PL/SQL avancé

Anne-Cécile Caron

Master MIAGE - BDA

1er trimestre 2010-2011

Objectif

Objectif : Exécution de requêtes SQL dans un programme PL/SQL en limitant le nombre d'échanges entre le moteur PL/SQL et le moteur SQL.

- Modification via un curseur.
- Récupération d'un ensemble de lignes dans une collection
- Instruction `FORALL` pour envoyer au moteur SQL un ensemble d'instructions de modifications.

Requêtes de modification

- Les requêtes de modification des données (`insert`, `delete`, `update`) s'écrivent telles quelles dans le programme.
- Il y a ouverture d'un curseur implicite.
- `SQL%rowcount` donne le nombre de lignes concernées par l'instruction de modification.

```
procedure passer_client(la_caisse caisse.identifiant%type) is
  begin
    delete from client
    where rang = 1 and idCaisse = la_caisse ;
    if SQL%rowcount = 0 then --pas de client a cette caisse
      raise PAS_DE_CLIENT ;
    end if ;
    update client set rang = rang-1
    where idCaisse = la_caisse ;
  end ;
```


Clause `Returning`

- Si l'instruction `insert`, `delete` ou `update` concerne exactement une ligne, on peut récupérer des informations sur la ligne sélectionnée grâce à la clause `Returning ... into ...`.

```
update emp set salary = salary*1.2 where empno = 12
returning dept into emp_dept;

delete from emp where empno = 299
returning first_name, last_name into emp_first_name, emp_last_name;

insert into insert into emp(empno,first_name,last_name)
values (genid.nextval, 'Sophie','Dupont') returning empno into un_id;
```

- Si l'instruction ne concerne aucune ligne, les variables sont indéfinies
- Si l'instruction concerne plusieurs lignes, exception `TOO_MANY_ROWS`

Requêtes d'interrogation

- `select ... into ...` si la requête ramène exactement une ligne (sinon `TOO_MANY_ROWS` ou `NO_DATA_FOUND`)
- Déclaration explicite d'un curseur.
- Syntaxe adaptée
  `for une_ligne in le_curseur loop ... end loop` si on parcourt tout le résultat.

```
function nb_clients(la_caisse caisse.identifiant%type) return NUMBER is
  nb NUMBER ;
begin
  select nbClients into nb
  from Caisse
  where identifiant = la_caisse ;
  return nb ;
exception
  when NO_DATA_FOUND then raise CAISSE_INCONNUE ;
end ;
```


Curseur explicite

```
procedure fermer_caisse(la_caisse caisse.identifiant%type) is
  cursor la_file_attente is
  select * from client where idCaisse = la_caisse ;
begin
  update caisse set ouverte=0 where identifiant = la_caisse;

  for un_client in la_file_attente loop

    dbms_output.put_line('client numero '||un_client.numero);

    update client set idCaisse=null, rang=null
    where numero = un_client.numero ;

    dbms_output.put_line('affectation de '||un_client.numero);
    affectation_client(un_client.numero) ;

  end loop ;
end ;
```


Modification via un curseur

- Dans l'exemple précédent, on exécute un update sur la table `Client`, qui modifie la ligne de la table correspondant à la ligne courante du curseur.
- On a vu lors du TP sur les transactions, qu'un `select ... for update` permettait de poser un verrou sur chaque ligne sélectionnée, ce qui offre la possibilité de modifier ces lignes par la suite.
- Avec un curseur lié à une requête `select ... for update`, on peut modifier (ou supprimer) la ligne courante en y faisant référence grâce à la clause `where current of ...` de l'instruction `update` (ou `delete`).

Modification via un curseur

- `Current of` fait référence à la dernière ligne acquise par un `fetch` ou dans une boucle `for`.
- Attention : on va retrouver les mêmes problèmes que lorsqu'on veut modifier une vue. Il faut que la requête liée au curseur permette la modification.
- Quand la requête porte sur plusieurs tables, il faut préciser la colonne qui va être modifiée : clause `for update of`. En effet, le "current of" fait référence à 1 rowid dans 1 bloc, il y a donc un problème d'ambiguité si la requête porte sur plusieurs tables.

Exemple

```
procedure fermer_caisse(la_caisse caisse.identifiant%type) is
  cursor la_file_attente is
  select * from client where idCaisse = la_caisse
  for update ;
begin
  update caisse set ouverte=0 where identifiant = la_caisse;

  for un_client in la_file_attente loop

    dbms_output.put_line('client numero '||un_client.numero);

    update client set idCaisse=null, rang=null
    where current of la_file_attente ;

    dbms_output.put_line('affectation de '||un_client.numero);
    affectation_client(un_client.numero) ;
  end loop ;
end ;
```


Communication entre PL/SQL et SQL

- PL/SQL envoie des instructions (interrogation ou modification) au moteur SQL.
- On peut améliorer les performances en diminuant le nombre d'échanges entre le moteur PL/SQL et le moteur SQL : *bulk SQL*
  - Récupérer plusieurs ligne dans une collection lors d'une interrogation, en utilisant `bulk collect ... into` dans un `select`.
  - Envoyer plusieurs instructions de modification en 1 seule fois, en utilisant l'instruction `forall`.
- Nécessité de manipuler des collections.

Les collections en PL/SQL

- Tableau associatif : association clé-valeur. (appelé `associative array`, ou `index-by table`) Ils servent dans les programmes PL/SQL là où on utilise habituellement une table de hashage.
- Table imbriquée : ensemble de valeurs, table de taille quelconque. (appelée `nested table`)
- Tableau (appelé `varray`).

Pour toutes ces collections, on peut utiliser les méthodes :

- `FIRST`, `LAST` qui donnent le premier et le dernier indice valide.
- `COUNT` qui donne le nombre effectif d'éléments
- `NEXT` et `PRIOR` qui donnent l'indice suivant et précédent d'un indice passé en paramètre.
- `EXISTS` qui prend un indice en paramètre et indique s'il existe un élément à cet indice

Tableaux associatifs

- Recherche à partir d'une clé qui peut être un nombre positif ou négatif, ou une chaîne de caractères.
- Les valeurs des clés ne se suivent pas forcément.
- Ces tables ne peuvent pas typer une colonne d'une table.
- on ajoute un élément par une simple affectation `T(nouvelle_clé) := valeur`
- on supprime une entrée (clé,valeur) par la méthode `T.delete(clé)`

Exemple

```
DECLARE
  TYPE population_type IS TABLE OF NUMBER INDEX BY VARCHAR2(64);
  continent_population population_type;
  howmany NUMBER;
  which VARCHAR2(64);
BEGIN
  continent_population('Australia') := 30000000;
  -- Looks up value associated with a string
  howmany := continent_population('Australia');
  continent_population('Antarctica') := 1000; -- Creates new entry
  continent_population('Antarctica') := 1001; -- Replacement
-- Returns 'Antarctica' as that comes first alphabetically.
  which := continent_population.FIRST;
-- Returns 'Australia' as that comes last alphabetically.
  which := continent_population.LAST;
END;
```


Tables imbriquées

- Utilisées pour stocker une sous-table dans une colonne d'une table (relationnel étendu).
- La taille n'est pas fixée à l'avance, et augmente dynamiquement lorsqu'on ajoute des éléments.
- Les indices ne se suivent pas forcément (à cause des `delete` qui peuvent laisser des trous) → utiliser `next` ou `prior` dans ce cas, pour parcourir les éléments de la table.

```
declare
  type tennismen_table is table of varchar2(15);
  -- initialisation avec un littéral
  les_joueurs tennismen_table := tennismen_table('Nadal','Federer','Monfils');
begin
  for i in les_joueurs.first .. les_joueurs.last loop
    dbms_output.put_line(les_joueurs(i)) ;
  end loop;
end ;
```

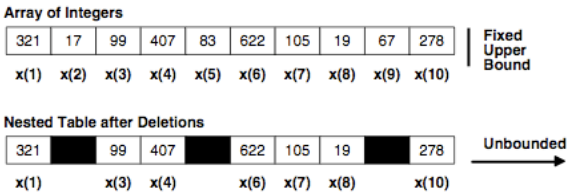

Exemple

```
declare
  type numlist is table of number;
  n numlist := numlist(1,3,5,7);
  counter integer ;
begin
  n.delete(2) ; -- supprime le 2nd élément
  counter := n.first ; -- vaut 1
  while counter is not null loop
    dbms_output.put_line(n(counter));
    counter := n.next(counter); -- le suivant de 1 vaut 3
    -- le suivant de 4 vaut null
  end loop;
end ;
```


Tableaux "classiques"

- taille maximale fixée à la déclaration (`t.limit` donne cette taille),
- les indices se suivent toujours.
- Peuvent typer une colonne d'une table en relationnel étendu.
- Pas de méthode `delete`

Figure 5–1 Array versus Nested Table



Exemple

```
declare
  type tab_entiers is varray(10) of integer ;
  mon_tab  tab_entier ;
begin
  mon_tab := tab_entier(4,10,7,3);
  -- mon_tab.count vaut 4, mon_tab.limit vaut 10
  -- on ajoute un 5e élément :
  mon_tab.extend(1); -- on "agrandit" le tableau
  mon_tab(5) := 23 ;
end ;
```


Les exceptions

- `COLLECTION_IS_NULL` : la variable collection vaut `null`
- `NO_DATA_FOUND` : l'indice désigne un élément supprimé d'une table imbriquée ou bien n'est pas une clé dans une table d'association.
- `SUBSCRIPT_BEYOND_COUNT` : l'indice est plus grand que le nombre d'éléments
- `SUBSCRIPT_OUTSIDE_LIMIT` : l'indice est en dehors de l'intervalle autorisé
- `VALUE_ERROR` : l'indice est `null` ou ne peut pas être converti dans le type de la clé, ou la valeur que l'on veut ranger dans la collection n'est pas du bon type.

`bulk collect ... into`

en anglais, "in bulk" = en vrac, en gros

- L'instruction `select ... into ...` permet de récupérer 1 ligne
- L'instruction `select ... bulk collect into ...` permet de récupérer un ensemble de lignes dans une variable collection. Si cette collection est un tableau (`varrays`), il doit être assez grand pour contenir toutes les lignes.
- Attention, `select c1, c2 bulk collect into t1, t2 ...` signifie que `t1` (resp. `t2`) est une collection d'éléments du même type que `c1` (resp. `c2`).
- plus généralement, le `bulk collect into` peut être utilisé pour les instructions suivantes :
  - `select ... into` (requête d'interrogation)
  - `fetch ... into` (traitement d'un curseur)
  - `returning ... into` (requête de modification)

Exemple 1

```
declare
  type emptab_type is table of emp%rowtype ;
  emptab emptab_type ;
begin
  select * bulk collect into emptab from emp where dept = 45 ;
  -- emptab.count = 0 si aucune ligne sélectionnée
  for i in emptab.first .. emptab.last loop
     dbms_output.put_line(emptab(i).first_name, emptab(i).last_name);
  end loop;
end ;
```

Exemple 2

```
declare
  type nomtab_type is table of emp.last_name%type ;
  nomtab nomtab_type;
  type salairetab_type is table of emp.salary%type ;
  salairetab salairetab_type;
begin
  select last_name,salary bulk collect into nomtab, salairetab
  from emp where dept = 45 ;
  for i in nomtab.first .. nomtab.last loop
     dbms_output.put_line(nomtab(i) || ' ' || salairetab(i));
  end loop;
end ;
```


Exemple de curseur

```
DECLARE
  TYPE DeptRecTab IS TABLE OF departments%ROWTYPE;
  dept_recs DeptRecTab;
  CURSOR c1 IS
    SELECT department_id, department_name, manager_id, location_id
      FROM departments WHERE department_id > 70;
      -- le select conserve toutes les colonnes de departements
BEGIN
  OPEN c1;
  FETCH c1 BULK COLLECT INTO dept_recs;
  -- il faut ensuite parcourir la collection dept_recs
END;
```


Gestion de la mémoire

- Le `bulk collect` permet de récupérer tout le résultat d'un `select` en une seule fois
- On gagne du temps, mais ça peut poser des problèmes de mémoire :
  - Les collections sont stockées dans la PGA (process global area), mémoire non partagée, alors que les résultats de requêtes sont stockés dans la SGA (system global area) mémoire partagée.
  - Le résultat d'une requête ne tient pas forcément dans la PGA
- On peut utiliser la clause `LIMIT` pour limiter le nombre de lignes que l'on veut acquérir à chaque `FETCH`

Exemple

```
declare
   TYPE numtab IS TABLE OF employees.employee_id%type ;
   CURSOR c1 IS
   SELECT employee_id FROM employees WHERE department_id = 80;
   empids    numtab;
begin
  open c1;
  loop -- à chaque itération, on lit 10 lignes (ou moins la dernière fois)
    FETCH c1 BULK COLLECT INTO empids LIMIT 10;
    EXIT WHEN empids.COUNT = 0;
--  exit when c1%NOTFOUND; -- incorrect, on peut perdre les dernières lignes
    -- on traite la collection lue
    for i in 1..empids.count loop
      dbms_output.put_line( 'Employee Id: ' || empids(i));
    end loop;
  end loop;
  close c1;
end;
```

forall

- Cette instruction permet d'exécuter plus efficacement des instructions SQL de modifications effectuées dans une boucle `for`
- Chaque itération de la boucle `for` doit exécuter une instruction SQL qui utilise les valeurs d'une collection
- Généralement, les indices de la collection sont consécutifs dans un intervalle `inf .. sup`. Si les indices ne sont pas consécutifs (e.g. dans une table imbriquée après suppression de données), on peut utiliser la clause `INDICES OF`
- Pour connaître le nombre de lignes affectées par la ième itération, on utilise `%BULK_ROWCOUNT`.

Exemple

```
declare
   TYPE NumList IS VARRAY(20) OF NUMBER;
   depts NumList := NumList(10, 30, 70);  -- department numbers
begin
   FORALL i IN depts.FIRST..depts.LAST
      delete from employees_temp where department_id = depts(i);

-- au lieu de :
   FOR i IN depts.FIRST..depts.LAST loop
      delete from employees_temp where department_id = depts(i);
    end loop ;
end ;
```


Exemple

```
declare
   TYPE cust_typ IS TABLE OF customer.cust_name%type ;
    cust_tab cust_typ ;
begin
-- on suppose qu'on a ajouté des éléments dans cust_tab,
-- puis supprimé certains d'entre-eux

FORALL i IN INDICES OF cust_tab
     INSERT INTO customer(cust_name) VALUES(cust_tab(i));

-- au lieu de
FOR i IN 1..cust_tab.LAST LOOP
     IF cust_tab.EXISTS(i) THEN
       INSERT INTO customer(cust_name) VALUES(cust_tab(i));
     END IF;
END LOOP;
```


Exemple avec BULK_ROWCOUNT

```
declare
   TYPE NumList IS TABLE OF NUMBER;
   depts NumList := NumList(30, 50, 60);
begin
   FORALL j IN depts.FIRST..depts.LAST
      DELETE FROM emp_temp WHERE department_id = depts(j);

-- combien de lignes sont affectées par chaque DELETE ?
   for i IN depts.FIRST..depts.LAST
  loop
      dbms_output.put_line('Iteration #' || i || ' deleted ' ||
         SQL%BULK_ROWCOUNT(i) || ' rows.');
   end loop;
end;
```

Traitement d'exception

- par défaut, lorsqu'une instruction de modification parmi celles exécutées dans un `forall` déclenche une exception, les instructions suivantes ne sont pas exécutées.
- On peut choisir de continuer l'exécution et de mémoriser les messages d'erreurs au fur et à mesure (une exception est finalement déclenchée à la fin du `forall`)
  - utiliser la clause `SAVE EXCEPTION` du `forall`
  - l'attribut `%BULK_EXCEPTIONS` est une collection de records, avec deux composantes : `ERROR_INDEX` (indice de l'instruction qui a déclenché une erreur) et `ERROR_CODE` (le code d'erreur Oracle).

Exemple

```
DECLARE
   TYPE empid_tab IS TABLE OF employees.employee_id%TYPE;
   emp_sr empid_tab;
-- create an exception handler for ORA-24381
   dml_errors EXCEPTION;
   PRAGMA EXCEPTION_INIT(dml_errors, -24381);
BEGIN
    -- on suppose qu'on a initialisé la table emp_sr
     FORALL i IN emp_sr.FIRST..emp_sr.LAST SAVE EXCEPTIONS
       update emp_temp set job_id = job_id || '_SR'
          where emp_sr(i) = emp_temp.employee_id;
EXCEPTION
  WHEN dml_errors THEN -- Now we figure out what failed and why.
   FOR i IN 1..SQL%BULK_EXCEPTIONS.COUNT LOOP
      dbms_output.put_line('iteration ' || SQL%BULK_EXCEPTIONS(i).ERROR_INDEX);
      dbms_output.put_line('Error message is ' ||
          sqlerrm(-SQL%BULK_EXCEPTIONS(i).ERROR_CODE));
          -- fct qui attend un nombre négatif
   END LOOP;
END;
```