Baze de date – Îndrumar de Laborator

Capitolul 6 – Cursoare, proceduri stocate, funcţii şi triggere

1. Cursoare

După cum am constatat în capitolul precedent, în PL/SQL prin comanda *SELECT ... INTO listă_variabile* nu putem prelua decât conţinutul unei singure linii rezultat, deoarece variabilele din *listă_variabile* sunt scalare (o variabilă poate prelua o singură valoare la un moment dat).

Pentru a folosi în PL/SQL interogări care generează mai multe linii, avem nevoie de un cursor.

Un cursor reprezintă un set de înregistrări obținut printr-o interogare şi stocat în memorie, care poate fi parcurs secvențial, uneori într-o singură direcție (de la prima înregistrare la ultima), alteori în ambele direcții, fiecare înregistrare păstrând, în unele cazuri, legătura cu linia tabelei din care provine [FO].

Cursorul se defineşte în zona de declaraţii a blocului PL/SQL, prin stabilirea numelui cursorului, a interogării asociate (comanda SELECT care se va executa la deschiderea cursorului), si a altor opţiuni. Apoi, în interiorul programului (după BEGIN) rularea sau deschiderea cursorului se face cu comanda OPEN, care va determina executarea interogării asociate cursorului, şi setarea poziţiei curente la prima linie din mulţimea de linii rezultat. Extragerea uneia sau mai multor linii din cursor se face cu comanda FETCH. După fiecare rulare a comenzii FETCH poziţia curentă în cadrul cursorului este actualizată conform regulii de parcurgere (implicit înainte). După ce am terminat de preluat liniile din cursor este recomandat să nu uităm să închidem cursorul (cu comanda CLOSE) pentru a elibera zona de memorie alocată acestuia.

Exemplu de cursor care preia numele şi prenumele funcţionarilor (angajaţii din tabela Employees care au ca Job_ID 'ST_CLERK'):

```
SET  SERVEROUTPUT  ON
DECLARE
    v_prenume   Employees.First_name%TYPE;
    v_nume      Employees.Last_name%TYPE;

    CURSOR  Nume_functionari  IS
       SELECT First_name, Last_name FROM Employees
                 WHERE Job_ID = 'ST_CLERK';
BEGIN
   OPEN  Nume_functionari;
   LOOP
         FETCH   Nume_functionari  INTO  v_prenume, v_nume;
         EXIT WHEN  Nume_functionari%NOTFOUND;
         DBMS_OUTPUT.PUT_LINE(v_prenume|| '  ' || v_nume);
   END LOOP;
   CLOSE Nume_functionari;

END;
/
```

Atributele cursorului pot fi consultate **după** ce s-a făcut primul FETCH si ele au următoarele valori:

- nume_cursor%ROWCOUNT: întoarce numărul de linii încărcate cu FETCH de la deschiderea cursorului.
- nume_cursor%FOUND: TRUE dacă ultimul FETCH a încărcat o nouă linie şi FALSE altfel.
- nume_cursor%NOTFOUND: funcţionează invers faţă de FOUND.
- nume_cursor%ISOPEN: TRUE dacă nume_cursor e deschis şi FALSE altfel

Pentru a simplifica preluarea datelor din cursoare putem utiliza *tipul de date înregistrare* (RECORD). De exemplu, putem declara o variabilă de tip înregistrare folosind expresia:

Nume_Tabela%ROWTYPE

Exemplu:

```
DECLARE
linie  Employees%ROWTYPE;  -- o variabilă ce va putea stoca o linie din tabela Employees
```

Apoi, în program, vom putea accesa fiecare din valorile atributelor coloanelor din înregistrarea *linie* cu operatorul **.** (punct), de exemplu: *linie.First_Name, linie.Last_Name* etc.

Exerciţiu:

- modificaţi programul de mai sus utilizând în loc de variabilele v_prenume şi v_nume o variabilă de tip înregistrare cu numele *linie* (va fi nevoie de SELECT * …).

Cursorul utilizat mai sus se numeşte cursor explicit. În Oracle există şi *cursoare implicite*. Un exemplu de cursor implicit este prezentat în programul următor:

```
SET  SERVEROUTPUT  ON
BEGIN
    FOR  persoana  IN
       (SELECT * FROM Employees WHERE Employee_id < 120)
    LOOP
       DBMS_OUTPUT.PUT_LINE('Prenume = ' || persoana.first_name ||
                                    ', Nume = ' || persoana.last_name);
    END LOOP;

END;
/
```

Notă: observaţi că variabila înregistrare „persoana” nu trebuie declarată, şi toate comenzile care ţin de utilizarea cursorului (OPEN, FETCH, CLOSE) se execută automat.

2. Proceduri stocate

Sub-programele (procedurile şi funcţiile) au apărut în programarea calculatoarelor datorită creşterii complexităţii programelor şi a nevoii de a grupa instrucţiuni. Astfel, dacă avem secvenţe de cod care se reiau de mai multe ori în procesul de calcul pentru rezolvarea unei anumite probleme, şi pentru care diferă eventual doar nişte valori de intrare, putem scrie secvenţa o singură dată într-o procedură sau funcţie, şi o apelăm ori de câte ori este nevoie în program. Pentru programatori, avantajele utilizării procedurilor şi a funcţiilor sunt majore, de exemplu putem utiliza funcţii complexe create de alţii, doar cunoscând parametrii de intrare/ieşire necesari, fără a trebui să cunoaştem algoritmul sau codul efectiv din funcţia respectivă.

O procedură stocată este o procedură care implementează o parte din algoritmii de calcul ai aplicațiilor şi care este memorată în baza de date la fel ca şi alte obiecte ale bazei de date [FI]. Este compilată şi optimizată o singură dată şi rămâne memorată în server pentru oricâte apeluri ulterioare. La compilarea procedurii, numele acesteia (stabilit prin CREATE PROCEDURE) devine un nume de obiect în dicționarul datelor, tipul obiectului fiind PROCEDURE.

Parametri de intrare/ieşire (input/output) ai procedurilor stocate

O procedură stocată în PL/SQL poate avea mai mulţi parametri de intrare (declaraţi cu IN), de ieşire (OUT) sau de intrare-ieşire (IN OUT). Parametrii de tip IN nu pot fi modificaţi în procedură.

Exemplu de procedură stocată:

```
CREATE OR REPLACE PROCEDURE  nume_angajat (
    id_angajat  IN    Employees.employee_id%TYPE ,
    v_nume    OUT Employees.last_name%TYPE )
IS
  -- aici putem avea declarații de variabile locale
BEGIN
      SELECT last_name  INTO  v_nume   FROM Employees
                WHERE Employee_id = id_angajat;
END;
```

Procedura *nume_angajat* primeşte valoarea unui id de angajat în variabila de intrare *id_angajat*, foloseşte această valoare în interogare, şi returnează programului principal care a apelat procedura (numit program apelant) conţinutul variabilei de ieşire *v_nume*. Toate variabilele din procedură sunt locale, adică vor dispărea după rularea procedurii. Procedura se compilează folosind comanda *run statement* sau *run script*. Observaţi că aceasta, după compilare, apare în fereastra din stânga a SQL Developer, la rubrica „Procedures", adică a fost stocată între obiectele bazei de date.

Apelarea unei proceduri stocate se face din programul principal scriind numele procedurii împreună cu parametrii de intrare-ieşire necesari:

```
SET  SERVEROUTPUT  ON;
DECLARE
  id           Employees.employee_id%TYPE;
  v_nume    Employees.last_name%TYPE;
BEGIN
  id := 206;
  nume_angajat(id , v_nume);  -- apelul procedurii stocate
  DBMS_OUTPUT.PUT_LINE('Numele angajatului nr. ' || TO_CHAR(id) ||' este ' || v_nume );
END;
/
```

3. Funcţii definite de utilizator

O funcție definită de utilizator este o funcție memorată în baza de date, la fel ca o procedură stocată; diferența între acestea este că o funcţie returnează întotdeauna o singura valoare şi poate fi folosită direct în expresii, pe când o procedură stocată poate să nu returneze nici o valoare [FI].
O funcţie are numai parametri de intrare, iar specificatorii IN / OUT / IN OUT nu se mai folosesc la funcţii (toţi parametrii fiind implicit de tip IN).

Exemplu de funcţie stocată:

```
CREATE OR REPLACE FUNCTION  Salariu_ang (id  NUMBER)
  RETURN  NUMBER  IS
    v_salariu  Employees.Salary%TYPE;
BEGIN
    SELECT Salary INTO v_salariu FROM Employees
              WHERE Employee_id = id;
    RETURN (v_salariu);
END;
```

Tipul de date al variabilei de ieşire este precizat la început între cuvintele RETURN şi IS. Orice funcţie trebuie să aibă măcar o instrucţiune RETURN, care va returna o valoare către programul apelant (la întâlnirea instrucţiunii RETURN funcţia se termină cu returnarea valorii).

Funcţia definită mai sus poate fi apelată de exemplu cu următorul program:

```
SET  SERVEROUTPUT  ON;
BEGIN
   DBMS_OUTPUT.PUT_LINE('Salariul angajatului cu id ' ||
```

```
TO_CHAR(206) ||' este ' || TO_CHAR( Salariu_ang(206) ));
END;
/
```

4. Declanşatoare (triggere)

Un trigger este o procedură stocată specială, care este executată automat atunci când se efectuează operaţii de actualizare a relaţiilor (INSERT, DELETE, UPDATE). Acestea permit reguli de validare complexe, de calcul automat a valorilor unor atribute, impunerea unor dependenţe funcţionale, realizarea de operaţii de jurnalizare etc.

Sintaxa de bază a unui trigger PL/SQL este [FI]:

CREATE [OR REPLACE] TRIGGER *nume_trigger*
{BEFORE | AFTER}
[INSERT , DELETE, UPDATE]
[FOR EACH ROW [WHEN condiție]]
{CALL procedura} | {bloc PL/SQL}

Opţiunea FOR EACH ROW are următorul rol: dacă se specifică, atunci trigger-ul se declanşează pentru fiecare linie asupra căreia se aplică operaţia respectivă; dacă se omite, trigger-ul se declanşează o singură dată la rularea unei comenzi, indiferent câte linii sunt afectate.

Orice declanşator poate fi de tip BEFORE (se execută înainte de rularea comenzii respective), sau AFTER (se execută după finalizarea comenzii). Declanşatoarele de tip BEFORE sunt potrivite pentru validare sau calculul automat al unor valori (la INSERT de exemplu), iar cele AFTER sunt potrivite pentru jurnalizări (ţinerea evidenţei modificărilor efectuate).

Într-un trigger avem acces la valorile coloanelor din tabele în felul următor:

- vechea valoare a coloanei se accesează cu **:OLD**.nume_coloana
- noua valoare a coloanei se poate seta cu **:NEW**.nume_coloana

Dacă definim un trigger pentru o comandă INSERT, nu vom putea utiliza decât **:NEW**.nume_coloana, deoarece o valoare „OLD" nu există (este NULL), neexistând linia nou inserată la declanşarea trigger-ului. La fel în cazul DELETE, valoarea „NEW" este NULL.

Exemplu de creare a unui trigger:

a) Rulaţi codul din Anexă, pentru crearea tabelelor Secţii şi Angajaţi.

b) Creaţi o secvenţă pentru incrementarea automată a cheii primare, care să înceapă cu valoarea 6:

```
CREATE SEQUENCE angajati_pk START WITH 6;
```

c) Compilaţi codul următorului trigger, care va seta automat, înaintea inserării unei linii în tabela Angajaţi, valoarea cheii primare IDAngajat:

```
CREATE OR REPLACE TRIGGER   angajati_insert
BEFORE INSERT ON  Angajati   FOR EACH ROW
BEGIN
      :NEW.IDAngajat := angajati_pk.NEXTVAL;
 END;
 /
```

d) Inseraţi câteva linii în tabela Angajaţi şi observaţi funcţionarea trigger-ului, afişând conţinutul tabelei Angajaţi. Exemplu de inserare a unei linii:

```
INSERT INTO Angajati (Nume, Prenume) VALUES ('FLOREA', 'Ioana');
```

Exemplu de trigger pentru jurnalizare:

a) Creaţi o tabelă auxiliară, folosită la jurnalizare, care va păstra data modificării salariului unui angajat. Fiecare linie va conţine: IDAngajat, data modificării salariului (data curentă a sistemului), salariul nou şi salariul vechi.

```
CREATE TABLE  ang_audit (
  ang_audit_id     NUMBER(6),
  up_date          DATE,
  sal_nou          NUMBER(5),
  sal_vechi        NUMBER(5) );
```

b) Compilaţi codul următor pentru crearea unui trigger de jurnalizare. Triggerul va fi de tip AFTER şi se va declanşa la fiecare UPDATE a unei valori de coloană dintr-o linie a tabelei Angajaţi.

```
CREATE OR REPLACE TRIGGER audit_sal
AFTER UPDATE OF Salariu ON Angajati FOR EACH ROW
BEGIN
   INSERT INTO ang_audit VALUES( :old.IDAngajat, SYSDATE, :new.Salariu, :old.Salariu );
END;
/
```

c) Schimbaţi salariul pentru un angajat din tabela Angajaţi, de exemplu:

```
UPDATE Angajati SET Salariu=2600 WHERE IDAngajat=4;
```

d) Măriţi salariul cu 10% la toţi angajaţii care au IDAngajat > 1. Observaţi modificările în tabelele Angajaţi şi ang_audit.

Probleme propuse

1) Adăugaţi la tabela Angajaţi o coloană „Salariu_net”, apoi, într-un program PL/SQL, folosind un cursor (explicit sau implicit), setaţi valoarea salariului net pentru toţi angajaţii din tabelă (salariul net este aprox. 70% din salariul brut).
2) Creaţi o procedură stocată cu trei parametri: un parametru de intrare = id_angajat şi doi parametri de ieşire: numar_angajati şi salariu_mediu. Procedura va primi un număr reprezentând un ID de angajat din tabela Angajaţi şi va returna numărul de angajaţi care lucrează în aceeaşi secţie cu acesta, precum şi salariul mediu brut al angajaţilor din secţia respectivă.
3) Creaţi o funcţie stocată care va primi un ID de angajat şi va returna: 1 dacă în tabela Angajaţi există măcar un angajat cu salariul mai mare decât angajatul precizat şi 0 altfel.
4) Creaţi un trigger care, înainte de ştergerea unei linii din tabela Angajaţi, să salveze într-o tabelă Arhivă: numele şi prenumele angajatului, data angajării şi data ştergerii.

Anexa

```
DROP TABLE Angajati;
DROP TABLE Sectii;

CREATE TABLE Sectii (
        IDSectie NUMBER  PRIMARY KEY,
        Nume VARCHAR2(50) NOT NULL,
        Buget NUMBER
);

INSERT INTO Sectii VALUES (1,'Productie',400000);
INSERT INTO Sectii VALUES (2,'Proiectare',300000);
INSERT INTO Sectii VALUES (3,'Cercetare',200000);

CREATE TABLE Angajati (
        IDAngajat NUMBER  PRIMARY KEY,
        Nume VARCHAR2(20)  NOT NULL,
        Prenume VARCHAR2(20)  NOT NULL,
        Datanasterii DATE,
        Adresa VARCHAR2(50),
        Salariu NUMBER  DEFAULT 2800,
        IDSectie NUMBER,
       FOREIGN KEY (IDSectie) REFERENCES Sectii (IDSectie)
  );
INSERT INTO Angajati VALUES (1,'Ionescu','Ion',DATE'1960-01-20','Bucuresti',4000,1);
INSERT INTO Angajati VALUES (2,'Popescu','Petre',DATE'1972-06-21','Bucuresti',3500,1);
INSERT INTO Angajati VALUES (3,'Vasilescu','Ana',DATE'1966-04-02','Bucuresti',3000,2);
INSERT INTO Angajati VALUES (4,'Ionescu','Ion',DATE'1970-11-12','Bucuresti',2500,3);
INSERT INTO Angajati  VALUES (5,'Petrescu','Vasile',DATE'1985-01-25','Iasi',2800,2);
```