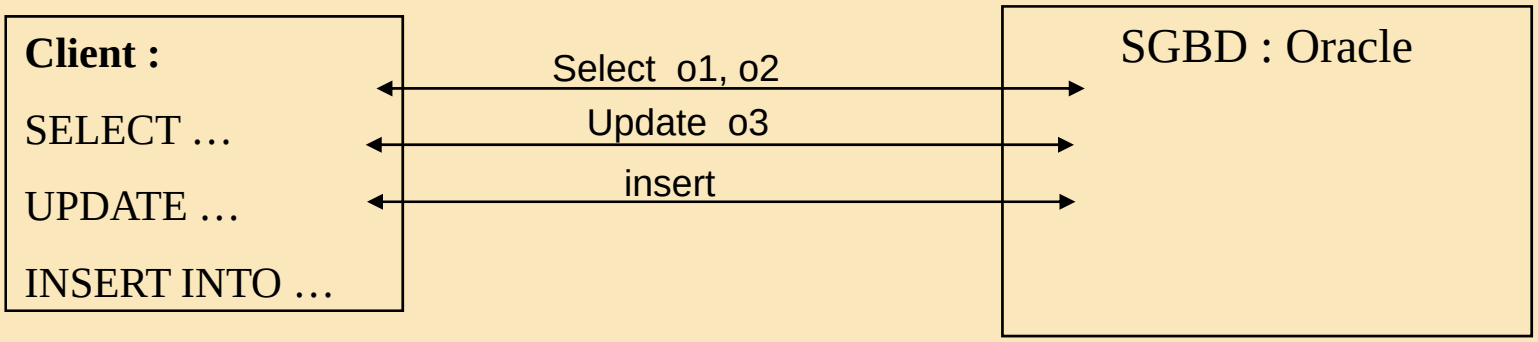
Rappel sur le PL/SQL et les packages

André Gamache, professeur associé
Département d'informatique et de génie logiciel
Faculté des sciences et de génie
Université Laval. Québec, Qc, Canada, G1K 7P4

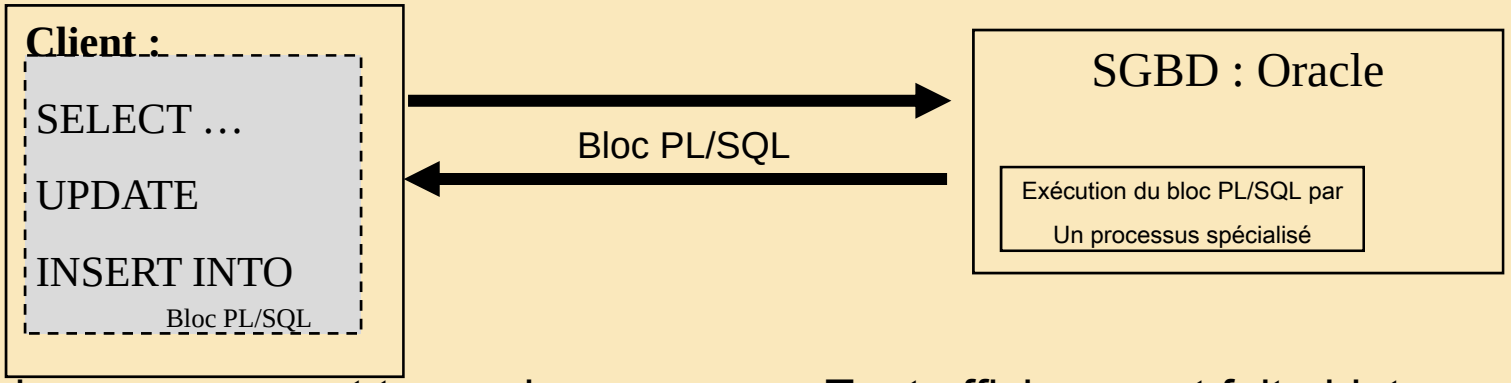
Courriel: andre.gamache@ift.ulaval.ca

Interactions entre une application et le serveur Oracle

Exécution : requête et DML dans un L3G



Exécution : bloc avec requête et DML



Le bloc anonyme est transmis au serveur. Tout affichage est fait objet par objet reçu du serveur.

SQL*PLUS

- SQL*PLUS est une application qui permet de formuler des clauses SQL et des blocs PL/SQL, de les transmettre au serveur pour leur vérification et le calcul de leur réponse. La réponse est retransmise par le serveur, objet par objet et afficher par l'utilitaire SQL*Plus avec un minimum de format!

- Syntaxe de terminaison:

  - Le ; à la fin d'une clause SQL déclenche la transmission, la vérification symtaxique et sémantique et l'exécution d'une clause SQL;

  - Le / du bloc PL/SQL marque sa fin et sa transmission au serveur pour vérification et exécution. Il apparaît obligatoirement en début de ligne.

    Toute erreur (ou presque) est visible avec la directive : SHOW ERRORS ?? Peu bavard cependant ??

Variables de substitution

- **&v et &&V**

  Une variable de substitution est valuée à l'exécution en sollicitant le terminal d'input (utilisateur)

  L'usage de &&V évite de solliciter à nouveau l'utilisateur en se référant à la dernière valeur de la variable de substitution dans le même bloc.

  Exemple:

```
Select &attrib
From Usine
Where &&attrib = 100;
```

Autres directives SQL/PLUS et non des clauses DDL

- SET SERVEROUTPUT ON : permet à une proc ou une fonction d'écrire dans le tampon (*pipe*): DBMS_OUTPUT.Put_Line('Bonjour ')
- SET CLEAR SCREEN : pour effacer l'écran et ramener le curseur au début
- SET PAUSE ON
- SHOW ALL -- pour lister toutes les directives et la valeurs pour la session de SQL/PLUS
- DEFINE_EDITOR = Word : pour associer un éditeur de texte
- Spool fichier.txt | OFF | OUT
- SET PAGESIZE 50 : définir la longueur d'une page en nb de lignes
- SET TERMOUT OFF | ON

Rappel PL/SQL

- Avantage : réduire le trafic réseau (par opposition aux ordres SQL) et faire un traitement avec un langage *computationnel* complet.

- N'est pas un L3G car n'accepte pas des données en interaction avec l'utilisateur: pas de input.

- Quatre sections dans un bloc nommé PL/SQL :
  - Signature : nom, liste des paramètres typés et le type de la réponse
  - DECLARE (facultatif) pour les déclarations des variables et des curseurs
  - BEGIN - END, avec toutes les instructions SQL et PL/SQL ;
  - EXCEPTION (facultatif) pour traiter les erreurs signalées par SGBD.

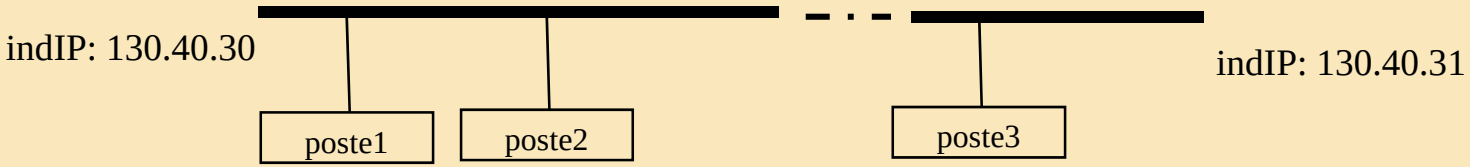
- Possibilité d'imbrication les blocs: bloc et sous-bloc: un bloc voit les variables de ses sous-blocs et non l'inverse.

- Un sous-bloc commence par BEGIN et finit par END et à sa sortie la suite se fait avec le bloc englobant ou s'il est absent c'es la fin de l'exécution.

Caractéristiques syntaxiques de PL/SQL

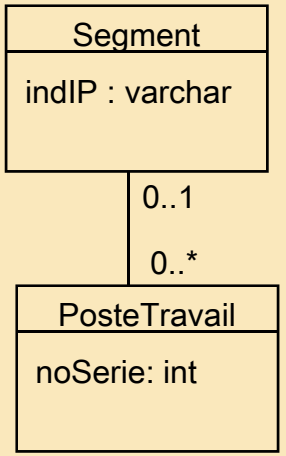
- Identifiant : Débute par une lettre; longueur max 30 car.
- Mots réservés : BEGIN, END; SELECT, …
- Littéral : valeur primitive typée : ex. 45 , 45.60, 'Oracle'
- Commentaire : de ligne : -- ou multiligne : /* … */
- Une constante de chaîne est formulée avec ' … ' ou " … " s'il faut distinguer les majuscules et les minuscules.

Schéma de la table PosteTravail (avec segment (1-*))



Segment:

indIP*	nomSeg	longueurS
130.40.30	ICARE	25
130.40.31	DEDALE	75



PosteTravail:

noSerie*	adrIP	typePoste	diskRestant	noSeg
p1	01	WIN95	950	130.40.30
P2	02	WIN95	875	130.40.30
P3	02	WINNT	400	130.40.30
p4	01	TX	980	130.40.31

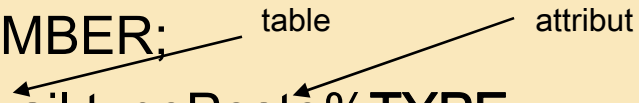
Création de la table objet PosteTravail

- CREATE OR REPLACE TYPE PosteTravail_t AS OBJECT (
  noSerie int,
  adrIP CHAR(2),
  typePoste CHAR(10),
  diskRestant NUMBER,
  noSeg varchar(9))
  /

- CREATE TABLE PosteTravail OF posteTravail_t;

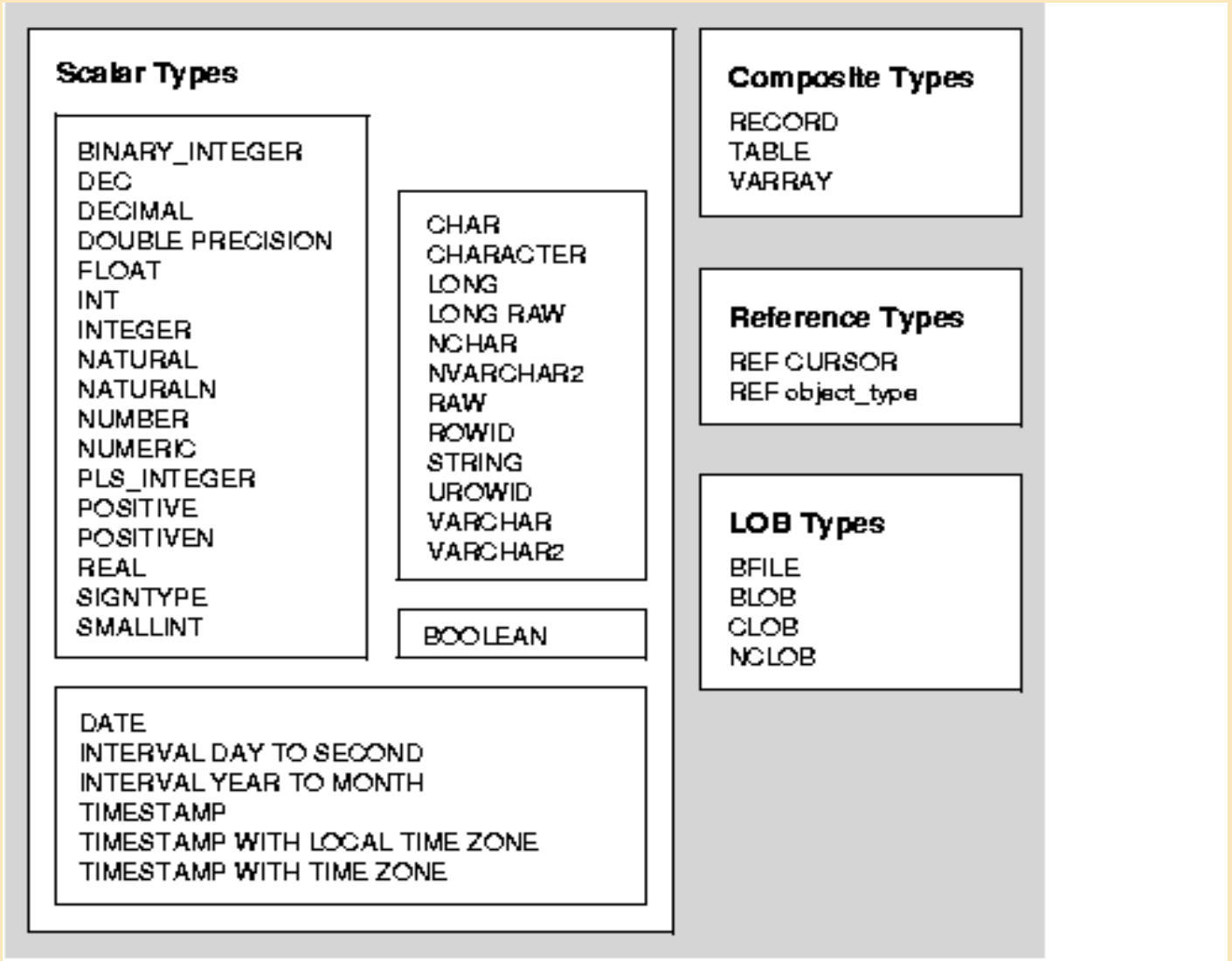
Exemple PL/SQL (suite)

```
--Bloc PL/SQL
DECLARE
    espaceDispo NUMBER;
    poste   PosteTravail.typePoste%TYPE;
BEGIN
    poste := 'Win98';
    SELECT SUM(diskRestant) INTO espaceDispo FROM PosteTravail
       WHERE typePoste = poste;
    INSERT INTO Table_de_sortie VALUES('IL reste' ||
       TO_CHAR(espaceDispo) || 'Mega pour les postes de type' || typePoste);
    DBMS_OUTPUT.PUT_LINE('espace libre :' ||TO_CHAR(espaceDispo)  ||
       "Mo pour postes de " || typePoste);
END;
/
```



Types atomiques de PL/SQL

Les types sont :



Conversion implicite

Plusieurs conversions sont implicites lorsqu'elles sont possibles et logiques.

Par exemple la chaîne '123' est implicitement transformée en type entier 123, tandis que la chaîne '12C' ne peut pas l'être et la transformation génère une erreur et la sortie du bloc.

Structures de contrôle : *PL/SQL computationnel complet*

- Structure alternative (IF *cond* THEN. ..ELSE ; ou ELSIF. ..END IF;)
- CASE *selector* WHEN *expression* THEN énoncé(s);

- Structures de répétition possibles sont :

  FOR condition LOOP... END LOOP;
  LOOP...EXIT WHEN condition END LOOP;
  WHILE (condition) LOOP... END LOOP;
  FOR variable curseur IN curseur LOOP. ..END LOOP;
  FOR variable curseur IN (select nomP from Client) LOOP. ..END LOOP;

- 4 sortes d' exception:
  1- Les erreurs internes de SQL prédéfinies par Oracle (NO_DATA_FOUND, DUP_VAL_ON_INDEX. ..),
  2- Les erreurs internes de SQL non prédéfinies par Oracle (NOT NULL. ..)
     OTHERS : exceptions pour effectuer des traitements quand une erreur se produit et qu'elle n'a pas été prise en compte par le programme.
  3- Les erreurs définies par l'utilisateur.

Alternative : IF … THEN …

- Le contrôle IF…THEN

Le scéhma de la table Employe:

Employe (noE int, nomE varchar(50), tauxcharge number(2,2), charge number (6,2), salaire number (8,2))

```
SQL>      Declare
                sal int;
                stat int;
               Begin
                select x.salaire into sal From Employe x
                Where x.matE = 567;
               IF sal > 4000.00 THEN stat := 1;
               End IF;
          End;
          /
```

Avec plusieurs énoncés dans la partie THEN : il faut utiliser un bloc

IF .. THEN … ELSE …END IF

- Le contrôle à deux sorties:

  Insert into employe values (100, 'Dios', 0.10, 500.00, 1500.00) et Set serveroutput on;

```
Declare
 v_employe  EMPLOYE%ROWTYPE;
Begin
     Select * into v_employe From Employe
     Where noE = 100;                          /*doit retourner 1 tuple seulement  */
 IF v_employe.salaire > 25000.00
     THEN Begin
      v_employe.charge := v_employe.charge * .5;
     v_employe.tauxCharge := 0.5;
     End;
 ELSE Begin
      DBMS_OUTPUT.put_line ('charge cadre' || '    ' || v_employe.charge);
     v_employe.tauxCharge := 0.75;
END;
     END IF;
Exception
When No_Data_Found Then DBMS_OUTPUT.put_line (‘La table_vide');
End;
```

```
charge cadre    500

Procédure PL/SQL terminée avec succès.
```

Structure itérative

- LOOP

```
[<<nom_boucke>>]
LOOP
    énoncés …
    EXIT [nom_boucle] WHEN  condition;
End Loop [nom_boucle];
```

```
Declare
     v_compteur1 number (6,0)  := 0;
     Begin
    LOOP
          v_compteur1 := v_compteur1 +1;
          DBMS_OUTPUT.put_line ( v_compteur1);
    EXIT WHEN v_compteur1 > 5;
    End Loop;
    End;
    /
```

Structure intérative WHILE

```
<<nom_boucle>> WHILE Condition LOOP
                    bloc
                    END LOOP  nom_boucle;
Declare
     v_compteur1 number (6,0)  := 0;
     Begin
    <<B1>>  WHILE v_compteur1 < 5 Loop
          v_compteur1 := v_compteur1 +1;
         DBMS_OUTPUT.put_line ( v_compteur1);
         End Loop B1;
   End;
   /
```

Structure itérative FOR .. IN

```
[<<nom_boucle>>]
    FOR indice J IN [REVERSE] expr1..expr2  LOOP
            bloc
    END LOOP;

DECLARE
Somme int :=0;
Begin
    FOR J IN 1..5 LOOP
            Somme := Somme + J;
            END LOOP;
    DBMS_OUTPUT.put_line ('La somme des indices :  ' || Somme);
    END;
```

Exemple de traitement des erreurs (exceptions)

```
DECLARE
      espaceDispo   NUMBER;
      poste     PosteTravail.typePoste%TYPE;
      ESPACE_INSUF    EXCEPTION;
BEGIN
     poste := 'Win98';
      SELECT SUM(diskRestant) INTO espaceDispo FROM PosteTravail
      WHERE typePoste = poste;
      IF espaceDispo < 900 THEN   RAISE ESPACE_INSUF ;
      END IF ;
      DBMS_OUTPUT.PUT_LINE('Il reste ' || TO_CHAR(espaceDispo) ||' Mega pour les
          postes de type' || poste);
EXCEPTION
      WHEN NO_DATA_FOUND THEN INSERT INTO exemple_blocPLSQL VALUES ('Il
          n''existe aucun poste du type: ' || poste);
      WHEN ESPACE_INSUF THEN DBMS_OUTPUT.PUT_LINE ( 'espace disque
          insuffisant dans le réseau');
END;
```

Raise_Application_Error()

- ATTENTION :
- La procédure **Raise_Application_Error** (entier_négatif, message) est un appel à une procédure du package DBMS_STANDARD du serveur et rendue disponible par Oracle, permettant d'activer une exception pré déclarée, de retourner un message d'erreur à l'environnement et faire un rollback des transactions actives.
- Si l'environnement est celui du module SQL*Plus, l'exécution de la clause SQL est terminée et le message d'erreur est affiché à l'écran.
- L'entier négatif doit être choisi dans l'intervalle –20000 et –20500.
- TRIGGER : Si la procédure est appelée par un trigger de BD, ce dernier se termine avec un rollback de l'action qui l'a déclenché et le message est aussi passé à l'environnement.

Type RECORD

- Le type RECORD permet de travailler avec des structures complexes définies dans un bloc PL/SQL. Le record est une structure transitoire dont le type de ses composants peut être spécifié au regard d'une table dont le schéma est dans le DD.

- Le RECORD peut inclure des attributs de types REF, NESTED TABLE OU VARRAY ou bien des attributs multimédias : BLOB, CLOB et BFILE.

```
DECLARE
type PosteWin98_r  IS RECORD (noSerie int,  adrIP CHAR(2),
                  typePoste CHAR(10) := 'Win98' , diskRestant NUMBER);
un_poste_98   posteWin98_r;
mon_poste     posteWin98_r;
BEGIN
      un_poste_98.noSerie := 5;
      un_poste_98.adrIP := 2;
      un_poste_98.diskRestant := 200;
      mon_poste := un_poste_98;  -- affectation de record à record
END;
```

Varray

- Structure de collection similaire à celle de la table PL/SQL pouvant stocker des éléments de même type.

- Structure avec une borne inférieure de 1 et une supérieure n qui est extensible.

Syntaxe pour la création d'un varray(n):

TYPE *nom_table* IS VARRAY(10) OF Employe.salaire%TYPE;

L'élément de la table *nom_table* est un salaire dont le type est celui de salaire de la table Employe.

À la création d'un varray aucune cellule n'est créée. Il faut les créer au fur à mesure du stockage des éléments par la méthode extend: tableNom.Extend

Bloc PL/SQL pour stocker des tuples dans une structure Varray

```
Declare
type empl_t is Varray(10) of Employe.nom%type;
tableNom  empl_t  := empl_t();
Cursor ouvriers is Select nom from Employe Where rownum <= 10;
indice int :=0;
Begin
      dbms_output.Put_line (indice);
      For n IN ouvriers LOOP
      indice  := indice + 1;
      tableNom.Extend;    -- création d'une première cellule du varray.
      Dbms_output.Put_line (indice);
      tableNom(indice) := n.nom;
      Dbms_output.put_line (tableNom(indice));
      End loop;
End;
```

Bloc PL/SQL pour stocker des valeurs dans une structure Varray

```
Declare
type matricule_t is Varray(10) of int;
tableMatricule  matricule_t   := matricule_t() ;
Begin
      For n IN 1..10 LOOP
          tableMatricule.Extend;    -- création d'une première cellule du varray.
      tableMatricule(n) := n;
       Dbms_Output.put_line (tableMatricule(n));
       End loop;
End;
```

CURSEUR

Deux sortes de curseur : explicite (le plus performant) et implicite

implicite (avec le mot clé INTO), et explicite (défini dans la zone de déclaration d'un bloc);

Curseur Oracle:

Espace mémoire réservé au niveau du serveur pour ranger les tuples (objets) et des informations au sujet de l'exécution de l'ordre et de donner un nom au curseur (explicite).

Un programme PL/SQL doit faire :

déclaration du curseur

ouverture du curseur

traitement des tuples (objets) de la réponse

fermeture du curseurs pour libérer l'espace et les objets traités.

Déclaration de curseur

CURSOR *nom_curseur* [(param type := valeur_int)] IS *requête_SQL* ;

La requête peut contenir des ordres SQL et les opérateurs Union, Intersect ou Minus.

Exemple:

```
SQL> Declare
    CURSOR  reponse1   IS
                    Select noE, nomE, salaire From Employe
                    ORDER BY nomE;

OPEN reponse1;   /* calcul de la réponse */
Fetch reponse1 into ….;  -- récupération d'un tuple de la réponse
Close reponse1;  /* fermeture du curseur et libération des données  */
```

Curseur (*Result Set*)

- Le curseur (CURSOR) permet de calculer et d'accéder au résultat d'une requête, d'une mise à jour, … avec 1 ou plusieurs objets dans un ensemble réponse qui est stockée sur le serveur.

- Utile pour effectuer un traitement séquentiel, record par record ou objet par objet.

- Il est possible de tester la valeur du curseur a tout moment avec les attributs du curseur : *syntaxe nonCurseur.attribut*

  - %FOUND : TRUE quand un tuple ou objet est retourné après un Fetch, FALSE dans les autres cas.
  - %NOTFOUND : TRUE si aucun objet n'est pas retourné avec un Fetch, FALSE dans les autres cas.
  - %ISOPEN qui retourne TRUE si le curseur a été ouvert, FALSE sinon FALSE.
  - %ROWCOUNT qui retourne un entier : zéro si le curseur est ouvert mais n'a pas été lu, puis la valeur s' incrémente de 1 à chaque lecture.

Exemple de curseur avec ses attributs

```
SQL>DECLARE
      CURSOR posteWin98 is SELECT noSerie, diskRestant FROM PosteTravail
          WHERE typePoste = 'Win98';
      espaceDispo NUMBER  :=0 ;
      nbRow NUMBER;
      n_poste     PosteTravail.noSerie%TYPE;
      espaceDisque    PosteTravail.diskRestant%TYPE;
BEGIN
      OPEN postesWin98;
      FETCH postesWin98 INTO n_poste, espaceDisque;
      nbrow : = postesWin98%ROWCOUNT;  --   1
      WHILE postesWin98%FOUND  LOOP
            espaceDispo := espaceDispo + espaceDisque;
            INSERT INTO exemple_curseur VALUES('Numero ' || n_poste || ' Disque ' ||
               TO_CHAR(espace_disque) || ' Cumul ' || TO_CHAR(espaceDispo) || 'Nombre tuples' ||
               TO_CHAR(nbrow) ) ;
            FETCH postesWin98 INTO n_poste, espaceDisque;
            nbrow := postesWin98%ROWCOUNT;
      END LOOP;
      CLOSE postesWin98;
END;
```

Utilisation de l'attribut de curseur %ROWTYPE

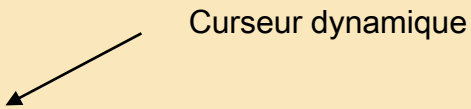
```
SQL> DECLARE
      CURSOR postesWin98 IS SELECT noSerie, diskRestant
          FROM PostTravail WHERE typePoste = 'Win98';
      EspaceDispo NUMBER;  nbRow NUMBER;
      curs_postesWin98   postesWin98%ROWTYPE;
BEGIN
      EspaceDispo := a;       -- à lire
      FOR curs_postesWin98 IN postesWin98  LOOP
            nbRow := postesWin98%ROWCOUNT;
            espaceDispon := espace_disponible  + curs_postesWin98.diskRestant;
            INSERT INTO Exemple_curseur VALUES('Numero' || c_postes_Win98.noSerie
               || Disque || TO_CHAR(curs_postesWin98.diskrestant) || 'Cumul' ||
               TO_CHAR(espacedisponible) || 'Nombre tuples ' || TO_CHAR(nbr_row));
      END LOOP;
END;
```

Ref cursor (dynamique)

- Un curseur dynamique n'est pas lié à une requête comme pour le curseur statique.

  Une variable de type curseur permet au curseur d'évoluer au cours du programme en lui associant diverses clauses SQL

- DECLARE




```
TYPE refPosteWin98   IS  REF CURSOR ;
desPostesWin98   refPosteWin98;
n_poste      PosteTravail.noSerie%TYPE;
…
```

(suite …au prochain écran)

Suite exemple de Ref cursor

```
BEGIN
    OPEN desPostesWin98  FOR
                SELECT noSerie FROM PosteTravail
                WHERE typePoste = 'Win98';  -- exécution du curseur

    FETCH desPostesWin98  INTO n_poste;
    WHILE desPostesWin98%FOUND LOOP
        DBMS_OUTPUT.PUT_LINE ('Numero ' || n_poste);
        FETCH desPostesWin98 INTO n_poste;
    END LOOP ;
    CLOSE desPostesWin98;  -- la var. curseur redevient disponible
END;
```

Exemple de curseur avec Return

```
DECLARE
    TYPE refPostesWin95 IS REF CURSOR  RETURN PosteTravail%ROWTYPE;
    desPostesWin95   refPostesWin95;  -- curseur
    records_poste     PosteTravail%ROWTYPE;
BEGIN
    OPEN desPostesWin95 FOR SELECT * FROM PosteTravail WHERE typePoste =
       'Win95';
    LOOP
        FETCH desPostesWin95 INTO records_poste;
        EXIT WHEN (desPostesWin95%NOTFOUND);
        INSERT INTO Exemple_curseur VALUES('Numero ' || records_poste.noSerie);
    END LOOP;
    CLOSE desPostesWin95;
END;
```

Manipulation d'un objet avec une variable PL/SQL

Contenu de la table Personne:

MAT	ADRESSE(NO, RUE)
125	ADR_T(1, 'a')

Manipulation d'un objet avec une variable PL/SQL du même type dans un bloc anonyme:

```
Declare
pers personne_t;
begin
select Value(p) into pers from Personne p Where p.mat = 100;
DBMS_OUTPUT.PUT_LINE (pers.mat);
end;
```

```
SQL> run
125
PL/SQL procedure successfully completed.
```

Fonctions et procédures cataloguées

- Les fonctions et les procédures cataloguées sont des programmes PL/SQL qui sont compilés et stockés dans le dictionnaire du SGBD sans être associées à une classe précise du schéma objet ou relationnel.

- Ce sont des blocs PL/SQL **nommés** permettant d'accéder à la base d'objets.

- Recompilation automatique:

  Lors d'un appel de fonction ou de procédure, il y a recompilation du programme que si un objet du dictionnaire référencé dans le code a été modifié. Après, l'exécutable est chargé en mémoire.

Fonctions et procédures cataloguées : avantages

- Sécurité: les droits d'accès portent aussi sur des programmes stockés. Ces droits sont délégués par l'instruction GRANT ,

  ex: GRANT EXECUTE ON p1 TO johanne; -- p1 est une procédure cataloguée

- Intégrité : les traitements sont exécutés dans le même bloc transmis au serveur de la BD (utilisation possible de COMMIT, ROLLBACK. ..) ;

- Performance: réduction du nombre d'appels Client/Serveur à la BD et partage d'un programme;

- Productivité : simplicité de la maintenance des applications par modularité, extensibilité et réutilisabilité, notamment avec les packages.

- Attention: Les procédures et les fonctions ne sont pas explicitement associées aux classes du schéma

Exemples de fonction et procédure cataloguées (dans le DD)

```
CREATE OR REPLACE FUNCTION DispoEsp (typeP IN CHAR) RETURN NUMBER IS
espaceDispo NUMBER;
BEGIN
SELECT SUM (diskRestant) INTO espaceDispo FROM PosteTravail WHERE typePoste
    = typeP;
RETURN espaceDispo;
END;
```

Il peut y avoir aussi des procédures sans valeur de retour.

```
CREATE OR REPLACE PROCEDURE ModifieIP(noserieP IN CHAR, nouveauIP IN
    varchar) IS
BEGIN
UPDATE PosteTravail SET adrIP = nouveauIP WHERE noSerie = noserieP;
END;
```

- Au besoin, les paramètres de sortie d'une proc ont un mode OUT, IN et IN OUT

Appel de fonction et procédure PL/SQL dans un bloc

```
DECLARE
      posteP   PosteTravail.typePoste%TYPE;
      posteModif   PosteTravail.noSerie%TYPE;   -- no serie du poste
BEGIN
      posteP := 'Win98';
      IF DispoEsp (posteP )< 900 THEN
      INSERT INTO ExempleProg VALUES('Moins de 900 MegaO pour les postes ' ||
          posteP);
      ELSE
      INSERT INTO ExempleProg VALUES ('Plus de 900 MegaO pour les postes ' ||
          posteP) ;
      END IF;
      posteP : = 'p1' ;
      ModifieIP(posteModif, '120.4');  -- procédure sans valeur retournée
END;
```

Spécifications possibles des paramètres: Formels et actuels (appel)

```
DECLARE
      posteP    PosteTravail.typeposte%TYPE;
      posteModif     PosteTravail.noSerie%TYPE;  -- var. du type de noSerie
BEGIN
      posteModif := 'p1';

      --Notation nommée : paramètre spécifié
      ModifieIP(typeP => posteModif,  nouveauIP => '120.4') ;
      ModifieIP(nouveauIP => '120.4', typeP => posteModif);  -- param non ordonné

      --Notation positionnelle
      ModifieIP(posteModif, '120.4');

      --Notation mixte
      ModifieIP(posteModif, nouveauIP => '01');
END;
```

Package

- Un package est un container de procédures, de fonctions et/ou de variables globales [et éventuellement de packages].

- Un package peut comprendre 4 sections: signature (spécification), déclaration, bloc exécutable et les exceptions.

- Un package permet de déclarer et d'implémenter des objets publics ou privés et de simuler, au besoin, le comportement de méthodes.

```
--spécification du package   -- par les signatures seulement
CREATE OR REPLACE PACKAGE ActionsPoste AS
    FUNCTION espaceDispo(typeP  IN CHAR) RETURN NUMBER;
    PROCEDURE modifieIP(poste IN CHAR, nouveauIP IN CHAR);
    PostePublic    PosteTravail.typePoste%TYPE;  -- var. globale publique
END ActionsPoste;
```

Body de package : 1 fonction et 1 procédure

```
CREATE OR REPLACE PACKAGE BODY ActionsPoste AS
FUNCTION EspaceDispo(typeP IN CHAR) RETURN NUMBER IS
    espaceDisponible NUMBER;
    BEGIN
        SELECT SUM(diskRestant) INTO
        espaceDisponible FROM   PosteTravail WHERE typeposte = typeP;
        RETURN espaceDisponible;
    END EspaceDispo;

PROCEDURE modifieIP(poste IN CHAR, nouveauIP IN CHAR) IS
    espaceDisponible NUMBER;
    BEGIN
    UPDATE PosteTravail SET adrIP = nouveauIP WHERE noSerie = poste;
    END modifieIP;
```

Définition du package (suite)

```
-- suite

FUNCTION maxSpace RETURN NUMBER IS
espaceMax   NUMBER;
BEGIN
      SELECT MAX(diskRestant) INTO espaceMax FROM PosteTravail;
      RETURN espaceMax;
END maxSpace;
END ActionsPoste;
```

Appel d'une méthode du type posteTravail_t dans une procédure (avec OR)

```
CREATE OR REPLACE PROCEDURE CompterNbClients IS
        nb NUMBER;
        poste   PosteTravail_t;
BEGIN
        poste := PosteTravail_t('p5', '02', 'UnixHP', NULL); --création d'une instance
        nb := poste.NbClients();   ← appel méthode par l'objet
        DBMS_OUTPUT.PUT_LINE('p5 a' || TO_CHAR(nb) ||' client');
END CompterNbClients;
```

PUT_LINE est une fonction du package DBMS_OUTPUT qui permet d'écrire dans un *buffer* et ensuite l'afficher sans format particulier :

Le préalable:

```
SQL> SET SERVEROUTPUT  ON
SQL> EXECUTE CompterNbClients()
```

← exécution de la proc cataloguée dans le dictionnaire de données de la base.

Appel d'une procédure à partir d'une requête SQL

Au lieu d'instancier un objet poste dans le programme, il est récupéré de la base :

-- parenthèses obligatoires dans l'appel de la méthode

```
SELECT p.NbClients()
FROM PosteTravail p
WHERE p.noSerie = 'p5';
```

Ainsi, l'intention est de faire un comptage mais sans connaissance du code au moment de l'appel, il n'y a aucune garantie que la méthode NbClients() se limite à cela : elle peut avoir été modifiée par la suite pour faire aussi des insertions!

Correction : Transmettre une info (niveau de pureté) supplémentaire par une directive Pragma.

** Seule les fonctions membres d'un type peuvent être appelées dans une requête SQL.

Effet de bord et *niveau de pureté* :directive Pragma

- Erreur avec : (calcul du nb clients au poste p5).

```
SELECT p.NbClients()
    FROM PosteTravail p WHERE p.noSerie = 'p5';
```

ORA-06571 Function NbClients does nto guarantie not to update database.

L'erreur potentielle est due à l'évolution éventuelle de la fonction qui peut introduire des effets de bord sur la base de données (L'intention est de faire un simple comptage et, en fait, on modifie aussi l'état de la base!).

- La solution à ce problème consiste à définir un niveau de pureté (*purity level*) pour chaque méthode : une directive est passée à la méthode par le *Pragma*.

Déclencheur (trigger)

- S'exécute suite à un événement sur la BD :
  - Row trigger (For Each Row) : s'exécute autant de fois que l'événement est déclenché suite à une modification massive dans une table :
    - Update Employe Set salaire = salaire * 1.5
  - Statement trigger (de table) : s'exécute une seule fois, même en présence d'une modification de plusieurs objets de la même table par un seul DML;
- Règles de pratique pour les déclencheurs :
  - Programmez uniquement les actions que vous ne pouvez pas définir avec une contrainte d'intégrité au niveau de la table (avec le DDL) . Un trigger ne devrait pas remplacer le travail à faire par une méthode.
  - Évitez de définir un déclencheur qui a des effets de bord, exemple un déclencheur récursif : évitez de générer une table en mutation! Optez pour le curseur de table si possible!

Sortes de déclencheurs

- Possibilité de déclencher avant ou après (BEFORE OU AFTER) une action de type INSERT, UPDATE ou DELETE. En plus de déclencheurs de relais, les INSTEAD OF .

- Un déclencheur peut être créé et modifié (CREATE OR REPLACE TRIGGER), supprime (DROP TRIGGER), activé ou désactivé (ALTER TRIGGER avec les directives ENABLE ou DISABLE).

- Accès aux anciennes et aux nouvelles valeurs des colonnes de l'enregistrement affecté par l'événement déclencheur :OLD ou :NEW aux noms des colonnes.

- Un déclencheur est lancé sur l'occurrence d'une action et de la vérification d'une pré-condition formulée par le WHEN (…)

- **Aucun Commit ni Rollback dans un trigger.**

Exemple: définition d'un déclencheur sur PosteTravail

- CREATE OR REPLACE TYPE Local_t AS OBJECT (noL int), nomL CHAR(10) , nbPoste NUMBER); --totalise le nb de postes

- CREATE OR REPLACE TYPE PosteTravail_t AS OBJECT (noSerie int), typePoste CHAR(10) , emplacement CHAR(10));

- Les tables objets:

CREATE TABLE Local OF Local_t (CONSTRAINT pk_Local PRIMARY KEY(noLocal)) ;

CREATE TABLE PosteTravail OF PosteTravail_t (CONSTRAINT pk_PosteTravail PRIMARY KEY (nserie) , CONSTRAINT fk_PosteLocal FOREIGN KEY(emplacement) REFERENCES Local(noLocal)) ;

Déclencheur INSERT, DELETE sur PosteTravail (suite)

```
CREATE OR REPLACE TRIGGER T_I_PosteTravail AFTER INSERT ON PosteTravail
    FOR EACH ROW When( NEW.noSerie is not null)
DECLARE  -- si variables à déclarer sinon Declare non nécessaire
BEGIN
      UPDATE Local SET nbPoste = nbPoste + 1
                                        WHERE noLocal = :NEW.emplacement;
END;
```

```
CREATE OR REPLACE TRIGGER T_D_PosteTravail AFTER DELETE ON PosteTravail
    FOR EACH ROW When( OLD.noSerie is not null)

BEGIN
      UPDATE local SET nbPoste = nbPoste -1
            WHERE noLocal = :OLD.emplacement;
END;
```

Déclencheur UPDATE sur PosteTravail

```
CREATE OR REPLACE TRIGGER  Trig_U_PosteTravail AFTER UPDATE OF
    emplacement ON PosteTravail FOR EACH ROW
BEGIN
     UPDATE Local SET nbPoste = nbPoste -1
                    WHERE noLocal = :OLD.emplacement;
     UPDATE Local SET nbPoste = nbPoste + 1
                    WHERE noLocal = :NEW.emplacement;
END;
```

Exemple de l'utilisation d'un trigger polyvalent avec intégration

```
CREATE OR REPLACE TRIGGER T_IDU_PosteTravail AFTER INSERT OR DELETE
OR UPDATE OF emplacement  OR UPDATE OF typePoste ON PosteTravail FOR EACH ROW
BEGIN
IF (INSERTING) THEN
      UPDATE Local SET nbPoste = nbPoste + 1  WHERE noLocal = :NEW.emplacement;
END IF;
IF (DELETING) THEN
      UPDATE Local SET nbPoste = nbPoste –1  WHERE noLocal = :OLD.emplacement;
END IF;
IF (UPDATING ('emplacement')) THEN
```

- UPDATE salle SET nbPoste = nbPoste -1
  WHERE noLocal = :OLD.emplacement;
- UPDATE Local SET nbPoste = nbPoste + 1
  WHERE noLocal = :NEW.emplacement ;

```
IF (UPDATING ('typePoste')) THEN  ….
END IF ;
END;
```

Déclencheur INSTEAD OF

- Le déclencheur INSTEAD OF permet l'insertion, la modification et la suppression de tuples à travers une **vue** relationnelle ou objet-relationnelle multitable qui pourrait être autrement non modifiable (*car la vue est le résultat d'une jointure*). C'est un trigger de relais.

```
CREATE VIEW EtudiantEnStage AS
        SELECT e.noE, e.nomE, e.bacE, s.themeS, s.lieuS
        FROM Etudiant e, Stage s
        WHERE e.noE = s.noE;
```

Rôle de relais : fait le travail en évitant la vue :

```
CREATE TRIGGER OF ChoixEtudiant INSTEAD OF INSERT ON EtudiantEnStage
    BEGIN
      INSERT INTO Etudiant VALUES (:NEW.noE, :NEW.nomE. :NEW.bac) ;
      INSERT INTO Stage VALUES (:NEW.noS, :NEW.themeS, :NEW.lieu, :New.noE);
END;
```

Exemple avec un curseur et table OR avec imbrication

```
CREATE TYPE PosteTravail_t;  -- définition incomplète
/

CREATE TYPE ClientRef_t AS OBJECT (client REF PosteTravail_t)
/

CREATE TYPE lesClients_t AS TABLE OF ClientRef_t
/

CREATE OR REPLACE TYPE PosteTravail_t AS OBJECT (noSerie int,  adrIP CHAR(2),
    typePoste CHAR(10), lesClients  lesClients_t)
/
```

```
SQL> CREATE TABLE requete2 (numero char(l0), type char(l0));

DECLARE
    noPosteServeur   PosteTravail.noSerie%TYPE;
    CURSOR postesServeurWinNT IS SELECT noSerie FROM PosteTravail
                                                    WHERE typePoste = 'WinNT';
BEGIN
      OPEN postesServeurWinNT;
      FETCH postesServeurWinNT INTO noPosteServeur;
      WHILE postesServeurWinNT%FOUND LOOP
            INSERT INTO requete2 values (
               SELECT n.noSerie, n.typePoste
                  FROM TABLE(SELECT lesClients FROM PosteTravail WHERE noSerie =
                     noPosteServeur) n) ;
            FETCH postesServeurWinNT INTO noPosteServeur;
      END LOOP ;
      CLOSE postesServeurWinNT;
END;
```

Références

Pour une documentation plus complète, consultez le site de Oracle :

http://www.csis.gvsu.edu/GeneralInfo/Oracle/appdev.920/a96594/adobjadv.htm#1002759

http://www.csis.gvsu.edu/GeneralInfo/Oracle/appdev.920/a96594/adobjvew.htm