorului.
- *PL/SQL Interpreter* depanează şi evaluează codul *PL/SQL* în timp real.
- *Program Unit Editor* creează şi editează codul *PL/SQL* sursă (*client-side*).
- *Stored Program Unit Editor* creează şi editează construcţii *PL/SQL server-side*, listează mesajele erorilor generate în timpul compilării
- *Database Trigger Editor* creează şi editează declanşatori bază de date.

Pentru a afişa codul unui subprogram, parametrii acestuia, precum şi alte informaţii legate de subprogram poate fi utilizată comanda *DESCRIBE*.

Proceduri *PL/SQL*

Procedura *PL/SQL* este un program independent care se găseşte compilat în schema bazei de date *Oracle*. Când procedura este compilată, identificatorul acesteia (stabilit prin comanda *CREATE PROCEDURE*) devine un nume 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 mânuire a excepţiilor]
END [*nume_procedură*];

unde parametrii au următoarea formă sintactică:

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

Clauza *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 BD în formă compilată. Dacă procedura există, atunci clauza *OR REPLACE* va avea ca efect ştergerea procedurii şi înlocuirea acesteia cu noua versiune. Dacă procedura există, iar *OR REPLACE* nu este prezent, atunci comanda *CREATE* va returna eroarea "*ORA-955: 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 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*, *commit*-ul sau *rollback*-ul acestor operaţii şi continuarea tranzacţiei principale.

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

Exemplu:

Să se creeze o procedură stocată care micşorează cu o cantitate dată (*cant*) valoarea poliţelor 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';
EXCEPTION
  WHEN NO_DATA_FOUND THEN
   RAISE_APPLICATION_ERROR (-20010,'nu exista ASIROM');
END;
/
```

Dacă sunt operaţii de reactualizare în subprograme şi există declanşatori relativ la aceste operaţii care nu trebuie consideraţi, 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 a unui declanşator;
- interactiv de utilizator utilizând un instrument *Oracle* (de exemplu, *iSQL*Plus*);
- explicit dintr-o aplicaţie (de exemplu, *SQL*Forms* sau utilizarea de precompilatoare).

Utilizarea (apelarea) unei proceduri se poate face:

1) în *iSQL*Plus* prin comanda:

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

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

Funcţii *PL/SQL*

Funcţia *PL/SQL* este similară unei proceduri cu excepţia că ea trebuie să întoarcă un rezultat. O funcţie fără comanda *RETURN* va genera 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 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*. O funcţie trebuie să aibă un *RETURN* în antet şi cel puţin un *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 mânuire a excepţiilor]
END [*nume_funcţie*];

Opţiunea *tip_de_date* specifică tipul valorii returnate de funcţie, tip care nu poate conţine specificaţii de mărime. Dacă totuşi sunt necesare aceste specificaţii se pot defini subtipuri, iar parametrii vor fi declaraţi de subtipul respectiv.

În interiorul funcţiei trebuie să apară *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, deoarece dupa ce valoarea este returnata, procesarea blocului inceteaza. Comanda *RETURN* (fără o expresie asociată) poate să apară şi într-o procedură. În acest caz, ea va avea ca efect revenirea la comanda ce urmează instrucţiunii apelante.

Opţiunea *DETERMINISTIC* ajută optimizorul *Oracle* în cazul unor apeluri repetate ale aceleaşi funcţii, având aceleaşi argumente. Ea asigură folosirea unui rezultat obţinut anterior.

În blocul *PL/SQL* al unei proceduri sau funcţii stocate (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.

***Exemplu*:**

Să se creeze o funcţie stocată care determină numărul operelor de artă realizate pe pânză, ce au fost achiziţionate la o anumită dată.

```
CREATE OR REPLACE FUNCTION  numar_opere
                  (v_a  IN  opera.data_achizitie%TYPE)
  RETURN NUMBER AS
  alfa  NUMBER;
BEGIN
  SELECT  COUNT(ROWID)
  INTO    alfa
  FROM    opera
  WHERE   material='panza'
  AND     data_achizitie = v_a;
  RETURN alfa;
END numar_opere;
/
```

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

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.

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

```
VARIABLE val NUMBER
EXECUTE :val := numar_opere(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 *iSQL*Plus.*

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

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

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

```
ACCEPT data PROMPT 'dati data achizitionare'
DECLARE
  num  NUMBER;
  v_data  opera.data_achizitie%TYPE := '&data';
BEGIN
  num := numar_opere(v_data);
  DBMS_OUTPUT.PUT_LINE('numarul operelor de arta
    achizitionate la data' || TO_CHAR(v_data) || este'
    || TO_CHAR(num));
END;
/
```

Exemplu:

Să se creeze o procedură stocată care pentru un anumit tip de operă de artă (dat ca parametru) calculează numărul operelor din muzeu de tipul respectiv, numărul de specialişti care au expertizat sau au restaurat aceste opere, numărul de expoziţii în care au fost expuse, precum şi valoarea nominală totală a acestora.

```
CREATE OR REPLACE PROCEDURE date_tip_opera
                     (v_tip  opera.tip%TYPE) AS
  FUNCTION nr_opere (v_tip  opera.tip%TYPE)
  RETURN NUMBER IS
    v_numar  NUMBER(3);
  BEGIN
    SELECT COUNT(*)
    INTO   v_numar
```

```
    FROM   opera
    WHERE  tip = v_tip;
    RETURN v_numar;
  END;
  FUNCTION valoare_totala (v_tip  opera.tip%TYPE)
  RETURN NUMBER IS
    v_numar  opera.valoare%TYPE;
  BEGIN
    SELECT SUM(valoare)
    INTO   v_numar
    FROM   opera
    WHERE  tip = v_tip;
    RETURN v_numar;
  END;
  FUNCTION nr_specialisti (v_tip  opera.tip%TYPE)
  RETURN NUMBER IS
    v_numar  NUMBER(3);
  BEGIN
    SELECT COUNT(DISTINCT studiaza.cod_specialist)
    INTO   v_numar
    FROM   studiaza, opera
    WHERE  studiaza.cod_opera = opera.cod_opera
    AND    opera.tip = v_tip;
    RETURN v_numar;
  END;
  FUNCTION nr_expozitii (v_tip  opera.tip%TYPE)
  RETURN NUMBER IS
    v_numar  NUMBER(3);
  BEGIN
    SELECT COUNT(DISTINCT figureaza_in.cod_expozitie)
    INTO   v_numar
    FROM   figureaza_in, opera
    WHERE  figureaza_in.cod_opera = opera.cod_opera
    AND    opera.tip = v_tip;
    RETURN v_numar;
  END;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Numarul operelor de arta este '||
                       nr_opere(v_tip));
  DBMS_OUTPUT.PUT_LINE('Valoarea oerelor de arta este '||
                       valoare_totala(v_tip));
  DBMS_OUTPUT.PUT_LINE('Numarul de specialisti este '||
                       nr_specialisti(v_tip));
  DBMS_OUTPUT.PUT_LINE('Numarul de expozitii este '||
                       nr_expozitii(v_tip);
END;
```

Instrucţiunea *CALL*

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

CALL este o comandă *SQL* care nu este validă într-un bloc *PL/SQL*. Poate fi utilizata in PL/SQL doar dinamic, prin intermediul comenzii *EXECUTE IMMEDIATE*. Pentru executarea acestei comenzi, utilizatorul trebuie să aibă privilegiul *EXECUTE* asupra subprogramului. Comanda poate fi executată interactiv din *iSQL*Plus*. Ea are sintaxa următoare:

CALL [*schema.*]*nume_subprogram* ([*lista_parametri actuali*])
[*@dblink_nume*] [***INTO*** :*variabila_host*]

Nume_subprogram este numele unui subprogram sau numele unei metode. Clauza *INTO* este folosită numai pentru variabilele de ieşire ale unei funcţii. Dacă clauza *@dblink_nume* lipseşte, sistemul se referă la baza de date locală, iar într-un sistem distribuit clauza specifică numele bazei care 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!
```

```
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 acestora se face prin comanda:

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

Comanda recompilează doar procedurile catalogate standard. Procedurile unui pachet se recompilează într-o altă manieră.

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.

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 asupra acestui obiect (subprogram) rămân aceleaşi.

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

Transferarea valorilor prin parametri

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

Dacă nu este specificat nimic, atunci implicit parametrul este considerat *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 fi *NULL*. Parametrul actual trebuie să fie o variabilă, nu poate fi o constantă sau o expresie.

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

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

Când este apelată o procedură *PL/SQL*, sistemul *Oracle* furnizează două metode pentru definirea parametrilor actuali:

- specificarea explicită prin nume;
- specificarea prin poziţie.

Exemplu:

```
CREATE PROCEDURE p1(a IN NUMBER, b IN VARCHAR2,
                    c IN DATE, d OUT NUMBER) AS…;
```

Sunt prezentate diferite moduri pentru apelarea acestei proceduri.

```
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;
```

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.

```
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*

În anumite condiţii, 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*.

În cazul unui apel, compilatorul compară parametri 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). Nu poate fi definită o versiune într-un bloc, iar altă versiune într-un bloc diferit.

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

Observaţii:

- Două programe *overload* trebuie să difere, cel puţin, prin tipul unuia dintre parametri. Două programe nu pot fi *overload* dacă parametri lor formali diferă numai prin subtipurile lor şi dacă aceste subtipuri se bazează pe acelaşi tip de date.
- Nu este suficient ca lista parametrilor programelor *overload* să difere numai prin numele parametrilor formali.
- Nu este suficient ca lista parametrilor programelor *overload* să difere numai prin tipul acestora (*IN*, *OUT*, *IN OUT*). *PL/SQL* nu poate face diferenţe (la apelare) între tipurile *IN* sau *OUT*.
- Nu este suficient ca funcţiile *overload* să difere doar prin tipul datei returnate (tipul datei specificate în clauza *RETURN* a funcţiei).

Exemplu:

Următoarele subprograme nu pot fi *overload*.

a)
```
FUNCTION alfa(par IN POSITIVE)…;
FUNCTION alfa(par IN BINARY_INTEGER)…;
```

b)
```
FUNCTION alfa(par IN NUMBER)…;
FUNCTION alfa(parar IN NUMBER)…;
```

c)
```
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 operelor de artă de un anumit tip. Prima funcţie va avea un argument reprezentând tipul operelor de artă, iar cea de a doua va avea două argumente, unul reprezentând tipul operelor de artă, iar celălalt reprezentând stilul operelor pentru care se calculează valoarea medie (adică funcţia va calcula media valorilor operelor de artă de un anumit tip şi care aparţin unui stil specificat).

```
DECLARE
  medie1 NUMBER(10,2);
  medie2 NUMBER(10,2);
  FUNCTION valoare_medie (v_tip  opera.tip%TYPE)
    RETURN NUMBER IS
    medie NUMBER(10,2);
  BEGIN
    SELECT AVG(valoare)
    INTO   medie
    FROM   opera
    WHERE  tip = v_tip;
    RETURN medie;
  END;
  FUNCTION valoare.medie (v_tip   opera.tip%TYPE,
                          v_stil  opera.stil%TYPE)
    RETURN NUMBER IS
    medie NUMBER(10,2);
  BEGIN
    SELECT AVG(valoare)
    INTO   medie
    FROM   opera
    WHERE  tip = v_tip AND stil = v_stil;
    RETURN medie;
  END;
BEGIN
  medie1 := valoare_medie('pictura');
  DBMS_OUTPUT.PUT_LINE('Media valorilor picturilor din
             muzeu este ' || medie1);
  medie2 := valoare_medie('pictura', 'impresionism');
  DBMS_OUTPUT.PUT_LINE('Media valorilor picturilor
           impresioniste din muzeu este ' || medie2);
END;
```

Procedură *versus* funcţie

Pot fi marcate câteva **deosebiri** esenţiale între funcţii şi proceduri.

- Procedura se execută ca o comandă *PL/SQL*, iar funcţia se invocă ca parte a unei expresii.
- Procedura poate returna (sau nu) una sau mai multe valori, iar funcţia trebuie să returneze (cel putin) o singură valoare.
- Procedura nu trebuie să conţină *RETURN tip_date*, iar funcţia trebuie să conţină această opţiune.

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

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

Recursivitate

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 unui cursor *FOR* sau între comenzile *OPEN* şi *CLOSE,* atunci la fiecare apel este deschis alt cursor. În felul acesta, programul poate depăşi limita pentru *OPEN_CURSORS* setată în parametrul de iniţializare *Oracle*.

Exemplu:

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

```
FUNCTION fibona(m  POSITIVE) RETURN INTEGER IS
BEGIN
  IF (m = 1) OR (m = 2) THEN
    RETURN 1;
  ELSE
    RETURN fibona(m-1) + fibona(m-2);
  END IF;
END fibona;
```

Declaraţii *forward*

Subprogramele sunt reciproc recursive dacă ele se apelează unul pe altul direct sau indirect. Declaraţiile *forward* permit definirea subprogramelor reciproc recursive.

În *PL/SQL*, un identificator trebuie declarat înainte de a-l folosi. De asemenea, un subprogram trebuie declarat înainte de a-l apela.

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

Procedura *beta* nu poate fi apelată deoarece nu este încă declarată. Problema se poate rezolva simplu, inversând ordinea celor două proceduri. Această soluţie nu este eficientă întotdeauna.

PL/SQL permite un tip special de declarare a unui subprogram numit *forward*. El constă dintr-o specificare de subprogram terminată prin ";".

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

Se pot folosi declaraţii *forward* pentru a defini subprograme într-o anumită ordine logică, pentru a defini subprograme reciproc recursive, 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 trebuie să rămână în aceeaşi unitate de program.

Utilizarea în expresii *SQL* a funcţiilor definite de utilizator

Începând cu *Release 7.1*, 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 pot fi folosite funcţii *SQL* standard.

Funcţiile *PL/SQL* pot să apară în:

- lista de selecţie 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 operele de artă (titlu, valoare, stare) a căror valoare este mai mare decât valoarea medie a tuturor operelor de artă din muzeu.

```
CREATE OR REPLACE FUNCTION valoare_medie
  RETURN NUMBER IS
  v_val_mediu opera.valoare%TYPE;
BEGIN
  SELECT AVG(valoare)
  INTO   v_val_mediu
  FROM   opera;
  RETURN v_val_mediu;
END;
```

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

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

Există restricţii referitoare la folosirea funcţiilor definite de utilizator într-o comandă *SQL*.

- funcţia definită de utilizator trebuie să fie o funcţie stocată (procedurile stocate nu pot fi apelate în expresii *SQL*), nu poate fi locală unui alt bloc;
- funcţia apelată dintr-o comandă *SELECT*, sau din comenzi paralelizate *INSERT*, *UPDATE* şi *DELETE* nu poate contine comenzi *LMD* care 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 executa comenzi *LCD* (*COMMIT*), *ALTER SYSTEM*, *SET ROLE* sau comenzi *LDD* (*CREATE*);
- funcţia nu poate să apară 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;
- 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 unei funcţii *PL/SQL* apelate dintr-o comandă *SQL* trebuie să fie specificaţi prin poziţie (specificarea prin nume nefiind permisă);
- parametrii formali ai unui subprogram funcţie trebuie să fie de tip specific bazei de date (*NUMBER*, *CHAR*, *VARCHAR2*, *ROWID*, *LONG*, *LONGROW*, *DATE*), nu tipuri *PL/SQL* (*BOOLEAN* sau *RECORD*);
- tipul returnat de un subprogram funcţie trebuie să fie un tip intern pentru *server*, nu un tip *PL/SQL* (nu poate fi *TABLE*, *RECORD* sau *BOOLEAN*);
- 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 opera(cod_opera, tip, data_achizitie,
valoare);
  VALUES (1358, 'gravura', SYSDATE, 700000);
  RETURN (p_val*7);
END;
/

UPDATE  opera
SET     valoare = calcul (550000)
WHERE   cod_opera = 7531;
```

Comanda *UPDATE* va returna o eroare deoarece tabelul *opera* este *mutating*. Reactualizarea este insa permisa asupra oricarui alt tabel diferit de *opera*.

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 (*DD*);

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

Vizualizarea *USER_OBJECTS* conţine informaţii generale despre toate obiectele manipulate în BD, în particular şi 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 când obiectul a fost creat;
- *LAST_DDL_TIME* – data ultimei modificări a obiectului;
- *TIMESTAMP* – data şi momentul ultimei recompilări;
- *STATUS* – starea obiectului (*VALID* sau *INVALID*).

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

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

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

Exemplu:

Să se listeze 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');
```

După ce subprogramul a fost creat, codul sursă al acestuia poate fi obţinut consultând vizualizarea *USER_SOURCE* din DD, care 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_opere*.

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

Exemplu:

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

```
CREATE OR REPLACE PROCEDURE sterge 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 sterge;
```

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

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

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

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.

Dependenţa subprogramelor

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 dacă o operaţie *LDD* este executată asupra unui obiect de care depinde.

Obiecte dependente:	Obiecte referite
View, Table	*Table, Secventa*
Procedure	*View*
Function	*Procedure*
Package Specification	*Function*
Package Body	*Synonym*
Database Trigger	*Package Specification*
Obiect definit de utilizator	*Obiect definit de utilizator*
Tip colectie	*Tip colectie*

Dacă se modifică definiţia unui obiect referit, obiectul dependent poate (sau nu) să continue să funcţioneze normal.

Există două tipuri de dependenţe:

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

În cazul dependenţelor locale, 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 *opera* şi că procedura *adaug* va reactualiza tabelul *opera* prin intermediul unei vizualizări *nou_opera*.

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 ('opera', 'nou_opera');
```

```
NAME        TYPE          REFENCED_NAME     REFENCED_TYPE
filtru      Procedure     opera             Table
adaug       Procedure     nou_opera         View
nou_opera   View          opera             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:

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

Exemplu:

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

NESTED_LEVEL      TYPE          NAME
           0      Table         opera
           1      View          nou_opera
           2      Procedure     adaug
           1      Procedure     filtru

SELECT  *
FROM    IDEPTREE;

DEPENDENCIES
TABLE  nume_schema.opera
       VIEW nume_schema.nou_opera
            PROCEDURE nume_schema.adaug
       PROCEDURE nume_schema.filtru
```

Dependenţele la distanţă sunt manipulate prin una din modalităţile alese de utilizator: modelul *timestamp* (implicit) sau modelul *signature*.

Fiecare unitate *PL/SQL* are un *timestamp* care este setat când unitatea este modificată (creata sau recompilata) 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 (referit) bazei are momentul ultimei modificări mai recent ca cel al obiectului dependent, atunci obiectul dependent va fi recompilat.

Modelul *signature* determină momentul la care obiectele bazei distante trebuie recompilate. Când este creată o procedură, o *signature* este depusă în dicţionarul datelor, alături de *p-code*. Aceasta conţine: numele construcţiei *PLSQL* (*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. Daca signatura nu se schimba, atunci executia continua.

Recompilarea procedurilor şi funcţiilor dependente este fără succes dacă:

- obiectul referit este distrus (*DROP*) sau redenumit (*RENAME*);
- tipul coloanei referite este schimbat;
- coloana referita este stearsa;
- o vizualizare referită este înlocuită printr-o vizualizare având alte coloane;
- lista parametrilor unei proceduri referite este modificată.

Recompilarea procedurilor şi funcţiilor dependente este cu succes dacă:

- tabelul referit are noi coloane;
- nici o coloana nou definita nu are restrictia NOT NULL;
- tipul coloanelor referite nu s-a schimbat;
- un tabel "private" este sters, dar exista un tabel "public" avand acelasi nume si structura;
- toate comenzile *INSERT* contin efectiv lista coloanelor;
- corpul *PL/SQL* a unei proceduri referite a fost modificat şi recompilat cu succes.

Cum pot fi minimizate erorile datorate dependenţelor?

- utilizând comenzi *SELECT* cu opţiunea *;
- incluzând lista coloanelor in cadrul comenzii *INSERT*;
- declarând variabile cu atributul *%TYPE*;
- declarând înregistrări cu atributul *%ROWTYPE*.

În concluzie:

- 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. 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,* singura modalitate 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 *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* era 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). O clauză specială (*AS EXTERNAL*) este utilizată (în cadrul comenzii *CREATE OR REPLACE PROCEDURE*) pentru crearea unui *wrapper*. 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*) între tipurile de date (clauza *PARAMETERS*). Ultimele versiuni renunta 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 când este necesar acest lucru. 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. prin utilizarea clauzei *AS LANGUAGE,* un *wrapper* poate include specificaţii de apelare. 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 *C* care este 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 de adresă al *server*-ului. *JVM* (*Java Virtual Machine*) de pe pe *server* va executa metoda *Java* direct, fără a fi necesar procesul *extproc*.

Maniera de a încărca depinde de limbajul în care este scrisă rutina (*C* sau *Java*).

- 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 spre 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);
```

Scrierea "c_calc" este corecta, iar " " implica ca stocarea este *case sensitive*, altfel implicit se depune numele cu litere mari.

Procedura poate fi apelata dintr-un bloc *PL/SQL*:

```
DECLARE
  emp_id   NUMBER;
  procent  NUMBER;
BEGIN
  …
  calc(emp_id, procent);
  …
END;
```

Rutina 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 aparţinând unui pachet, metode ale unui tip obiect, declanşatori bază de date, comenzi *SQL* care apelează funcţii (în acest caz trebuie utilizată 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 programe *Java* este folosirea interactivă în *iSQL*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*). Un parametru input poate fi vector, tabel *PL/SQL*, *REF CURSOR*. 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*:**

```
CREATE FUNCTION ff(p SYS_REFCURSOR)
        RETURN cartype PIPELINED IS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN  …  END;
```

In timpul executiei paralele, fiecare instanta a functiei tabel va crea o tranzactie independenta.

Urmatoarele comenzi sunt incorecte.

```
UPDATE ff(CURSOR (SELECT * FROM tab))
  SET col = valoare;
INSERT INTO ff(…)
  VALUES('orice', 'vrei');
```

***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*).

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. 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 începută de altă tranzacţie, numită tranzacţie principală. Tranzacţia autonomă permite suspendarea tranzacţiei principale, executarea de comenzi *SQL*, *commit*-ul şi *rollback*-ul acestor operaţii.

Odată începută, tranzacţia autonomă este independentă în sensul că nu împarte 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.

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

```
CREATE PACKAGE exemplu AS
  …
  FUNCTION autono(x INTEGER) RETURN real;
END exemplu;
CREATE PACKAGE BODY exemplu AS
  …
  FUNCTION autono(x INTEGER) RETURN real IS
    PRAGMA AUTONOMOUS_TRANSACTION;
    z real;
  BEGIN
  …
  END;
END exemplu;
```

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

Observaţii:

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

Cu toate că o tranzacţie autonomă este începută de altă tranzacţie, ea **nu** este o tranzacţie încuibărită deoarece:

- nu partajează resurse cu tranzacţia principală;
- nu depinde de tranzacţia principală (de exemplu, dacă tranzacţia principală este *rollback*, atunci tranzacţiile încuibărite sunt de asemenea *rollback*, dar tranzacţia autonomă nu este *rollback*);
- schimbările *commit* din tranzacţii autonome sunt vizibile imediat altor tranzacţii, pe când cele de la tranzacţii încuibărite sunt vizibile doar după ce tranzacţia principală este *commit*.

Pachete în *PL/SQL*

Pachetul (*package*) permite încapsularea într-o unitate logică în baza de date a procedurilor, funcţiilor, cursoarelor, tipurilor, constantelor, variabilelor şi excepţiilor.

Pachetele sunt unităţi de program care sunt compilate, depanate şi testate, sunt obiecte ale bazei de date care grupează tipuri, obiecte şi subprograme *PL/SQL* având o legătură logică între ele.

De ce sunt importante?

Atunci când este referenţiat un pachet (când este apelată pentru prima dată o construcţie a pachetului), întregul pachet este încărcat în *SGA*, zona globală a sistemului, şi este pregătit pentru execuţie. Plasarea pachetului în *SGA* (zona globala sistem) reprezintă avantajul vitezei de execuţie, deoarece *server*-ul nu mai trebuie să aducă informaţia despre pachet de pe disc, aceasta fiind deja în memorie. Prin urmare, apeluri ulterioare ale unor construcţii din acelaşi pachet, nu solicită operaţii *I*/*O* de pe disc. De aceea, ori de câte ori apare cazul unor proceduri şi funcţii înrudite care trebuie să fie executate împreună, este convenabil ca acestea să fie grupate într-un pachet stocat. Este de subliniat că în memorie există o singură copie a unui pachet, pentru toţi utilizatorii.

Spre deosebire de subprograme, pachetele nu pot:

- fi apelate,
- transmite parametri,
- fi încuibărite.

Un pachet are două parţi, fiecare fiind stocată separat în dicţionarul datelor.

- Specificarea pachetului (*package specification*) – partea „vizibilă”, adică interfaţa cu aplicaţii sau cu alte unităţi program. Se declară tipuri, constante, variabile, excepţii, cursoare şi subprograme folositoare utilizatorului.
- Corpul pachetului (*package body*) – partea „acunsă”, mascată de restul aplicaţiei, adică realizarea specificaţiei. Corpul defineşte cursoare şi subprograme, implementând specificaţia. Obiectele conţinute în corpul pachetului sunt fie private, fie publice.

Prin urmare, specificaţia defineşte interfaţa utilizatorului cu pachetul, iar corpul pachetului conţine codul care implementează operaţiile definite în specificaţie. Crearea unui pachet se face în două etape care presupun crearea specificaţiei pachetului şi crearea corpului pachetului.

Un pachet poate cuprinde, fie doar partea de specificaţie, fie specificaţia şi corpul pachetului. Dacă conţine doar specificaţia, atunci evident pachetul conţine doar definiţii de tipuri şi declaraţii de date.

Corpul pachetului poate fi schimbat fără schimbarea specificaţiei pachetului. Dacă specificaţia este schimbată, aceasta invalidează automat corpul pachetului, deoarece corpul depinde de specificaţie.

Specificaţia şi corpul pachetului sunt unităţi compilate separat. Corpul poate fi compilat doar după ce specificaţia a fost compilată cu succes.

Un pachet are următoarea formă generală:

CREATE PACKAGE nume_pachet {IS | AS} -- specificaţia
/* interfaţa utilizator, care conţine: declaraţii de tipuri şi obiecte publice, specificaţii de subprograme */
***END* [*nume_pachet*];**
CREATE PACKAGE BODY nume_pachet {IS | AS} -- corpul
/* implementarea, care conţine: declaraţii de obiecte şi tipuri private, corpuri de subprograme specificate în partea de interfaţă */
[*BEGIN*]
/* instrucţiuni de iniţializare, executate o singură dată când pachetul este invocat prima oară de către sesiunea utilizatorului */
***END* [*nume_pachet*];**

Specificaţia unui pachet

Specificaţia unui pachet cuprinde declararea procedurilor, funcţiilor, constantelor, variabilelor şi excepţiilor care pot fi accesibile utilizatorilor, adică declararea obiectelor de tip *PUBLIC* din pachet. Acestea pot fi utilizate în proceduri sau comenzi care nu aparţin pachetului, dar care au privilegiul *EXECUTE* asupra acestuia.

Variabilele declarate în specificaţia unui pachet sunt globale pachetului şi sesiunii. Ele sunt iniţializate (implicit) prin valoarea *NULL*, evident dacă nu este specificată explicit o altă valoare.

CREATE* [*OR REPLACE*] *PACKAGE* [*schema*.]*nume_pachet
[*AUTHID* {*CURRENT_USER* | *DEFINER*}]
{*IS* | *AS*}
***specificaţie_PL/SQL*;**

Specificaţie_PL/SQL poate include declaraţii de tipuri, variabile, cursoare, excepţii, funcţii, proceduri, pragma etc. În secţiunea declarativă, un obiect trebuie declarat înainte de a fi referit.

Opţiunea *OR REPLACE* este specificată dacă există deja corpul pachetului. Clauzele *IS* şi *AS* sunt echivalente, dar dacă se foloseşte *PROCEDURE BUILDER* este necesară opţiunea *IS*.

Clauza *AUTHID* specifică faptul ca subprogramele pachetului se execută cu drepturile proprietarului (implicit) sau ale utilizatorului curent. De asemenea, această clauză precizează dacă referinţele la obiecte sunt rezolvate în schema proprietarului subprogramului sau a utilizatorului curent.

Corpul unui pachet

Corpul unui pachet conţine codul *PL/SQL* pentru obiectele declarate în specificaţia acestuia şi obiectele private pachetului. De asemenea, corpul poate include o secţiune declarativă în care sunt specificate definiţii locale de tipuri, variabile, constante, proceduri şi funcţii locale. Obiectele private sunt vizibile numai în interiorul corpului pachetului şi pot fi accesate numai de către funcţiile şi procedurile din pachetul respectiv. Corpul pachetului este opţional şi nu este necesar să fie creat dacă specificaţia pachetului nu conţine declaraţii de proceduri sau funcţii.

Este importantă ordinea în care subprogramele sunt definite în interiorul corpului pachetului. O variabilă trebuie declarată înainte ca să fie referită de altă variabilă sau subprogram, iar un subprogram privat trebuie declarat sau definit înainte de a fi apelat de alte subprograme.

CREATE* [*OR REPLACE*] *PACKAGE BODY* [*schema*.]*nume_pachet
{*IS* | *AS*}
***corp_pachet*;**

Un pachet este instanţiat când este apelat prima dată. Aceasta presupune că pachetul este citit de pe disc în memorie şi este executat codul compilat a subprogramului apelat. În acest moment, memoria este alocată tuturor variabilelor definite în pachet.

În multe cazuri este necesar să se facă o iniţializare atunci când pachetul este instanţiat prima dată într-o sesiune. Aceasta se realizează prin adăugarea unei secţiuni de iniţializare (opţională) în corpul pachetului secţiune încadrată între cuvintele cheie *BEGIN* şi *END*. Secţiunea conţine un cod de iniţializare care este executat atunci când pachetul este invocat pentru prima dată.

Crearea pachetului face ca acesta să fie disponibil pentru utilizatorul care l-a creat sau orice cont de utilizator căruia i s-a acordat privilegiul *EXECUTE*.

Referinţa la o declaraţie sau la un obiect specificat în pachet se face prefixând numele obiectului cu numele pachetului. În corpul pachetului, obiectele din specificaţie pot fi referite fără a specifica numele pachetului.

Procesul de creare a specificaţei şi corpului unui pachet urmează acelaşi algoritm ca cel întâlnit în crearea subprogramelor *PL/SQL* independente.

- sunt verificate erorile sintactice şi semantice, iar modulul este depus în dicţionarul datelor;
- sunt verificate instrucţiunile *SQL* individuale, adică dacă obiectele referite există şi dacă utilizatorul le poate accesa;
- sunt comparate declaraţiile de subprograme din specificaţia pachetului cu cele din corpul pachetului (dacă au acelaşi număr şi tip de parametri). Orice eroare detectată la compilarea specificaţiei sau a corpului pachetului este marcată în dicţionarul datelor.

După ce specificaţia şi corpul pachetului sunt compilate, ele devin obiecte în schema curentă. În vizualizarea *USER_OBJECTS* din dicţionarul datelor, vor fi două noi linii:

OBJECT_TYPE	***OBJECT NAME***
PACKAGE	***nume_pachet***
PACKAGE BODY	***nume_pachet***

Modificarea şi suprimarea pachetelor

Modificarea unui pachet presupune de fapt recompilarea sa (pentru a putea modifica metoda de acces şi planul de execuţie) şi se realizează prin comanda:

***ALTER PACKAGE* [*schema*.]*nume_pachet COMPILE* [*PACKAGE* | *BODY*]**

Schimbarea corpului pachetului nu cere recompilarea construcţiilor dependente, în timp ce schimbări în specificaţia pachetului solicită recompilarea fiecărui subprogram stocat care referenţiază pachetul.

Dacă se doreşte modificarea sursei, utilizatorul poate recrea pachetul (cu opţiunea *REPLACE*) pentru a-l înlocui pe cel existent.

***DROP PACKAGE* [*schema.*]*nume_pachet* [*PACKAGE* | *BODY*]**

Dacă în cadrul comenzii apare opţiunea *BODY* este distrus doar corpul pachetului, în caz contrar sunt distruse atât specificaţia, cât şi corpul pachetului. Dacă pachetul este distrus, toate obiectele dependente de acesta devin invalide. Dacă este distrus numai corpul, toate obiectele dependente de acesta rămân valide. În schimb, nu pot fi apelate subprogramele declarate în specificaţia pachetului, până când nu este recreat corpul pachetului.

Pentru ca un utilizator să poată distruge un pachet trebuie ca fie pachetul să aparţină schemei utilizatorului, fie utilizatorul să posede privilegiul de sistem *DROP ANY PROCEDURE*.

Una din posibilităţile interesante oferite de pachetele *PL/SQL* este aceea de a crea proceduri/funcţii *overload*. Procesul implică definirea unui număr de proceduri cu acelaşi nume, dar care diferă prin numărul şi tipul parametrilor pe care le folosesc în fiecare instanţă a procedurii implementată separat în corpul pachetului. Acest tip de programare este folositor când este necesară o singură funcţie care să execute aceeaşi operaţie pe obiecte de tipuri diferite (diferite tipuri de parametri de intrare). Când este apelată o procedură *overload* sistemul decide pe baza tipului şi numărului de parametri care instanţă a procedurii va fi executată. Numai subprogramele locale sau aparţinând unui pachet pot fi *overload*. Subprogramele *stand-alone* nu pot fi *overload*.

Utilizarea unui pachet se realizează în funcţie de mediul (*SQL* sau *PL/SQL*) care solicită un obiect din pachetul respectiv.

1) În *PL/SQL* se face prin referirea:

***nume_pachet.nume_componentă* [(*listă_de_argumente*)];**

2) În *SQL*Plus* se face prin comanda:

***EXECUTE nume_pachet.nume_componentă* [(*listă_de_argumente*)]**

Exemplu:

Să se creeze un pachet ce include o procedură prin care se verifică dacă o combinaţie specificată dintre atributele *cod_artist* şi *stil* este o combinaţie care există în tabelul *opera*.

```
CREATE PACKAGE verif_pachet IS
  PROCEDURE verifica
      (p_idartist IN opera.cod_artist%TYPE,
       p_stil     IN opera.stil%TYPE);
END verif_pachet;
/
CREATE OR REPLACE PACKAGE BODY verif_pachet IS
  i NUMBER := 0;
  CURSOR opera_cu IS
    SELECT cod_artist, stil
    FROM   opera;
  TYPE opera_table_tip IS TABLE OF opera_cu%ROWTYPE
    INDEX BY BINARY INTEGER;
  art_stil opera_table_tip;
  PROCEDURE verifica
      (p_idartist IN opera.cod_artist%TYPE,
       p_stil     IN opera.stil%TYPE);
  IS
  BEGIN
    FOR k IN art_stil.FIRST..art_stil.LAST LOOP
      IF p_idartist = art_stil(k).cod_artist
         AND p_stil = art_stil(k).stil THEN
           RETURN;
      END IF;
    END LOOP;
    RAISE_APPLICATION_ERROR (-20777,'nu este buna
                             combinatia');
  END verifica;
BEGIN
  FOR ope_in IN opera_cu LOOP
    art_stil(i) := ope_in;
    i := i+1;
  END LOOP;
END verif_pachet;
/
```

Utilizarea în *PL/SQL* a unui obiect (*verifica*) din pachet se face prin:

```
verif_pachet.verifica (7935, 'impresionism');
```

Utilizarea în *SQL*Plus* a unui obiect (*verifica*) din pachet se face prin:

```
EXECUTE verif_pachet.verifica (7935, 'impresionism')
```

Observaţii:

- Un declanşator nu poate apela o procedură sau o funcţie ce conţine comenzile *COMMIT, ROLLBACK, SAVEPOINT*. Prin urmare, pentru flexibilitatea apelului (de către declanşatori) subprogramelor conţinute în pachete, trebuie verificat că nici una din procedurile sau funcţiile pachetului nu conţin aceste comenzi.
- Procedurile şi funcţiile conţinute într-un pachet pot fi referite din fişiere *iSQL**Plus, din subprograme stocate *PL/SQL*, din aplicaţii client (de exemplu, *Oracle Forms* sau *Power Builder*), din declanşatori (bază de date), din programe aplicaţie scrise în limbaje de generaţia a 3-a.
- Într-un pachet nu pot fi referite variabile gazdă.
- Într-un pachet, mai exact în corpul acestuia, sunt permise declaraţii *forward*.
- Functiile unui pachet pot fi utilizate (cu restrictii) in comenzi *SQL*.

Dacă un subprogram dintr-un pachet este apelat de un subprogram *stand-alone* trebuie remarcat că:

- dacă corpul pachetului se schimbă, dar specificaţia pachetului nu se schimbă, atunci subprogramul care referă o construcţie a pachetului rămâne valid;
- dacă specificaţia pachetului se schimbă, atunci subprogramul care referă o construcţie a pachetului, precum şi corpul pachetului sunt invalidate.

Dacă un subprogram *stand-alone* referit de un pachet se schimbă, atunci întregul corp al pachetului este invalidat, dar specificaţia pachetului rămâne validă.

Pachete predefinite

PL/SQL conţine pachete predefinite utilizabile pentru dezvoltare de aplicaţii şi care sunt deja compilate în baza de date. Aceste pachete adaugă noi funcţionalităţi limbajului, protocoale de comunicaţie, acces la fişierele sistemului etc. Apelarea unor proceduri din aceste pachete solicită prefixarea numelui procedurii cu numele pachetului.

Dintre cele mai importante pachete predefinite se remarcă:

- *DBMS_OUTPUT* (permite afişarea de informaţii);
- *DBMS_DDL* (furnizează accesul la anumite comenzi *LDD* care pot fi folosite în programe *PL/SQL*);

- *UTL_FILE* (permite citirea din fişierele sistemului de operare, respectiv scrierea în astfel de fişiere);
- *UTL_HTTP* (foloseşte *HTTP* pentru accesarea din *PL/SQL* a datelor de pe *Internet*);
- *UTL_TCP* (permite aplicaţiilor *PL/SQL* să comunice cu *server*-e externe utilizând protocolul *TCP/IP*);
- *DBMS_JOB* (permite planificarea programelor *PL/SQL* pentru execuţie şi execuţia acestora);
- *DBMS_SQL* (accesează baza de date folosind *SQL* dinamic);
- *DBMS_PIPE* (permite operaţii de comunicare între două sau mai multe procese conectate la aceeaşi instanţă *Oracle*);
- *DBMS_LOCK* (permite folosirea exclusivă sau partajată a unei resurse),
- *DBMS_SNAPSHOT* (permite exploatarea clişeelor);
- *DBMS_UTILITY* (oferă utilităţi *DBA*, analizează obiectele unei scheme particulare, verifică dacă *server*-ul lucrează în mod paralel etc.);
- *DBMS_LOB* (realizează accesul la date de tip *LOB*, permiţând compararea datelor *LOB*, adăugarea de date la un *LOB*, copierea datelor dintr-un *LOB* în altul, ştergerea unor porţiuni din date *LOB*, deschiderea, închiderea şi regăsirea de informaţii din date *BFILE* etc).

DBMS_STANDARD este un pachet predefinit fundamental prin care se declară tipurile, excepţiile, subprogramele care sunt utilizabile automat în programele *PL/SQL*. Conţinutul pachetului este vizibil tuturor aplicaţiilor. Pentru referirea componentelor sale nu este necesară prefixarea cu numele pachetului. De exemplu, utilizatorul poate folosi ori de câte ori are nevoie în aplicaţia sa funcţia *ABS* (*x*), aparţinând pachetului *DBMS_STANDARD*, care reprezintă valoarea absolută a numărului *x*, fără a prefixa numele funcţiei cu numele pachetului.

Pachetul *DBMS_OUTPUT*

DBMS_OUTPUT permite afişarea de informaţii atunci când se execută un program *PL/SQL* (trimite mesagele din orice bloc *PL/SQL* intr-un buffer in *BD*).

DBMS_OUTPUT lucrează cu un *buffer* (conţinut în *SGA*) în care poate fi scrisă informaţie utilizând procedurile *PUT*, *PUT_LINE* şi *NEW_LINE*. Această informaţie poate fi regăsită folosind procedurile *GET_LINE* şi *GET_LINES*. Procedura *DISABLE* dezactivează toate apelurile la pachetul *DBMS_OUTPUT* (cu excepţia procedurii *ENABLE*) şi curăţă *buffer*-ul de orice informaţie.

Inserarea în *buffer* a unui sfârşit de linie se face prin procedura *NEW_LINE*.

Procedura *PUT* depune (scrie) informaţie în *buffer*, informaţie care este de tipul *NUMBER, VARCHAR2* sau *DATE*. *PUT_LINE* are acelaşi efect ca procedura *PUT*, dar inserează şi un sfârşit de linie. Procedurile *PUT* şi *PUT_LINE* sunt *overload*, astfel încât informaţia poate fi scrisă în format nativ (*VARCHAR2, NUMBER* sau *DATE*).

Procedura *GET_LINE* regăseşte o singură linie de informaţie (de dimensiune maximă 255) din *buffer* (dar sub formă de şir de caractere). Procedura *GET_LINES* regăseşte mai multe linii (*nr_linii*) din *buffer* şi le depune într-un tablou (*nume_tab*) *PL/SQL* având tipul şir de caractere. Valorile sunt plasate în tabel începând cu linia zero. Specificaţia este următoarea:

```
TYPE string255_table IS TABLE OF VARCHAR2(255)
  INDEX BY BINARY_INTEGER;
PROCEDURE GET_LINES
  (nume_tab OUT string255_table,
   nr_linii IN OUT INTEGER);
```

Parametrul *nr_linii* este şi parametru de tip *OUT*, deoarece numărul liniilor solicitate poate să nu coincidă cu numărul de linii din *buffer*. De exemplu, pot fi solicitate 10 linii, iar în *buffer* sunt doar 6 linii. Atunci doar primele 6 linii din tabel sunt definite.

Dezactivarea referirilor la pachet se poate realiza prin procedura *DISABLE*, iar activarea referirilor se face cu ajutorul procedurii *ENABLE*.

Exemplu:

Următorul exemplu plasează în *buffer* (apelând de trei ori procedura *PUT*) toate informaţiile într-o singură linie.

```
DBMS_OUTPUT.PUT(:opera.valoare||:opera.cod_artist);
DBMS_OUTPUT.PUT(:opera.cod_opera);
DBMS_OUTPUT.PUT(:opera.cod_galerie);
```

Dacă aceste trei comenzi sunt urmate de comanda

```
DBMS_OUTPUT.NEW_LINE;
```

atunci informaţia respectivă va fi găsită printr-un singur apel *GET_LINE*. Altfel, nu se va vedea nici un efect al acestor comenzi deoarece *PUT* plasează informaţia în *buffer*, dar nu adaugă sfârşit de linie.

Când este utilizat pachetul *DBMS_OUTPUT* pot să apară erorile *buffer overflow* şi *line length overflow*. Tratarea acestor erori se face apelând procedura *RAISE_APPLICATION_ERROR* din pachetul standard *DBMS_STANDARD*.

Pachetul *DBMS_SQL*

Pachetul *DBMS_SQL* permite folosirea dinamică a comenzilor *SQL* în proceduri stocate sau în blocuri anonime şi analiza gramaticală a comenzilor *LDD*. Aceste comenzi nu sunt incorporate în programul sursă, ci sunt depuse în şiruri de caractere. O comandă *SQL* dinamică este o instrucţiune *SQL* care conţine variabile ce se pot schimba în timpul execuţiei. De exemplu, pot fi utilizate instrucţiuni *SQL* dinamice pentru:

- a crea o procedură care operează asupra unui tabel al cărui nume nu este cunoscut decât în momentul execuţiei;
- a scrie şi executa o comandă *LDD*;
- a scrie şi executa o comandă *GRANT, ALTER SESSION* etc.

În *PL/SQL* aceste comenzi nu pot fi executate static. Pachetul *DBMS_SQL* permite, de exemplu, ca într-o procedură stocată să foloseşti comanda *DROP TABLE*. Evident, folosirea acestui pachet pentru a executa comenzi *LDD* poate genera interblocări. De exemplu, pachetul este utilizat pentru a şterge o procedură care însă este utilizată.

SQL dinamic suportă toate tipurile de date *SQL*, dar nu suportă cele specifice *PL/SQL*. Unica excepţie o constituie faptul că o înregistrare *PL/SQL* poate să apară în clauza *INTO* a comenzii *EXECUTE IMMEDIATE*.

Orice comandă *SQL* trebuie să treacă prin nişte etape, cu observaţia că anumite etape pot fi evitate. Etapele presupun: analizarea gramaticală a comenzii, adică verificarea sintactică a comenzii, validarea acesteia, asigurarea că toate referinţele la obiecte sunt corecte şi asigurarea că există privilegiile referitoare la acele obiecte (*parse*); obţinerea de valori pentru variabilele de legătură din comanda *SQL* (*binding variables*); executarea comenzii (*execute*); selectarea rândurilor rezultatului şi încărcarea acestor rânduri (*fetch*).

Dintre subprogramele pachetului *DBMS_SQL*, care permit implementarea etapelor amintite anterior se remarcă:

- *OPEN_CURSOR* (deschide un nou cursor, adică se stabileşte o zonă de memorie în care este procesată comanda *SQL*);
- *PARSE* (stabileşte validitatea comenzii *SQL*, adică se verifică sintaxa instrucţiunii şi se asociază cursorului deschis);
- *BIND VARIABLE* (leaga valoarea data de variabila corespunzatoare din comanda *SQL* analizata)
- *EXECUTE* (execută comanda *SQL* şi returnează numărul de linii procesate);

- *FETCH_ROWS* (regăseşte o linie pentru un cursor specificat, iar pentru mai multe linii foloseşte un *LOOP*);
- *CLOSE_CURSOR* (închide cursorul specificat).

Să se construiască o procedură care foloseşte *SQL* dinamic pentru a şterge liniile unui tabel specificat (*num_tab*). Subprogramul furnizează ca rezultat numărul liniilor şterse (*nr_lin*).

```
CREATE OR REPLACE PROCEDURE sterge_linii
  (num_tab IN VARCHAR2, nr_lin OUT NUMBER)
AS
  nume_cursor INTEGER;
BEGIN
  nume_cursor := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE (nume_cursor, 'DELETE FROM' ||
                  num_tab, DBMS_SQL.V7);
  nr_lin := DBMS_SQL.EXECUTE (nume_cursor);
  DBMS_SQL.CLOSE_CURSOR (nume_cursor);
END;
```

Argumentul *DBMS_SQL.V7* reprezintă modul (versiunea 7) în care *Oracle* tratează comenzile *SQL*. Ştergerea efectivă a liniilor tabelului *opera* se realizează:

```
VARIABLE linii_sterse NUMBER
EXECUTE sterge_linii ('opera', :linii_sterse)
PRINT linii_sterse
```

Pentru a executa o instrucţiune *SQL* dinamic poate fi utilizată şi comanda *EXECUTE IMMEDIATE*. Comanda conţine o clauză opţională *INTO* care este utilizabilă pentru interogări ce returnează o singură linie. Pentru o cerere care returnează mai multe linii trebuie folosite comenzile *OPEN FOR*, *FETCH*, *CLOSE*.

```
CREATE OR REPLACE PROCEDURE sterge_linii
  (num_tab IN VARCHAR2, nr_lin OUT NUMBER)
IS
BEGIN
  EXECUTE IMMEDIATE 'DELETE FROM' || num_tab;
  nr_lin := SQL%ROWCOUNT;
END;
```

Procedura se poate apela printr-o secvenţă identică cu cea prezentată anterior.

Pachetul *DBMS_DDL*

Pachetul *DBMS_DDL* furnizează accesul la anumite comenzi *LDD* care pot fi folosite în subprograme *PL/SQL* stocate. În felul acesta, pot fi accesate (în *PL/SQL*) comenzile *ALTER* sau *ANALYZE*.

Pachetul include procedura *ALTER_COMPILE* care permite recompilarea programului modificat (procedură, funcţie, declanşator, pachet, corp pachet).

ALTER_COMPILE (*tip_obiect*, *nume_schema*, *nume_obiect*);

Procedura este echivalentă cu instrucţiunea *SQL*:

***ALTER PROCEDURE | FUNCTION | PACKAGE* [*schema.*] *nume COMPILE* [*BODY*]**

Cu ajutorul procedurii *ANALYZE_OBJECT* poate fi analizat un obiect de tip *table*, *cluster* sau *index*. Procedura furnizează statistici referitoare la obiectele amintite. De exemplu, se pot obţine numărul liniilor unui tabel, numărul de blocuri ale unui tabel, lungimea medie a unei linii, numărul valorilor distincte ale unei col., numărul elementelor *null* dintr-o coloană, distribuţia datelor (histograma) etc.

***ANALYZE_OBJECT* (*tip_obiect, nume_schema, nume_obiect, metoda, număr_linii_estimate, procent, opţiune_metoda, nume_partiţie*);**

Metodele care pot fi utilizate sunt *COMPUTE*, *ESTIMATE* sau *DELETE*. Prin aceste metode se cuantifică distribuţia datelor şi caracteristicile de stocare. *DELETE* determină ştergerea statisticilor (depuse în *DD*) referitoare la obiectul analizat. *COMPUTE* calculează statistici referitoare la un obiect analizat şi le depune în *DD*, iar *ESTIMATE* estimează statistici. Statisticile calculate sau estimate sunt utilizate pentru optimizarea planului de execuţie a comenzilor *SQL* care accesează obiectele analizate.

Procedura este echivalentă cu instrucţiunea:

***ANALYZE TABLE | CLUSTER | INDEX* [*nume_schema*] *nume_obiect* [*metoda*]**
***STATISTICS* [*SAMPLE n*] [*ROWS | PERCENT*]]**

Dacă *nume_schema* este *null*, atunci se presupune că este vorba de schema curentă. Dacă *tip_obiect* este diferit de *table*, *index* sau *cluster*, se declanşează eroarea *ORA - 20001*. Parametrul *procent* reprezintă procentajul liniilor de estimat şi este ignorat dacă este specificat numărul liniilor de estimat (*număr_linii_estimate*). Implicit, ultimele patru argumente ale procedurii iau valoarea *null*.

Argumentul *opţiune_metoda* poate avea forma:

[*FOR TABLE*] [*FOR ALL INDEXES*] [*FOR ALL* [*INDEXED*] *COLUMNS*] [*SIZE n*]

Pentru metoda *ESTIMATE* trebuie să fie prezentă una dintre aceste opţiuni.

Exemplu:

Utilizând pachetul *DBMS_DDL* şi metoda *COMPUTE*, să se creeze o procedură care analizează un obiect furnizat ca parametru acestei proceduri.

```
CREATE OR REPLACE PROCEDURE analiza
  (p_obiect_tip  IN VARCHAR2,
   p_obiect_nume IN VARCHAR2);
IS
BEGIN
  DBMS_DDL.ANALYZE_OBJECT(p_obiect_tip, USER,
                          UPPER(p_obiect_nume), 'COMPUTE');
END;
/
```

Procedura se testează (relativ la tabelul *opera*) în felul următor:

```
EXECUTE analiza ('TABLE', 'opera')
SELECT LAST_ANALYZED
FROM   USER_TABLES
WHERE  TABLE_NAME = 'opera';
```

Pachetul *DBMS_JOB*

Pachetul *DBMS_JOB* este utilizat pentru planificarea programelor *PL/SQL* în vederea execuţiei. Cu ajutorul acestui pachet se pot executa programe *PL/SQL* la momente determinate de timp, se pot şterge sau suspenda programe din lista de planificări pentru execuţie, se pot rula programe de întreţinere în timpul perioadelor de utilizare scăzută etc.

Dintre subprogramele acestui pachet se remarcă:

- *SUBMIT* – adaugă un nou *job* în coada de aşteptare a *job*-urilor;
- *REMOVE* – şterge un *job* specificat din coada de aşteptare a *job*-urilor;
- *RUN* – execută imediat un *job* specificat;
- *BROKEN* – dezactivează execuţia unui *job* care este marcat ca *broken* (implicit, orice *job* este *not broken*, iar un *job* marcat *broken* nu se execută);
- *CHANGE* – modifică argumentele *WHAT*, *NEXT_DATE*, *INTERVAL*;
- *WHAT* – furnizează descrierea unui *job* specificat;
- *NEXT_DATE* – dă momentul următoarei execuţii a unui *job*;
- *INTERVAL* – furnizează intervalul între diferite execuţii ale unui *job*.

Fiecare dintre subprogramele pachetului are argumente specifice. De exemplu, procedura *DBMS_JOB.SUBMIT* are ca argumente:

- *JOB* – de tip *OUT*, un identificator pentru *job* (*BINARY_INTEGER*);

- *WHAT* – de tip *IN*, codul *PL/SQL* care va fi executat ca un *job* (*VARCHAR2*);
- *NEXT_DATE* – de tip *IN*, data următoarei execuţii a *job*-ului (implicit este *SYSDATE*);
- *INTERVAL* – de tip *IN*, funcţie care furnizează intervalul dintre execuţiile *job*-ului (*VARCHAR2*, implicit este *null*);
- *NO_PARSE* – de tip *IN*, variabilă logică care indică dacă *job*-ul trebuie analizat gramatical (*BOOLEAN*, implicit este *FALSE*).

Dacă unul dintre parametri *WHAT*, *INTERVAL* sau *NEXT_DATE* are valoarea *null*, atunci este folosită ultima valoare asignată acestora.

***Exemplu*:**

Să se utilizeze pachetul *DBMS_JOB* pentru a plasa pentru execuţie în coada de aşteptare a *job*-urilor, procedura *verifica* din pachetul *verif_pachet.*

```
VARIABLE num_job NUMBER
BEGIN
  DBMS_JOB.SUBMIT(
    job => :num_job,
    what                                                    =>
'verif_pachet.verifica(8973,''impresionism'');'
    next_date => TRUNC(SYSDATE+1),
    interval => 'TRUNC(SYSDATE+1)');
  COMMIT;
END;
/
PRINT num_job
```

Vizualizarea *DBA_JOBS* din dicţionarul datelor furnizează informaţii referitoare la starea *job*-urilor din coada de aşteptare, iar vizualizarea *DBA_JOBS_RUNNING* conţine informaţii despre *job*-urile care sunt în curs de execuţie. Vizualizările pot fi consultate doar de utilizatorii care au privilegiul *SYS.DBA_JOBS*.

Exemplu:

```
SELECT JOB, LOG_USER, NEXT_DATE, BROKEN, WHAT
FROM   DBA_JOBS;
```

Pachetul *UTL_FILE*

Pachetul *UTL_FILE* permite programului *PL/SQL* citirea din fişierele sistemului de operare, respectiv scrierea în aceste fişiere. El este utilizat pentru exploatarea fişierelor text.

Folosind componentele acestui pachet (functiile *FOPEN* si *IS_OPEN*; procedurile *GET_LINE*, *PUT*, *PUT_LINE*, *PUTF*, *NEW_LINE*, *FCLOSE*, *FCLOSEALL*, *FFLUSH*) se pot deschide fisiere, obtine text din fisiere, scrie text in fisiere, inchide fisiere.

Pachetul procesează fişierele într-o manieră clasică:

- verifică dacă fişierul este deschis (funcţia *IS_OPEN*);
- dacă fişierul nu este deschis, îl deschide şi returnează un *handler* de fişier (de tip *UTL_FILE.FILE_TYPE*) care va fi utilizat în următoarele operaţii *I/O* (funcţia *FOPEN*);
- procesează fişierul (citire/scriere din/în fişier);
- închide fişierul (procedura *FCLOSE* sau *FCLOSEALL*).

Functia *IS_OPEN* verifica daca un fisier este deschis. Are antetul:

```
FUNCTION IS_OPEN (handler_fisier IN FILE_TYPE)
  RETURN BOOLEAN;
```

Functia *FOPEN* deschide un fisier si returneaza un handler care va fi utilizat in urmatoarele operatii I/O. Parametrul *open_mode* este un string care specifica modul cum a fost deschis fisierul.

```
FUNCTION FOPEN (locatia     IN VARCHAR2,
                nume_fisier IN VARCHAR2,
                open_mode   IN VARCHAR2)
  RETURN UTL_FILE.FILE_TYPE;
```

Prin procedura *GET_LINE*, pachetul *UTL_FILE* citeşte o linie de text din fişierul deschis pentru citire şi o plasează într-un *buffer* de tip şir de caractere, iar prin procedurile *PUT* şi *PUT_LINE* scrie un text din *buffer* în fişierul deschis pentru scriere sau adăugare.

Utilizarea componentelor acestui pachet pentru procesarea fişierelor sistemului de operare poate declanşa excepţii, dintre care remarcam:

- *INVALID_PATH* – numele sau locaţia fişierului sunt invalide;
- *INVALID_MODE* – parametrul *OPEN_MODE* (prin care se specifică dacă fişierul este deschis pentru citire, scriere, adăugare) este invalid;
- *INVALID_FILEHANDLE* – *handler*-ul de fişier obţinut în urma deschiderii este invalid;

- *INVALID_OPERATION* – operaţie invalidă asupra fişierului;
- *READ_ERROR* – o eroare a sistemului de operare a apărut în timpul operaţiei de citire;
- *WRITE_ERROR* – o eroare a sistemului de operare a apărut în timpul operaţiei de scriere;
- *INTERNAL_ERROR* – o eroare nespecificată a apărut în *PL/SQL*.

Exceptiile trebuie prefixate cu numele pachetului. Evident, pot aparea si erorile *NO_DATA_FOUND* si *VALUE_ERROR*.

Pachetele *DBMS_PIPE* şi *DBMS_ALERT*

Pachetul *DBMS_PIPE* permite operaţii de comunicare între două sau mai multe sesiuni conectate la aceeaşi bază de date. De exemplu, pachetul poate fi utilizat pentru comunicarea dintre o procedură stocată şi un program *Pro*C*. Comunicarea se face prin conducte (*pipe*). O conductă este o zonă de memorie utilizată de un proces pentru a transmite informaţie altui proces. Informaţia trimisă prin conductă este depusă într-un *buffer* din *SGA*. Toate informaţiile din conductă sunt pierdute atunci când instanţa este închisă.

Conductele sunt asincrone, ele operând independent de tranzacţii. Dacă un anumit mesaj a fost transmis, nu există nici o posibilitate de oprire a acestuia, chiar dacă sesiunea care a trimis mesajul este derulată înapoi (*rollback*).

Pachetul *DBMS_PIPE* este utilizat pentru a trimite mesaje în conductă (*DBMS_PIPE.SEND_MESSAGE*), mesaje ce constau din date de tip *VARCHAR2*, *NUMBER*, *DATE*, *RAW* sau *ROWID*. Tipurile obiect definite de utilizator şi colecţiile nu sunt acceptate de acest pachet.

De asemenea, pachetul poate realiza primirea de mesaje din conductă în *buffer*-ul local (*DBMS_PIPE.RECEIVE_MESSAGE*), accesarea următorului articol din *buffer* (*DBMS_PIPE.UNPACK_MESSAGE*), crearea unei noi conducte (*DBMS_PIPE*. *CREATE_PIPE*) etc.

DBMS_ALERT este similar pachetului *DBMS_PIPE*, fiind utilizat tot pentru comunicarea dintre sesiuni conectate la aceeaşi bază de date. Există totuşi câteva deosebiri esenţiale.

- *DBMS_ALERT* asigură o comunicare sincronă.
- Un mesaj trimis prin *DBMS_PIPE* este primit de un singur destinatar (cititor) chiar dacă există mai mulţi pe conductă, pe când cel trimis prin *DBMS_ALERT* poate fi primit de mai mulţi cititori simultan.
- Dacă două mesaje sunt trimise printr-o conducta (înainte ca ele să fie citite), ambele vor fi primite de destinatar prin *DBMS_PIPE*. În cazul pachetului *DBMS_ALERT*, doar cel de al 2-lea mesaj va fi primit.

Pachete predefinite furnizate de Oracle9i

Oracle9i furnizează o varietate de pachete predefinite care simplifică administrarea bazei de date şi oferă noi funcţionalităţi legate de noile caracteristici ale sistemului. Dintre pachetele introduse în versiunea *Oracle9i* se remarcă:

- *DBMS_REDEFINITION* – permite reorganizarea *online* a tabelelor;
- *DBMS_LIBCACHE* – permite extragerea de comenzi *SQL* şi *PL/SQL* dintr-o instanţă distantă într-una una locală (vor fi compilate local, dar nu executate);
- *DBMS_LOGMNR_CDC_PUBLISH* – realizează captarea schimbărilor din tabelele bazei de date (identifică datele adăugate, modificate sau şterse şi editează aceste informaţii într-o formă utilizabilă în aplicaţii);
- *DBMS_LOGMNR_CDC_SUBSCRIBE* – face posibilă vizualizarea şi interogarea schimbărilor din datele care au fost captate cu pachetul *DBMS_LOGMNR_CDC_PUBLISH*;
- *DBMS_METADATA* – furnizează informaţii despre obiectele bazei de date;
- *DBMS_RESUMABLE* – permite setarea limitelor de spaţiu şi timp pentru o operaţie specificată, operaţia fiind suspendată dacă sunt depăşite aceste limite;
- *DBMS_XMLQUERY*, *DBMS_XMLSAVE*, *DBMS_XMLGEN* – permit prelucrarea şi conversia datelor *XML* (*XMLGEN* converteşte rezultatul unei cereri *SQL* în format *XML*, *XMLQUERY* este similară lui *XMLGEN*, doar că este scrisă în *C*, iar *XMLSAVE* face conversia din format *XML* în date ale bazei);
- *UTL_INADDR* – returnează numele unei gazde locale sau distante a cărei adresă *IP* este cunoscută şi reciproc, returnează adresa *IP* a unei gazde căreia i se cunoaşte numele (de exemplu, *www.oracle.com*);
- *DBMS_AQELM* – furnizează proceduri şi funcţii pentru gestionarea configuraţiei cozilor de mesaje asincrone prin *e-mail* şi *HTTP*;
- *DBMS_FGA* – asigură întreţinerea unor funcţii de securitate;
- *DBMS_FLASHBACK* – permite trecerea la o versiune a bazei de date corespunzătoare unei unităţi de timp specificate sau unui *SCN* (*system change number*) dat, în felul acesta putând fi recuperate linii şterse sau mesaje *e-mail* distruse;
- *DBMS_TRANSFORM* – furnizează subprograme ce permit transformarea unui obiect (expresie *SQL* sau funcţie *PL/SQL*) de un anumit tip (sursă) într-un obiect având un tip (destinaţie) specificat;

Folosirea dinamică a comenzilor *SQL*

SQL dinamic este o parte integrantă a limbajului *SQL* care permite folosirea dinamică a comenzilor sale în proceduri stocate sau în blocuri anonime. Spre deosebire de comenzile statice, care nu se schimbă în timp real, comenzile dinamice se schimbă de la o execuţie la alta. Comenzile dinamice *SQL* pot depinde de anumite valori de intrare furnizate de utilizator sau de procesarea realizată în programul aplicaţie. Ele nu sunt incorporate în programul sursă, ci sunt depuse în şiruri de caractere.

SQL dinamic este o tehnică de programare care permite construirea dinamică a comenzilor la momentul execuţiei (adica, direct în faza de execuţie a blocului *PL/SQL*). Textul comenzii nu este cunoscut la compilare. De exemplu, se creează o procedură care operează asupra unui tabel al cărui nume este cunoscut doar când se execută procedura. În momentul compilării este cunoscută definiţia tabelelor, dar nu şi numele acestora. Există aplicaţii (de exemplu, legate de *data warehouse*) în care la fiecare unitate de timp (de exemplu, sfert de oră) sunt generate noi tabele, toate având aceeaşi structură.

Utilitatea tehnicii *SQL* dinamic este justificată de motive majore, dintre care se remarcă:

- necesitatea de a executa în *PL/SQL*, comenzi *SQL* care nu pot fi apelate în codul *PL/SQL* (de exemplu, *CREATE*, *DROP*, *GRANT*, *REVOKE*, *ALTER SESSION*, *SET ROLE*);
- necesitatea unei flexibilităţi în tratarea comenzilor (de exemplu, posibilitatea de a avea diferite condiţii în clauza *WHERE* a comenzii *SELECT*);
- necunoaşterea completă, la momentul implementării, a comenzii *SQL* care trebuie executată.

Pentru execuţia dinamică a comenzilor *SQL* în *PL/SQL* există două tehnici:

- utilizarea pachetului *DBMS_SQL*;
- *SQL* dinamic nativ.

Dacă s-ar face o comparaţie între *SQL* dinamic nativ şi funcţionalitatea pachetului *DBMS_SQL*, se poate sublinia că *SQL* dinamic nativ:

- este mai uşor de utilizat,
- solicita mai putin cod,
- este mai rapid,
- poate incarca direct linii in recorduri *PL/SQL*,
- suportă toate tipurile acceptate de *SQL* static in *PL/SQL*, inclusiv tipuri definite de utilizator.

Pachetul *DBMS_SQL*, în raport cu *SQL* dinamic nativ:

- poate fi folosit în programe *client-side*;
- suportă comenzi *SQL* mai mari de 32 *KB*;
- permite încărcarea înregistrărilor (procedura *FETCH_ROWS*);
- acceptă comenzi cu clauza *RETURNING* pentru reactualizarea şi ştergerea de linii multiple;
- suportă posibilităţile oferite de comanda *DESCRIBE* (procedura *DESCRIBE_COLUMNS*);
- analizează validitatea unei comenzi *SQL* o singură dată (procedura *PARSE*), permiţând ulterior mai multe utilizări ale comenzii pentru diferite mulţimi de argumente.

SQL dinamic nativ a fost introdus în *Oracle8i*, asigurând plasarea de comenzi *SQL* dinamic în codul *PL/SQL*. Comanda de bază utilizată pentru procesarea dinamică nativă a comenzilor *SQL* şi a blocurilor *PL/SQL* anonime este *EXECUTE IMMEDIATE*. Comanda are următoarea sintaxă:

EXECUTE IMMEDIATE *şir_dinamic*
 [***INTO*** {*def_variabila* [, *def_variabila ...*] | *record*}]
 [***USING*** [***IN*** | ***OUT*** | ***IN OUT***] *argument_bind*
 [, [***IN*** | ***OUT*** | ***IN OUT***] *argument_bind ...*]]
 [{***RETURNING*** | ***RETURN***}
 INTO *argument_bind* [, *argument_bind ...*]];

- *şir_dinamic* este o expresie (şir de caractere) care reprezintă o comandă *SQL* (fără caracter de terminare) sau un bloc *PL/SQL* (având caracter de terminare);
- *def_variabila* reprezintă variabila în care se stochează valoarea coloanei selectate;
- *record* reprezintă înregistrarea în care se depune o linie selectată;
- *argument_bind*, dacă se referă la valori de intrare (*IN*) este o expresie (comandă *SQL* sau bloc *PL/SQL*), iar dacă se referă la valori de ieşire (*OUT*) este o variabilă ce va conţine valoarea selectată de comanda *SQL* sau de blocul *PL/SQL*.
- *INTO* este folosită pentru cereri care întorc o singură linie, iar clauza *USING* pentru a reţine argumentele de legătură.
- Pentru procesarea unei cereri care returnează mai multe linii sunt necesare instrucţiunile *OPEN...FOR*, *FETCH* şi *CLOSE*.
- Prin clauza *RETURNING* sunt precizate variabilele care conţin rezultatele.

***Observaţii*:**

- *SQL* dinamic suportă toate tipurile *SQL*, dar nu acceptă tipuri de date specifice *PL/SQL* (unica excepţie este tipul *RECORD*, care poate să apară în clauza *INTO*).
- În subprogramele *PL/SQL* pot să fie executate dinamic comenzi *SQL* care se referă la obiecte aparţinând unei baze de date distante.
- În anumite situaţii, o comandă *LDD* poate crea o interblocare. De exemplu, o procedură poate genera o interblocare dacă în corpul procedurii există o comandă care şterge chiar procedura respectivă. Prin urmare, niciodată nu pot fi utilizate comenzile *ALTER* sau *DROP* referitoare la un subprogram sau pachet în timp ce se lucrează cu pachetul sau subprogramul respectiv.

***Exemplu*:**

Să se construiască o procedură care poate şterge orice tabel din baza de date. Numele tabelului şters este transmis ca parametru acestei proceduri.

```
CREATE PROCEDURE sterge_tabel (nume_tabel IN VARCHAR2)
AS
BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE :tab'
    USING nume_tabel;
END;
```

La execuţia acestei proceduri va fi semnalată o eroare, deoarece nu pot fi utilizate variabile de legătură ca argument, pentru a transmite numele obiectelor dintr-o schemă. Prin urmare comanda corectă este:

```
EXECUTE IMMEDIATE 'DROP TABLE ' || nume_tabel;
```

***Exemplu*:**

Inserarea unei linii intr-un tabel cu doua coloane.

```
CREATE PROCEDURE add_linie (nume_tabel VARCHAR2, id NUMBER,
                            nume VARCHAR2)
IS
BEGIN
  EXECUTE IMMEDIATE 'INSERT INTO ' || nume_tabel ||
                    'VALUES (:1, :2)' USING id, nume;
END;
```

***Exemplu*:**

Valoarea *null* nu poate să apară în clauza *USING*. Comanda următoare este incorectă.

```
EXECUTE IMMEDIATE 'UPDATE opera SET valoare = :x'
  USING null;
```

Totuşi, dacă este necesară folosirea valorii *null*, pot fi utilizate variabile neiniţializate. O soluţie pentru corectarea erorii anterioare este dată de secvenţa:

```
DECLARE
  val_null  NUMBER;
BEGIN
  EXECUTE IMMEDIATE 'UPDATE opera SET valoare = :x'
    USING val_null;
END;
```

***Exemplu*:**

Să se utilizeze *SQL* dinamic pentru adaugarea unei coloane intr-un tabel.

```
CREATE PROCEDURE adaug_col (nume_tabel VARCHAR2,
                            col_spec   VARCHAR2)
IS
  instr  VARCHAR2(100) := 'ALTER TABLE ' || nume_tabel
                          ||' ADD ' || col_spec;
BEGIN
  EXECUTE IMMEDIATE instr;
END;
```

Adaugarea efectiva a unei noi coloane se face prin:

```
EXECUTE adaug_col ('carte', val_actual NUMBER(7,2)')
```

***Exemplu*:**

Să se obţină numărul operelor de artă din muzeu a căror valoare depăşeşte o limită precizată (cerere *single-row*).

```
CREATE OR REPLACE FUNCTION numar_opera (val_opera NUMBER)
  RETURN NUMBER AS
  sir_cerere  VARCHAR2(500);
  num_oper    NUMBER;
BEGIN
  sir_cerere :=
    'SELECT COUNT(*) FROM opera ' ||
      'WHERE valoare >= :alfa';
  EXECUTE IMMEDIATE sir_cerere
    INTO  num_oper
    USING val_opera;
  RETURN num_oper;
END;
```

***Exemplu*:**

Exemplul care urmează furnizează o modalitate de utilizare corectă a argumentelelor în clauza *USING*.

a) Pentru comenzi *SQL*, asocierea cu argumentele de legătură (*bind*) din clauza *USING* este prin poziţie.

```
sql_com := 'INSERT INTO alfa VALUES (:x, :x, :y, :x)';
EXECUTE IMMEDIATE sql_com USING a, a, b, a
```

b) Pentru blocuri *PL/SQL* executate dinamic, asocierea cu argumentele *bind* din clauza *USING* se face prin nume.

```
DECLARE
  x  NUMBER := 7;
  y  NUMBER := 23;
  v_bloc  VARCHAR2(200);
BEGIN
  v_bloc := 'BEGIN calcule(:a, :a, :b, :a); END;';
  EXECUTE IMMEDIATE v_bloc USING x, y;
END;
```

În exemplul care urmează va fi ilustrat modul de utilizare a comenzii *EXECUTE IMMEDIATE* pentru executarea comenzilor *LDD*, comenzilor *LMD* de reactualizare şi a blocurilor *PL/SQL* anonime. Şirul care trebuie executat poate fi un literal inclus între apostrofuri (de exemplu, *CREATE TABLE* sau *DROP TABLE*) sau poate fi un şir de caractere (de exemplu, blocuri anonime). Caracterul „;“ nu trebuie inclus decât pentru blocurile anonime.

***Exemplu*:**

```
DECLARE
  v_sql_sir     VARCHAR2(200);
  v_plsql_bloc  VARCHAR2(200);
BEGIN
  -- creare tabel
  EXECUTE IMMEDIATE
    'CREATE TABLE model_tabel (col1 VARCHAR2(30))';
  FOR contor IN 1..10 LOOP
    v_sql_sir :=
      'INSERT INTO model_tabel
         VALUES (''Linia '' || contor)';
    EXECUTE IMMEDIATE v_sql_sir;
  END LOOP;
  -- tipareste continut tabel utilizand un bloc anonim
  v_plsql_bloc :=
    'BEGIN
      FOR cont IN (SELECT * FROM model_tabel) LOOP
        DBMS_OUTPUT.PUT_LINE (cont.col1);
      END LOOP;
     END;';
  -- executie bloc anonim
  EXECUTE IMMEDIATE v_plsql_bloc;
  -- sterge tabel
  EXECUTE IMMEDIATE 'DROP TABLE model_tabel';
END;
```

Comanda *EXECUTE IMMEDIATE* poate fi utilizată şi pentru execuţia unor comenzi în care intervin variabile de lagatură. Exemplul următor ilustrează această situaţie, marcând şi modul de folosire a clauzei *USING*.

Exemplu:

```
DECLARE
  v_sql_sir     VARCHAR2(200);
  v_plsql_bloc  VARCHAR2(200);
BEGIN
  v_sql_sir :=
    'INSERT INTO opera (cod_opera, titlu, valoare)
      VALUES (:cod, :descriere, :val)';
  EXECUTE IMMEDIATE v_sql_sir USING 'c17', 'Modista', 15;
  v_plsql_bloc :=
    'BEGIN
      UPDATE artist SET nume = ''Gauguin''
      WHERE  cod_artist = :xx;
     END;';
  EXECUTE IMMEDIATE v_plsql_bloc USING 'a37';
END;
```

Pentru executarea cererilor multiple (care intorc mai multe linii) este necesară o abordare similară celei descrise în cazul cursoarelor dinamice, prin utilizarea:

- declaratiei tipului *REF CURSOR* si a unei variabile cursor bazata pe acest tip,
- comenzilor *OPEN - FOR*, *FETCH* şi *CLOSE*.

Exemplul care urmează prezinta maniera în care pot fi executate diferite cereri, utilizând *SQL* dinamic nativ.

Exemplu:

```
CREATE OR REPLACE PACKAGE nativ AS
  TYPE t_ref  IS REF CURSOR;
  FUNCTION opera_cerere (p_clauza IN VARCHAR2)
    RETURN t_ref;
  FUNCTION opera_alta_cerere (p_stil IN VARCHAR2)
    RETURN t_ref;
END nativ;

CREATE OR REPLACE PACKAGE BODY nativ AS
  FUNCTION opera_cerere (p_clauza IN VARCHAR2)
    RETURN t_ref IS
    v_retur_cursor  t_ref;
    v_sql_comanda   VARCHAR2(500);
  BEGIN
```

```
    v_sql_comanda := 'SELECT * FROM opera ' || p_clauza;
    OPEN v_retur_cursor FOR v_sql_comanda;
    RETURN v_retur_cursor;
  END opera_cerere;

  FUNCTION opera_alta_cerere (p_stil IN VARCHAR2)
    RETURN t_ref IS
    v_retur_cursor  t_ref;
    v_sql_comanda   VARCHAR2(500);
  BEGIN
    v_sql_comanda := 'SELECT * FROM opera WHERE  stil = :s';
    OPEN v_retur_cursor FOR v_sql_comanda USING p_stil;
    RETURN v_retur_cursor;
  END opera_alta_cerere;
END nativ;

DECLARE
  v_opera         opera%ROWTYPE;
  v_opera_cursor  nativ.t_ref;
BEGIN
  -- deschide cursor
  v_opera_cursor :=
    nativ.opera_cerere ('WHERE valoare < 1000000');
  -- parcurge cursor si tipareste rezultate
  DBMS_OUTPUT.PUT_LINE ('Urmatoarele opere au valoarea mai
                         mica de un milion de dolari');
  LOOP
    FETCH v_opera_cursor INTO v_opera;
    EXIT WHEN v_opera_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_opera.titlu || ' ' ||
                         v_opera.stil);
  END LOOP;
  CLOSE v_opera_cursor;

  -- se procedeaza similar pentru functia opera_alta_cerere
  -- deschide cursor
  v_opera_cursor :=
    nativ.opera_alta_cerere ('impresionism');
  -- parcurge cursor si tipareste rezultate
  DBMS_OUTPUT.PUT_LINE ('Urmatoarele opere de arta apartin
                         impresionismului');
  LOOP
    FETCH v_opera_cursor INTO v_opera;
    EXIT WHEN v_opera_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (v_opera.titlu  || ' ' ||
                          v_opera.valoare);
  END LOOP;
  CLOSE v_opera_cursor;
END;
```

Comanda *EXECUTE IMMEDIATE* poate fi utilizată pentru cereri care întorc o singură linie, cu sau fără variabile *bind*. Următorul exemplu prezintă, în doua cazuri, modul de implementare a acestei posibilităţi.

***Exemplu*:**

```
DECLARE
  v_sql_cerere    VARCHAR2(200);
  v_galerie       galerie%ROWTYPE;
  v_nume_galerie  galerie.nume_galerie%TYPE;
BEGIN
  -- selectare in variabila
  v_sql_cerere :=
    'SELECT nume_galerie ' ||
    'FROM   galerie ' ||
    'WHERE  cod_galerie = ''g73''';
  EXECUTE IMMEDIATE v_sql_cerere INTO v_nume_galerie;
  DBMS_OUTPUT.PUT_LINE (v_nume_galerie);
  -- selectare in record, utilizand variabile bind
  v_sql_cerere :=
    'SELECT * ' ||
    'FROM   galerie ' ||
    'WHERE nume_galerie = :nume_galerie';
  EXECUTE IMMEDIATE v_sql_cerere INTO v_galerie
    USING v_nume_galerie;
  DBMS_OUTPUT.PUT_LINE (v_galerie.cladire);
END;
```

Structura *Bulk bind* permite executarea cererilor care returnează mai multe linii, toate liniile returnate putând fi obţinute printr-o singură operaţie. Puterea acestei structuri poate fi combinată cu facilităţile oferite de *SQL* dinamic. Utilizând comenzile *FETCH*, *EXECUTE IMMEDIATE*, *FORALL*, clauzele *RETURNING INTO*, *COLLECT INTO* şi atributul cursor *%BULK_ROWCOUNT* se pot construi comenzi *SQL* care se execută dinamic utilizând tehnica *bulk bind*. Comenzile vor avea o structură adaptată pentru rezolvarea dinamică a comenzilor *SQL*.

În acest caz, comanda *FETCH* are forma sintactică:

FETCH *cursor_dinamic*
BULK COLLECT INTO *variabila* [, *variabila ...*];

De remarcat că dacă numărul variabilelor este mai mare decât numărul de coloane, *Oracle* va declanşa o eroare.

Comanda *FORALL* va avea următoarea structură modificată:

FORALL *index* ***IN*** *lim_inf .. lim_sup*
EXECUTE IMMEDIATE *şir_dinamic*
USING *argument_bind* | *argument_bind*(*index*)
[, *argument_bind* | *argument_bind*(*index*) ...]
[{***RETURNING*** | ***RETURN***} ***BULK COLLECT***
INTO *argument_bind* [, *argument_bind* ...]];

Atributul *şir_dinamic* acceptă comenzile *INSERT*, *UPDATE* şi *DELETE*, dar nu comanda *SELECT*.

Liniile de valori returnate de comanda *EXECUTE IMMEDIATE* pot fi depuse într-o colecţie. Comanda îşi modifică structura în următoarea manieră:

EXECUTE IMMEDIATE *şir_dinamic*
[[***BULK COLLECT***] ***INTO*** *variabila* [, *variabila* ...]
[***USING*** *argument_bind* [, *argument_bind* ...]]
[{***RETURNING*** | ***RETURN***}
BULK COLLECT INTO *argument_bind* [, *argument_bind* ...]];

***Exemplu*:**

Exemplul care urmează arată modul de utilizare a clauzei *BULK COLLECT* în comenzile *FETCH* şi *EXECUTE IMMEDIATE*.

```
DECLARE
  TYPE  opera_ref IS REF CURSOR;
  TYPE  tab_num IS TABLE OF NUMBER;
  TYPE  car_tab IS TABLE OF VARCHAR2(30);
  oper  opera_ref;
  num1  tab_num;
  car1  car_tab;
  num2  tab_num;
BEGIN
  OPEN oper FOR 'SELECT cod_opera, valoare FROM opera';
  FETCH oper BULK COLLECT INTO car1, num1;
  CLOSE oper;
  EXECUTE IMMEDIATE 'SELECT valoare FROM opera'
    BULK COLLECT INTO num2;
END;
```

Numai comenzile *INSERT*, *UPDATE* şi *DELETE* pot avea variabile *bind* ca argumente *OUT*.

***Exemplu*:**

```
DECLARE
  TYPE tab IS TABLE OF VARCHAR2(60);
  v_val    tab;
  bun      NUMBER := 100000;
  com_sql  VARCHAR2(200);
```

```
BEGIN
  com_sql := 'UPDATE opera SET valoare = :1
              RETURNING titlu INTO :2';
  EXECUTE IMMEDIATE com_sql
    USING bun RETURNING BULK COLLECT INTO v_val;
END;
```

Pentru a utiliza variabile *bind* ca intrări într-o comandă *SQL* (diferită de *SELECT*) se pot folosi comanda *FORALL* şi clauza *USING*.

***Exemplu*:**

```
DECLARE
  TYPE  num  IS TABLE OF NUMBER;
  TYPE  car  IS TABLE OF VARCHAR2(30);
  num1  num;
  car1  car;
BEGIN
  num1 := num(1, 2, 3, 4, 5);
  FORALL i IN 1..5
    EXECUTE IMMEDIATE
       'UPDATE opera SET val = val*1.1
        WHERE  cod_opera = :1
        RETURNING titlu INTO :2'
      USING num1(i) RETURNING BULK COLLECT INTO car1;
  …
END;
```

SQL dinamic poate fi utilizat pentru a compila cod *PL/SQL*.

***Exemplu*:**

```
CREATE PROCEDURE compilare_plsql (name  VARCHAR2,
                                  tip_plsql  VARCHAR2,
                                  optiune  VARCHAR2 := NULL)
IS
  instr  VARCHAR2(200) := 'ALTER '|| tip_plsql ||
                     ' ' || name || ' COMPILE';
BEGIN
  IF optiune IS NOT NULL THEN
    instr := instr || ' ' || optiune;
  END IF;
  EXECUTE IMMEDIATE instr;
END;
```

Exemple de apelare:

```
EXECUTE compilare_plsql ('listare', 'procedure')
EXECUTE compilare_plsql ('pachet', 'package', body')
```

Efectul este:

```
ALTER PROCEDURE listare COMPILE
ALTER PACKAGE pachet COMPILE BODY
```

SQL dinamic poate fi apelat şi din cadrul altor limbaje.

- Limbajele *C/C++* apelează *SQL* dinamic prin *OCI* (*Oracle Call Interface*) sau poate fi utilizat un precompilator *Pro*C/C++* pentru adăugarea comenzilor dinamice *SQL* la codul *C*.
- Limbajul *Cobol* poate utiliza comenzi dinamice *SQL* folosind un precompilator *Pro*Cobol*.
- Limbajul *Java* prin intermediul lui *JDBC* (interfaţa pentru conectarea limbajului la baze de date relaţionale) poate utiliza comenzi dinamice *SQL*.

***Exemplu*:**

Să se şteargă toţi indecşii din schema personală.

```
BEGIN
  FOR j IN (SELECT INDEX_NAME
            FROM   USER_INDEXES) LOOP
    EXECUTE IMMEDIATE
      'DROP INDEX' || j.INDEX_NAME ||' ';
  END LOOP;
END;
```

Presupunand ca in schema sunt 500 indecsi, executia blocului va genera eroarea *ORA - 01555*, deoarece nu exista suficient spatiu de lucru.

De ce? Raspunsul este simplu: *COMMIT*-ul generat de *LDD* (stergerea unui index apartine de *LDD*) si faptul ca *LDD* modifica vizualizarile *DD* prin stergerea indecsilor.

Cum trebuie procedat in acest caz? Trebuie ca sa nu fie deschis cursorul in timpul permanentizarii.

```
DECLARE
  i_index  VARCHAR2(30);
BEGIN
  LOOP
    SELECT INDEX_NAME INTO i_index
    FROM USER_INDEXES
    WHERE ROWNUM = 1;
    EXECUTE IMMEDIATE 'DROP INDEX' || i_index;
  END LOOP;
EXCEPTION
WHEN NO_DATA _FOUND THEN NULL;
END;
```

Declanşatori în *PL/SQL*

Un declanşator (*trigger*) este un bloc *PL/SQL* sau apelul (*CALL*) unei proceduri *PL/SQL*, care se execută automat ori de câte ori are loc un anumit eveniment „declanşator“. Evenimentul poate consta din:

- modificarea unui tabel sau a unei vizualizări,
- acţiuni sistem
- anumite acţiuni utilizator.

Blocul *PL/SQL* poate fi asociat unui tabel, unei vizualizări, unei scheme sau unei baze de date.

La fel ca şi pachetele, declanşatorii nu pot fi locali unui bloc sau unui pachet, ei trebuie depuşi ca obiecte independente în baza de date.

Folosirea declanşatorilor garantează faptul că atunci când o anumită operaţie este efectuată, automat sunt executate nişte acţiuni asociate. Evident, nu trebuie introduşi declanşatori care ar putea să substituie funcţionalităţi oferite deja de sistem. De exemplu, nu are sens să fie definiţi declanşatori care să implementeze regulile de integritate ce pot fi definite, mai simplu, prin constrângeri declarative.

Tipuri de declanşatori

Declanşatorii pot fi:

- la nivel de bază de date (*database triggers*);
- la nivel de aplicaţie (*application triggers*).

Declanşatorii bază de date se execută automat ori de câte ori are loc:

- o acţiune (comandă *LMD*) asupra datelor unui tabel;
- o acţiune (comandă *LMD*) asupra datelor unei vizualizări;
- o comandă *LDD* (*CREATE*, *ALTER*, *DROP*) referitoare la anumite obiecte ale schemei sau ale bazei;
- un eveniment sistem (*SHUTDOWN*, *STARTUP*);
- o acţiune a utilizatorului (*LOGON*, *LOGOFF*);
- o eroare (*SERVERERROR*, *SUSPEND*).

Declanşatorii bază de date sunt de trei tipuri:

- declanşatori *LMD* – activaţi de comenzi *LMD* (*INSERT*, *UPDATE* sau *DELETE*) executate asupra unui tabel al bazei de date;

- declanşatori *INSTEAD OF* – activaţi de comenzi *LMD* executate asupra unei vizualizări (relaţionale sau obiect);
- declanşatori sistem – activaţi de un eveniment sistem (oprirea sau pornirea bazei), de comenzi *LDD* (*CREATE*, *ALTER*, *DROP*), de conectarea (deconectarea) unui utilizator. Ei sunt definiţi la nivel de schemă sau la nivel de bază de date.

Declanşatorii asociaţi unui tabel (stocaţi în baza de date) vor acţiona indiferent de aplicaţia care a efectuat operaţia *LMD*. Dacă operaţia *LMD* se referă la o vizualizare, declanşatorul *INSTEAD OF* defineşte acţiunile care vor avea loc, iar dacă aceste acţiuni includ comenzi *LMD* referitoare la tabele, atunci declanşatorii asociaţi acestor tabele sunt şi ei, la rândul lor, activaţi.

Dacă declanşatorii sunt asociaţi unei baze de date, ei se declanşează pentru fiecare eveniment, pentru toţi utilizatorii. Dacă declanşatorii sunt asociaţi unei scheme sau unui tabel, ei se declanşează numai dacă evenimentul declanşator implică acea schemă sau acel tabel. Un declanşator se poate referi la un singur tabel sau la o singură vizualizare.

Declanşatorii aplicaţie se execută implicit ori de câte ori apare un eveniment particular într-o aplicaţie (de exemplu, o aplicaţie dezvoltată cu *Developer Suite*). *Form Builder* utilizează frecvent acest tip de declanşatori (*form builder triggers*). Ei pot fi declanşaţi prin apăsarea unui buton, prin navigarea pe un câmp etc. În acest capitol se va face referinţă doar la declanşatorii bază de date.

Atunci când un pachet sau un subprogram este depus în dicţionarul datelor, alături de codul sursă este depus şi *p-codul* compilat. În mod similar se întâmplă şi pentru declanşatori. Prin urmare, un declanşator poate fi apelat fără recompilare. Declanşatorii pot fi invalidaţi în aceeeaşi manieră ca pachetele şi subprogramele. Dacă declanşatorul este invalidat, el va fi recompilat la următoarea activare.

Crearea declanşatorilor *LMD*

Declanşatorii *LMD* sunt creaţi folosind comanda *CREATE TRIGGER*.

Numele declanşatorului trebuie să fie unic printre numele declanşatorilor din cadrul aceleaşi scheme, dar poate să coincidă cu numele altor obiecte ale acesteia (de exemplu, tabele, vizualizări sau proceduri).

La crearea unui declanşator este obligatorie una dintre opţiunile *BEFORE* sau *AFTER*, prin care se precizează momentul în care este executat corpul declanşatorului. Acesta nu poate depăşi 32*KB*.

CREATE [***OR REPLACE***] ***TRIGGER*** [*schema.*]*nume_declanşator*
{***BEFORE*** | ***AFTER***}
{***DELETE*** | ***INSERT*** | ***UPDATE*** [***OF*** *coloana*[*, coloana ...*]] }
[***OR*** {***DELETE*** | ***INSERT*** | ***UPDATE*** [***OF*** *coloana*[*, coloana ...*]] ...}
ON [*schema.*]*nume_tabel*
[***REFERENCING*** {***OLD*** [***AS***] *vechi* ***NEW*** [***AS***] *nou*
| ***NEW*** [***AS***] *nou* ***OLD*** [***AS***] *vechi* }]
[***FOR EACH ROW***]
[***WHEN*** (*condiţie*)]
corp_declanşator (*bloc PL/SQL sau apelul unei proceduri*);

Până la versiunea *Oracle8i*, corpul unui declanşator trebuia să fie un bloc *PL/SQL*. În ultimele versiuni, corpul poate consta doar dintr-o singură comandă *CALL*. Procedura apelată poate fi un subprogram *PL/SQL* stocat, o rutină *C* sau o metodă *Java*. În acest caz, *CALL* nu poate conţine clauza *INTO* care este specifică funcţiilor, iar pentru a referi coloanele tabelului asociat declanşatorului, acestea trebuie prefixate de atributele *:NEW* sau *:OLD*. De asemenea, în expresia parametrilor nu pot să apară variabile *bind*.

Declararea unui declanşator trebuie să cuprindă tipul comenzii *SQL* care duce la executarea declanşatorului şi tabelul asociat acestuia. În ceea ce priveşte tipul comenzii *SQL* care va duce la executarea declaşatorului, sunt incluse următoarele tipuri de opţiuni: *DELETE*, *INSERT*, *UPDATE* sau o combinare a acestora cu operatorul logic *OR*. Cel puţin una dintre opţiuni este obligatorie.

În declararea declanşatorului este specificat tabelul asupra căruia va fi executat declanşatorul. *Oracle9i* admite tablouri imbricate. Dacă declanşatorul este de tip *UPDATE*, atunci pot fi enumerate coloanele pentru care acesta se va executa.

În corpul fiecărui declanşator pot fi cunoscute valorile coloanelor atât înainte de modificarea unei linii, cât şi după modificarea acesteia. Valoarea unei coloane înainte de modificare este referită prin atributul *OLD*, iar după modificare, prin atributul *NEW*. Prin intermediul clauzei opţionale *REFERENCING* din sintaxa comenzii de creare a declanşatorilor, atributele *NEW* şi *O*LD pot fi redenumite. In interiorul blocului PL/SQL, coloanele prefixate prin OLD sau *NEW* sunt considerate variabile externe, deci trebuie prefixate cu ":".

Un declanşator poate activa alt declanşator, iar acesta la rândul său poate activa alt declanşator etc. Această situaţie (declanşatori în cascadă) poate avea însă efecte imprevizibile. Sistemul *Oracle* permite maximum 32 declanşatori în cascadă. Numărul acestora poate fi limitat (utilizând parametrul de iniţializare *OPEN_CURSORS*), deoarece pentru fiecare execuţie a unui declanşator trebuie deschis un nou cursor.

Declanşatorii la nivel de baze de date pot fi de două feluri:

- la nivel de instrucţiune (*statement level trigger*);
- la nivel de linie (*row level trigger*).

Declanşatori la nivel de instrucţiune

Declanşatorii la nivel instrucţiune sunt executaţi o singură dată pentru instrucţiunea declanşatoare, indiferent de numărul de linii afectate (chiar dacă nici o linie nu este afectată). Un declanşator la nivel de instrucţiune este util dacă acţiunea declanşatorului nu depinde de informaţiile din liniile afectate.

***Exemplu*:**

Programul de lucru la administraţia muzeului este de luni până vineri, în intervalul (8:00 a.m. - 10:00 p.m.). Să se construiască un declanşator la nivel de instrucţiune care împiedică orice activitate asupra unui tabel al bazei de date, în afara acestui program.

```
CREATE OR REPLACE PROCEDURE verifica IS
BEGIN
  IF ((TO_CHAR(SYSDATE,'D') BETWEEN 2 AND 6)
    AND
      TO_DATE(TO_CHAR(SYSDATE,'hh24:mi'), 'hh24:mi')
        NOT BETWEEN TO_DATE('08:00','hh24:mi')
                AND TO_DATE('22:00','hh24:mi'))
    THEN
      RAISE_APPLICATION_ERROR (-27733, 'nu puteti reactualiza
         acest tabel deoarece sunteti in afara programului');
  END IF;
END verifica;
/
CREATE OR REPLACE TRIGGER BIUD_tabel1
  BEFORE INSERT OR UPDATE OR DELETE ON tabel1
BEGIN
  verifica;
END;
/
```

Declanşatori la nivel de linie

Declanşatorii la nivel de linie sunt creaţi cu opţiunea *FOR EACH ROW*. În acest caz, declanşatorul este executat pentru fiecare linie din tabelul afectat, iar dacă evenimentul declanşator nu afectează nici o linie, atunci declanşatorul nu este executat. Dacă opţiunea *FOR EACH ROW* nu este inclusă, declanşatorul este considerat implicit la nivel de instrucţiune.

Declanşatorii la nivel linie nu sunt performanţi dacă se fac frecvent reactualizări pe tabele foarte mari.

Restricţiile declanşatorilor pot fi incluse prin specificarea unei expresii booleene în clauza *WHEN*. Acestă expresie este evaluată pentru fiecare linie afectată de către declanşator. Declanşatorul este executat pentru o linie, doar dacă expresia este adevărată pentru acea linie. Clauza *WHEN* este validă doar pentru declanşatori la nivel de linie.

***Exemplu*:**

Să se implementeze cu ajutorul unui declanşator constrângerea că valorile operelor de artă nu pot fi reduse (trei variante).

Varianta 1:

```
CREATE OR REPLACE TRIGGER verifica_valoare
  BEFORE UPDATE OF valoare ON opera
  FOR EACH ROW
  WHEN (NEW.valoare < OLD.valoare)
BEGIN
  RAISE_APPLICATION_ERROR (-20222, 'valoarea unei opere de
                           arta nu poate fi micsorata');
END;
```

Varianta 2:

```
CREATE OR REPLACE TRIGGER verifica_valoare
  BEFORE UPDATE OF valoare ON opera
  FOR EACH ROW
BEGIN
  IF (:NEW.valoare < :OLD.valoare) THEN
     RAISE_APPLICATION_ERROR (-20222, 'valoarea unei opere de
                              arta nu poate fi micsorata');
  END IF;
END;
```

Varianta 3:

```
CREATE OR REPLACE TRIGGER verifica_valoare
  BEFORE UPDATE OF valoare ON opera
  FOR EACH ROW
  WHEN (NEW.valoare < OLD.valoare)
  CALL procedura -- care va face actiunea RAISE …
/
```

Accesul la vechile şi noile valori ale coloanelor liniei curente, afectată de evenimentul declanşator, se face prin: *OLD.nume_coloană* (vechea valoare), respectiv prin *NEW.nume_coloană* (noua valoare). În cazul celor trei comenzi *LMD*, aceste valori devin:

INSERT	: *NEW*.nume_coloană → noua valoare (: *OLD*.nume_coloană → *NULL*);
UPDATE	: *NEW*.nume_coloană → noua valoare : *OLD*.nume_coloană → vechea valoare;
DELETE	(: *NEW*.nume_coloană → *NULL*) : *OLD*.nume_coloană → vechea valoare.

***Exemplu*:**

Se presupune că pentru fiecare galerie există două câmpuri (*min_valoare* şi *max_valoare*) în care se reţin limitele minime şi maxime ale valorile operelor din galeria respectivă. Să se implementeze cu ajutorul unui declanşator constrângerea că, dacă aceste limite s-ar modifica, valoarea oricarei opere de artă trebuie să ramană cuprinsă între noile limite.

```
CREATE OR REPLACE TRIGGER verifica_limite
  BEFORE UPDATE OF min_valoare, max_valoare ON galerie
  FOR EACH ROW
DECLARE
  v_min_val  opera.valoare%TYPE;
  v_max_val  opera.valoare%TYPE;
  e_invalid  EXCEPTION;
BEGIN
  SELECT MIN(valoare), MAX(valoare)
  INTO   v_min_val, v_max_val
  FROM   opera
  WHERE  cod_galerie = :NEW.cod_galerie;
  IF (v_min_val < :NEW.min_valoare) OR
     (v_max_val > :NEW.max_valoare) THEN
     RAISE e_invalid;
  END IF;
EXCEPTION
  WHEN e_invalid THEN
    RAISE_APPLICATION_ERROR (-20567, 'Exista opere de
    arta ale caror valori sunt in afara domeniului
    permis');
END verifica_limite;
/
```

Ordinea de execuţie a declanşatorilor

PL/SQL permite definirea a 12 tipuri de declanşatori care sunt obţinuţi prin combinarea proprietăţii de moment (timp) al declanşării (*BEFORE*, *AFTER*), cu proprietatea nivelului la care acţionează (nivel linie, nivel intrucţiune) şi cu tipul operaţiei ataşate declanşatorului (*INSERT*, *UPDATE*, *DELETE*).

De exemplu, *BEFORE INSERT* acţionează o singură dată, înaintea executării unei instrucţiuni *INSERT*, iar *BEFORE INSERT FOR EACH ROW* acţionează înainte de inserarea fiecărei noi înregistrări.

Declanşatorii sunt activaţi când este executată o comandă *LMD*. La apariţia unei astfel de comenzi se execută câteva acţiuni care vor fi descrise în continuare.

1. Se execută declanşatorii la nivel de instrucţiune *BEFORE*.
2. Pentru fiecare linie afectată de comanda *LMD*:
   2.1. se execută declanşatorii la nivel de linie *BEFORE*;
   2.2. se blochează şi se modifică linia afectată (se execută comanda *LMD*), se verifică constrângerile de integritate (blocarea rămâne valabilă până în momentul în care tranzacţia este permanentizată);
   2.3. se execută declanşatorii la nivel de linie *AFTER*.
3. Se execută declanşatorii la nivel de instrucţiune *AFTER*.

Începând cu versiunea *Oracle8i* algoritmul anterior se schimbă, în sensul că verificarea constrângerii referenţiale este amânată după executarea declanşatorului la nivel linie.

***Obsevaţii*:**

- În expresia clauzei *WHEN* nu pot fi incluse funcţii definite de utilizator sau subcereri *SQL*.
- În clauza *ON* poate fi specificat un singur tabel sau o singură vizualizare.
- În interiorul blocului *PL/SQL*, coloanele tabelului prefixate cu *OLD* sau *NEW* sunt considerate variabile externe şi deci, trebuie precedate de caracterul „:“.
- Condiţia de la clauza *WHEN* poate conţine coloane prefixate cu *OLD* sau *NEW*, dar în acest caz, acestea nu trebuie precedate de „:“.
- Declanşatorii bază de date pot fi definiţi numai pe tabele (excepţie, declanşatorul *INSTEAD OF* care este definit pe o vizualizare). Totuşi, dacă o comandă *LMD* este aplicată unei vizualizări, pot fi activaţi declanşatorii asociaţi tabelelor care definesc vizualizarea.
- Corpul unui declanşator nu poate conţine o interogare sau o reactualizare a unui tabel aflat în plin proces de modificare, pe timpul acţiunii declanşatorului (*mutating table*).
- Blocul *PL/SQL* care descrie acţiunea declanşatorului nu poate conţine comenzi pentru gestiunea tranzacţiilor (*COMMIT*, *ROLLBACK*, *SAVEPOINT*). Controlul tranzacţiilor este permis, însă, în procedurile stocate. Dacă un declanşator apelează o procedură stocată care execută o comandă referitoare la controlul tranzacţiilor, atunci va apărea o eroare la execuţie şi tranzacţia va fi anulată.
- Comenzile *LDD* nu pot să apară decât în declanşatorii sistem.
- Corpul declanşatorului poate să conţină comenzi *LMD*.
- În corpul declanşatorului pot fi referite şi utilizate coloane *LOB*, dar nu pot fi modificate valorile acestora.
- Nu este indicată crearea declanşatorilor recursivi.

- În corpul declanşatorului se pot insera date în coloanele de tip *LONG* şi *LONGRAW*, dar nu pot fi declarate variabile de acest tip.
- Dacă un tabel este suprimat (se şterge din dicţionarul datelor), automat sunt distruşi toţi declanşatorii asociaţi tabelului.
- Este necesară limitarea dimensiunii unui declanşator. Dacă acesta solicită mai mult de 60 linii de cod, atunci este preferabil ca o parte din cod să fie inclusă într-o procedură stocată şi aceasta să fie apelată din corpul declanşatorului.

Sunt două diferenţe esenţiale între declanşatori şi procedurile stocate:

- declanşatorii se invocă implicit, iar procedurile explicit;
- instrucţiunile *LCD* (*COMMIT*, *ROLLBACK*, *SAVEPOINT*) nu sunt permise în corpul unui declanşator.

Predicate condiţionale

În interiorul unui declanşator care poate fi executat pentru diferite tipuri de instrucţiuni *LMD* se pot folosi trei funcţii booleene prin care se stabileşte tipul operaţiei executate. Aceste predicate condiţionale (furnizate de pachetul standard *DBMS_STANDARD*) sunt *INSERTING*, *UPDATING* şi *DELETING*.

Funcţiile booleene nu solicită prefixarea cu numele pachetului şi determină tipul operaţiei (*INSERT*, *DELETE*, *UPDATE*). De exemplu, predicatul *INSERTING* ia valoarea *TRUE* dacă instrucţiunea declanşatoare este *INSERT*. Similar sunt definite predicatele *UPDATING* şi *DELETING*. Utilizând aceste predicate, în corpul declanşatorului se pot executa secvenţe de instrucţiuni diferite, în funcţie de tipul operaţiei *LMD*.

În cazul în care corpul declanşatorului este un bloc *PL/SQL* complet (nu o comandă *CALL*), pot fi utilizate atât predicatele *INSERTING*, *UPDATING*, *DELETING*, cât şi identificatorii *:OLD*, *:NEW*, *:PARENT*.

***Exemplu*:**

Se presupune că în tabelul *galerie* se păstrează (într-o coloană numită *total_val*) valoarea totală a operelor de artă expuse în galeria respectivă.

```
UPDATE galerie
SET    total_val =
           (SELECT SUM(valoare)
            FROM   opera
            WHERE opera.cod_galerie = galerie.cod_galerie);
```

Reactualizarea acestui câmp poate fi implementată cu ajutorul unui declanşator în următoarea manieră:

```
CREATE OR REPLACE PROCEDURE creste
          (v_cod_galerie  IN galerie.cod_galerie%TYPE,
           v_val         IN galerie.total_val%TYPE) AS
BEGIN
  UPDATE  galerie
  SET     total_val = NVL (total_val, 0) + v_val
  WHERE   cod_galerie = v_cod_galerie;
END creste;
/
CREATE OR REPLACE TRIGGER calcul_val
  AFTER INSERT OR DELETE OR UPDATE OF valoare ON opera
  FOR EACH ROW
BEGIN
  IF DELETING THEN
    creste (:OLD.cod_galerie, -1*:OLD.valoare);
  ELSIF UPDATING THEN
    creste (:NEW.cod_galerie, :NEW.valoare - :OLD.valoare);
  ELSE /* inserting */
    creste (:NEW.cod_galerie, :NEW.valoare);
  END IF;
END;
/
```

Declanşatori *INSTEAD OF*

PL/SQL permite definirea unui nou tip de declanşator, numit *INSTEAD OF*, care oferă o modalitate de actualizare a vizualizărilor obiect şi a celor relaţionale.

Sintaxa acestui tip de declanşator este similară celei pentru declanşatori *LMD*, cu două excepţii:

- clauza {*BEFORE* | *AFTER*} este înlocuită prin *INSTEAD OF*;
- clauza *ON* [*schema.*]*nume_tabel* este înlocuită printr-una din clauzele *ON* [*schema.*]*nume_view* sau *ON NESTED TABLE* (*nume_coloană*) *OF* [*schema.*]*nume_view.*

Declanşatorul *INSTEAD OF* permite reactualizarea unei vizualizări prin comenzi *LMD*. O astfel de modificare nu poate fi realizată în altă manieră, din cauza regulilor stricte existente pentru reactualizarea vizualizărilor. Declanşatorii de tip *INSTEAD OF* sunt necesari, deoarece vizualizarea pe care este definit declanşatorul poate, de exemplu, să se refere la *join*-ul unor tabele, şi în acest caz, nu sunt actualizabile toate legăturile.

O vizualizare nu poate fi modificată prin comenzi *LMD* dacă vizualizarea conţine operatori pe mulţimi, funcţii grup, clauzele *GROUP BY*, *CONNECT BY*, *START WITH*, operatorul *DISTINCT* sau *join*-uri.

Declanşatorul *INSTEAD OF* este utilizat pentru a executa operaţii *LMD* direct pe tabelele de bază ale vizualizării. De fapt, se scriu comenzi *LMD* relative la o vizualizare, iar declanşatorul, în locul operaţiei originale, va opera pe tabelele de bază.

De asemenea, acest tip de declanşator poate fi definit asupra vizualizărilor ce au drept câmpuri tablouri imbricate, declanşatorul furnizând o modalitate de reactualizare a elementelor tabloului imbricat.

În acest caz, el se declanşează doar în cazul în care comenzile *LMD* operează asupra tabloului imbricat (numai când elementele tabloului imbricat sunt modificate folosind clauzele *THE*() sau *TABLE*()) şi nu atunci când comanda *LMD* operează doar asupra vizualizării. Declanşatorul permite accesarea liniei „părinte“ ce conţine tabloul imbricat modificat.

Observaţii:

- Spre deosebire de declanşatorii *BEFORE* sau *AFTER*, declanşatorii *INSTEAD OF* se execută în locul instrucţiunii *LMD* (*INSERT*, *UPDATE*, *DELETE*) specificate.
- Opţiunea *UPDATE OF* nu este permisă pentru acest tip de declanşator.
- Declanşatorii *INSTEAD OF* se definesc pentru o vizualizare, nu pentru un tabel.
- Declanşatorii *INSTEAD OF* acţionează implicit la nivel de linie.
- Dacă declanşatorul este definit pentru tablouri imbricate, atributele *:OLD* şi *:NEW* se referă la liniile tabloului imbricat, iar pentru a referi linia curentă din tabloul „părinte“ s-a introdus atributul *:PARENT*.

***Exemplu*:**

Se consideră *nou_opera*, respectiv *nou_artist*, copii ale tabelelor *opera*, respectiv *artist* şi *vi_op_ar* o vizualizare definită prin compunerea naturală a celor două tabele. Se presupune că pentru fiecare artist există un câmp (*sum_val*) ce reprezintă valoarea totală a operelor de artă expuse de acesta în muzeu.

Să se definească un declanşator prin care reactualizările executate asupra vizualizării *vi_op_ar* se vor transmite automat tabelelor *nou_opera* şi *nou_artist*.

```
CREATE TABLE nou_opera AS
  SELECT cod_opera, cod_artist, valoare, tip, stil
  FROM   opera;

CREATE TABLE nou_artist AS
  SELECT cod_artist, nume, sum_val
  FROM   artist;

CREATE VIEW vi_op_ar AS
  SELECT cod_opera,o.cod_artist,valoare,tip,nume,
         sum_val
```

```
  FROM   opera o, artist a
  WHERE  o.cod_artist = a.cod_artist
CREATE OR REPLACE TRIGGER react
INSTEAD OF INSERT OR DELETE OR UPDATE ON vi_op_ar
FOR EACH ROW
BEGIN
  IF INSERTING THEN
    INSERT INTO nou_opera
    VALUES (:NEW.cod_opera, :NEW.cod_artist, :NEW.valoare,
            :NEW.tip);
    UPDATE nou_artist
    SET    sum_val = sum_val + :NEW.valoare
    WHERE  cod_artist = :NEW.cod_artist;
  ELSIF DELETING THEN
    DELETE FROM nou_opera
    WHERE  cod_opera = :OLD.cod_opera;
    UPDATE nou_artist
    SET    sum_val = sum_val - :OLD.valoare
    WHERE  cod_artist = :OLD.cod_artist;
  ELSIF UPDATING ('valoare') THEN
    UPDATE nou_opera
    SET    valoare = :NEW.valoare
    WHERE  cod_opera = :OLD.cod_opera;
    UPDATE nou_artist
    SET   sum_val = sum_val + (:NEW.valoare - :OLD.valoare)
    WHERE  cod_artist = :OLD.cod_artist;
  ELSIF UPDATING ('cod_artist') THEN
    UPDATE nou_opera
    SET    cod_artist = :NEW.cod_artist
    WHERE  cod_opera = :OLD.cod_opera;
    UPDATE nou_artist
    SET    sum_val = sum_val - :OLD.valoare
    WHERE  cod_artist = :OLD.cod_artist;
    UPDATE nou_artist
    SET    sum_val = sum_val + :NEW.valoare
    WHERE  cod_artist = :NEW.cod_artist;
  END IF;
END;
/
```

Declanşatori sistem

Declanşatorii sistem sunt activaţi de comenzi *LDD* (*CREATE*, *DROP*, *ALTER*) şi de anumite evenimente sistem (*STARTUP*, *SHUTDOWN*, *LOGON*, *LOGOFF*, *SERVERERROR*, *SUSPEND*). Un declanşator sistem poate fi definit la nivelul bazei de date sau la nivelul schemei.

Sintaxa pentru crearea unui astfel de declanşator este următoarea:

CREATE [***OR REPLACE***] ***TRIGGER*** [*schema.*]*nume_declanşator*
{***BEFORE*** | ***AFTER***}
{*lista_evenimente_LDD* | *lista_evenimente_bază*}
ON {***DATABASE*** | ***SCHEMA***}
[***WHEN*** (*condiţie*)]
corp_declanşator;

Cuvintele cheie *DATABASE* sau *SCHEMA* specifică nivelul declanşatorului.

Există restricţii asupra expresiilor din condiţia clauzei *WHEN*. De exemplu, declanşatorii *LOGON* şi *LOGOFF* pot verifica doar identificatorul (*userid*) şi numele utilizatorului (*username*), iar declanşatorii *LDD* pot verifica tipul şi numele obiectelor definite, identificatorul şi numele utilizatorului.

Evenimentele amintite anterior pot fi asociate clauzelor *BEFORE* sau *AFTER*. De exemplu, un declanşator *LOGON* (*AFTER*) se activează după ce un utilizator s-a conectat la baza de date, un declanşator *CREATE* (*BEFORE* sau *AFTER*) se activează înainte sau după ce a fost creat un obiect al bazei, un declanşator *SERVERERROR* (*AFTER*) se activează ori de câte ori apare o eroare (cu excepţia erorilor: *ORA-01403*, *ORA-01422*, *ORA-01423*, *ORA-01034*, *ORA-04030*).

Declanşatorii *LDD* se activează numai dacă obiectul creat este de tip *table*, *cluster*, *function*, *procedure*, *index*, *package*, *role*, *sequence*, *synonym*, *tablespace*, *trigger*, *type*, *view* sau *user*.

Pentru declanşatorii sistem se pot utiliza funcţii speciale care permit obţinerea de informaţii referitoare la evenimentul declanşator. Ele sunt funcţii *PL/SQL* stocate care trebuie prefixate de numele proprietarului (*SYS*).

Printre cele mai importante funcţii care furnizează informaţii referitoare la evenimentul declanşator, se remarcă:

- *SYSEVENT* – returnează evenimentul sistem care a activat declanşatorul (este de tip *VARCHAR2*(20) şi este aplicabilă oricărui eveniment);
- *DATABASE_NAME* – returnează numele bazei de date curente (este de tip *VARCHAR2*(50) şi este aplicabilă oricărui eveniment);
- *SERVER_ERROR* – returnează codul erorii a cărei poziţie în stiva erorilor este dată de argumentul de tip *NUMBER* al funcţiei (este de tip *NUMBER* şi este aplicabilă evenimentului *SERVERERROR*);
- *LOGIN_USER* – returnează identificatorul utilizatorului care activează declanşatorul (este de tip *VARCHAR2*(30) şi este aplicabilă oricărui eveniment);

- *DICTIONARY_OBJ_NAME* – returnează numele obiectului la care face referinţă comanda *LDD* ce a activat declanşatorul (este de tip *VARCHAR2*(30) şi este aplicabilă evenimentelor *CREATE*, *ALTER*, *DROP*).

***Exemplu*:**

```
CREATE OR REPLACE TRIGGER logutiliz
  AFTER CREATE ON SCHEMA
BEGIN
  INSERT INTO ldd_tab(user_id, object_name, creation_date)
  VALUES      (USER, SYS.DICTIONARY_OBJ_NAME, SYSDATE);
END logutiliz;
```

Evenimentul *SERVERERROR* poate fi utilizat pentru a urmări erorile care apar în baza de date. Codul erorii este furnizat, prin intermediul declanşatorului, de funcţia *SERVER_ERROR*, iar mesajul asociat erorii poate fi obţinut cu procedura *DBMS_UTILITY.FORMAT_ERROR_STACK*.

***Exemplu*:**

```
CREATE TABLE erori (
  moment       DATE,
  utilizator   VARCHAR2(30),
  nume_baza    VARCHAR2(50),
  stiva_erori  VARCHAR2(2000) );
/
CREATE OR REPLACE TRIGGER logerori
  AFTER SERVERERROR ON DATABASE
BEGIN
  INSERT INTO erori
  VALUES (SYSDATE, SYS.LOGIN_USER, SYS.DATABASE_NAME,
          DBMS_UTILITY.FORMAT_ERROR_STACK);
END logerori;
/
```

Modificarea şi suprimarea declanşatorilor

Opţiunea *OR REPLACE* din cadrul comenzii *CREATE TRIGGER* recreează declanşatorul, dacă acesta există. Clauza permite schimbarea definiţiei unui declanşator existent fără suprimarea acestuia.

Similar procedurilor şi pachetelor, un declanşator poate fi suprimat prin:

DROP TRIGGER [*schema.*]*nume_declanşator*;

Uneori acţiunea de suprimare a unui declanşator este prea drastică şi este preferabilă doar dezactivarea sa temporară. În acest caz, declanşatorul va continua să existe în dicţionarul datelor.

Modificarea unui declanşator poate consta din recompilarea (*COMPILE*), redenumirea (*RENAME*), activarea (*ENABLE*) sau dezactivarea (*DISABLE*) acestuia şi se realizează prin comanda:

ALTER TRIGGER [*schema.*]*nume declanşator*
{***ENABLE*** | ***DISABLE*** | ***COMPILE*** | ***RENAME TO*** *nume_nou*}
{***ALL TRIGGERS***}

Dacă un declanşator este activat, atunci sistemul *Oracle* îl execută ori de câte ori au loc operaţiile precizate în declanşator asupra tabelului asociat şi când condiţia de restricţie este îndeplinită. Dacă declanşatorul este dezactivat, atunci sistemul *Oracle* nu îl va mai executa. După cum s-a mai subliniat, dezactivarea unui declanşator nu implică ştergerea acestuia din dicţionarul datelor.

Toţi declanşatorii asociaţi unui tabel pot fi activaţi sau dezactivaţi utilizând opţiunea *ALL TRIGGERS* (*ENABLE ALL TRIGGERS*, respectiv *DISABLE ALL TRIGGERS*). Declanşatorii sunt activaţi în mod implicit atunci când sunt creaţi.

Pentru activarea (*enable*) unui declansator, *server*-ul *Oracle*:

- verifica integritatea constrangerilor,
- garanteaza ca declansatorii nu pot compromite constrangerile de integritate,
- garanteaza consistenta la citire a vizualizarilor,
- gestioneaza dependentele.

Activarea şi dezactivarea declanşatorilor asociaţi unui tabel se poate realiza şi cu ajutorul comenzii *ALTER TABLE*.

Un declanşator este compilat în mod automat la creare. Dacă un *site* este neutilizabil atunci când declanşatorul trebuie compilat, sistemul *Oracle* nu poate valida comanda de accesare a bazei distante şi compilarea eşuează.

Informaţii despre declanşatori

În DD există vizualizări ce conţin informaţii despre declanşatori şi despre starea acestora (*USER_TRIGGERS, USER_TRIGGER_COL, ALL_TRIGGERS, DBA_TRIGGERS* etc.). Aceste vizualizări sunt actualizate ori de câte ori un declanşator este creat sau suprimat.

Atunci când declanşatorul este creat, codul său sursă este stocat în vizualizarea *USER_TRIGGERS*. Vizualizarea *ALL_TRIGGERS* conţine informaţii despre toţi declanşatorii din baza de date. Pentru a detecta dependenţele declanşatorilor poate fi consultată vizualizarea *USER_DEPENDENCIES*, iar *ALL_DEPENDECIES* conţine informaţii despre dependenţele tuturor obiectelor din baza de date. Erorile rezultate din compilarea declanşatorilor pot fi analizate din vizualizarea *USER_ERRORS*, iar prin comanda *SHOW ERRORS* se vor afişa erorile corespunzătoare ultimului declanşator compilat.

În operaţiile de gestiune a bazei de date este necesară uneori reconstruirea instrucţiunilor *CREATE TRIGGER*, atunci când codul sursă original nu mai este disponibil. Aceasta se poate realiza utilizând vizualizarea *USER_TRIGGERS*.

Vizualizarea include numele declanşatorului (*TRIGGER_NAME*), tipul acestuia (*TRIGGER_TYPE*), evenimentul declanşator (*TRIGGERING_EVENT*), numele proprietarului tabelului (*TABLE_OWNER*), numele tabelului pe care este definit declanşatorul (*TABLE_NAME*), clauza *WHEN* (*WHEN_CLAUSE*), corpul declanşatorului (*TRIGGER_BODY*), antetul (*DESCRIPTION*), starea acestuia (*STATUS*) care poate să fie *ENABLED* sau *DISABLED* şi numele utilizate pentru a referi parametrii *OLD* şi *NEW* (*REFERENCING_NAMES*). Dacă obiectul de bază nu este un tabel sau o vizualizare, atunci *TABLE_NAME* este *null*.

***Exemplu*:**

Presupunând că nu este disponibil codul sursă pentru declanşatorul *alfa*, să se reconstruiască instrucţiunea *CREATE TRIGGER* corespunzătoare acestuia.

```
SELECT  'CREATE OR REPLACE TRIGGER ' || DESCRIPTION ||
        TRIGGER_BODY
FROM    USER_TRIGGERS
WHERE   TRIGGER_NAME = 'ALFA';
```

Cu această interogare se pot reconstrui numai declanşatorii care aparţin contului utilizator curent. O interogare a vizualizărilor *ALL_TRIGGERS* sau *DBA_TRIGGERS* permite reconstruirea tuturor declanşatorilor din sistem, dacă se dispune de privilegii *DBA*.

***Exemplu*:**

```
SELECT  USERNAME
FROM    USER_USERS;
```

Aceasta cerere furnizeaza numele "proprietarului" (creatorului) declansatorului si nu numele utilizatorului care a reactualizat tabelul.

Privilegii sistem

Sistemul furnizează privilegii sistem pentru gestiunea declanşatorilor:

- *CREATE TRIGGER* (permite crearea declanşatorilor în schema personală);
- *CREATE ANY TRIGGER* (permite crearea declanşatorilor în orice schemă cu excepţia celei corespunzătoare lui *SYS*);
- *ALTER ANY TRIGGER* (permite activarea, dezactivarea sau compilarea declanşatorilor în orice schemă cu excepţia lui *SYS*);
- *DROP ANY TRIGGER* (permite suprimarea declanşatorilor la nivel de bază de date în orice schemă cu excepţia celei corespunzătoate lui *SYS*);
- *ADMINISTER DATABASE TRIGGER* (permite crearea sau modificarea unui declanşator sistem referitor la baza de date);

- *EXECUTE* (permite referirea, în corpul declanşatorului, a procedurilor, funcţiilor sau pachetelor din alte scheme).

Tabele *mutating*

Asupra tabelelor şi coloanelor care pot fi accesate de corpul declanşatorului există anumite restricţii. Pentru a analiza aceste restricţii este necesară definirea tabelelor în schimbare (*mutating*) şi constrânse (*constraining*).

Un tabel *constraining* este un tabel pe care evenimentul declanşator trebuie să-l consulte fie direct, printr-o instrucţiune *SQL*, fie indirect, printr-o constrângere de integritate referenţială declarată. Tabelele nu sunt considerate *constraining* în cazul declanşatorilor la nivel de instrucţiune. Comenzile *SQL* din corpul unui declanşator nu pot modifica valorile coloanelor care sunt declarate chei primare, externe sau unice (*PRIMARY KEY*, *FOREIGN KEY*, *UNIQUE KEY*) într-un tabel *constraining*.

Un tabel *mutating* este tabelul modificat de instrucţiunea *UPDATE*, *DELETE* sau *INSERT*, sau un tabel care va fi actualizat prin efectele acţiunii integrităţii referenţiale *ON DELETE CASCADE*. Chiar tabelul pe care este definit declanşatorul este un tabel *mutating*, ca şi orice tabel referit printr-o constrângere *FOREING KEY*. Tabelele nu sunt considerate *mutating* pentru declanşatorii la nivel de instrucţiune, cu excepţia celor declanşaţi ca efect al opţiunii ON *DELETE CASCADE*. Vizualizările nu sunt considerate *mutating* în declanşatorii *INSTEAD OF*.

Regula care trebuie respectată la utilizarea declanşatoriilor este:
comenzile *SQL* din corpul unui declanşator nu pot consulta sau modifica date dintr-un tabel *mutating*.

Exceptia! Dacă o comandă *INSERT* afectează numai o înregistrare, declanşatorii la nivel de linie (*BEFORE* sau *AFTER*) pentru înregistrarea respectivă nu tratează tabelul ca fiind *mutating*. Acesta este unicul caz în care un declanşator la nivel de linie poate citi sau modifica tabelul. Comanda *INSERT INTO tabel SELECT ...* consideră tabelul *mutating* chiar dacă cererea returnează o singură linie.

***Exemplu*:**

```
CREATE OR REPLACE TRIGGER cascada
  AFTER UPDATE OF cod_artist ON artist
  FOR EACH ROW
BEGIN
  UPDATE opera
  SET    opera.cod_artist = :NEW.cod_artist
  WHERE  opera.cod_artist = :OLD.cod_artist
END;
```

```
UPDATE   artist
SET      cod_artist = 71
WHERE    cod_artist = 23;
```

La execuţia acestei secvenţe este semnalată o eroare. Tabelul *artist* referenţiază tabelul *opera* printr-o constrângere de cheie externă. Prin urmare, tabelul *opera* este *constraining*, iar declanşatorul *cascada* încearcă să schimbe date în tabelul *constraining*, ceea ce nu este permis. Exemplul va funcţiona corect dacă nu este definită sau activată constrângerea referenţială între cele două tabele.

***Exemplu*:**

Să se implementeze cu ajutorul unui declanşator restricţia că într-o sală pot să fie expuse maximum 10 opere de artă.

```
CREATE OR REPLACE TRIGGER TrLimitaopere
  BEFORE INSERT OR UPDATE OF cod_sala ON opera
  FOR EACH ROW
DECLARE
  v_Max_opere CONSTANT NUMBER := 10;
  v_opere_curente      NUMBER;
BEGIN
  SELECT  COUNT(*) INTO v_opere_curente
  FROM    opera
  WHERE   cod_sala = :NEW.cod_sala;
  IF v_opere_curente + 1 > v_Max_opere THEN
    RAISE_APPLICATION_ERROR(-20000,'Prea multe opere de
      arta in sala avand codul ' || :NEW.cod_sala);
  END IF;
END TrLimitaopere;
```

Cu toate că declanşatorul pare să producă lucrul dorit, totuşi după o reactualizare a tabelului *opera* în următoarea manieră:

```
INSERT INTO opera (cod_opera, cod_sala)
VALUES (756893, 10);
```

se obţine următorul mesaj de eroare:

```
ORA-04091: tabel opera is mutating, trigger/function
may not see it
ORA-04088: error during execution of trigger
```

Eroarea *ORA-04091* apare deorece declanşatorul *TrLimitaopere* consultă chiar tabelul (*opera*) la care este asociat declanşatorul (*mutating*).

Tabelul *opera* este *mutating* doar pentru un declanşator la nivel de linie. Aceasta înseamnă că tabelul poate fi consultat în interiorul unui declanşator la nivel de instrucţiune. Totuşi, limitarea numărului operelor de artă nu poate fi făcută în interiorul unui declanşator la nivel de instrucţiune, din moment ce este necesară valoarea *:NEW.cod_sala* în corpul declanşatorului.

```
CREATE OR REPLACE PACKAGE PSalaDate AS
  TYPE t_cod_sala  IS TABLE OF opera.cod_sala%TYPE
                INDEX BY BINARY_INTEGER;
  TYPE t_cod_opera IS TABLE OF opera.cod_opera%TYPE
                INDEX BY BINARY_INTEGER;
  v_cod_sala    t_cod_sala;
  v_cod_opera   t_cod_opera;
  v_NrIntrari  BINARY_INTEGER := 0;
END PSalaDate;

CREATE OR REPLACE TRIGGER TrLLimitaSala
  BEFORE INSERT OR UPDATE OF cod_sala ON opera
  FOR EACH ROW
BEGIN
  PSalaDate.v_NrIntrari := PSalaDate.v_NrIntrari + 1;
  PSalaDate.v_cod_sala(PSalaDate.v_NrIntrari) :=
                                      :NEW.cod_sala;
  PSalaDate.v_cod_opera(PSalaDate.v_NrIntrari) :=
                                      :NEW.cod_opera;
END TrLLimitasala;

CREATE OR REPLACE TRIGGER TrILimitaopere
  AFTER INSERT OR UPDATE OF cod_sala ON opera
DECLARE
  v_Max_opere      CONSTANT NUMBER := 10;
  v_opere_curente  NUMBER;
  v_cod_operax     opera.cod_opera%TYPE;
  v_cod_salax      opera.cod_sala%TYPE;
BEGIN
  FOR v_LoopIndex IN 1..PsalaDate.v_NrIntrari LOOP
    v_cod_operax := PsalaDate.v_cod_opera(v_LoopIndex);
    v_cod_salax := PsalaDate.v_cod_sala(v_LoopIndex);
    SELECT  COUNT(*)
    INTO    v_opere_curente
    FROM    opera
    WHERE   cod_sala = v_cod_salax;
    IF v_opere_curente > v_Max_opere THEN
      RAISE_APPLICATION_ERROR(-20000, 'Prea multe opere de
      arta in sala' || v_cod_salax || 'din cauza inserarii
      operei avand codul' || v_cod_operax)
    END IF;
  END LOOP;
  /* Reseteaza contorul deoarece urmatoarea executie
     va folosi date noi */
  PSalaDate.v_NrIntrari := 0;
END TrILimitaopere;
```

O solutie pentru acestă problemă este crearea a doi declanşatori, unul la nivel de linie şi altul la nivel de instrucţiune. În declanşatorul la nivel de linie se înregistrează valoarea lui *:NEW.cod_opera*, dar nu va fi interogat tabelul *opera*.

Interogarea va fi făcută în declanşatorul la nivel de instrucţiune şi va folosi valoarea înregistrată în declanşatorul la nivel de linie.

O modalitate pentru a înregistra valoarea lui *:NEW.cod_opera* este utilizarea unui tablou indexat în interiorul unui pachet.

***Exemplu*:**

Să se creeze un declanşator care:

a) dacă este eliminată o sală, va şterge toate operele expuse în sala respectivă;

b) dacă se schimbă codul unei săli, va modifica această valoare pentru fiecare operă de artă expusă în sala respectivă.

```
CREATE OR REPLACE TRIGGER sala_cascada
  BEFORE DELETE OR UPDATE OF cod_sala ON sala
  FOR EACH ROW
BEGIN
  IF DELETING THEN
    DELETE FROM opera
    WHERE  cod_sala = :OLD.cod_sala;
  END IF;
  IF UPDATING AND :OLD.cod_sala != :NEW.cod_sala THEN
    UPDATE opera
    SET    cod_sala = :NEW.cod_sala
    WHERE  cod_sala = :OLD.cod_sala;
  END IF;
END sala_cascada;
```

Declanşatorul anterior realizează constrângerea de integritate *UPDATE* sau *ON DELETE CASCADE*, adică ştergerea sau modificarea cheii primare a unui tabel „părinte“ se va reflecta şi asupra înregistrărilor corespunzătoare din tabelul „copil“.

Executarea acestuia, pe tabelul *sala* (tabelul „părinte“), va duce la efectuarea a două tipuri de operaţii pe tabelul *opera* (tabelul „copil“).

La eliminarea unei săli din tabelul *sala*, se vor şterge toate operele de artă corespunzătoare acestei săli.

```
DELETE FROM  sala
WHERE  cod_sala = 773;
```

La modificarea codului unei săli din tabelul *sala*, se va actualiza codul sălii atât în tabelul *sala*, cât şi în tabelul *opera*.

```
UPDATE  sala
SET     cod_sala = 777
WHERE   cod_sala = 333;
```

Se presupune că asupra tabelului *opera* există o constrângere de integritate:

```
FOREIGN KEY (cod_sala) REFERENCES sala(cod_sala)
```

În acest caz sistemul *Oracle* va afişa un mesaj de eroare prin care se precizează că tabelul *sala* este *mutating*, iar constrângerea definită mai sus nu poate fi verificată.

```
ORA-04091: table MASTER.SALA is mutating,
           trigger/function may not see it
```

Pachetele pot fi folosite pentru încapsularea detaliilor logice legate de declanşatori. Exemplul următor arată un mod simplu de implementare a acestei posibilităţi. Este permisă apelarea unei proceduri sau funcţii stocate din blocul *PL/SQL* care reprezintă corpul declanşatorului.

***Exemplu*:**

```
CREATE OR REPLACE PACKAGE pachet IS
  PROCEDURE procesare_trigger(pvaloare IN NUMBER,
                              pstare   IN VARCHAR2);
END pachet;

CREATE OR REPLACE PACKAGE BODY pachet IS
  PROCEDURE procesare_trigger(pvaloare IN NUMBER,
                              pstare   IN VARCHAR2) IS
  BEGIN
  …
  END procesare_trigger;
END pachet;

CREATE OR REPLACE TRIGGER gama
  AFTER INSERT ON opera
  FOR EACH ROW
BEGIN
  pachet.procesare_trigger(:NEW.valoare,:NEW.stare)
END;
```

Tratarea erorilor

Mecanismul de gestiune a erorilor permite utilizatorului să definească şi să controleze comportamentul programului atunci când acesta generează o eroare. În acest fel, aplicaţia nu este oprită, revenind într-un regim normal de execuţie.

Într-un program *PL/SQL* pot să apară erori la compilare sau erori la execuţie.

Erorile care apar în timpul compilării sunt detectate de motorul *PL/SQL* şi sunt comunicate programatorului care va face corecţia acestora. Programul nu poate trata aceste erori deoarece nu a fost încă executat.

Erorile care apar în timpul execuţiei nu mai sunt tratate interactiv. În program trebuie prevăzută apariţia unei astfel de erori şi specificat modul concret de tratare a acesteia. Atunci când apare eroarea este declanşată o excepţie, iar controlul trece la o secţiune separată a programului, unde va avea loc tratarea erorii.

Gestiunea erorilor în *PL/SQL* face referire la conceptul de excepţie. Excepţia este un eveniment particular (eroare sau avertisment) generat de *server*-ul *Oracle* sau de aplicaţie, care necesită o tratare specială. În *PL/SQL* mecanismul de tratare a excepţiilor permite programului să îşi continue execuţia şi în prezenţa anumitor erori.

Excepţiile pot fi definite, activate, tratate la nivelul fiecărui bloc din program (program principal, funcţii şi proceduri, blocuri interioare acestora). Execuţia unui bloc se termină întotdeauna atunci când apare o excepţie, dar se pot executa acţiuni ulterioare apariţiei acesteia, într-o secţiune specială de tratare a excepţiilor.

Posibilitatea de a da nume fiecărei excepţii, de a izola tratarea erorilor într-o secţiune particulară, de a declanşa automat erori (în cazul excepţiilor interne) îmbunătăţeşte lizibilitatea şi fiabilitatea programului. Prin utilizarea excepţiilor şi rutinelor de tratare a excepţiilor, un program *PL/SQL* devine robust şi capabil să trateze atât erorile aşteptate, cât şi cele neaşteptate ce pot apărea în timpul execuţiei.

Secţiunea de tratare a erorilor

Pentru a gestiona excepţiile, utilizatorul trebuie să scrie câteva comenzi care preiau controlul derulării blocului *PL/SQL*. Aceste comenzi sunt situate în secţiunea de tratare a erorilor dintr-un bloc *PL/SQL* şi sunt cuprinse între cuvintele cheie *EXCEPTION* şi *END*, conform următoarei sintaxe generale:

EXCEPTION
 WHEN *nume_excepţie1* [***OR*** *nume_excepţie2 ...*] ***THEN***
  secvenţa_de_instrucţiuni_1;
 [***WHEN*** *nume_excepţie3* [***OR*** *nume_excepţie4 ...*] ***THEN***
  secvenţa_de_instrucţiuni_2;]
 ...
 [***WHEN OTHERS THEN***
  secvenţa_de_instrucţiuni_n;]
END;

De remarcat că *WHEN OTHERS* trebuie să fie ultima clauză şi trebuie să fie unică. Toate excepţiile care nu au fost analizate vor fi tratate prin această clauză. Evident, în practică nu se utilizează forma *WHEN OTHERS THEN NULL*.

În *PL/SQL* există două tipuri de excepţii:

- excepţii interne, care se produc atunci când un bloc *PL/SQL* nu respectă o regulă *Oracle* sau depăşeşte o limită a sistemului de operare;
- excepţii externe definite de utilizator (*user-defined error*), care sunt declarate în secţiunea declarativă a unui bloc, subprogram sau pachet şi care sunt activate explicit în partea executabilă a blocului *PL/SQL*.

Excepţiile interne *PL/SQL* sunt de două tipuri:

- excepţii interne predefinite (*predefined Oracle Server error*);
- excepţii interne nepredefinite (*non-predefined Oracle Server error*).

Funcţii pentru identificarea excepţiilor

Indiferent de tipul excepţiei, aceasta are asociate două elemente:

- un cod care o identifică;
- un mesaj cu ajutorul căruia se poate interpreta excepţia respectivă.

Cu ajutorul funcţiilor *SQLCODE* şi *SQLERRM* se pot obţine codul şi mesajul asociate excepţiei declanşate. Lungimea maximă a mesajului este de 512 caractere.

De exemplu, pentru eroarea predefinită *ZERO_DIVIDE*, codul *SQLCODE* asociat este -1476, iar mesajul corespunzător erorii, furnizat de *SQLERRM*, este „*divide by zero error*“.

Codul erorii este:

- un număr negativ, în cazul unei erori sistem;
- numărul +100, în cazul excepţiei *NO_DATA_FOUND*;
- numărul 0, în cazul unei execuţii normale (fără excepţii);
- numărul 1, în cazul unei excepţii definite de utilizator.

Funcţiile *SQLCODE* şi *SQLERRM* nu se pot utiliza direct ca parte a unei instrucţiuni *SQL*. Valorile acestora trebuie atribuite unor variabile locale.

Rezultatul funcţiei *SQLCODE* poate fi asignat unei variabile de tip numeric, iar cel al funcţiei *SQLERRM* unei variabile de tip caracter. Variabilele locale astfel definite pot fi utilizate în comenzi *SQL*.

***Exemplu*:**

Să se scrie un bloc *PL/SQL* prin care să se exemplifice situaţia comentată.

```
DECLARE
  eroare_cod    NUMBER;
  eroare_mesaj  VARCHAR2(100);
BEGIN
  …
EXCEPTION
  …
  WHEN OTHERS THEN
    eroare_cod := SQLCODE;
    eroare_mesaj := SUBSTR(SQLERRM,1,100);
    INSERT  INTO erori
    VALUES  (eroare_cod, eroare_mesaj);
END;
```

Mesajul asociat excepţiei declanşate poate fi furnizat şi de funcţia *DBMS_UTILITY.FORMAT_ERROR_STACK*.

Excepţii interne

Excepţiile interne se produc atunci când un bloc *PL/SQL* nu respectă o regulă *Oracle* sau depăşeşte o limită a sistemului de exploatare.

Aceste excepţii pot fi independente de structura bazei de date sau pot să apară datorită nerespectării constrângerilor statice implementate în structură (*PRIMARY KEY*, *FOREIGN KEY*, *NOT NULL*, *UNIQUE*, *CHECK*).

Atunci când apare o eroare *Oracle*, excepţia asociată ei se declanşează implicit. De exemplu, dacă apare eroarea *ORA-01403* (deoarece o comandă *SELECT* nu returnează nici o linie), atunci implicit *PL/SQL* activează excepţia *NO_DATA_FOUND*. Cu toate că fiecare astfel de excepţie are asociat un cod specific, ele trebuie referite prin nume.

Excepţii interne predefinite

Excepţiile interne predefinite nu trebuie declarate în secţiunea declarativă şi sunt tratate implicit de către *server*-ul *Oracle*. Ele sunt referite prin nume (*CURSOR_ALREADY_OPEN*, *DUP_VAL_ON_INDEX*, *NO_DATA_FOUND* etc.). *PL/SQL* declară aceste excepţii în pachetul *DBMS_STANDARD*.

Nume excepţie	**Cod eroare**	**Descriere**
ACCES_INTO_NULL	*ORA-06530*	Asignare de valori atributelor unui obiect neiniţializat.
CASE_NOT_FOUND	*ORA-06592*	Nu este selectată nici una din clauzele *WHEN* ale lui *CASE* şi nu există nici clauza *ELSE* (excepţie specifică lui *Oracle9i*).
COLLECTION_IS_NULL	*ORA-06531*	Aplicarea unei metode (diferite de *EXISTS*) unui tabel imbricat sau unui vector neiniţializat.
CURSOR_ALREADY_OPEN	*ORA-06511*	Deschiderea unui cursor care este deja dechis.
DUP_VAL_ON_INDEX	*ORA-00001*	Detectarea unei dubluri într-o coloană unde acestea sunt interzise.
INVALID_CURSOR	*ORA-01001*	Operaţie ilegală asupra unui cursor.
INVALID_NUMBER	*ORA-01722*	Conversie nepermisă de la tipul şir de caractere la număr.
LOGIN_DENIED	*ORA-01017*	Nume sau parolă incorecte.
NO_DATA_FOUND	*ORA-01403*	Comanda *SELECT* nu returnează nici o înregistrare.
NOT_LOGGED_ON	*ORA-01012*	Programul *PL/SQL* apelează baza fără să fie conectat la *Oracle*.
SELF_IS_NULL	*ORA-30625*	Apelul unei metode când instanţa este *NULL*.
PROGRAM_ERROR	*ORA-06501*	*PL/SQL* are o problemă internă.
ROWTYPE_MISMATCH	*ORA-06504*	Incompatibilitate între parametrii actuali şi formali, la deschiderea unui cursor parametrizat.
STORAGE_ERROR	*ORA-06500*	*PL/SQL* are probleme cu spaţiul de memorie.
SUBSCRIPT_BEYOND_COUNT	*ORA-06533*	Referire la o componentă a unui *nested table* sau *varray*, folosind un index mai mare decât numărul elementelor colecţiei respective.
SUBSCRIPT_OUTSIDE_LIMIT	*ORA-06532*	Referire la o componentă a unui tabel imbricat sau vector, folosind un index care este în afara domeniului (de exemplu, -1).
SYS_INVALID_ROWID	*ORA-01410*	Conversia unui şir de caractere într-un *ROWID* nu se poate face deoarece şirul nu reprezintă un *ROWID* valid.
TIMEOUT_ON_RESOURCE	*ORA-00051*	Expirarea timpului de aşteptare pentru eliberarea unei resurse.
TRANSACTION_BACKED_OUT	*ORA-00061*	Tranzacţia a fost anulată datorită unei interblocări.
TOO_MANY_ROWS	*ORA-01422*	*SELECT…INTO* întoarce mai multe linii.
VALUE_ERROR	*ORA-06502*	Apariţia unor erori în conversii, constrângeri sau erori aritmetice.
ZERO_DIVIDE	*ORA-01476*	Sesizarea unei împărţiri la zero.

***Exemplu*:**

Să se scrie un bloc *PL/SQL* prin care să se afişeze numele artiştilor de o anumită naţionalitate care au opere de artă expuse în muzeu.

1) Dacă rezultatul interogării returnează mai mult decât o linie, atunci să se trateze excepţia şi să se insereze în tabelul *mesaje* textul „mai mulţi creatori“.

2) Dacă rezultatul interogării nu returnează nici o linie, atunci să se trateze excepţia şi să se insereze în tabelul *mesaje* textul „nici un creator“.

3) Dacă rezultatul interogării este o singură linie, atunci să se insereze în tabelul *mesaje* numele artistului şi pseudonimul acestuia.

4) Să se trateze orice altă eroare, inserând în tabelul *mesaje* textul „alte erori au apărut“.

```
SET VERIFY OFF
ACCEPT national PROMPT 'Introduceti nationalitatea:'
DECLARE
  v_nume_artist   artist.nume%TYPE;
  v_pseudonim     artist.pseudonim%TYPE;
  v_national      artist.national%TYPE:='&national';
BEGIN
  SELECT   nume, pseudonim
  INTO     v_nume_artist, v_pseudonim
  FROM     artist
  WHERE    national = v_national;
  INSERT   INTO mesaje (rezultate)
  VALUES   (v_nume_artist||'-'||v_pseudonim);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
     INSERT   INTO mesaje (rezultate)
     VALUES   ('nici un creator');
  WHEN TOO_MANY_ROWS THEN
     INSERT   INTO mesaje (rezultate)
     VALUES   ('mai multi creatori');
  WHEN OTHERS THEN
     INSERT   INTO mesaje (rezultate)
     VALUES   ('alte erori au aparut');
END;
/
SET VERIFY ON
```

Aceeaşi excepţie poate să apară în diferite circumstanţe. De exemplu, excepţia *NO_DATA_FOUND* poate fi generată fie pentru că o interogare nu întoarce un rezultat, fie pentru că se referă un element al unui tablou *PL/SQL* care nu a fost definit (nu are atribuită o valoare). Dacă într-un bloc *PL/SQL* apar ambele situaţii, este greu de stabilit care dintre ele a generat eroarea şi este necesară restructurarea blocului, astfel încât acesta să poată diferenţia cele două situaţii.

Excepţii interne nepredefinite

Excepţiile interne nepredefinite sunt declarate în secţiunea declarativă şi sunt tratate implicit de către *server*-ul *Oracle*. Ele pot fi gestionate prin clauza *OTHERS*, în secţiunea *EXCEPTION*.

Diferenţierea acestor erori este posibilă doar cu ajutorul codului. După cum s-a mai specificat, codul unei excepţii interne este un număr negativ, în afară de excepţia *NO_DATA_FOUND*, care are codul +100.

O altă metodă pentru tratarea unei erori interne nepredefinite (diferită de folosirea clauzei *OTHERS* drept detector universal de excepţii) este utilizarea directivei de compilare (pseudo-instrucţiune) *PRAGMA EXCEPTION_INIT*. Această directivă permite asocierea numelui unei excepţii cu un cod de eroare intern. În felul acesta, orice excepţie internă poate fi referită printr-un nume şi se pot scrie rutine speciale pentru tratarea acesteia. Directiva este procesată în momentul compilării, şi nu la execuţie.

Directiva trebuie să apară în partea declarativă a unui bloc, pachet sau subprogram, după definirea numelui excepţiei. *PRAGMA EXCEPTION_INIT* poate să apară de mai multe ori într-un program. De asemenea, pot fi asignate mai multe nume pentru acelaşi cod de eroare.

În acest caz, tratarea erorii se face în următoarea manieră:

1) se declară numele excepţiei în partea declarativă sub forma:

nume_excepţie ***EXCEPTION;***

2) se asociază numele excepţiei cu un cod eroare standard *Oracle*, utilizând comanda:

PRAGMA EXCEPTION_INIT (*nume_excepţie*, *cod_eroare*)*;*

3) se referă excepţia în secţiunea de gestiune a erorilor (excepţia este tratată automat, fără a fi necesară comanda *RAISE*).

***Exemplu*:**

Dacă există opere de artă create de un anumit artist, să se tipărească un mesaj prin care utilizatorul este anunţat că artistul respectiv nu poate fi şters din baza de date (violarea constrângerii de integritate având codul eroare *Oracle* -2292).

```
SET VERIFY OFF
DEFINE p_nume = Monet
DECLARE
  opera_exista  EXCEPTION;
  PRAGMA EXCEPTION_INIT(opera_exista,-2292);
BEGIN
  DELETE FROM artist WHERE nume = '&p_nume';
```

```
  COMMIT;
EXCEPTION
  WHEN opera_exista THEN
    DBMS_OUTPUT.PUT_LINE ('nu puteti sterge artistul cu
         numele ' || '&p_nume' || ' deoarece exista in
         muzeu opere de arta create de acesta');
END;
/
SET VERIFY ON
```

Excepţii externe

PL/SQL permite utilizatorului să definească propriile sale excepţii. Aceste excepţii pot să apară în toate secţiunile unui bloc, subprogram sau pachet. Excepţiile externe sunt definite în partea declarativă a blocului, deci posibilitatea de referire la ele este asigurată. În mod implicit, toate excepţiile externe au asociat acelaşi cod (+1) şi acelaşi mesaj (*USER DEFINED EXCEPTION*).

Tratarea unei astfel de erori se face într-o manieră similară modului de tratare descris anterior. Activarea excepţiei externe este făcută explicit, folosind comanda *RAISE* însoţită de numele excepţiei. Comanda opreşte execuţia normală a blocului *PL/SQL* şi transferă controlul „administratorului“ excepţiilor.

Declararea şi prelucrarea excepţiilor externe respectă următoarea sintaxă:

DECLARE
nume_excepţie ***EXCEPTION***; -- *declarare excepţie*
BEGIN
...
RAISE *nume_excepţie*; *--declanşare excepţie*
-- codul care urmează nu mai este executat
...
EXCEPTION
WHEN *nume_excepţie* ***THEN***
-- definire mod de tratare a erorii
...
END;

Excepţiile trebuie privite ca nişte variabile, în sensul că ele sunt active în secţiunea în care sunt declarate. Ele nu pot să apară în instrucţiuni de atribuire sau în comenzi *SQL*.

Este recomandat ca fiecare subprogram să aibă definită o zonă de tratare a excepţiilor. Dacă pe parcursul execuţiei programului intervine o eroare, atunci acesta generează o excepţie şi controlul se transferă blocului de tratare a erorilor.

***Exemplu*:**

Să se scrie un bloc *PL/SQL* care afişează numărul creatorilor operelor de artă din muzeu care au valoarea mai mare sau mai mică cu 100000$ decât o valoare specificată. Să se tipărească un mesaj adecvat, dacă nu există nici un artist care îndeplineşte această condiţie.

```
VARIABLE g_mesaj VARCHAR2(100)
SET VERIFY OFF
ACCEPT p_val PROMPT 'va rog specificati valoarea:'
DECLARE
  v_val       opera.valoare%TYPE := &p_val;
  v_inf       opera.valoare%TYPE := v_val - 100000;
  v_sup       opera.valoare%TYPE := v_val + 100000;
  v_numar     NUMBER(7);
  e_nimeni    EXCEPTION;
  e_mai_mult  EXCEPTION;
BEGIN
  SELECT  COUNT(DISTINCT cod_autor)
  INTO    v_numar
  FROM    opera
  WHERE   valoare BETWEEN v_inf AND v_sup;
  IF v_numar = 0 THEN
     RAISE  e_nimeni;
  ELSIF v_numar > 0 THEN
     RAISE e_mai_mult;
  END IF;
EXCEPTION
  WHEN e_nimeni THEN
    :g_mesaj:='nu exista nici un artist cu valoarea
         operelor cuprinsa intre '||v_inf ||' si '||v_sup;
  WHEN e_mai_mult THEN
    :g_mesaj:='exista '||v_numar||' artisti cu valoarea
         operelor cuprinsa intre '||v_inf||' si '||v_sup;
  WHEN OTHERS THEN
    :g_mesaj:='au aparut alte erori';
END;
/

SET VERIFY ON
PRINT g_mesaj
```

Activarea unei excepţii externe poate fi făcută şi cu ajutorul procedurii *RAISE_APPLICATION_ERROR*, furnizată de pachetul *DBMS_STANDARD*.

RAISE_APPLICATION_ERROR poate fi folosită pentru a returna un mesaj de eroare unităţii care o apelează, mesaj mai descriptiv (non standard) decât identificatorul erorii. Unitatea apelantă poate fi *SQL*Plus*, un subprogram *PL/SQL* sau o aplicaţie *client*.

Procedura are următorul antet:

RAISE_APPLICATION_ERROR (*numar_eroare* ***IN NUMBER***,
mesaj_eroare ***IN VARCHAR2***, [{***TRUE*** | ***FALSE***}]);

Atributul *numar_eroare* este un număr cuprins între –20000 şi –20999, specificat de utilizator pentru excepţia respectivă, iar *mesaj_eroare* este un text asociat erorii, care poate avea maximum 2048 octeţi.

Parametrul boolean este opţional. Dacă acest parametru este *TRUE*, atunci noua eroare se va adăuga listei erorilor existente, iar dacă este *FALSE* (valoare implicită) atunci noua eroare va înlocui lista curentă a erorilor (se retine ultimul mesaj de eroare).

O aplicaţie poate apela *RAISE_APPLICATION_ERROR* numai dintr-un subprogram stocat (sau metodă). Dacă *RAISE_APPLICATION_ERROR* este apelată, atunci subprogramul se termină şi sunt returnate codul şi mesajul asociate erorii respective.

Procedura *RAISE_APPLICATION_ERROR* poate fi folosită în secţiunea executabilă, în secţiunea de tratare a erorilor şi chiar simultan în ambele secţiuni.

- În secţiunea executabilă:

```
DELETE FROM  opera  WHERE  material = 'carton';
IF SQL%NOTFOUND THEN
  RAISE_APPLICATION_ERROR(-20201,'info incorecta');
END IF;
```

- În secţiunea de tratare a erorilor:

```
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20202,'info invalida');
END;
```

- În ambele secţiuni:

```
DECLARE
  e_material  EXCEPTION;
  PRAGMA EXCEPTION_INIT (e_material, -20777);
BEGIN
  …
  DELETE FROM opera WHERE valoare < 100001;
  IF SQL%NOTFOUND THEN
    RAISE_APPLICATION_ERROR(-20777,
       'nu exista opera cu aceasta valoare');
  END IF;
EXCEPTION
  WHEN e_material THEN
    -- trateaza eroarea aceasta
  …
END;
```

RAISE_APPLICATION_ERROR facilitează comunicaţia dintre *client* şi *server*, transmiţând aplicaţiei *client* erori specifice aplicaţiei de pe *server* (de obicei, un declanşator). Prin urmare, procedura este doar un mecanism folosit pentru comunicaţia *server* → *client* a unei erori definite de utilizator, care permite ca procesul *client* să trateze excepţia.

***Exemplu*:**

Să se implementeze un declanşator care nu permite acceptarea în muzeu a operelor de artă având valoarea mai mică de 100000$.

```
CREATE OR REPLACE TRIGGER minim_valoare
BEFORE INSERT ON opera
FOR EACH ROW
BEGIN
  IF :NEW.valoare < 100000 THEN
    RAISE_APPLICATION_ERROR
      (-20005,'operele de arta trebuie sa aiba valoare
               mai mare de 100000$');
  END IF;
END;
```

Pe staţia *client* poate fi scris un program care detectează şi tratează eroarea.

```
DECLARE
  /* declarare excepţie */
  nu_accepta  EXCEPTION;
  /* asociază nume,codului eroare folosit in trigger */
  PRAGMA EXCEPTION_INIT(nu_accepta,-20005);
BEGIN
  /* incearca sa inserezi */
  INSERT   INTO opera …;
EXCEPTION
  /* tratare exceptie */
  WHEN nu_accepta THEN
    DBMS_OUTPUT.PUT_LINE(SQLERRM);
   /* SQLERRM va returna mesaj din RAISE_APPLICATION_ERROR */
END;
```

Cazuri speciale în tratarea excepţiilor

Dacă se declanşează o excepţie într-un bloc simplu, atunci se face saltul la partea de tratare (*handler*) a acesteia, iar după ce este terminată tratarea erorii se iese din bloc (instrucţiunea *END*).

Prin urmare, dacă excepţia se propagă spre blocul care include blocul curent, restul acţiunilor executabile din subbloc sunt „pierdute“. Dacă după o eroare se doreşte totuşi continuarea prelucrării datelor, este suficient ca instrucţiunea care a declanşat excepţia să fie inclusă într-un subbloc.

După ce subblocul a fost terminat, se continuă secvenţa de instrucţiuni din blocul principal.

***Exemplu*:**

```
BEGIN
  DELETE …
  SELECT …--poate declansa exceptia A
          --nu poate fi efectuat INSERT care urmeaza
  INSERT INTO …
EXCEPTION
  WHEN A THEN …
END;
```

Deficienţa anterioară se poate rezolva incluzând într-un subbloc comanda *SELECT* care a declanşat excepţia.

```
BEGIN
  DELETE …
   BEGIN
     SELECT …
     …
   EXCEPTION
    WHEN A THEN …
    /* dupa ce se trateaza exceptia A, controlul este
      transferat blocului de nivel superior, de fapt
      comenzii INSERT */
   END;
  INSERT  INTO …
  …
EXCEPTION
…
END;
```

Uneori este dificil de aflat care comandă *SQL* a determinat o anumită eroare, deoarece există o singură secţiune pentru tratarea erorilor unui bloc. Sunt sugerate două soluţii pentru rezolvarea acestei probleme.

1) Introducerea unui contor care să identifice instrucţiunea *SQL*.

```
DECLARE
  v_sel_cont  NUMBER(2):=1;
BEGIN
  SELECT …
  v_sel_cont:=2;
  SELECT …
  v_sel_cont:=3;
  SELECT …
EXCEPTION
  WHEN NO_DATA_FOUND THEN
```

```
    INSERT INTO log_table(info)
    VALUES ('comanda SELECT ' || TO_CHAR(v_sel_cont) ||
           ' nu gaseste date');
END;
```

2) Introducerea fiecărei instrucţiuni *SQL* într-un subbloc.

```
BEGIN
  BEGIN
    SELECT …
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      INSERT INTO log_table(info)
      VALUES('SELECT 1 nu gaseste date');
  END;
  BEGIN
    SELECT …
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      INSERT INTO log_table(info)
      VALUES('SELECT 2 nu gaseste date');
  END;
  …
END;
```

Activarea excepţiilor

Pentru activarea unei excepţii există două metode:

- activarea explicită a excepţiei (definite de utilizator sau predefinite) în interiorul blocului, cu ajutorul comenzii *RAISE*;
- activarea automată a excepţiei asociate unei erori *Oracle*.

Excepţiile pot fi sesizate în secţiunea executabilă, declarativă sau în cea de tratare a excepţiilor. La aceste niveluri ale programului, o excepţie poate fi gestionată în moduri diferite.

Pentru a reinvoca o excepţie, după ce a fost tratată în blocul curent, se foloseşte instrucţiunea *RAISE*, dar fără a fi însoţită de numele excepţiei. În acest fel, după executarea instrucţiunilor corespunzătoare tratării excepţiei, aceasta se transmite şi blocului „părinte“. Pentru a fi recunoscută ca atare de către blocul „părinte“, excepţia trebuie să nu fie definită în blocul curent, ci în blocul „părinte“ (sau chiar mai sus în ierarhie), în caz contrar ea putând fi captată de către blocul „părinte“ doar la categoria *OTHERS*.

Pentru a executa acelaşi set de acţiuni în cazul mai multor excepţii nominalizate explicit, în secţiunea de prelucrare a excepţiilor se poate utiliza operatorul *OR*.

Pentru a evita tratarea fiecărei erori în parte, se foloseşte secţiunea *WHEN OTHERS* care va cuprinde acţiuni pentru fiecare excepţie care nu a fost tratată, adică pentru captarea excepţiilor neprevăzute sau necunoscute. Această secţiune trebuie utilizată cu atenţie deoarece poate masca erori critice sau poate împiedica aplicaţia să răspundă în mod corespunzător.

Propagarea excepţiilor

Dacă este declanşată o eroare în secţiunea executabilă şi blocul curent are un *handler* pentru tratarea ei, atunci blocul se termină cu succes, iar controlul este dat blocului imediat exterior.

Dacă se produce o excepţie care nu este tratată în blocul curent, atunci excepţia se propagă spre blocul „părinte“, iar blocul *PL/SQL* curent se termină fără succes. Procesul se repetă până când fie se găseşte într-un bloc modalitatea de tratare a erorii, fie se opreşte execuţia şi se semnalează situaţia apărută (*unhandled exception error*).

Dacă este declanşată o eroare în partea declarativă a blocului, aceasta este propagată către blocul imediat exterior, chiar dacă există un *handler* al acesteia în blocul corespunzător secţiunii declarative.

La fel se întâmplă dacă o eroare este declanşată în secţiunea de tratare a erorilor. La un moment dat, într-o secţiune *EXCEPTION*, poate fi activă numai o singură excepţie.

Instrucţiunea *GOTO* nu permite:

- saltul la secţiunea de tratare a unei excepţii;
- saltul de la secţiunea de tratare a unei excepţii, în blocul curent.

Comanda *GOTO* permite totuşi saltul de la secţiunea de tratare a unei excepţii la un bloc care include blocul curent.

***Exemplu*:**

Exemplul următor marchează un salt ilegal în blocul curent.

```
DECLARE
  v_var  NUMBER(10,3);
BEGIN
  SELECT dim2/NVL(valoare,0)
  INTO   v_var
  FROM   opera
  WHERE  dim1 > 100;
  <<eticheta>>
  INSERT  INTO politaasig(cod_polita, valoare)
  VALUES  (7531, v_var);
EXCEPTION
  WHEN ZERO_DIVIDE THEN v_var:=0;
    GOTO <<eticheta>>; --salt ilegal in blocul curent
END;
```

În continuare, vor fi analizate modalităţile de propagare a excepţiilor în cele trei cazuri comentate: excepţii sesizate în secţiunea declarativă, în secţiunea executabilă şi în secţiunea de tratare a erorilor.

Excepţie sesizată în secţiunea executabilă

Excepţia este sesizată şi tratată în subbloc. După aceea, controlul revine blocului exterior.

```
DECLARE
  A  EXCEPTION;
BEGIN
  …
  BEGIN
    RAISE A; -- exceptia A sesizata in subbloc
  EXCEPTION
    WHEN A THEN …-- exceptia tratata in subbloc
    …
  END;
-- aici este reluat controlul
END;
```

Excepţia este sesizată în subbloc, dar nu este tratată în acesta şi atunci se propagă spre blocul exterior. Regula poate fi aplicată de mai multe ori.

```
DECLARE
  A  EXCEPTION;
  B  EXCEPTION;
BEGIN
  BEGIN
    RAISE B; --exceptia B sesizata in subbloc
  EXCEPTION
    WHEN A THEN …
     --exceptia B nu este tratata in subbloc
  END;
EXCEPTION
  WHEN B THEN …
  /* exceptia B s-a propagat spre blocul exterior unde a
  fost tratata, apoi controlul trece in exteriorul blocului */
END;
```

Excepţie sesizată în secţiunea declarativă

Dacă în secţiunea declarativă este generată o excepţie, atunci aceasta se propagă către blocul exterior, unde are loc tratarea acesteia. Chiar dacă există un *handler* pentru excepţie în blocul curent, acesta nu este executat.

***Exemplu*:**

Să se realizeze un program prin care să se exemplifice propagarea erorilor apărute în secţiunea declarativă a unui bloc *PL/SQL*.

Programul calculează numărul creatorilor de opere de artă care au lucrări expuse în muzeu.

```
BEGIN
  DECLARE
    nr_artisti   NUMBER(3) := 'XYZ';
  BEGIN
    SELECT  COUNT (DISTINCT cod_autor)
    INTO    nr_artisti
    FROM    opera;
  EXCEPTION
    WHEN OTHERS THEN
     DBMS_OUTPUT.PUT_LINE('Eroare bloc intern:' || SQLERRM);
  END;
EXCEPTION
  WHEN OTHERS THEN
   DBMS_OUTPUT.PUT_LINE('Eroare bloc extern:' || SQLERRM );
END;
```

Deoarece la iniţializarea variabilei *nr_artisti* apare o neconcordanţă între tipul declarat şi cel asignat, este generată eroarea internă *VALUE_ERROR*. Cum eroarea a apărut în partea declarativă a blocului intern, deşi acesta conţine un *handler OTHERS* care ar fi putut capta eroarea, *handler*-ul nu este executat, eroarea fiind propagată către blocul extern unde este tratată în *handler*-ul *OTHERS* asociat. Aceasta se poate remarca deoarece la execuţie se obţine mesajul: „*Eroare bloc extern*: *ORA-06502: PL/SQL*: *numeric or value error*“.

Excepţie sesizată în secţiunea *EXCEPTION*

Dacă excepţia este sesizată în secţiunea *EXCEPTION*, ea se propagă imediat spre blocul exterior.

```
BEGIN
  DECLARE
    A  EXCEPTION;
    B  EXCEPTION;
  BEGIN
    RAISE A; --sesizare exceptie A
  EXCEPTION
    WHEN A THEN
      RAISE B; --sesizare exceptie B
    WHEN B THEN …
     /* exceptia este propagata spre blocul exterior
      cu toate ca exista aici un handler pentru ea */
  END;
EXCEPTION
  WHEN B THEN …
  --exceptia B este tratata in blocul exterior
END;
```

Informaţii despre erori

Pentru a obţine textul corespunzător erorilor la compilare, poate fi utilizată vizualizarea *USER_ERRORS* din dicţionarul datelor. Pentru informaţii adiţionale referitoare la erori pot fi consultate vizualizările *ALL_ERRORS* sau *DBA_ERRORS*.

Vizualizarea *USER_ERRORS* are câmpurile:
NAME (numele obiectului),
TYPE (tipul obiectului),
SEQUENCE (numărul secvenţei),
LINE (numărul liniei din codul sursă în care a apărut eroarea),
POSITION (poziţia în linie unde a apărut eroarea),
TEXT (mesajul asociat erorii).

***Exemplu*:**

Să se afişeze erorile de compilare din procedura *alfa*.

```
SELECT LINE, POSITION, TEXT
FROM   USER_ERRORS
WHERE  NAME = 'ALFA';
```

LINE specifică numărul liniei în care apare eroarea, dar acesta nu corespunde liniei efective din fişierul text (se referă la codul sursă depus în *USER_SOURCE*).