R3.07 - SQL dans un langage de programmation

Abdelbadie Belmouhcine, Mohammed Yasser Khayata

Institut Universitaire de Technologie de Vannes - Université Bretagne Sud
abdelbadie.belmouhcine@univ-ubs.fr

19 décembre 2023

Ressources

- Saad, M. (2016). *PL/SQL sous Oracle 12c : Guide du développeur*. Éditions ENI
- Soutou, C. (2020). *Programmer avec Oracle : SQL, PL-SQL, XML, JSON, PHP, Java*. Éditions Eyrolles

Introduction au PL/SQL

Système de Gestion de Bases de Données (SGBD)

- **SGBD** est un **ensemble de programmes** permettant de **créer** et **maintenir** une **base de données**
- Activités prises en charge :
  - **Définition** d'une base de données (spécification des **types de données** à stocker)
  - **Construction** d'une base de données (**stockage des données** proprement dites)
  - **Manipulation** des données (**ajouter**, **supprimer** et **retrouver des données**)

Exemples de SGBD

Oracle, MySQL, SQL Server, PostgreSQL, DB2, SyBase, Microsoft Access, ...

Base de données relationnelle

- Collection **bien structurée** de **données opérationnelles interreliées**
- Entité **cohérente logiquement**, véhiculant une certaine **sémantique**
- Mise à disposition de **plusieurs utilisateurs**
- Utilisée pour des **traitements par lots** et **à réponse immédiate**
- Constituée d'un **ensemble de tables** (**relations**)

Table

- Collection de **données structurées** relative à un **domaine bien défini**

Exemples de tables

étudiants d'un établissement universitaire, produits d'une entreprise commerciale

- Chaque **ligne** de la table correspond à un **enregistrement**
- Chaque **ligne** est composée de **colonnes**, appelées **champs de données** ou **attributs** de la table

SQL (Structured Query Language)

SQL est un **langage** de **requêtes** ensembliste et assertionnel proposé par le **SGBDR**. Il sert à la **création**, à l'**administration**, à l'**interrogation** et aux **manipulations** des **bases de données relationnelles**

La norme **SQL 2011** est la plus répandue aujourd'hui

Sous-Langages de SQL I

- Le langage SQL compte trois sous-langages :
  - Langage de Définition des Données (LDD) :
    - `CREATE TABLE` : Créer une table
    - `ALTER TABLE` : Ajouter ou modifier une colonne
    - `DROP TABLE` : Supprimer une table
    - `CREATE INDEX` : Créer un index
    - `DROP INDEX` : Supprimer un index
    - `TRUNCATE` : Supprimer tous les enregistrements d'une table

Sous-Langages de SQL II

- Langage de Manipulation des Données (LMD) :
  - `INSERT` : Insérer des enregistrements dans une table
  - `UPDATE` : Modifier des données dans une table
  - `DELETE` : Supprimer des enregistrements dans une table
  - `SELECT` : Extraire des données à partir d'une base
  - `COMMIT` : Valider les opérations de mise à jour pour la transaction en cours
  - `ROLLBACK` : Annuler les opérations de mise à jour pour la transaction en cours
- Langage de Contrôle des Données (LCD) :
  - `GRANT` : Attribuer des privilèges à un utilisateur
  - `REVOKE` : Supprimer des privilèges d'un utilisateur

Caractéristiques de SQL

SQL permet d'exprimer des contraintes (d'attribut ou de table)

- de clé primaire (`PRIMAREY KEY`)
- de clé étrangère (`REFERENCES`)
- d'existence (`NOT NULL`)
- d'unicité (`UNIQUE`)
- de vérification (`CHECK`)

Mais il ne permet pas

- d'exprimer toutes les contraintes ➡ ***Déclencheurs***
- d'appliquer des traitements « complexes » sur les données ➡ ***Langage non procédural***

Présentation et historique I

PL/SQL

- Acronyme : **Procedural Language / Structured Query Language**
- Extension de **SQL** : les manipulations de données optimisées par le **SGBDR** cohabitent avec les éléments habituels de la programmation structurée
- Disponible dans **Oracle** Database depuis la version 7
- Combinaison des avantages d'un langage de programmation classique avec les structures algorithmiques et les possibilités de manipulation de données offertes par **SQL**

Présentation et historique II

Évolution de PL/SQL

- Développé par **Oracle** à la fin des années **1980**
- Capacités élargies au début des années **1990**
- Évolution vers un langage de programmation **procédural** et **orienté objet** avec l'avènement des concepts orientés objet dans Oracle 8i et 9i
- À partir de la version 11g, **PL/SQL** est passé d'un langage interprété à un **langage compilé**. Le code **PL/SQL** est stocké dans le **tablespace** système après une compilation directe

Avantages de PL/SQL

Les manipulations de données dans le développement d'une application

- Interface d'accès aux données : JDBC / PDO ➡ ***R3.01***
- Patron de conception Data Access Object (DAO) ➡ ***R4.01***
- Interface Object-Relational Mapping (ORM) : JPA / Doctrine

Le coût de ces approches peuvent limiter les performances

Le PL/SQL permet d'automatiser de manière efficace et transparente certains traitements grâce aux SGBDR, c'est un complément essentiel

Moteur PL/SQL

- Lorsque le moteur **PL/SQL** reçoit un **bloc** pour exécution, il effectue les opérations suivantes :
  - ☞ séparation des commandes **SQL** et **PL/SQL**
  - ☞ passage des commandes **SQL** au **processeur SQL** (SQL statement executor)
  - ☞ passage des **instructions procédurales** au **processeur d'instructions procédurales** (procedural statement executor)

Unités PL/SQL

- Les **unités** de PL/SQL sont :
  - bloc anonyme PL/SQL
  - fonction
  - procédure
  - déclencheur
  - bibliothèque
  - paquetage (package)
  - corps de paquetage
  - type
  - corps de type

Bloc PL/SQL

Structure d'un programme PL/SQL I

- Un programme est structuré en **blocs d'instructions** de 3 types :
  - les **procédures anonymes**
  - les **procédures nommées**
  - les **fonctions nommées**

Structure d'un bloc anonyme

```
DECLARE
    -- déclaration de toutes les variables et tous les types, constantes,
      curseurs et exceptions utilisateur référencés dans la section exécutable
BEGIN
    -- instructions PL/SQL (affectations, instructions conditionnelles,
      boucles, appels de procédure, etc.) ainsi que les commandes SQL
    EXCEPTION
        -- récupération des erreurs
END;
```

Structure d'un programme PL/SQL II

- Les blocs, comme les instructions, se terminent par un « ; »
- Seuls **BEGIN** et **END** sont obligatoires

Identificateur

- **30** caractères au plus
- **commence** par une **lettre**
- peut contenir **lettres**, **chiffres**, _, **$** et **#**
- **pas** sensible à la **casse**
- ne correspond **pas** à un **mot réservé** (`SELECT`, `INSERT`, `UPDATE`, `COMMIT`, `ROLLBACK`, `SUM`, `MAX`, ...)
- **doit** être distinct des noms de **tables** et de **colonnes**

Littéraux

- Les **dates** et les **chaînes de caractères** sont délimitées par des **simples quotes** (`'20-NOV-2023'` ou `'texte'`)
- Les valeurs numériques :
  - Écriture standard : nombre réel avec partie entière et décimale (ex. `-5.25`)
  - Écriture scientifique : mantisse et exposant (ex. `2E5` = 2×10^5 = `200000`)

Commentaires

- Il est évidemment possible (et recommandé) de commenter un programme

```
-- Commentaire

-- Commentaire d'une (fin de) ligne

/*
Commentaire
de plusieurs
lignes
*/
```

Types de données en PL/SQL

- **Types scalaires**
  - Types de données `SQL`
  - `BOOLEAN`
  - `PLS_INTEGER`
  - `BINARY_INTEGER`
  - `REF CURSOR` (curseur de référence)
  - Sous-types définis par l'utilisateur
- **Types composés**
  - Tableaux
  - Enregistrements
  - ...

Types de données - Caractères

- `CHAR[(n)]`
  - Chaîne de caractères de longueur fixe
  - Longueur en PL/SQL : 1 à 32767 octets
  - Longueur en SQL : 1 à 2000 octets
  - Par défaut, n est égal à 1
- `NCHAR[(n)]`
  - Identique à CHAR, mais avec représentation en unicode
- `VARCHAR2(n)`
  - Chaîne de caractères de longueur variable
  - Longueur en PL/SQL : 1 à 32767 octets
  - Longueur en SQL : 1 à 4000 octets
- `NVARCHAR2(n)`
  - Identique à VARCHAR2, avec représentation en unicode
- `LONG`
  - Semblable à VARCHAR2
  - Longueur en PL/SQL : max 32760 octets
  - Longueur max en SQL : 2 Go
- `RAW(n)`
  - Données binaires de longueur variable
  - Longueur en PL/SQL : 1 à 32767 octets
  - Longueur en SQL : 1 à 2000 octets
  - Aucune conversion de caractères lors du transfert
- `LONG RAW`
  - Semblable à LONG, avec données binaires en hexadécimal
  - Longueur max en PL/SQL : 32760 octets
  - Longueur max en SQL : 2Go

Types de données - Numériques I

- `NUMBER[(n[, m)]]`
  - Valeur numérique décimale
  - n chiffres significatifs (38 par défaut)
  - m positions décimales ou entières (0 par défaut)
  - Plage : -2^{418} à 2^{418}
- `PLS_INTEGER` et `BINARY_INTEGER`
  - Entiers de -2^{31} à 2^{31}
  - `PLS_INTEGER` plus rapide (utilise les registres du processeur)

Le type `PLS_INTEGER` et ses sous-types peuvent être convertis implicitement vers les types de données suivants : `CHAR`, `VARCHAR2`, `NUMBER` et `LONG`. Aussi, tous les types de données précédents, à l'exception du type `LONG`, et tous les sous-types de `PLS_INTEGER`, peuvent être implicitement convertis en `PLS_INTEGER`

Types de données - Numériques II

- `BINARY_FLOAT`
  - Nombre à virgule flottante simple précision
  - 5 octets par valeur
  - Valeurs entre -3.4×10^{38} et 3.4×10^{38}
  - Plus petite valeur positive : 1.2×10^{-38}
  - Plus grande valeur négative : -1.2×10^{-38}
- `BINARY_DOUBLE`
  - Nombre à virgule flottante double précision
  - 9 octets par valeur
  - Valeurs entre -1.79×10^{308} et 1.79×10^{308}
  - Plus petite valeur positive : 2.3×10^{-308}
  - Plus grande valeur négative : -2.3×10^{-308}

`SIMPLE_FLOAT` et `SIMPLE_DOUBLE` sont des sous-types de `BINARY_FLOAT` et `BINARY_DOUBLE` avec la contrainte `NOT NULL`

Types de données - Grands objets

- `CLOB`
  - Permet de stocker un flot de caractères, par exemple un texte
  - Taille maximale de 128 To en PL/SQL et 4 Go en SQL
- `NCLOB`
  - Identique à CLOB, mais les données de ce type sont représentées par le unicode
- `BLOB`
  - Permet de stocker un objet binaire non structuré, comme le multimédia (images, sons, vidéo, etc.)
  - Taille maximale de l'objet de 128 To en PL/SQL et 4 Go en SQL
- `BFILE`
  - Permet de stocker des données binaires dans un fichier externe à la base

Types de données - Autres types

- `DATE`
  - Permet de stocker des dates, constituées du siècle, de l'année, du mois... des secondes
  - Plage de type DATE de 1/1/-4712 à 31/12/9999. Par défaut, heures, minutes et secondes valent 00:00:00
- `TIMESTAMP[(n)]`
  - Permet de stocker des dates et heures avec la granularité des fractions de secondes
  - Précision des fractions de secondes de 0 à 9 (par défaut n=6)
- `TIMESTAMP WITH TIME ZONE`
  - Permet de stocker des dates et heures de type TIMESTAMP avec la prise en compte des fuseaux horaires
- `TIMESTAMP WITH LOCAL TIME ZONE`
  - Permet de faire la distinction entre une heure de serveur et une heure du client
- `INTERVAL YEAR TO MONTH`
  - Permet de stocker la différence de deux dates avec la précision mois/année
- `INTERVAL DAY TO SECOND`
  - Permet d'enregistrer une période de temps en jours, heures, minutes et secondes
- `BOOLEAN`
  - Stocke les valeurs booléennes logiques (VRAI, FAUX, NULL représente une valeur inconnue)
  - **Existe seulement en PL/SQL**

Déclarations de variables

```
DECLARE
   sexe CHAR(1);
   /* déclaration d'une variable pour stocker le sexe d'une personne (M ou F)
      */
   i BINARY_INTEGER:= 0;
   /* déclaration d'un compteur initialisé à 0 */
   date_commande DATE:= SYSDATE;
   /* déclaration d'une variable pour stocker la date de création d'une
      commande. La variable est initialisée à la date du jour.*/
   message VARCHAR2(30):='Bonjour';
   /* déclaration d'une variable pour stocker un message, qui est initialisée
      à 'Bonjour' */
   note FLOAT := 10.5;
   /* déclaration d'une variable pour stocker la note, qui  est  initialisée à
       10,5 */
BEGIN
   NULL;
END;
```

Déclaration par un type de données

```
DECLARE
   nom EMPLOYE.ENOM%TYPE;
   sal EMPLOYE.SALAIRE%TYPE;
   /* déclaration de deux variables nom et sal ayant les mêmes type de données
      que les champs ENOM et SALAIRE de la table EMPLOYE */

   min_salaire sal%type;
   /* déclaration d'une variable min_salaire ayant le même type de données que
      la variable sal */

BEGIN
   NULL;
END;
```

Déclarations de constantes

```
DECLARE
   pi NUMBER(8,5) :=3.14159; -- type de données SQL
   min_salaire CONSTANT REAL := 1000.00; -- type de données SQL
   trouve CONSTANT BOOLEAN := FALSE; -- type de données PL/SQL

BEGIN
   NULL;
END;
```

Paquetage DBMS_OUTPUT en Oracle I

- **DBMS_OUTPUT** assure la gestion des **entrées-sorties** dans les **blocs PL/SQL**. Les procédures clés sont :
  - `ENABLE` : active les entrées-sorties
  - `DISABLE` : désactive les entrées-sorties
  - `PUT (expression)` : affiche l'expression à l'écran
  - `NEW_LINE` : effectue un retour à la ligne
  - `PUT_LINE (expression)` : appelle `PUT` suivi d'un appel de `NEW_LINE`
  - `GET_LINE (out chaîne, out état)` : lit une ligne dans les paramètres chaîne, avec état prenant la valeur 0 si la valeur lue est valide
  - `GET_LINES (out chaîne, in-out nbre_lignes)` : permet la lecture de plusieurs lignes

Paquetage DBMS_OUTPUT en Oracle II

Avant d'utiliser ce paquetage, on doit l'activer au préalable avec la commande `SET SERVEROUTPUT ON`. L'appel de toute procédure d'un paquetage se réalise avec l'instruction `nom_paquetage.nom_procédure(paramètres)`

Exemple

```
-- activation du paquetage sous SQL*PLUS
SET SERVEROUTPUT ON

DECLARE
   nom VARCHAR2(25) := 'THOMAS';
   fonct VARCHAR2(25) :='PRESIDENT';
   sal NUMBER := 5000;

BEGIN

   DBMS_OUTPUT.ENABLE; -- Activation du paquetage sous PL/SQL
   DBMS_OUTPUT.PUT_LINE('Votre nom est         '|| nom||',');
   DBMS_OUTPUT.PUT_LINE('votre fonction est    '|| fonct||',');
   DBMS_OUTPUT.PUT_LINE('et votre salaire est '|| sal||'.');

END;
```

Exemple

```sql
-- activation du paquetage sous SQL*PLUS
SET SERVEROUTPUT ON

DECLARE
   nom VARCHAR2(25) := 'THOMAS';
   fonct VARCHAR2(25) :='PRESIDENT';
   sal NUMBER := 5000;

BEGIN

   DBMS_OUTPUT.ENABLE; -- Activation du paquetage sous PL/SQL
   DBMS_OUTPUT.PUT_LINE('Votre nom est         '|| nom||',');
   DBMS_OUTPUT.PUT_LINE('votre fonction est    '|| fonct||',');
   DBMS_OUTPUT.PUT_LINE('et votre salaire est '|| sal||'.');

END;
```

Votre nom est THOMAS ,
votre fonction est PRESIDENT ,
et votre salaire est 5000.

Affectation

- L'**affectation** de variable s'effectue :
  - directement avec « := »
  - via une requête `SELECT` avec la directive `INTO`

Conflits de noms

- Si une **variable** porte le **même nom** qu'**une colonne**, c'est la **colonne** qui l'emporte ce qui peut provoquer de graves erreurs
- Pour éviter cela, on peut préfixer par « v_ » le nom d'une **variable** [a]

a. et par « p_ » le nom d'un **paramètre** (bonne pratique pour les procédures et fonctions)

Exemple

Affectation simple

```
SET SERVEROUTPUT ON
DECLARE
   a INTEGER;
   b INTEGER;
   c INTEGER;
BEGIN -- affectation des variables
   a := 1;
   DBMS_OUTPUT.PUT_LINE('A = '|| a);
   b := a+3;
   DBMS_OUTPUT.PUT_LINE('B = '|| b);
   a := 3;
   DBMS_OUTPUT.PUT_LINE('A = '|| a);
   b := 5;
   DBMS_OUTPUT.PUT_LINE('B = '|| b);
   c := a+b;
   DBMS_OUTPUT.PUT_LINE('C = '|| c);
   c := b-a;
   DBMS_OUTPUT.PUT_LINE('C = '|| c);
END;
```

Exemple

Affectation simple

```
SET SERVEROUTPUT ON
DECLARE
   a INTEGER;
   b INTEGER;
   c INTEGER;
BEGIN -- affectation des variables
   a := 1;
   DBMS_OUTPUT.PUT_LINE('A = '|| a);
   b := a+3;
   DBMS_OUTPUT.PUT_LINE('B = '|| b);
   a := 3;
   DBMS_OUTPUT.PUT_LINE('A = '|| a);
   b := 5;
   DBMS_OUTPUT.PUT_LINE('B = '|| b);
   c := a+b;
   DBMS_OUTPUT.PUT_LINE('C = '|| c);
   c := b-a;
   DBMS_OUTPUT.PUT_LINE('C = '|| c);
END;
```

A = 1
B = 4
A = 3
B = 5
C = 8
C = 2

Exemple

Affectation et initialisation

```
SET SERVEROUTPUT ON
DECLARE
   /* on peut faire l'affectation dans la partie déclarative : initialisation
      */
   salaire NUMBER;
   heure_travail NUMBER := 40;
   salaire_horaire NUMBER := 22.50;
   bonus NUMBER := 100;
   pays VARCHAR2(50);
   nom VARCHAR2(50) := 'thomas';
   max_sal NUMBER := 700;
   valide BOOLEAN;
BEGIN -- on peut faire l'affectation dans le corps du bloc
   salaire := (salaire_horaire * heure_travail) + bonus;
   nom := UPPER(nom);
   pays := 'France';
   valide := (salaire > max_sal);
   DBMS_OUTPUT.PUT_LINE(nom || ' habite en '|| pays);
   DBMS_OUTPUT.PUT_LINE(' Il gagne '|| salaire);
END;
```

Exemple

Affectation et initialisation

```
SET SERVEROUTPUT ON
DECLARE
   /* on peut faire l'affectation dans la partie déclarative : initialisation
     */
   salaire NUMBER;
   heure_travail NUMBER := 40;
   salaire_horaire NUMBER := 22.50;
   bonus NUMBER := 100;
   pays VARCHAR2(50);
   nom VARCHAR2(50) := 'thomas';
   max_sal NUMBER := 700;
   valide BOOLEAN;
BEGIN -- on peut faire l'affectation dans le corps du bloc
   salaire := (salaire_horaire * heure_travail) + bonus;
   nom := UPPER(nom);
   pays := 'France';
   valide := (salaire > max_sal);
   DBMS_OUTPUT.PUT_LINE(nom || ' habite en '|| pays);
   DBMS_OUTPUT.PUT_LINE(' Il gagne '|| salaire);
END;
```

L'employé THOMAS a pour matricule 7800

Exemple

Affectation par select

```
SET SERVEROUTPUT ON
DECLARE
   num NUMBER;
   nom_emp VARCHAR2(30) := 'THOMAS';
BEGIN
-- affectation des variables par l'instruction SELECT ... INTO
   SELECT MATRICULE INTO num
   FROM EMPLOYE
   WHERE ENOM= nom_emp;
   DBMS_OUTPUT.PUT_LINE('L'employé '||nom_emp||' a pour matricule ' ||num);
END;
```

Exemple

Affectation par select

```
SET SERVEROUTPUT ON
DECLARE
   num NUMBER;
   nom_emp VARCHAR2(30) := 'THOMAS';
BEGIN
-- affectation des variables par l'instruction SELECT ... INTO
   SELECT MATRICULE INTO num
   FROM EMPLOYE
   WHERE ENOM= nom_emp;
   DBMS_OUTPUT.PUT_LINE('L'employé '||nom_emp||' a pour matricule ' ||num);
END;
```

THOMAS habite en France
Il gagne 1000

Priorité

```
SET SERVEROUTPUT ON

DECLARE
   a INTEGER := 1+3**3;
   b INTEGER := (1+3)**3;
   c INTEGER := ((1+3)*(4+5))/6;
   d INTEGER := 2**2*3**2;
   e INTEGER := (2**2)*(3**2);

BEGIN

   DBMS_OUTPUT.PUT_LINE('a = ' || a);
   DBMS_OUTPUT.PUT_LINE('b = ' || b);
   DBMS_OUTPUT.PUT_LINE('c = ' || c);
   DBMS_OUTPUT.PUT_LINE('d = ' || d);
   DBMS_OUTPUT.PUT_LINE('e = ' || e);

END;
```

Priorité

```
SET SERVEROUTPUT ON

DECLARE
   a INTEGER := 1+3**3;
   b INTEGER := (1+3)**3;
   c INTEGER := ((1+3)*(4+5))/6;
   d INTEGER := 2**2*3**2;
   e INTEGER := (2**2)*(3**2);

BEGIN

   DBMS_OUTPUT.PUT_LINE('a = ' || a);
   DBMS_OUTPUT.PUT_LINE('b = ' || b);
   DBMS_OUTPUT.PUT_LINE('c = ' || c);
   DBMS_OUTPUT.PUT_LINE('d = ' || d);
   DBMS_OUTPUT.PUT_LINE('e = ' || e);

END;
```

a = 28
b = 64
c = 6
d = 36
e = 36

Blocs imbriqués et portée d'un objet

```
SET SERVEROUTPUT ON
-- bloc parent
DECLARE
   x INTEGER :=10;
BEGIN
   DBMS_OUTPUT.PUT_LINE('Bloc parent');
   DBMS_OUTPUT.PUT_LINE('x = ' || x);
-- sous bloc
   DECLARE
      y INTEGER;
   BEGIN
      x :=x+5;
      DBMS_OUTPUT.PUT_LINE('Sous Bloc');
      DBMS_OUTPUT.PUT_LINE('x = ' || x);

   END;
 END;
```

Blocs imbriqués et portée d'un objet

```
SET SERVEROUTPUT ON
-- bloc parent
DECLARE
   x INTEGER :=10;
BEGIN
   DBMS_OUTPUT.PUT_LINE('Bloc parent');
   DBMS_OUTPUT.PUT_LINE('x = ' || x);
-- sous bloc
   DECLARE
      y INTEGER;
   BEGIN
      x :=x+5;
      DBMS_OUTPUT.PUT_LINE('Sous Bloc');
      DBMS_OUTPUT.PUT_LINE('x = ' || x);

   END;
 END;
```

Bloc parent
x = 10
Sous Bloc
x = 15

Variables de substitution

```
SET SERVEROUTPUT ON
DECLARE
   mat NUMBER := &s_mat; -- variable de substitution
   -- On suppose que le matricule saisi est correcte
   nom EMPLOYE.ENOM%TYPE;
   fonct EMPLOYE.FONCTION%TYPE;
   sal EMPLOYE.SALAIRE%TYPE;
BEGIN
   SELECT ENOM, FONCTION, SALAIRE INTO nom, fonct, sal FROM EMPLOYE WHERE
      MATRICULE = mat;
   DBMS_OUTPUT.PUT_LINE ('L'employé de nom '||nom);
   DBMS_OUTPUT.PUT_LINE ('Sa fonction est  '|| fonct);
   DBMS_OUTPUT.PUT_LINE ('Son salaire est  '|| sal);
END;
```

Variables de substitution

```
SET SERVEROUTPUT ON
DECLARE
   mat NUMBER := &s_mat; -- variable de substitution
   -- On suppose que le matricule saisi est correcte
   nom EMPLOYE.ENOM%TYPE;
   fonct EMPLOYE.FONCTION%TYPE;
   sal EMPLOYE.SALAIRE%TYPE;
BEGIN
   SELECT ENOM, FONCTION, SALAIRE INTO nom, fonct, sal FROM EMPLOYE WHERE
      MATRICULE = mat;
   DBMS_OUTPUT.PUT_LINE ('L'employé de nom '||nom);
   DBMS_OUTPUT.PUT_LINE ('Sa fonction est  '|| fonct);
   DBMS_OUTPUT.PUT_LINE ('Son salaire est  '|| sal);
END;
```

Entrez une valeur pour s_mat : 7800
ancien :... mat NUMBER := &s_mat ; ...
nouveau :... mat NUMBER := 7800 ; ...
L'employé de nom THOMAS
Sa fonction est PRESIDENT
Son salaire est 5000

Variables de substitution - Accept

```
ACCEPT s_mat PROMPT 'Entrer Matricule Employé : '
SET SERVEROUTPUT ON
DECLARE
   nom EMPLOYE.ENOM%TYPE;
   fonct EMPLOYE.FONCTION%TYPE;
   sal EMPLOYE.SALAIRE%TYPE;
BEGIN
   SELECT ENOM, FONCTION, SALAIRE FROM EMPLOYE WHERE MATRICULE = &s_mat;
   DBMS_OUTPUT.PUT_LINE ('L'employé de nom '||nom);
   DBMS_OUTPUT.PUT_LINE ('Sa fonction est  '|| fonct);
   DBMS_OUTPUT.PUT_LINE ('Son salaire est  '|| sal);
END;
```

Variables de substitution - Accept

```
ACCEPT s_mat PROMPT 'Entrer Matricule Employé : '
SET SERVEROUTPUT ON
DECLARE
   nom EMPLOYE.ENOM%TYPE;
   fonct EMPLOYE.FONCTION%TYPE;
   sal EMPLOYE.SALAIRE%TYPE;
BEGIN
   SELECT ENOM, FONCTION, SALAIRE FROM EMPLOYE WHERE MATRICULE = &s_mat;
   DBMS_OUTPUT.PUT_LINE ('L'employé de nom '||nom);
   DBMS_OUTPUT.PUT_LINE ('Sa fonction est  '|| fonct);
   DBMS_OUTPUT.PUT_LINE ('Son salaire est  '|| sal);
END;
```

Entrer Matricule Employé : 7800
ancien :... WHERE MATRICULE = s_mat ; ...
nouveau : ... WHERE MATRICULE = 7800 ; ...
L'employé de nom THOMAS
Sa fonction est PRESIDENT
Son salaire est 5000

Variables de session (hôtes)

```
VARIABLE g_x NUMBER;
VARIABLE g_y NUMBER;
DECLARE
   z NUMBER;
BEGIN
 :g_x := 10;
 :g_y := 15;
 z    := :g_x;
 :g_x := :g_y;
 :g_y := z;
END;
/
PRINT g_x;
PRINT g_y;
```

Variables de session (hôtes)

```
VARIABLE g_x NUMBER;
VARIABLE g_y NUMBER;
DECLARE
   z NUMBER;
BEGIN
 :g_x := 10;
 :g_y := 15;
 z    := :g_x;
 :g_x := :g_y;
 :g_y := z;
END;
/
PRINT g_x;
PRINT g_y;
```

```
      G_X
---------
       15
      G_Y
---------
       10
```

Structures de contrôles

Structures de contrôle en PL/SQL

- Structures conditionnelles :
  - L'instruction `IF` : Utilisée pour exécuter un ensemble d'instructions si une condition est vraie
  - L'instruction `CASE` : Utilisée pour effectuer différentes actions en fonction de la valeur d'une expression
- Structures itératives :
  - La boucle de base `LOOP` : Répète une série d'instructions sans condition globale
  - La boucle `FOR` : Contrôle le nombre d'itérations grâce à un indice

La boucle `FOR` peut aussi être utilisée avec une **collection indexée** ou un **curseur** lors de l'itération à travers les enregistrements d'une requête SQL

  - La boucle `WHILE` : Contrôle les itérations selon la vérification d'un prédicat

Conditions logiques en PL/SQL

- Une **condition** en PL/SQL est une expression booléenne évaluée comme **TRUE**, **FALSE** ou **NULL**

Opérateur	Description
<	Inférieur
>	Supérieur
<=	Inférieur ou égal
>=	Supérieur ou égal
=	Égal
<>, !=, ¬=, ^=	Différent
IS NULL	Vérifie si l'opérande est **NULL**
IS NOT NULL	Vérifie si l'opérande n'est pas **NULL**
LIKE	Comparaison partielle avec **%** et **_**
BETWEEN	Appartenance à un intervalle
IN	Appartenance à une liste de valeurs prédéfinies
AND	**Et** logique
OR	**Ou** logique
NOT	**Négation** logique

Traitements conditionnels I

- On dispose du :

IF

```
-- Branchement conditionnel
IF condition THEN
    instructions;
[ELSIF condition THEN
    instructions;]
[ELSE
    instructions;]
END IF;
```

Traitements conditionnels II

- ainsi que du :

CASE simple

```
-- Case simple
CASE selecteur
   WHEN expression_1 THEN instructions_1;
   WHEN expression_2 THEN instructions_2;
   ...
   WHEN expression_n THEN instructions_n;
[ELSE
   instructions_else;]
END CASE;
```

Si aucune condition n'est vraie et que l'instruction `ELSE` n'existe pas dans la commande, alors l'exception prédéfinie `CASE_NOT_FOUND` est soulevée

Traitements conditionnels III

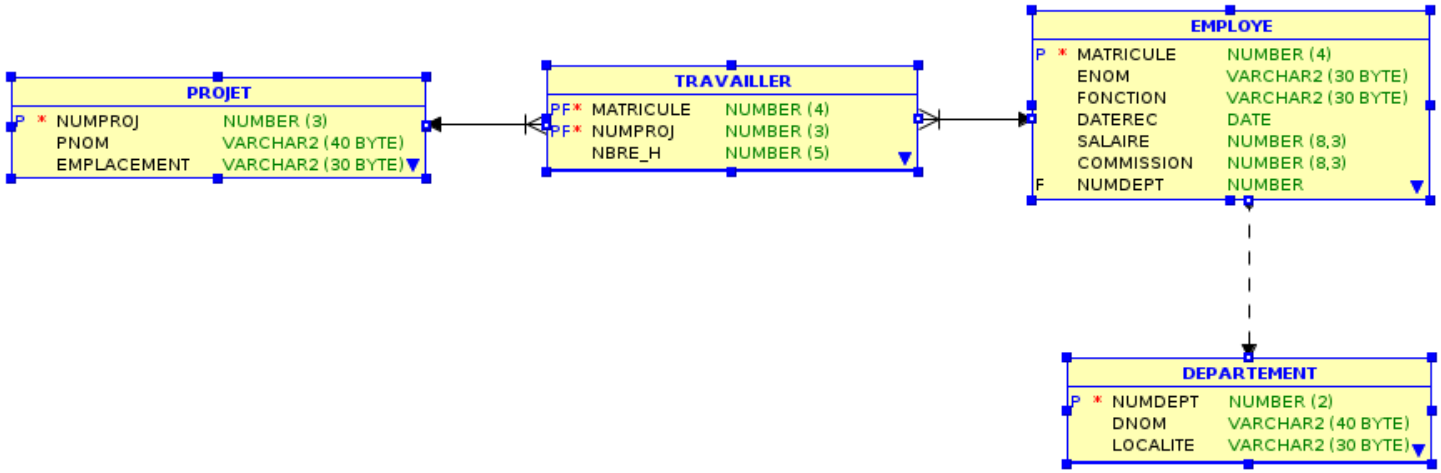
CASE de recherche

```
-- Case de recherche
CASE
   WHEN condition_1 THEN instructions_1;
   WHEN condition_2 THEN instructions_2;
   ...
   WHEN condition_n THEN instructions_n;
[ELSE
   instructions_else;]
END CASE;
```

Si aucune condition n'est vraie et que l'instruction `ELSE` n'existe pas dans la commande, alors l'exception prédéfinie `CASE_NOT_FOUND` est soulevée

Exemple - IF

- Soit le SR suivant :



- Écrire un bloc **PL/SQL** qui permet d'augmenter le salaire d'un employé particulier, selon le nombre des heures de travail (quota) dans des projets :
  - `salaire = salaire + 0.1 * quota, si quota <= 100,`
  - `salaire = salaire + 0.2 * quota, si 100 < quota <= 200,`
  - `salaire = salaire + 0.3 * quota, si 200 < quota <= 300,`
  - `salaire = salaire + 0.4 * quota, si quota >300`

Exemple - IF

Solution

```
DECLARE
-- déclaration des variables
   quota TRAVAILLER.NBRE_H%TYPE;
   bonus EMPLOYE.SALAIRE%TYPE;
   sal EMPLOYE.SALAIRE%TYPE;
   emp_id EMPLOYE.MATRICULE%TYPE :=7566;
BEGIN
-- sélection de nombre des heures de travail
   SELECT SUM(NBRE_H) INTO quota FROM TRAVAILLER WHERE MATRICULE = emp_id;
-- sélection de salaire
   SELECT SALAIRE INTO sal FROM EMPLOYE WHERE MATRICULE = emp_id;
-- détermination de bonus
   IF quota <= 100 THEN bonus := 0.1*quota;
   ELSIF quota <= 200 THEN bonus := 0.2*quota;
   ELSIF quota <= 300 THEN bonus := 0.3*quota;
   ELSE bonus := 0.4*quota;
   END IF;
-- modification de salaire
   UPDATE EMPLOYE SET SALAIRE = SALAIRE + bonus WHERE MATRICULE = emp_id;
-- sélection de salaire après modification
   SELECT SALAIRE INTO sal FROM EMPLOYE WHERE MATRICULE = emp_id;
END;
```

Exemple - CASE simple

- Écrire un bloc **PL/SQL** qui, à partir d'une appréciation donnée, affiche la mention associée

Exemple - CASE simple

- Écrire un bloc **PL/SQL** qui, à partir d'une appréciation donnée, affiche la mention associée

Solution

```
SET SERVEROUTPUT ON
DECLARE
   app CHAR(1);

BEGIN
   app := 'B';
-- affichage de la mention
   CASE app
       WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
       WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Très bien');
       WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Bien');
       WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Passable');
       WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Faible');
   ELSE
       DBMS_OUTPUT.PUT_LINE(' appréciation inconnue !!!');
   END CASE;
END;
```

Exemple - CASE de recherche

- Écrire un bloc **PL/SQL** qui, à partir d'une moyenne donnée, affiche la mention associée

Exemple - CASE de recherche

- Écrire un bloc **PL/SQL** qui, à partir d'une moyenne donnée, affiche la mention associée

Solution

```
SET SERVEROUTPUT ON
DECLARE
   moyenne NUMBER(4,2) :=13.50;
BEGIN
  CASE
   WHEN moyenne between 18 and 20  THEN
    DBMS_OUTPUT.PUT_LINE('Excellent');
   WHEN moyenne between 16 and 17.99 THEN
    DBMS_OUTPUT.PUT_LINE('Très Bien');
   WHEN moyenne between 14 and 15.99 THEN
    DBMS_OUTPUT.PUT_LINE('Bien');
   WHEN moyenne between 12 and 13.99 THEN
    DBMS_OUTPUT.PUT_LINE('Assez bien');
   WHEN moyenne between 10 and 11.99 THEN
    DBMS_OUTPUT.PUT_LINE('Passable');
   WHEN moyenne between 0 and 9.99 THEN
    DBMS_OUTPUT.PUT_LINE('Redouble');
   ELSE
    DBMS_OUTPUT.PUT_LINE('Moyenne inconnue !!!');
  END CASE;
END;
```

Traitements répétitifs I

- On dispose de la :

boucle « générale »

```
-- Boucle "générale"
LOOP
   instructions;
   ...
   [EXIT [WHEN condition_1;]]
   ...
   [CONTINUE [WHEN condition_2;]]
   ...
END LOOP;
```

Une boucle **sans EXIT** est une boucle **infinie**

Traitements répétitifs II

- ainsi que de la :

boucle « tant que »

```
-- Boucle "tant que"
WHILE condition LOOP
    instructions;
    ...
END LOOP;
```

Traitements répétitifs III

- sans oublier la :

boucle « pour »

```
-- Boucle "pour"
FOR compteur IN [REVERSE] inf..sup LOOP
    instructions;
    ...
END LOOP;
```

Le compteur est déclaré **implicitement** comme un **entier**. Il n'est pas nécessaire de le déclarer

Traitements répétitifs IV

Parmi toutes ces boucles, on utilisera principalement la boucle « **pour** », mais avec un **curseur**...

Curseurs

Curseurs PL/SQL

- Une **zone mémoire privée** de `SQL` contenant le **résultat** d'une requête
- Deux types de **curseurs** :
  - **Curseurs implicites** : Déclarés automatiquement lors de l'exécution d'une instruction `SQL`. L'utilisateur n'a pas de contrôle sur ces curseurs
  - **Curseurs explicites** : Générés et gérés par le programmeur pour traiter un requêtes `SELECT` renvoyant plusieurs lignes (définis dans la section `DECLARE`)

Curseurs implicites

- Géré implicitement par PL/SQL lors de l'exécution d'instructions comme SELECT, INSERT, DELETE ou UPDATE
- Le **programmeur** ne peut pas le contrôler, mais peut obtenir des informations à partir de ses attributs
- Principaux attributs :
  - SQL%ISOPEN : Toujours FALSE, car le curseur implicite se ferme après l'exécution de l'instruction
  - SQL%FOUND : TRUE si l'instruction SELECT ou LMD affecte au moins une ligne
  - SQL%NOTFOUND : TRUE si l'instruction SELECT ou LMD ne renvoie aucune ligne
  - SQL%ROWCOUNT : Retourne le nombre de lignes renvoyées par SELECT ou affectées par la dernière LMD

Exemple

Attributs des curseurs implicites

```
SET SERVEROUTPUT ON
BEGIN
-- modification des projets dont l'emplacement est PARIS
   UPDATE PROJET
   SET EMPLACEMENT='BORDEAUX'
   WHERE EMPLACEMENT='PARIS';
-- vérification si des modifications sont effectuées
   IF SQL%FOUND THEN DBMS_OUTPUT.PUT_LINE ('Modification effectuée');
   ELSE DBMS_OUTPUT.PUT_LINE ('Pas de modification');
   END IF;
-- vérification si des modifications sont effectuées
   IF SQL%NOTFOUND THEN DBMS_OUTPUT.PUT_LINE('Pas de modification');
   ELSE DBMS_OUTPUT.PUT_LINE('Modification effectuée');
   END IF;
-- Suppression des lignes de la table TRAVAILLER
   DELETE FROM TRAVAILLER;
-- le nombre des lignes supprimées
   DBMS_OUTPUT.PUT_LINE('Le nombre des lignes supprimées de la table
      TRAVAILLER est  '|| SQL%ROWCOUNT);
  ROLLBACK;
END;
```

Curseurs explicites

- Construit et géré par le **programmeur**
- Offre un contrôle plus programmatique par rapport au **curseur implicite**
- Permet de traiter un ensemble de lignes de manière efficace

Déclaration et définition

```
-- Curseur explicite
DECLARE
   -- Déclaration
   CURSOR nom_curseur  RETURN return_type;
   -- Définition
   CURSOR nom_curseur IS requête_select;
```

Exemple

Déclaration et définition

```
DECLARE
-- déclaration c1
   CURSOR c1 RETURN PROJET%ROWTYPE;
-- déclaration et définition c2
   CURSOR c2 IS SELECT *  FROM EMPLOYE  WHERE SALAIRE > 4000;
-- définition c1
   CURSOR c1 RETURN PROJET%ROWTYPE IS SELECT * FROM PROJET;
-- déclaration c3
   CURSOR c3 RETURN TRAVAILLER%ROWTYPE;
-- définition c3
   CURSOR c3 IS SELECT * FROM TRAVAILLER WHERE MATRICULE = 780;
BEGIN
   NULL;
END;
```

Boucle FOR avec curseur

- Utilisé pour traiter les lignes d'un curseur explicite
- Gestion automatique de l' **ouverture**, de la **lecture**, du **test de sortie**, et de la **fermeture** du **curseur**

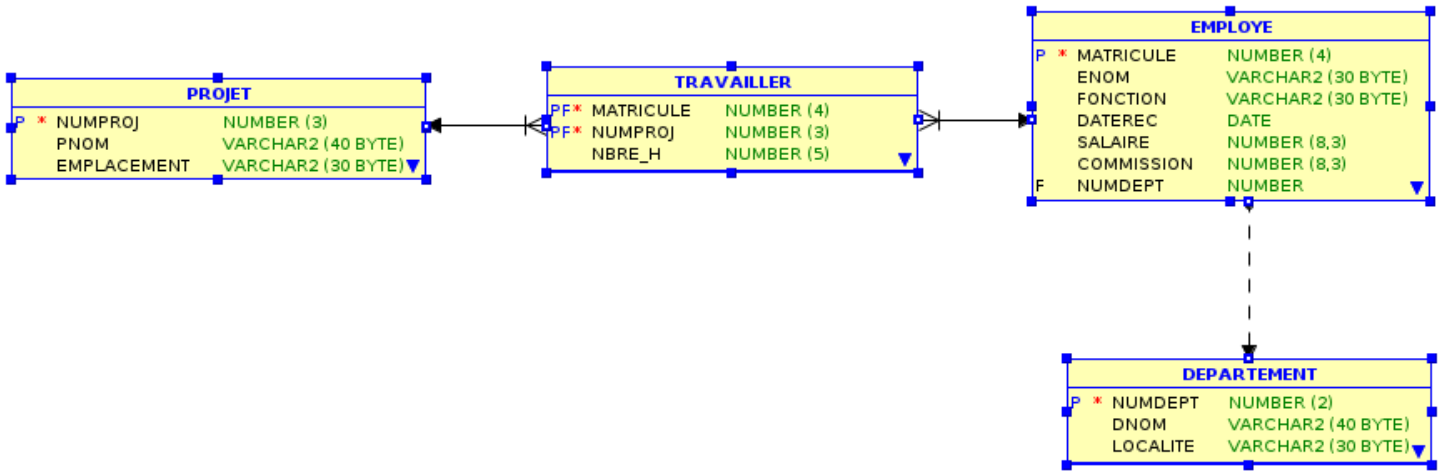
Synthaxe

```
-- FOR et Curseur
FOR nom_enregistrement IN nom_curseur LOOP
    ...
END LOOP;
```

- Il faut noter que :
  - L'enregistrement qui contrôle la boucle ne doit pas être déclaré. Il est visible seulement dans celle-ci
  - L'accès aux valeurs des colonnes se fait par la notation pointée
  - Il faut utiliser les attributs d'un curseur, si nécessaire

Exemple - Curseurs (I)

- Soit le SR suivant :



- Écrire un bloc PL/SQL qui permet d'afficher les villes où a travaillé l'employé ayant le matricule 7600

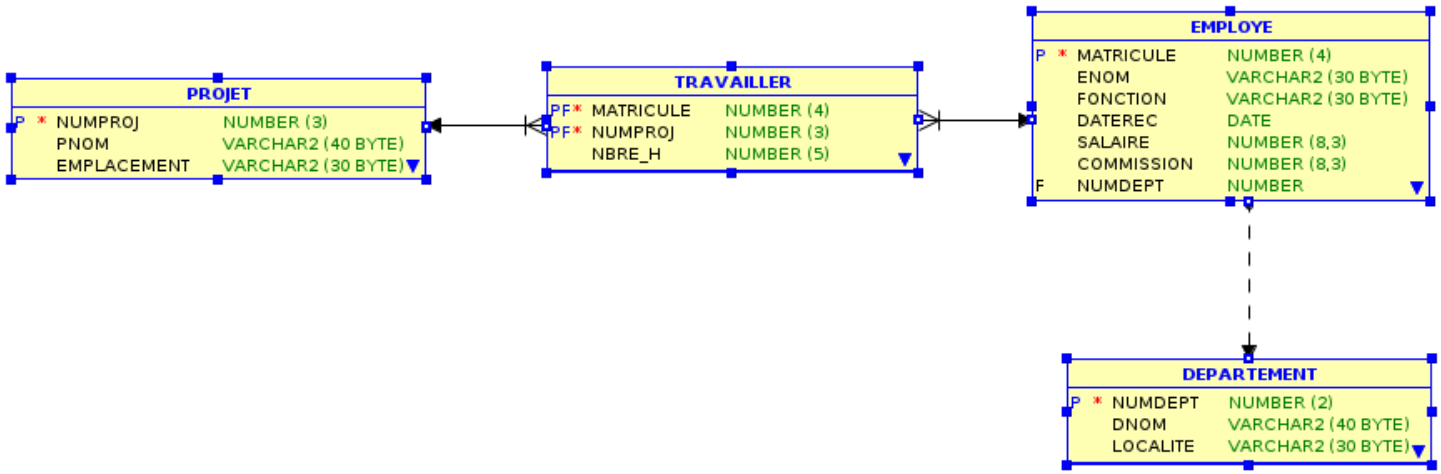
Exemple - Curseurs (I)

Solution (Parmi d'autres)

```
SET SERVEROUTPUT ON
DECLARE
-- déclaration du curseur
   CURSOR c IS SELECT DISTINCT EMPLACEMENT FROM TRAVAILLER T, PROJET P WHERE T
      .NUMPROJ = P.NUMPROJ AND MATRICULE=7600 ORDER BY EMPLACEMENT;
BEGIN
   DBMS_OUTPUT.PUT_LINE('L'employé de matricule 7600 a travaillé dans les
      villes :');
   FOR rec IN c LOOP -- lecture du curseur
      DBMS_OUTPUT.PUT_LINE('-- ' || rec.EMPLACEMENT);
   END LOOP;
END;
```

Exemple - Curseurs (II)

- Soit le SR suivant :



- Écrire un bloc PL/SQL qui permet d'afficher les noms et les salaires des employés ayant la fonction 'MANAGER'

Exemple - Curseurs (II)

Solution

```
SET SERVEROUTPUT ON
DECLARE
-- déclaration du curseur
    CURSOR emp_curseur IS SELECT ENOM, SALAIRE FROM EMPLOYE WHERE FONCTION = '
      MANAGER';
BEGIN
   -- chargement et accès aux éléments du curseur
   FOR rec1 in emp_curseur LOOP
    DBMS_OUTPUT.PUT_LINE( rec1.enom||' a le salaire '|| rec1.salaire);
   END LOOP;
END;
```

Curseurs FOR UPDATE

- Permet de baser une mise à jour sur les valeurs existant dans les lignes du curseur
  - Oracle verrouille les lignes à l'ouverture du curseur
  - Il empêche les modifications par d'autres utilisateurs avant la mise à jour
  - Il libère le verrou à la fin de la transaction

Syntaxe

```
-- Curseur FOR UPDATE
DECLARE
   CURSOR nom_curseur IS requête_select
                         FOR UPDATE
                         [OF nom_colonne [,...]]
                         [{NOWAIT | WAIT nbre_secondes}];
```

- `nom_colonne` désigne une ou plusieurs colonnes sur lesquelles est appliquée la clause `FOR UPDATE`
- `NOWAIT` : Ne pas attendre si les lignes sont verrouillées par une autre session
- `WAIT nbre_secondes` : Attendre le nombre de secondes spécifié

CURRENT OF

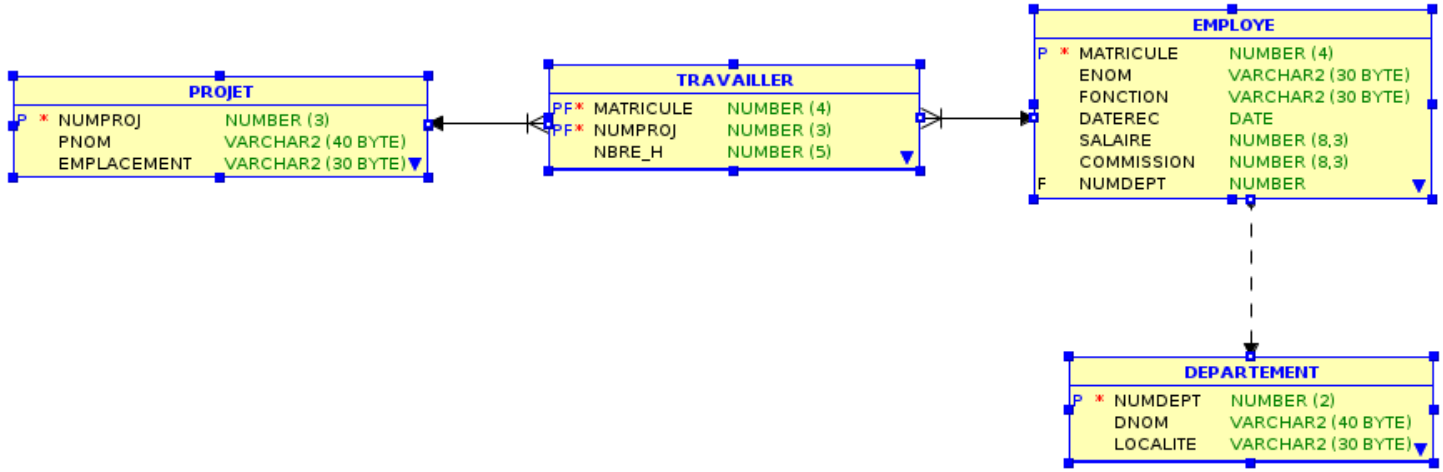
- Souvent utilisé avec `DELETE` ou `UPDATE`
- Permet de référencer la ligne courante du curseur
- Évite d'utiliser une condition `WHERE` sur la valeur d'un champ

Syntaxe

```
-- DELETE / UPDATE avec CURRENT OF
DELETE FROM ma_table WHERE CURRENT OF mon_curseur;
UPDATE ma_table SET mon_champ = nouvelle_valeur WHERE CURRENT OF mon_curseur;
```

Exemple

- Soit le SR suivant :



- Écrire un bloc `PL/SQL` qui utilise le curseur `FOR UPDATE` pour :
  - augmenter le salaire des `managers` de `20 %` ;
  - augmenter le salaire des `analystes` de `30 %` ;
  - affecter une commission de `10 %` du salaire aux `secrétaires`

Exemple

Solution

```
DECLARE
-- déclaration du curseur
   CURSOR emp_curseur IS SELECT * FROM EMPLOYE FOR UPDATE;
BEGIN
   FOR rec IN emp_curseur LOOP -- accès aux éléments du curseur
    -- mise à jour de la table employe
      IF rec.fonction= 'MANAGER' THEN UPDATE EMPLOYE SET SALAIRE = SALAIRE *
     1.2 WHERE CURRENT OF emp_curseur;
      ELSIF rec.fonction= 'ANALYSTE' THEN UPDATE EMPLOYE SET SALAIRE = SALAIRE
       * 1.3 WHERE CURRENT OF emp_curseur;
      ELSIF rec.fonction= 'SECRÉTAIRE' THEN UPDATE EMPLOYE SET COMMISSION =
     SALAIRE * 0.1 WHERE CURRENT OF emp_curseur;
      END IF;
   END LOOP;
END;
```

Le serveur Oracle libère le verrou à la fin de la transaction. Donc, vous ne pouvez pas faire l'opération de validation (`COMMIT`) ou d'annulation (`ROLLBACK`) entre deux `FETCH` ou avant la fermeture du curseur

On ne peut pas utiliser la commande `FETCH` sur un curseur `FOR UPDATE`, après avoir fait l'opération de validation (`COMMIT`) ou d'annulation (`ROLLBACK`). Le pointage dans le curseur est perdu

Exemple

```
DECLARE
-- déclaration du curseur
   CURSOR emp_curseur IS SELECT * FROM EMPLOYE FOR UPDATE;
BEGIN
    -- accès aux éléments du curseur
    FOR rec IN emp_curseur LOOP
        -- mise à jour de la table employe
        IF rec.fonction= 'MANAGER' THEN
            UPDATE EMPLOYE SET SALAIRE = SALAIRE * 0 WHERE CURRENT OF
     emp_curseur;
       END IF;
       -- validation
       COMMIT;
    END LOOP;
END;
```

Exemple

```
DECLARE
-- déclaration du curseur
   CURSOR emp_curseur IS SELECT * FROM EMPLOYE FOR UPDATE;
BEGIN
    -- accès aux éléments du curseur
    FOR rec IN emp_curseur LOOP
        -- mise à jour de la table employe
        IF rec.fonction= 'MANAGER' THEN
            UPDATE EMPLOYE SET SALAIRE = SALAIRE * 0 WHERE CURRENT OF
     emp_curseur;
        END IF;
        -- validation
        COMMIT;
    END LOOP;
END;
```

```
ORA-01002: fetch out of sequence ORA-06512: at line 6
ORA-06512: at line 6
ORA-06512: at "SYS.DBMS_SQL", line 1721
```

Curseurs paramétrés

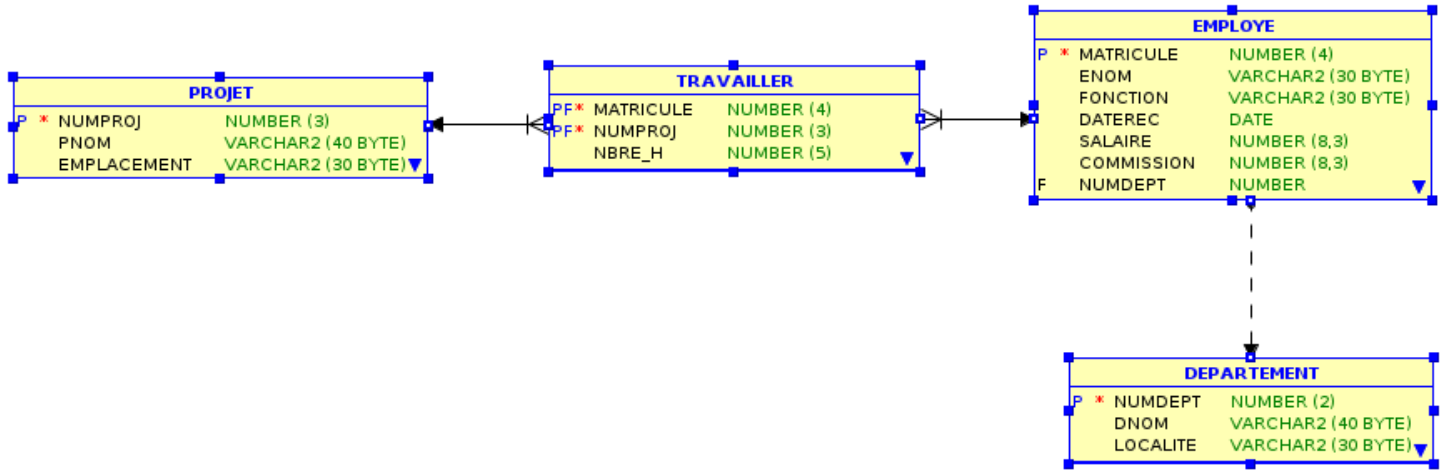
- Permettent une plus grande flexibilité
- Paramètres formels spécifiés dans la déclaration (nom et type)
- Paramètres effectifs (valeurs des paramètres) fournis lors de l'ouverture du curseur
- Réutilisable dans le même bloc avec différentes valeurs

Syntaxe

```
-- Curseur paramétré
CURSOR nom_curseur(nom_parametre type_donnees , ...) IS requête_select;
```

Exemple

- Soit le SR suivant :



- Écrire un bloc PL/SQL qui permet d'afficher les noms et les salaires des employés qui ont des salaires supérieurs à une valeur donnée (paramètre du curseur) et qui sont recrutés après une date donnée (paramètre du curseur avec une valeur par défaut)

Exemple

Solution

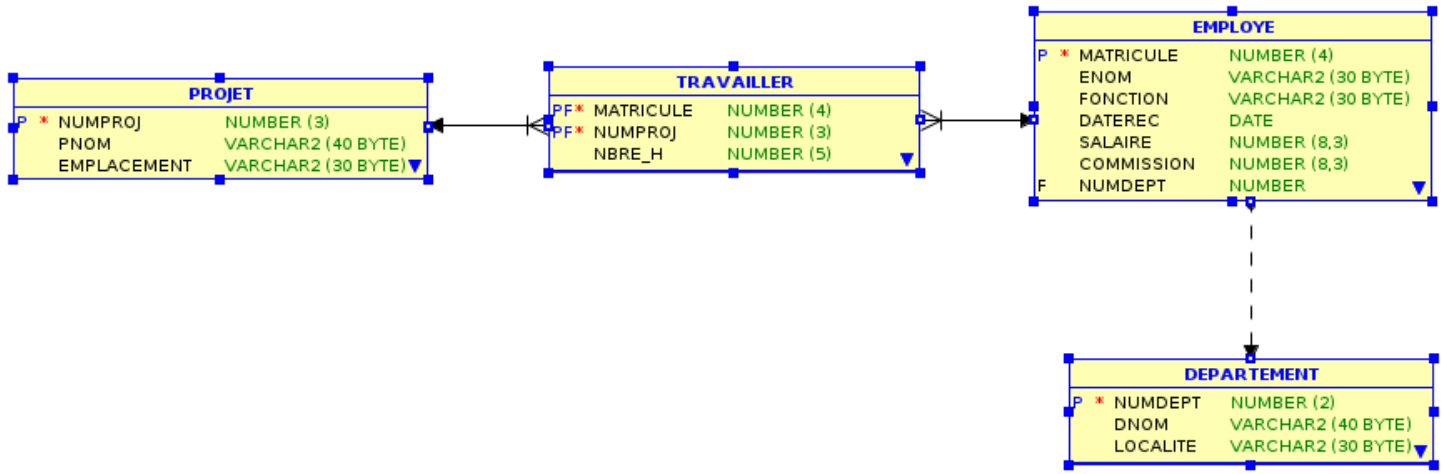
```
SET SERVEROUTPUT ON
DECLARE
   CURSOR c(max_sal NUMBER, date_rec DATE DEFAULT TO_DATE('20-10-1990','dd-mm-
      yyyy')) IS
   SELECT * FROM EMPLOYE WHERE SALAIRE > max_sal AND DATEREC > date_rec ORDER
      BY SALAIRE;
BEGIN
   FOR enr IN c(2000) LOOP
      DBMS_OUTPUT.PUT_LINE(enr.enom|| ' a le salaire ' || enr.salaire );
   END LOOP;
END;
```

Curseur local anonyme avec SELECT

- La commande **SELECT** agit comme un **curseur local anonyme**
  - Pas de déclaration explicite nécessaire
  - Aucun attribut accessible
  - Manipulation directe des résultats d'une requête SQL

Exemple

- Soit le SR suivant :



- Écrire un bloc PL/SQL qui permet d'afficher les matricules et les noms des employés du département administratif, sans qu'on utilise la déclaration d'un curseur

Exemple

Solution

```
SET SERVEROUTPUT ON
-- pas de déclaration du curseur
BEGIN
-- parcours du curseur
 DBMS_OUTPUT.PUT_LINE('Les personnels du département administratif  sont : ');
   FOR ligne IN ( SELECT MATRICULE, ENOM FROM EMPLOYE, DEPARTEMENT WHERE DNOM
      = 'ADMINISTRATIF' AND EMPLOYE.NUMDEPT = DEPARTEMENT.NUMDEPT)
   LOOP
    DBMS_OUTPUT.PUT_LINE('Matricule : ' || ligne.matricule || ' -- Nom : ' ||
      ligne.enom);
   END LOOP;
END;
```

Exceptions

Gestion des exceptions

- Une **exception** interrompt le déroulement normal d'un bloc PL/SQL
- La section optionnelle **EXCEPTION** permet de programmer le traitement des erreurs

Syntaxe

```
-- Traitement des exceptions
EXCEPTION
   WHEN exception_1 [OR exception_2...] THEN
     instruction_1;
     Instruction_2;
   ...
     WHEN exception3[OR exception4...] THEN
      Instruction_3;
      Instruction_4;
   ...
   [WHEN OTHERS THEN
      Instruction_n;
      Instruction_n+1;
   ...]
```

Types d'exceptions

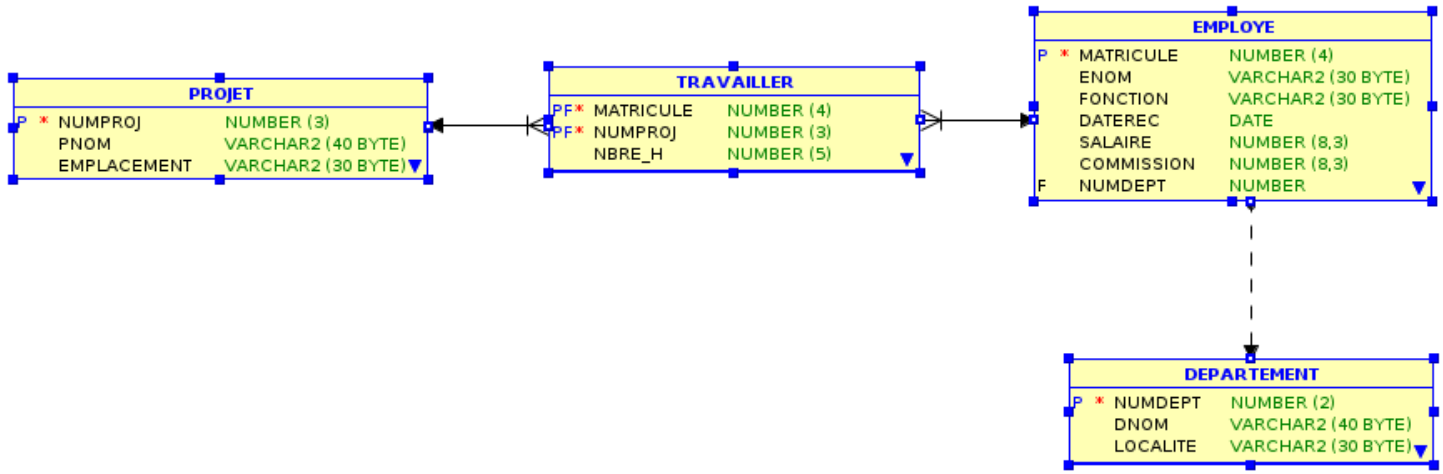
- Erreurs Oracle prédéfinies :
  - Exemples : `ORA-06500` (`STORAGE_ERROR`)
  - Non déclarées et déclenchées **automatiquement** par Oracle
- Erreurs Oracle non prédéfinies :
  - Déclarées dans la section déclarative
  - Déclenchées **implicitement** par Oracle
  - Utilisation de `PRAGMA EXCEPTION_INIT`
- Erreurs définies par l'utilisateur :
  - Gestion entièrement à la charge de l'utilisateur
  - Définies dans la section déclarative
  - Déclenchées **explicitement**

Quelques exceptions Oracle prédéfinies (STANDARD)

Nom de l'exception	Code erreur	Commentaires
ACCESS_INTO_NULL	-6530	Affectation de valeurs aux attributs d'un objet non initialisé
CASE_NOT_FOUND	-6592	Aucun des choix dans les clauses WHEN d'une instruction CASE n'a été sélectionné et il n'y a aucune clause ELSE
COLLECTION_IS_NULL	-6531	Application d'une méthode autre que EXISTS sur une collection NESTED table ou VARRAY non initialisée
DUP_VAL_ON_INDEX	-0001	Insertion des valeurs en double dans une colonne définie avec un index unique (créé par la commande CREATE UNIQUE INDEX ou par la contrainte PRIMARY KEY)
INVALID_NUMBER	-1722	Échec de conversion d'une chaîne de caractères en NUMBER
LOGIN_DENIED	-1017	Connexion avec un nom utilisateur ou un mot de passe incorrect
NO_DATA_FOUND	-1403	Requête ne retournant aucun résultat
NOT_LOGGED_ON	-1012	Appel à une base de données sans avoir une connexion Oracle
ROWTYPE_MISMATCH	-6504	Incompatibilité de types entre une variable externe et une variable PL/SQL
SELF_IS_NULL	-0625	Appel d'une méthode d'un type sur un objet NULL (extension objet)
SYS_INVALID_ROWID	-01410	Échec de conversion d'une chaîne de caractères en ROWID
TOO_MANY_ROWS	-01422	Requête SELECT retournant plusieurs lignes
VALUE_ERROR	-06502	Erreur arithmétique (conversion, troncature, taille) d'un NUMBER
ZERO_DIVIDE	-01476	Division par zéro

Exemple (I)

- Soit le SR suivant :



- Écrire un bloc PL/SQL qui permet d'afficher si un nom particulier existe dans la table Employé

Exemple (I)

Solution

```
SET SERVEROUTPUT ON
DECLARE
    num NUMBER;
    vnom VARCHAR2(30) := 'MARTIN';
BEGIN
    SELECT MATRICULE INTO num FROM EMPLOYE
    WHERE ENOM = vnom;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Aucun employé porte le nom ' || vnom);
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('Plusieurs employés portent le nom'||vnom);
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Il y a un problème...');
END;
```

Exemple (I)

Solution

```
SET SERVEROUTPUT ON
DECLARE
    num NUMBER;
    vnom VARCHAR2(30) := 'MARTIN';
BEGIN
    SELECT MATRICULE INTO num FROM EMPLOYE
    WHERE ENOM = vnom;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Aucun employé porte le nom ' || vnom);
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('Plusieurs employés portent le nom'||vnom);
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Il y a un problème...');
END;
```

Aucun employé ne porte le nom MARTIN

Exemple (II)

- Écrire un bloc PL/SQL qui permet d'afficher la racine carrée d'un nombre positif ou un message d'erreur

Exemple (II)

Solution

```
SET SERVEROUTPUT ON
DECLARE
  var NUMBER  := &svar;
BEGIN
  DBMS_OUTPUT.PUT_LINE(' Racine carrée de '||var||' est '||SQRT(var));
EXCEPTION
  WHEN VALUE_ERROR THEN
     DBMS_OUTPUT.PUT_LINE('Erreur... La valeur doit être un nombre positif');
END;
```

Exemple (II)

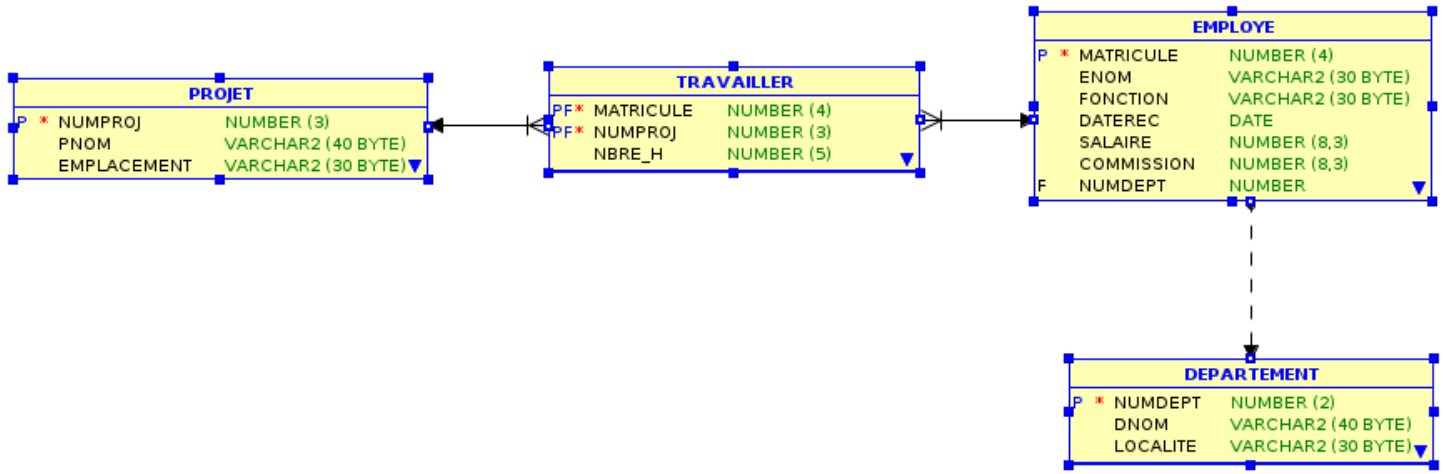
Solution

```
SET SERVEROUTPUT ON
DECLARE
  var NUMBER  := &svar;
BEGIN
  DBMS_OUTPUT.PUT_LINE(' Racine carrée de '||var||' est '||SQRT(var));
EXCEPTION
  WHEN VALUE_ERROR THEN
     DBMS_OUTPUT.PUT_LINE('Erreur... La valeur doit être un nombre positif');
END;
```

```
Entrez une valeur pour svar : -10
ancien : ... var NUMBER(10,2) := &svar; ...
nouveau : ... var NUMBER(10,2) := -10; ...
Erreur... La valeur doit être un nombre positif
```

Exemple (III)

- Soit le SR suivant :



- Écrire un bloc PL/SQL qui permet d'insérer un projet dans la table Projet. Si ce dernier existe déjà, le programme affiche un message d'erreur

Exemple (III)

Solution

```
SET SERVEROUTPUT ON
BEGIN
   INSERT INTO PROJET
   VALUES(10,'CONSTRUCTION D"UN CENTRE','PARIS');
EXCEPTION
WHEN DUP_VAL_ON_INDEX THEN
   DBMS_OUTPUT.PUT_LINE('Projet existe déjà');
END;
```

Exemple (III)

Solution

```
SET SERVEROUTPUT ON
BEGIN
   INSERT INTO PROJET
   VALUES(10,'CONSTRUCTION D"UN CENTRE','PARIS');
EXCEPTION
WHEN DUP_VAL_ON_INDEX THEN
   DBMS_OUTPUT.PUT_LINE('Projet existe déjà');
END;
```

Le projet existe déjà

Exceptions de l'utilisateur

Déclaration d'une exception

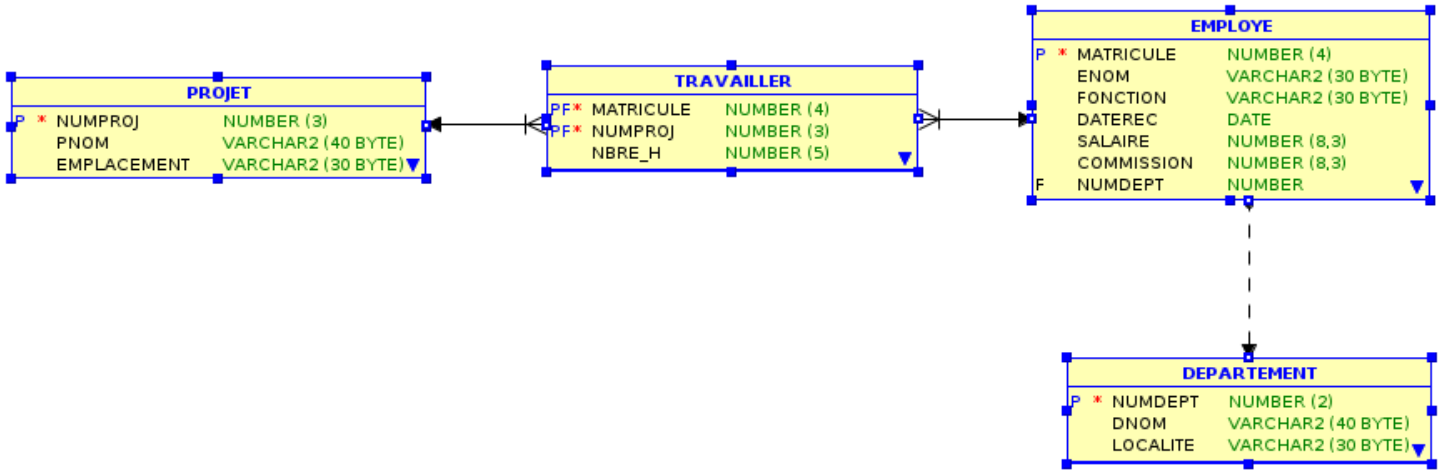
```
-- Déclaration d'une exception
nom_exception EXCEPTION;
```

Déclenchement d'une exception

```
-- Déclenchement d'une exception
RAISE nom_exception;
```

Exemple (I)

- Soit le SR suivant :



- Écrire un bloc PL/SQL qui permet de mettre à jour le nom d'un département. L'utilisateur entre le numéro du département et son nouveau nom. Si l'utilisateur entre un numéro du département qui n'existe pas, aucune ligne n'est mise à jour dans la table DEPARTEMENT. Une exception est produite et un message s'affiche pour avertir l'utilisateur qui a saisi un numéro de département incorrect

Exemple (I)

Solution

```
SET SERVEROUTPUT ON
DECLARE
   -- déclaration d'une exception
   excep_dept EXCEPTION;
BEGIN
   UPDATE DEPARTEMENT SET DNOM = '&nom_dept' WHERE NUMDEPT = &num_dept;
   -- déclenchement de l'exception
   IF SQL%NOTFOUND THEN RAISE excep_dept;
   END IF;
   COMMIT;
EXCEPTION
   -- traitement de l'exception
   WHEN excep_dept THEN DBMS_OUTPUT.PUT_LINE('Numéro département invalide');
END;
```

Exemple (I)

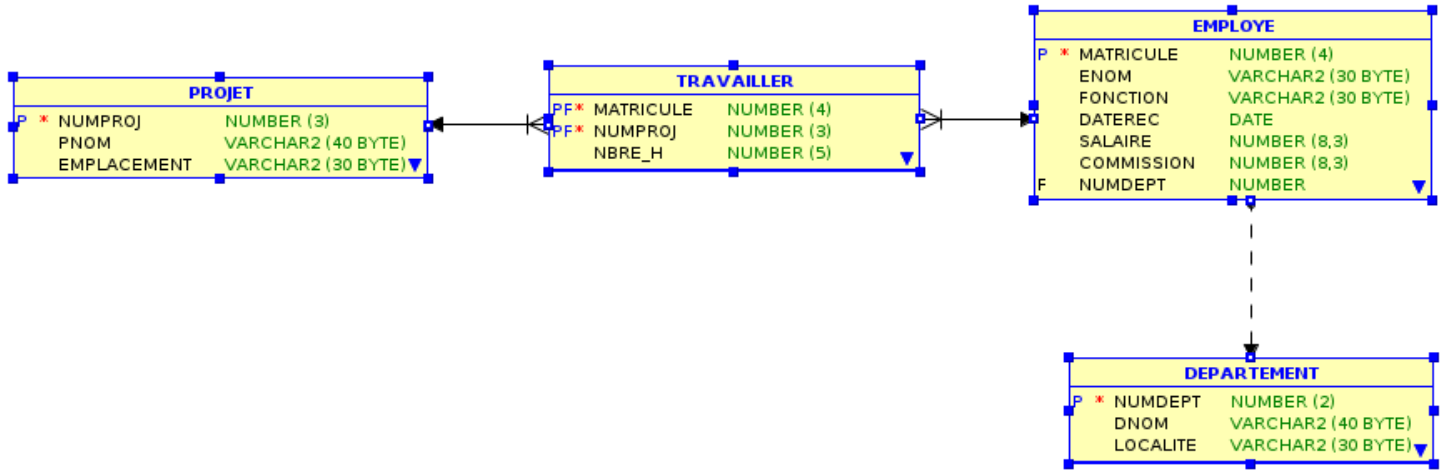
Solution

```
SET SERVEROUTPUT ON
DECLARE
   -- déclaration d'une exception
   excep_dept EXCEPTION;
BEGIN
   UPDATE DEPARTEMENT SET DNOM = '&nom_dept' WHERE NUMDEPT = &num_dept;
   -- déclenchement de l'exception
   IF SQL%NOTFOUND THEN RAISE excep_dept;
   END IF;
   COMMIT;
EXCEPTION
   -- traitement de l'exception
   WHEN excep_dept THEN DBMS_OUTPUT.PUT_LINE('Numéro département invalide');
END;
```

```
Entrez une valeur pour nom_dept : INFORMATIQUE
    ancien : ... SET DNOM = '&nom_dept' ...
 nouveau : ... SET DNOM = 'INFORMATIQUE' ...
    Entrez une valeur pour num_dept : 15
ancien : ... WHERE NUMDEPT = &num_dept; ...
   nouveau : ... WHERE NUMDEPT = 15; ...
        Numéro département invalide
```

Exemple (II)

- Soit le SR suivant :



- Écrire un bloc PL/SQL qui permet de calculer le nombre d'heures travaillées d'un employé particulier. L'utilisateur entre le numéro d'employé. Si l'utilisateur entre un numéro employé négatif, une exception est produite. Si l'utilisateur entre un numéro employé positif mais que ce numéro n'existe pas dans la table EMPLOYE, une autre exception est produite. Sinon, le programme affiche le nombre d'heures travaillées et affiche aussi qu'il n'y a aucune exception déclenchée

Exemple (II)

Solution

```
DECLARE
   num_emp EMPLOYE.MATRICULE%TYPE  := &smat;
   total_heure NUMBER;
   nbre NUMBER  := 0;
   excep_emp1 EXCEPTION;
   excep_emp2 EXCEPTION;
BEGIN
   IF num_emp< 0 THEN RAISE excep_emp1;
   ELSE SELECT COUNT(*) INTO nbre FROM EMPLOYE WHERE MATRICULE = num_emp;
      IF  nbre=0 THEN RAISE excep_emp2;
      ELSE SELECT SUM(NBRE_H) INTO total_heure FROM TRAVAILLER WHERE MATRICULE
       = num_emp;
      DBMS_OUTPUT.PUT_LINE('L'emplyé a travaillé '|| total_heure||' heures');
      END IF;
   END IF;
   DBMS_OUTPUT.PUT_LINE(' Aucune exception n'a été soulevée');
EXCEPTION
   WHEN excep_emp1 THEN DBMS_OUTPUT.PUT_LINE('La matricule doit être positive'
     );
   WHEN excep_emp2 THEN DBMS_OUTPUT.PUT_LINE('La matricule n'existe pas dans
     la table employé');
END;
```

Procédure RAISE_APPLICATION_ERROR

- Utilisée pour publier des messages et codes d'erreurs personnalisés (exceptions utilisateur)
- Évite le renvoi d'exceptions non traitées

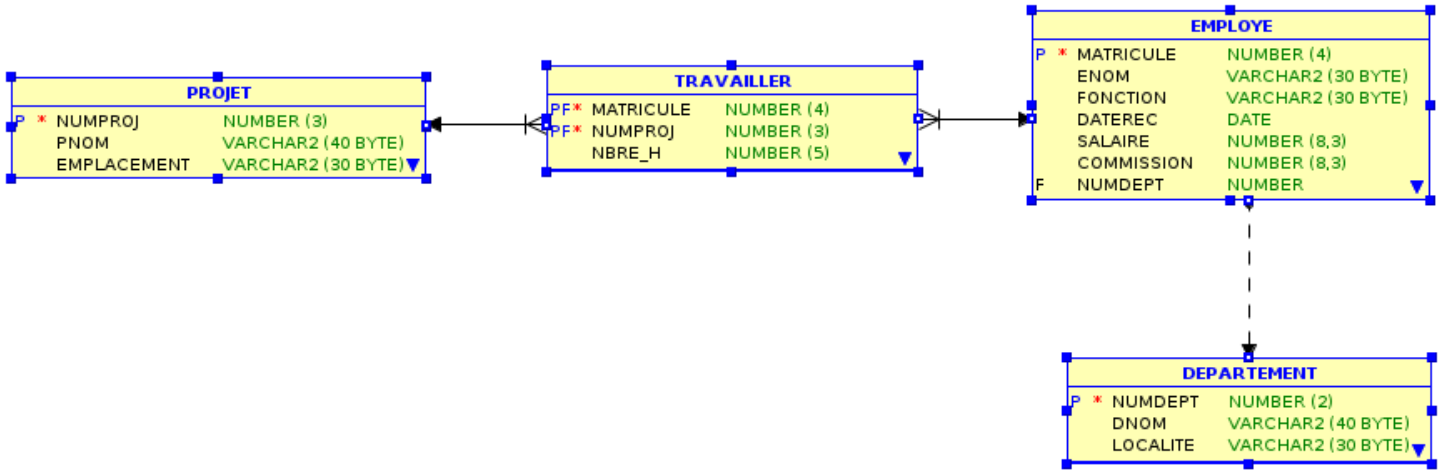
Syntaxe

```
-- RAISE_APPLICATION_ERROR
RAISE_APPLICATION_ERROR(numero_erreur, message
                                        [,{TRUE | FALSE}]);
```

- `numero_erreur` : code d'erreur entre `-20 999` et `-20 000`
- `message` : description de l'erreur (jusqu'à 2048 octets)
- `TRUE | FALSE` : optionnel, spécifie le comportement en cas de propagation en cascade des erreurs

Exemple (I)

- Soit le SR suivant :



- Écrire un bloc PL/SQL qui permet d'insérer un nouvel employé. Si le salaire introduit est inférieur à un seuil, annuler l'insertion et afficher un message d'erreur en utilisant la procédure `RAISE_APPLICATION_ERROR` dans la section `EXCEPTION`

Exemple (I)

Solution

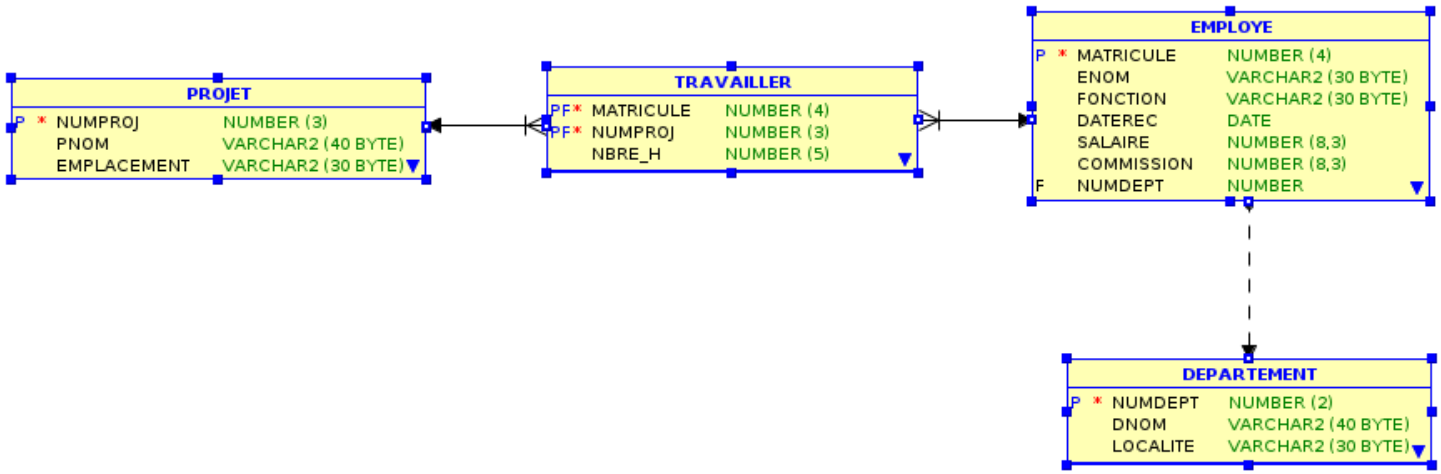
```
SET SERVEROUTPUT ON
DECLARE
   err_sal EXCEPTION;
   min_sal NUMBER := 1000;
   nov_sal NUMBER := 500;
BEGIN
   INSERT INTO EMPLOYE VALUES(7900, 'BERNARD', 'MANAGER', to_date('25-12-2015'
      ,'dd-mm-yyyy'), nov_sal, NULL, 20);
   IF nov_sal < min_sal THEN
    RAISE err_sal;
   END IF;
   COMMIT;
EXCEPTION
   WHEN err_sal THEN
      ROLLBACK;
      /* utilisation de la procédure RAISE_APPLICATION_ERROR dans la
      partie exception */
      RAISE_APPLICATION_ERROR(-20102, 'Salaire est inferieur a '||min_sal);
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Erreur...');
END;
```

Exemple (I)

```
ERREUR à la ligne 1 :
ORA-20102 : Salaire est inférieur à 1000
ORA-06512 : à ligne 21
```

Exemple (II)

- Soit le SR suivant :



- Récrire le bloc PL/SQL précédent, mais en utilisant la procédure RAISE_APPLICATION_ERROR dans la section exécutable

Exemple (II)

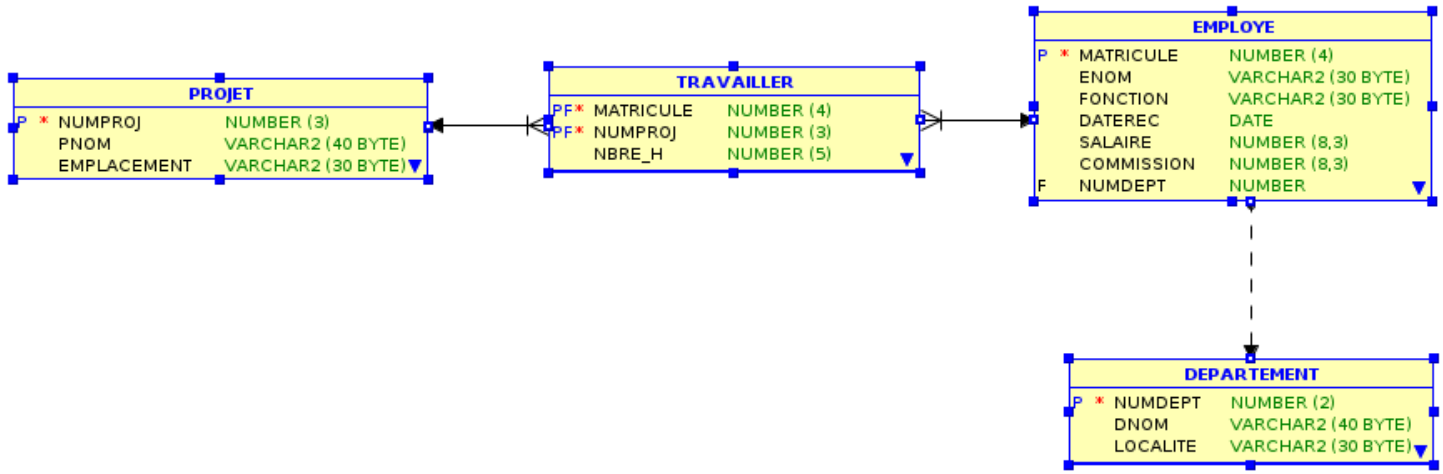
Solution

```
SET SERVEROUTPUT ON
DECLARE
   min_sal NUMBER := 1000;
   nov_sal NUMBER := 500;
BEGIN
   INSERT INTO EMPLOYE VALUES(7900, 'BERNARD', 'MANAGER', to_date('25-12-2015'
      ,'dd-mm-yyyy'), nov_sal, NULL, 20);
   IF nov_sal < min_sal THEN
       ROLLBACK;
       /* utilisation de la procédure RAISE_APPLICATION_ERROR dans la
       partie exécutable */
       RAISE_APPLICATION_ERROR(-20102, 'Salaire est inferieur a '||min_sal);

   END IF;
   COMMIT;
END;
```

Exemple (III)

- Soit le SR suivant :



- Écrire un bloc PL/SQL qui permet d'afficher si un nom particulier existe dans la table employé, en utilisant la procédure `RAISE_APPLICATION_ERROR` avec les exceptions prédéfinies

Exemple (III)

Solution

```
SET SERVEROUTPUT ON
DECLARE
    num NUMBER;
    vnom VARCHAR2(30) := 'MARTIN';
BEGIN
    SELECT MATRICULE INTO num FROM EMPLOYE WHERE ENOM = vnom;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
       RAISE_APPLICATION_ERROR(-20100,'Aucun employé porte le nom ' || vnom);
    WHEN TOO_MANY_ROWS THEN
       RAISE_APPLICATION_ERROR(-20101,'Plusieurs employés portent le nom'||
      vnom);
    WHEN OTHERS THEN
       DBMS_OUTPUT.PUT_LINE('Il y a un problème...');
END;
```

Déclencheurs

Déclencheurs (Trigger)

- Un ensemble d'instructions déclenché automatiquement par le SGBD
- Déclenché à chaque fois qu'un **événement particulier** se produit sur une **table** ou une **vue**
- Permet de programmer des règles de gestion non définies par des **contraintes au niveau des tables**
- Une **table** ou **vue** peut avoir plusieurs **déclencheurs** ou aucun

Événements de déclenchement

- Déclencheurs **LMD** :
  - Avant ou après `INSERT`, `UPDATE` ou `DELETE`
- Déclencheurs **LDD** :
  - Avant ou après `CREATE`, `ALTER` ou `DROP` sur des objets
- Déclencheurs d'**instance** :
  - Événements système, démarrage/arrêt de la base de données (`startup` ou `shutdown`), (`NO_DATA_FOUND`, `DUP_VAL_ON_INDEX`, etc.), connexions/déconnexions
- Déclencheurs de **sécurité** :
  - Ouverture et fermeture de session utilisateur
- Applications :
  - Prévenir les transactions invalides
  - Vérifier des contraintes d'intégrité complexes
  - Appliquer des règles de gestion avancées
  - Renforcer la sécurité
  - Générer automatiquement des valeurs de colonnes
  - Compléter des procédures d'audit `Oracle`

Classification des déclencheurs

❖ Type d'événement :

- ☞ `INSERT`
- ☞ `UPDATE`
- ☞ `DELETE`

❖ Moment de l'exécution :

- ☞ `BEFORE` : **avant** l'événement
- ☞ `AFTER` : **après** l'événement

❖ Événements non atomiques :

- ☞ Triggers de **table** (`STATEMENT`)
- ☞ Triggers de **ligne** (`ROW`)

❖ Utilisation :

- ☞ Trigger de **ligne** : pour chaque ligne affectée par l'événement
- ☞ Trigger de **table** : une seule exécution, opérations de groupe

Équivalents lorsque la requête manipule une seule ligne

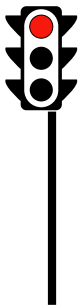
Création d'un déclencheur I

Syntaxe

```
-- Déclencheur LMD
CREATE [OR REPLACE ] TRIGGER nom_trigger
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name] ON nom_table
[REFERENCING OLD AS nomvieux NEW AS nomnew]
[FOR EACH ROW]
 WHEN (condition)
BEGIN
   --- instructions
END;
```

- `BEFORE | AFTER | INSTEAD OF` : Chronologie entre les instructions du déclencheur et l'événement
- `INSERT [OR] | UPDATE [OR] | DELETE` : Requêtes SQL déclenchant le déclencheur
- `OF col_name` : Colonne associée au déclencheur
- `ON nom_table` : Table associée au déclencheur
- `REFERENCING` : Renommer des variables
- `FOR EACH ROW` : Différencie les déclencheurs LMD (au niveau ligne ou table)
- `WHEN (condition)` : Condition pour exécuter le corps du déclencheur

Création d'un déclencheur II

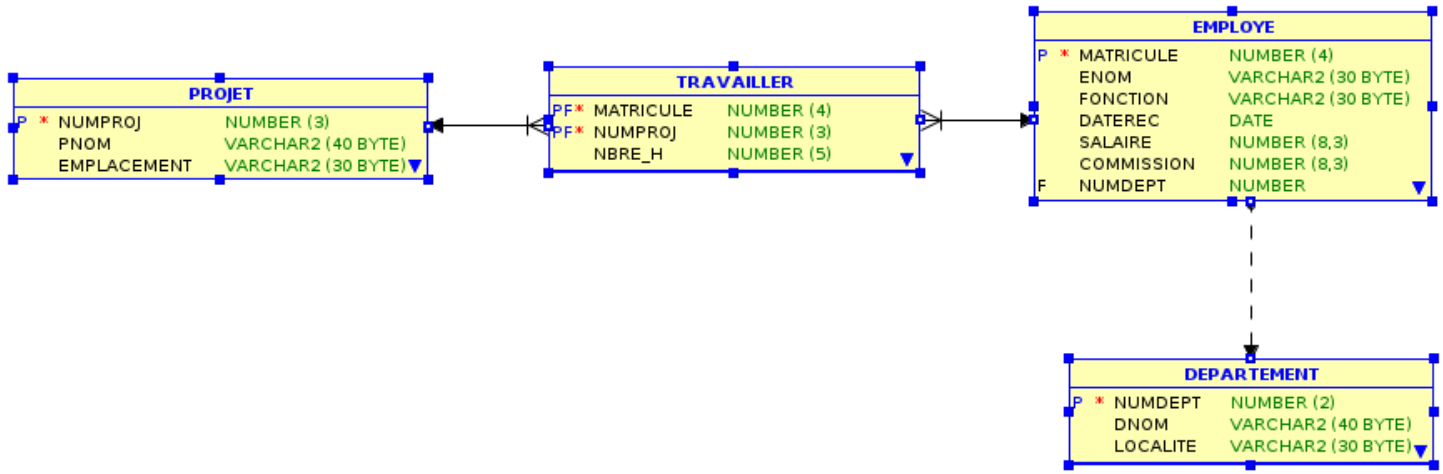


Un **trigger** ne peut valider aucune transaction. Ainsi, les instructions : `COMMIT`, `ROLLBACK`, `SAVEPOINT` et `SET CONSTRAINT`, sont interdites

- Il est possible de **combiner plusieurs événements** dans le même **trigger**, appelé **événement composé**
  - Exemple : `INSERT OR UPDATE OR DELETE` déclenche le **trigger** sur l'une de ces opérations
  - Pour identifier l'événement qui a déclenché le **trigger**, on utilise les conditions prédéfinies :
    - `INSERTING` : True lors d'une opération d'`INSERT`
    - `UPDATING` : True lors d'une opération d'`UPDATE`
    - `DELETING` : True lors d'une opération de `DELETE`

Exemple (I)

- Soit le SR suivant :



- Créer un trigger de façon à n'autoriser les opérations de mise à jour (`INSERT`, `UPDATE` ou `DELETE`) dans la table `EMPLOYE` que pendant une plage horaire fixe de la semaine

Exemple (I)

Solution

```
CREATE OR REPLACE TRIGGER horaire_travail_emp
BEFORE INSERT OR UPDATE OR DELETE ON EMPLOYE
BEGIN
   IF (TO_CHAR(SYSDATE,'DY') IN('SAM', 'DIM')) OR (TO_CHAR(SYSDATE,'HH24') NOT
       BETWEEN 8 AND 18) THEN
      IF INSERTING THEN RAISE_APPLICATION_ERROR(-20101, 'Insertion interdite à
       cette heure');
      ELSIF DELETING THEN RAISE_APPLICATION_ERROR(-20102, 'Suppression
      interdite à cette heure');
      ELSIF UPDATING('SALAIRE') THEN RAISE_APPLICATION_ERROR(-20103, '
      Modification salaire interdite à cette heure');
      ELSE RAISE_APPLICATION_ERROR(-20104, 'Modification interdite à cette
      heure');
      END IF;
   END IF;
END;
```

Exemple (I)

Il est 19:30

```
UPDATE EMPLOYE
SET SALAIRE = SALAIRE * 1.1;
```

Exemple (I)

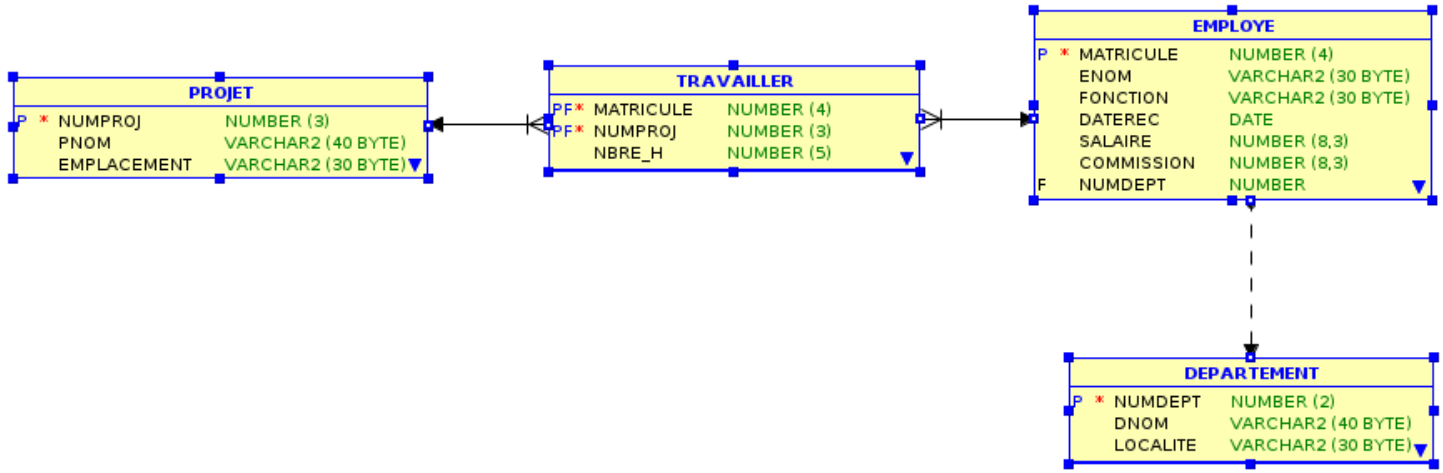
Il est 19:30

```
UPDATE EMPLOYE
SET SALAIRE = SALAIRE * 1.1;
```

```
                    UPDATE EMPLOYE
                           *
                    ERREUR à la ligne 1 :
ORA-20103 : Modification salaire interdite à cette heure
   ORA-06512 : à "MFS.HORAIRE_TRAVAIL_EMP", ligne 12
ORA-04088 : erreur lors d'exécution du déclencheur 'MFS.HORAIRE_TRAVAIL_EMP'
```

Exemple (II)

- Soit le SR suivant :



- Créer un trigger qui enregistre dans une table `AUDIT_TABLE` la trace de la modification de la table `EMPLOYE`. On mémorise ici le moment de la modification et l'utilisateur qui l'a provoqué. Le trigger n'est donc exécuté qu'une seule fois par modification de la table `EMPLOYE`

Exemple (II)

Solution

Dans ce cas, on va avoir besoin d'une table d'audit pour enregistrer la trace :

```
CREATE TABLE AUDIT_TABLE
(
   NOM_TABLE VARCHAR2(30),
   DATE_OP DATE,
   UTILISATEUR VARCHAR2(30),
   ACTION VARCHAR2(30)
);
```

```
CREATE OR REPLACE TRIGGER tlog
AFTER INSERT OR UPDATE ON EMPLOYE
BEGIN
   INSERT INTO AUDIT_TABLE
   VALUES ('EMPLOYE', SYSDATE, sys_context('USERENV', 'CURRENT_USER'), 'INSERT
     /UPDATE ON EMPLOYE');
END;
```

Utilisation des variables OLD et NEW dans les déclencheurs

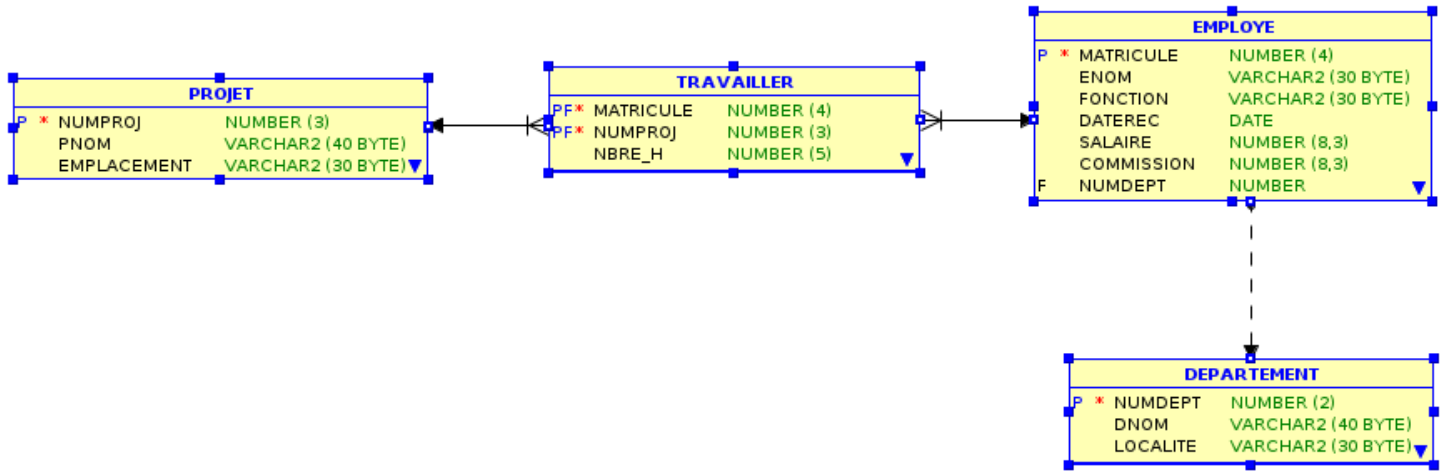
- Les variables `OLD` et `NEW` sont utilisables uniquement dans les **triggers de lignes** (`FOR EACH ROW`)
- Référence aux valeurs avant et après la modification :
  - `OLD` : Valeur avant la modification
  - `NEW` : Valeur après la modification
- On peut faire référence aux colonnes en les préfixant avec `:NEW.nom_colonne` ou `:OLD.nom_colonne`

Opération	Valeur OLD	Valeur NEW
INSERT	NULL	Valeur insérée
UPDATE	Valeur avant modification	Valeur après modification
DELETE	Valeur avant suppression	NULL

Vous pouvez limiter l'action d'un trigger de lignes en indiquant une condition dans la clause WHEN ➡ Pas de préfixe :

Exemple (I)

- Soit le SR suivant :



- Créer un trigger qui permet d'empêcher la diminution du salaire d'un employé

Exemple (I)

Solution

```
CREATE OR REPLACE TRIGGER pas_dim_salaire
BEFORE UPDATE ON EMPLOYE
FOR EACH ROW -- obligatoire si on utilise OLD et NEW
BEGIN
   IF (:OLD.SALAIRE > :NEW.SALAIRE)  THEN
      RAISE_APPLICATION_ERROR(-20105,'Pas de diminution de salaire !');
   END IF;
END;
```

```
UPDATE EMPLOYE
SET SALAIRE = SALAIRE * 0.9;
```

Exemple (I)

Solution

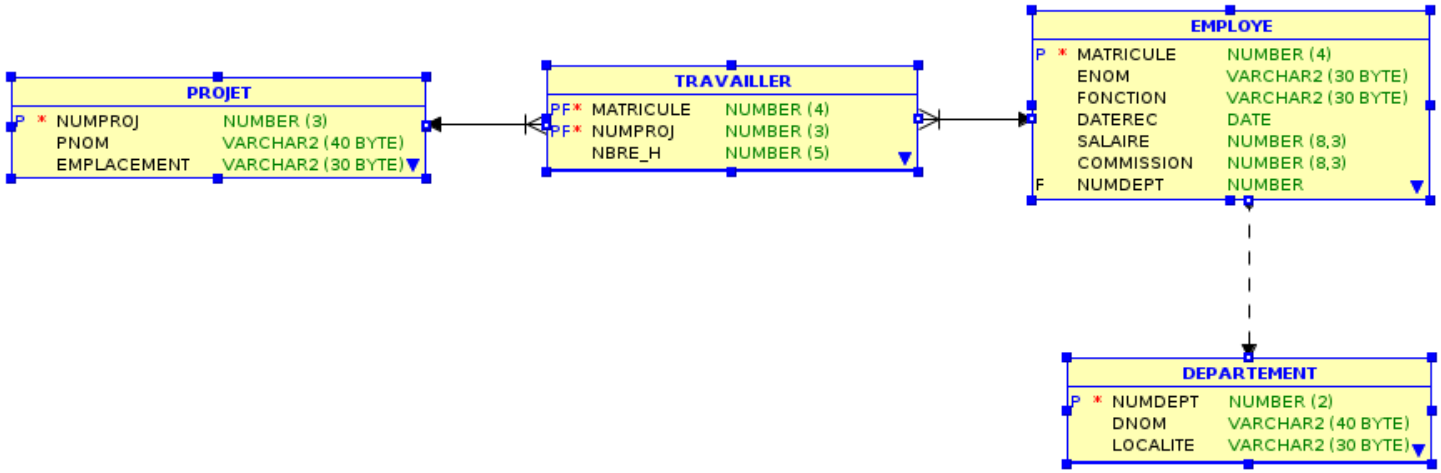
```
CREATE OR REPLACE TRIGGER pas_dim_salaire
BEFORE UPDATE ON EMPLOYE
FOR EACH ROW -- obligatoire si on utilise OLD et NEW
BEGIN
   IF (:OLD.SALAIRE > :NEW.SALAIRE)  THEN
      RAISE_APPLICATION_ERROR(-20105,'Pas de diminution de salaire !');
   END IF;
END;
```

```
UPDATE EMPLOYE
SET SALAIRE = SALAIRE * 0.9;
```

```
UPDATE EMPLOYE
*
ERREUR à la ligne 1 :
ORA-20105 : Pas de diminution de salaire !
ORA-06512 : à "MFS.PAS_DIM_SALAIRE", ligne 4
ORA-04088 : erreur lors d'exécution du déclencheur 'MFS.PAS_DIM_SALAIRE'
```

Exemple (II)

- Soit le SR suivant :



- Créer un trigger qui permet d'affecter une valeur à la commission d'un employé ayant la fonction `'DIRECTEUR'`. Cette affectation est la suivante :
  - `commission = 0`, si l'opération est l'insertion ou l'opération est la modification et l'ancienne valeur de la commission est `NULL`
  - `commission =` $\frac{\texttt{ancienne valeur du salaire}}{\texttt{nouvelle valeur du salaire}}$, si l'opération est la modification et la commission est `NOT NULL`

Exemple (II)

Solution

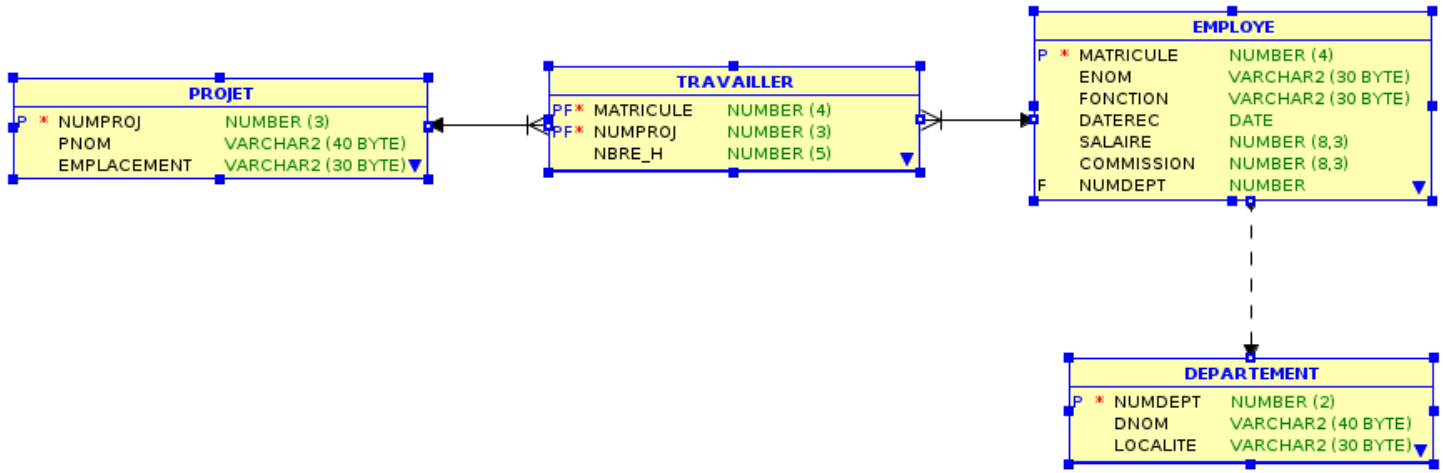
```
CREATE or REPLACE TRIGGER derive_commission
BEFORE UPDATE OR INSERT OF SALAIRE, FONCTION ON EMPLOYE
FOR EACH ROW
WHEN (NEW.FONCTION = 'DIRECTEUR')
BEGIN
   IF INSERTING THEN
      :NEW.COMMISSION := 0;
   ELSE
      IF :OLD.COMMISSION IS NULL THEN
         :NEW.COMMISSION := 0;
      ELSE
         :NEW.COMMISSION :=(:NEW.SALAIRE / :OLD.SALAIRE);
     END IF;
  END IF;
END;
```

Déclencheur INSTEAD OF

- Un **trigger LMD** permettant la mise à jour d'une vue multitable
- Utilisé lorsque la vue ne peut pas être modifiée directement par `INSERT`, `UPDATE` ou `DELETE`
- Le déclencheur effectue des actions au lieu d'**insérer**, de **modifier** ou de **supprimer** une vue
- Caractéristiques :
  - Utilise la clause `FOR EACH ROW` (implicitement)
  - S'applique uniquement sur des vues
  - Ne nécessite pas les options `BEFORE` et `AFTER`
  - Non conditionnel
  - Peut lire des valeurs anciennes et nouvelles

Exemple

- Soit le SR suivant :



- Créer la vue `travail_emp` pour afficher les informations sur les employés, leurs départements, leurs projets et leurs nombres des heures de travail pour chaque projet. Ensuite, créer un déclencheur `INSTEAD OF` pour traiter l'instruction `INSERT` destinées à la vue. Le déclencheur insère des lignes dans les tables de base de la vue

Exemple

Solution

```
CREATE OR REPLACE VIEW travail_emp AS SELECT EMPLOYE.MATRICULE, EMPLOYE.ENOM,
      EMPLOYE.FONCTION, PROJET.NUMPROJ, PROJET.PNOM, DEPARTEMENT.NUMDEPT,
      DEPARTEMENT.DNOM, TRAVAILLER.NBRE_H FROM EMPLOYE, TRAVAILLER, PROJET,
      DEPARTEMENT WHERE EMPLOYE.MATRICULE = TRAVAILLER.MATRICULE AND PROJET.
      NUMPROJ = TRAVAILLER.NUMPROJ AND EMPLOYE.NUMDEPT = DEPARTEMENT.NUMDEPT;
CREATE OR REPLACE TRIGGER travail_emp_insert INSTEAD OF INSERT ON travail_emp
BEGIN
   INSERT INTO DEPARTEMENT(NUMDEPT, DNOM) VALUES (:NEW.NUMDEPT,:NEW.DNOM);
   INSERT INTO PROJET(NUMPROJ, PNOM) VALUES (:NEW.NUMPROJ,:NEW.PNOM);
   INSERT INTO EMPLOYE (MATRICULE, ENOM, FONCTION, NUMDEPT) VALUES (:NEW.
     MATRICULE,:NEW.ENOM,:NEW.FONCTION,:NEW.NUMDEPT);
   INSERT INTO TRAVAILLER VALUES (:NEW.MATRICULE,:NEW.NUMPROJ,:NEW.NBRE_H);
   EXCEPTION
     WHEN DUP_VAL_ON_INDEX THEN RAISE_APPLICATION_ERROR(-20107,'Duplicate info
      ');
END travail_emp_insert;
```

Restrictions des tables de mutation

- Une **table de mutation** est une table en cours de modification par une instruction **LMD**
- Restrictions :
  - Pas d'interrogation ni de modification pendant cette opération
  - Manipulation interdite sur la table du déclencheur dans son corps
  - Erreur `ORA-04091` en cas de violation
- Concerne les déclencheurs de ligne (`FOR EACH ROW`)
- S'applique également aux déclencheurs de table avec `ON DELETE CASCADE`

Si l'événement `INSERT` insère une seule ligne, les déclencheurs `BEFORE/AFTER ...FOR EACH ROW` ne considèrent pas la **table en mutation**

Déclencheurs LDD

- Cas d'utilisations :
  - Empêcher certaines modifications sur un schéma ou une base de données
  - Déclencher un événement en réponse à une modification du schéma
  - Enregistrer des modifications ou des événements dans le schéma ou la base de données

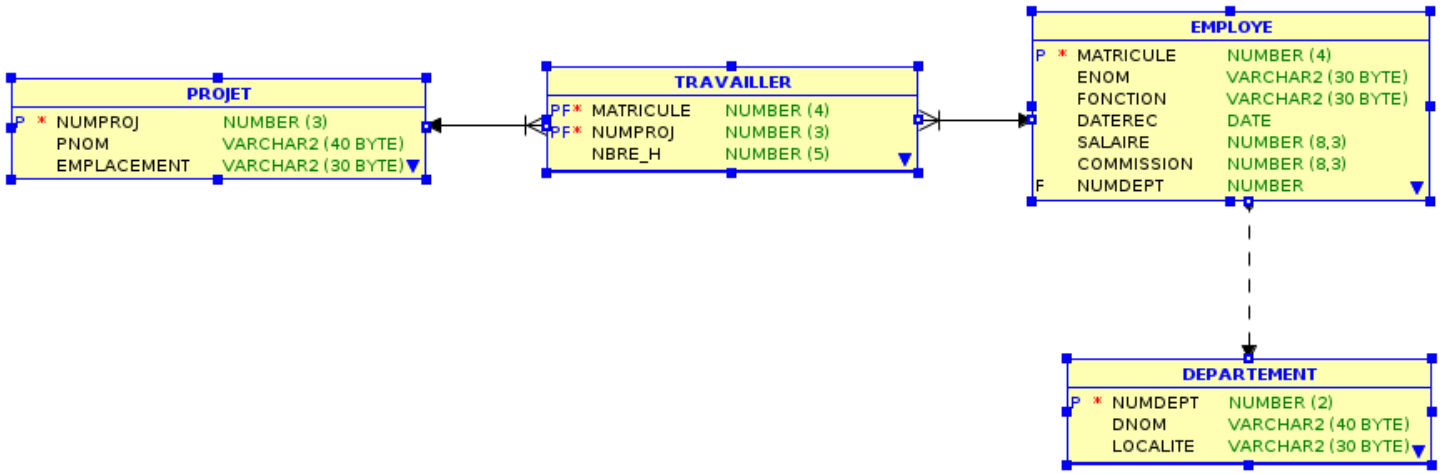
Syntaxe d'un déclencheur LDD

```
-- Déclencheur LDD
CREATE [OR REPLACE] TRIGGER [schéma.] nom_trigger
BEFORE | AFTER
  { action_structure_base [OR action_structure_base]... }
ON { [schéma.] SCHEMA | DATABASE }
Bloc PL/SQL(DECLARE variables BEGIN instructions END ; )
```

- `CREATE`, `ALTER`, `DROP`, `GRANT`, `RENAME`, `REVOKE`, `TRUNCATE`
- Utilisation de `BEFORE` ou `AFTER` pour spécifier le moment de l'exécution
- Directive `SCHEMA | DATABASE` pour préciser la source du déclencheur

Exemple

- Soit le SR suivant :



- Créer un déclencheur qui permet d'interdire toute suppression se produisant un samedi ou un dimanche

Exemple

Solution

```
CREATE OR REPLACE TRIGGER interdire_suppression
BEFORE DROP ON DATABASE
BEGIN
   IF TO_CHAR(SYSDATE,'DAY') IN ('SAMEDI','DIMANCHE') THEN
      RAISE_APPLICATION_ERROR (-20106, 'Désolé pas de destruction ce jour...')
      ;
   END IF;
END;
```

Activation et désactivation des Triggers

- Activation par défaut lors de la création
- Un trigger désactivé continue d'exister

Désactivation

```
ALTER TRIGGER nom_trigger DISABLE;
```

Réactivation

```
ALTER TRIGGER nom_trigger ENABLE;
```

Activation ou désactivation de tous les triggers

```
ALTER TRIGGER [ ENABLE | DISABLE ] ALL TRIGGERS;
```

Compilation d'un Trigger

- Compilation automatique lors de la création
- Le statut peut devenir invalide

Re-compilation

```
ALTER TRIGGER nom_trigger COMPILE;
```

Erreurs de compilation

```
SHOW ERRORS TRIGGER nom_trigger;
```

Suppression d'un Trigger

Suppression d'un trigger

```
DROP TRIGGER nom_trigger;
```

- Si une table est supprimée, tous les triggers associés sont également supprimés
- La suppression d'un déclencheur supprime les informations le concernant

Recherche des Triggers

- Les triggers stockés sont des éléments de la base de données
- La vue `USER_TRIGGERS` contient des informations sur les triggers

Afficher la structure de la vue `USER_TRIGGERS`

```
DESC USER_TRIGGERS;
```

Afficher les noms et les types de tous les triggers

```
SELECT TRIGGER_NAME, TRIGGER_TYPE FROM USER_TRIGGERS;
```

Afficher le code du trigger `PAS_DIM_SALAIRE`

```
SELECT TRIGGER_NAME, TRIGGER_TYPE, TRIGGER_BODY FROM USER_TRIGGERS WHERE
      TRIGGER_NAME = 'PAS_DIM_SALAIRE';
```

Fonctions et procédures

Sous-programmes en PL/SQL

- Un sous-programme est un bloc PL/SQL portant un nom
- Une fois pré-compilé et stocké sur le serveur Oracle, il devient un objet de base de données
- Unité de traitement avec des paramètres en entrée et en sortie
- Assure la **modularité**, l'**extensibilité**, la **réutilisation**, et la **facilité de maintenance** des applications
- Dans l'environnement Oracle, un **sous-programme** est appelé « **fonction cataloguée** » ou « **procédure cataloguée** » (**stockée dans la base de données**)
  - Les **procédures** réalisent des actions
  - Les **fonctions** retournent un **unique résultat**
  - Seule la **procédure** peut avoir **plusieurs paramètres en sortie** (recommandé)

Avantages de l'utilisation des sous-programmes

- Simplification et réutilisabilité :
  - Décomposition d'un problème en sous-problèmes faciles à résoudre
    - Lisibles et faciles à comprendre
    - Faciles à maintenir : détection rapide d'erreurs, corrections aisées
    - Faciles à faire évoluer : ajout de fonctionnalités
    - Réutilisables : résolution répétée d'une suite d'actions
- Performance :
  - Créés et compilés une seule fois
  - Exécutions ultérieures sans nouvelle vérification de syntaxe
  - Allègement des échanges client-serveur en stockant les procédures fréquemment utilisées au niveau du serveur
- Sécurité :
  - Objets de la base de données `Oracle` avec droits d'accès
  - Autorisation d'utilisation sans accès direct aux tables utilisées

Création d'une procédure

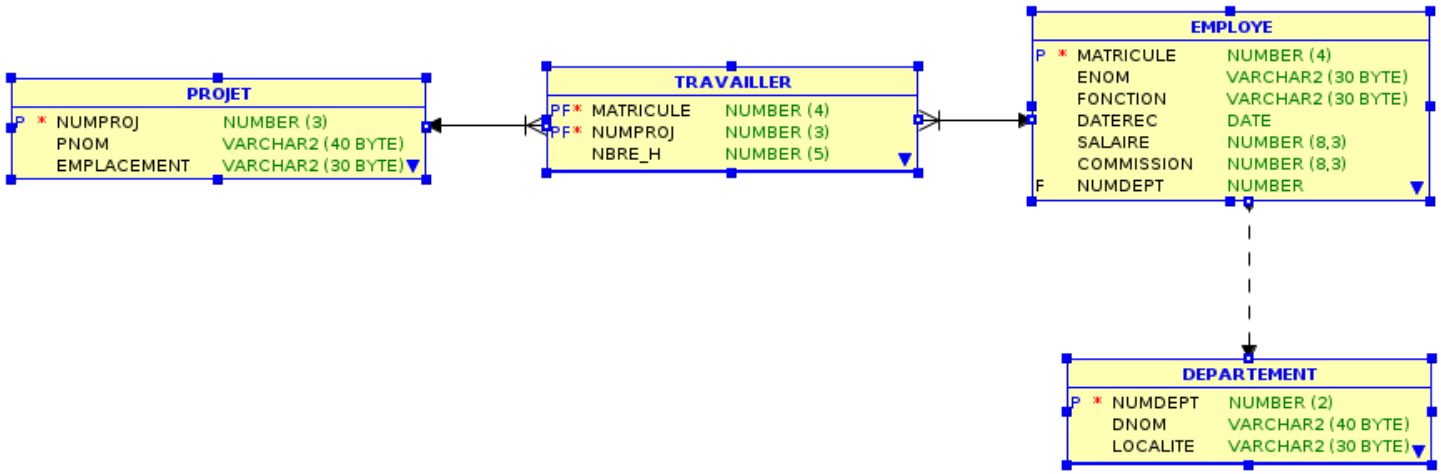
Syntaxe

```
CREATE [OR REPLACE] PROCEDURE nom_procedure[(parametre[parametre...])] { IS |
      AS }
   /* Déclarations locales de : variables, constantes...*/
   [<Section déclaration locale>]
   BEGIN
      ...
      ...
  [EXCEPTION
      ...
      ...
  ]
END [nom_procedure];
```

- L'exécution de la requête **CREATE PROCEDURE** déclenche :
  - **Compilation du code source :** génération de pseudocode si aucune erreur n'est détectée
  - **Stockage du code source :** même en cas de détection d'une erreur, le code source est stocké dans la base
  - **Stockage du pseudocode :** pour éviter la recompilation à chaque appel, le pseudocode est stocké dans la base de données

Exemple

- Soit le SR suivant :



- Écrire une procédure `maj_commision` qui permet d'augmenter de `10%` la commission des employés qui ont travaillé plus que `400` heures

Exemple

Solution

```
CREATE OR REPLACE PROCEDURE maj_commission
IS
   CURSOR cur_emp IS SELECT MATRICULE, SUM(NBRE_H) FROM TRAVAILLER GROUP BY
     MATRICULE HAVING SUM(NBRE_H)>400;
BEGIN
   FOR c IN cur_emp LOOP
      UPDATE EMPLOYE
      SET COMMISSION = COMMISSION * 1.1
      WHERE MATRICULE = c.MATRICULE;
   END LOOP;
END maj_commission;
```

La procédure nommée `maj_commission` est compilée et stockée dans la base de données pour une exécution ultérieure

Appel de procédures stockées

- L'appel des procédures peut se faire de différentes manières :
  - Dans le corps d'une autre **procédure**, d'une **fonction**, ou d'un **trigger**
  - Par elle-même (**récursivité**)
  - En utilisant un **bloc PL/SQL anonyme**
  - En utilisant la commande **EXECUTE**
  - Dans une application (programme hôte utilisant des ordres SQL imbriqués)

Exécuter la procédure `maj_commission`, en utilisant la commande EXECUTE

```
EXECUTE maj_commission
```

Exécuter la procédure `maj_commission`, en utilisant un bloc PL/SQL anonyme

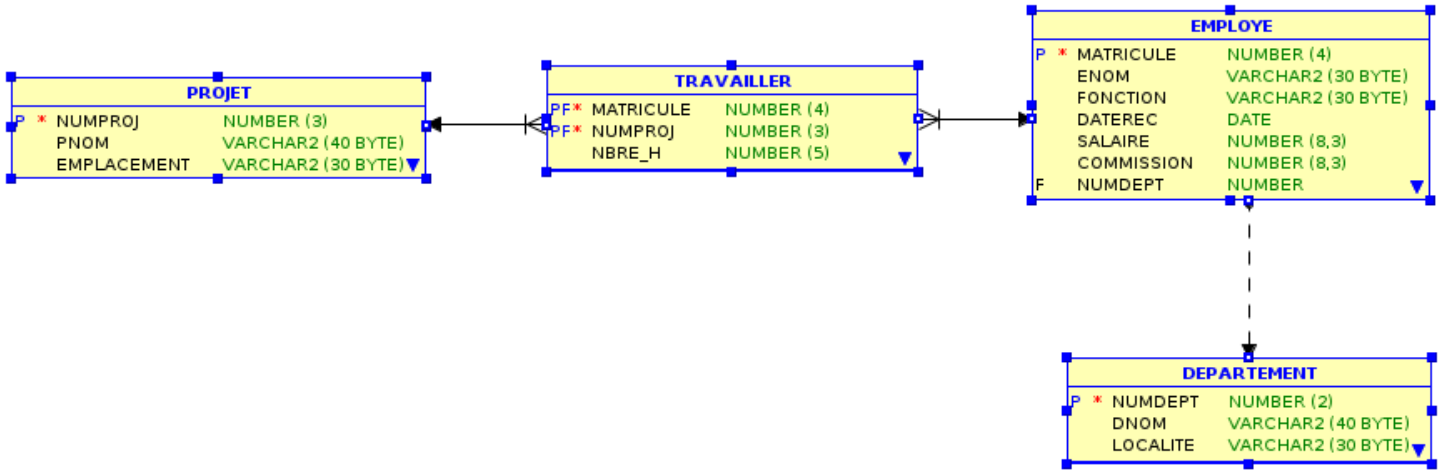
```
BEGIN
   maj_commission;
END;
```

Procédures locales

- En plus des procédures stockées, PL/SQL prend en charge les procédures locales
- Les procédures locales sont déclarées dans le corps d'un bloc PL/SQL sans utiliser **CREATE PROCEDURE**
- Elles sont spécifiques à ce bloc et ne peuvent pas être appelées en dehors de celui-ci
- Utiles pour encapsuler des logiques spécifiques à un bloc sans créer d'objets de base de données
- Déclarées à l'intérieur de la section DECLARE d'un bloc PL/SQL (**regarder l'exemple suivant**)

Exemple

- Soit le SR suivant :



- Écrire un bloc `PL/SQL` comportant une procédure `modifier_salaire` qui permet de modifier le salaire d'un employé particulier, puis utiliser cette procédure dans le bloc

Exemple

Solution

```
DECLARE
   mat NUMBER(6)   := 7800;
   bonus NUMBER(6) := 100;
   merit NUMBER(4) := 50;
   nom EMPLOYE.ENOM%TYPE;
   sal EMPLOYE.SALAIRE%TYPE;
   PROCEDURE modifier_salaire(mat_emp NUMBER, montant  NUMBER) IS
   BEGIN
     UPDATE EMPLOYE SET SALAIRE = SALAIRE + montant WHERE MATRICULE = mat_emp;
   END modifier_salaire;
BEGIN
   SELECT ENOM, SALAIRE INTO nom, sal  FROM EMPLOYE WHERE MATRICULE = mat;
   DBMS_OUTPUT.PUT_LINE(nom || ' a le salaire '||sal) ;
   DBMS_OUTPUT.PUT_LINE('Appel de la procédure modifier_salaire') ;
   modifier_salaire(mat, bonus); -- paramètres effectifs
   SELECT ENOM, SALAIRE INTO nom, sal FROM EMPLOYE WHERE MATRICULE = mat;
   DBMS_OUTPUT.PUT_LINE(nom || ' a le salaire '||sal) ;
   DBMS_OUTPUT.PUT_LINE('Appel de la procédure modifier_salaire') ;
   modifier_salaire(mat, merit + bonus); -- paramètres effectifs
   SELECT ENOM, SALAIRE INTO nom, sal FROM EMPLOYE WHERE MATRICULE = mat;
   DBMS_OUTPUT.PUT_LINE(nom || ' a le salaire '||sal) ; ROLLBACK;
END;
```

Exemple

```
THOMAS a le salaire 5000
Appel de la procédure modifier_salaire
THOMAS a le salaire 5100
Appel de la procédure modifier_salaire
THOMAS a le salaire 5250
Procédure PL/SQL terminée avec succès
```

Modes de passage de paramètres en PL/SQL

- PL/SQL offre trois modes de passage de paramètres pour les procédures et les fonctions :
  - IN : le paramètre est passé en entrée de procédure
  - OUT : le paramètre est valorisé dans la procédure et renvoyé à l'environnement appelant
  - IN OUT : le paramètre est passé en entrée de la procédure et il est renvoyé à l'environnement appelant

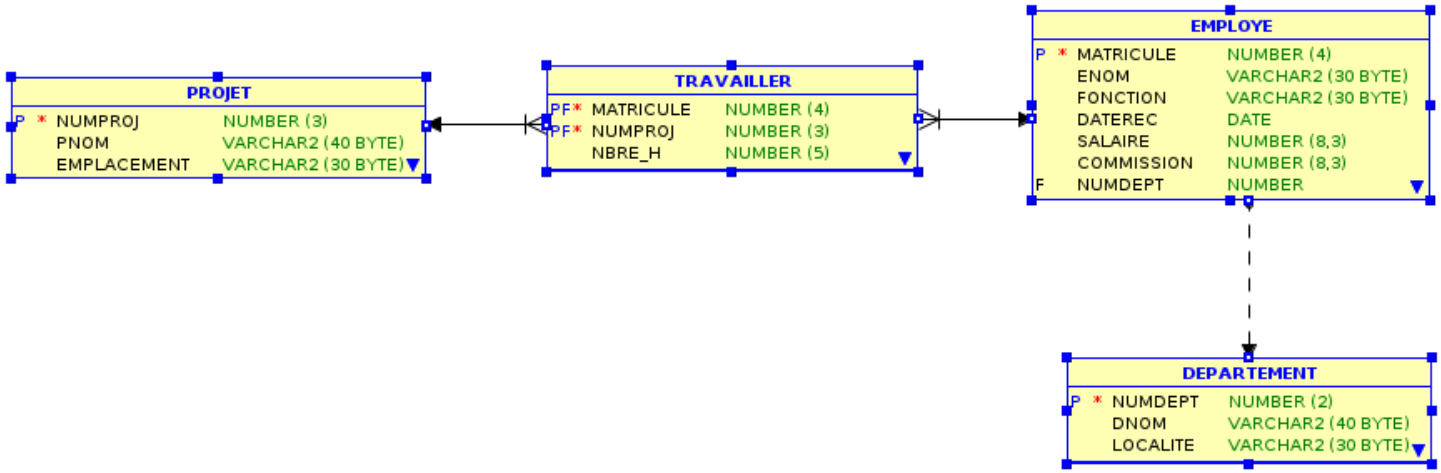
Syntaxe

```
nom_param [IN | OUT | IN OUT] [NOCOPY] nom_type [{ := | DEFAULT} expression];
```

Oracle dispose d'un indicateur de compilation, appelé NOCOPY, utilisé lors de la déclaration des paramètres OUT ou IN OUT. NOCOPY permet de transmettre directement le paramètre, c'est-à-dire que le compilateur PL/SQL passe le paramètre par référence plutôt que par valeur. Les paramètres IN sont toujours passés en NOCOPY

Exemple (I)

- Soit le SR suivant :



- Écrire un bloc `PL/SQL` contenant une procédure `maj_salaire` qui permet de modifier le salaire d'un employé particulier. On va utiliser deux paramètres `IN` pour passer les valeurs du nouveau salaire et du matricule de l'employé

Exemple (I)

Solution

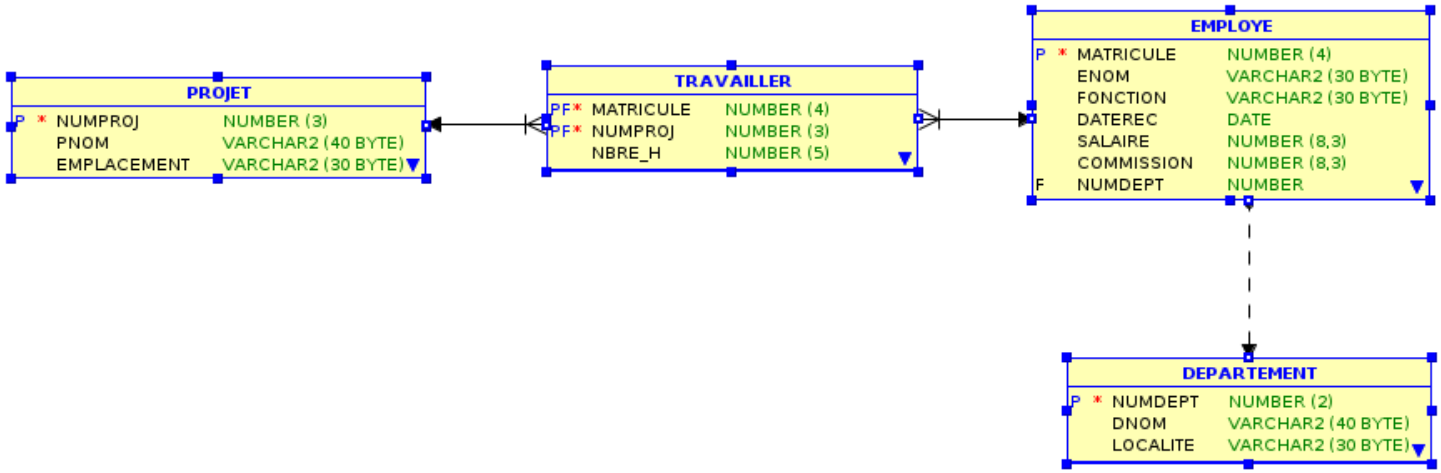
```
DECLARE
   sal1 EMPLOYE.SALAIRE%TYPE;
   mat1 EMPLOYE.MATRICULE%TYPE;
   nom  EMPLOYE.ENOM%TYPE;
   PROCEDURE maj_salaire(sal IN EMPLOYE.SALAIRE%TYPE, mat IN EMPLOYE.MATRICULE
      %TYPE) IS
      BEGIN
        UPDATE EMPLOYE
        SET SALAIRE = sal
        WHERE MATRICULE = mat;
      END;
BEGIN
   mat1 :=7800;
   SELECT ENOM, SALAIRE INTO nom, sal1 FROM EMPLOYE WHERE MATRICULE = mat1;
   DBMS_OUTPUT.PUT_LINE(nom || ' a le salaire '||sal1);
   sal1 :=5500;
   DBMS_OUTPUT.PUT_LINE('Appel de la procédure maj_salaire');
   maj_salaire( sal1, mat1);
   SELECT ENOM, SALAIRE INTO nom, sal1 FROM EMPLOYE WHERE MATRICULE = mat1;
   DBMS_OUTPUT.PUT_LINE(nom || ' a le salaire '||sal1);
   ROLLBACK;
END;
```

Exemple (I)

```
THOMAS a le salaire 5000
Appel de la procédure maj_salaire
THOMAS a le salaire 5500
```

Exemple (II)

- Soit le SR suivant :



- Écrire une procédure `trouver_emp` qui permet de retourner le nom et la fonction d'un employé donné. On va utiliser un paramètre `IN` pour passer l'identifiant de l'employé et deux paramètres `OUT` pour stocker le nom et la fonction

Exemple (II)

Solution

```
CREATE OR REPLACE PROCEDURE trouver_emp(mat_emp IN NUMBER,nom_emp OUT VARCHAR2
      ,fonct_emp OUT VARCHAR2) AS
BEGIN
   SELECT ENOM, FONCTION INTO nom_emp, fonct_emp FROM EMPLOYE WHERE MATRICULE
     = mat_emp;
EXCEPTION
    WHEN OTHERS THEN
       DBMS_OUTPUT.PUT_LINE(mat_emp||' Erreur  de recherche !');
END trouver_emp;
```

Exemple (II)

Test

```
SET SERVEROUTPUT ON
DECLARE
   mat EMPLOYE.MATRICULE%TYPE := 7800;
   nom  EMPLOYE.ENOM%TYPE;
   fonct EMPLOYE.FONCTION%TYPE;
BEGIN
-- appel de la procédure
   trouver_emp(mat, nom, fonct);
   DBMS_OUTPUT.PUT_LINE('L'employé de matricule ' || mat);
   DBMS_OUTPUT.PUT_LINE('Son nom est ' || nom);
   DBMS_OUTPUT.PUT_LINE('Sa fonction est ' || fonct);
END;
```

Exemple (II)

Test

```
SET SERVEROUTPUT ON
DECLARE
   mat EMPLOYE.MATRICULE%TYPE := 7800;
   nom  EMPLOYE.ENOM%TYPE;
   fonct EMPLOYE.FONCTION%TYPE;
BEGIN
-- appel de la procédure
   trouver_emp(mat, nom, fonct);
   DBMS_OUTPUT.PUT_LINE('L'employé de matricule ' || mat);
   DBMS_OUTPUT.PUT_LINE('Son nom est ' || nom);
   DBMS_OUTPUT.PUT_LINE('Sa fonction est ' || fonct);
END;
```

```
L'employé de matricule 7800
Son nom est THOMAS
Sa fonction est PRESIDENT
```

Qu'affiche le programme ?

```
SET SERVEROUTPUT ON
DECLARE
   n NUMBER := 111;
   PROCEDURE proced (para1 IN NUMBER,para2 IN OUT NUMBER,para3 IN OUT NOCOPY
      NUMBER) IS
   BEGIN
     para2 := 222;
     DBMS_OUTPUT.put_line
     ('affichage procedure para1 ' || para1);
     para3 := 333;
     DBMS_OUTPUT.put_line
     ('affichage procedure para1 ' || para1);
   END;
BEGIN
   proced(n, n, n);
   DBMS_OUTPUT.put_line('affichage bloc principal n '|| n);
END;
```

Qu'affiche le programme ?

```
SET SERVEROUTPUT ON
DECLARE
   n NUMBER := 111;
   PROCEDURE proced (para1 IN NUMBER,para2 IN OUT NUMBER,para3 IN OUT NOCOPY
      NUMBER) IS
   BEGIN
     para2 := 222;
     DBMS_OUTPUT.put_line
     ('affichage procedure para1 ' || para1);
     para3 := 333;
     DBMS_OUTPUT.put_line
     ('affichage procedure para1 ' || para1);
   END;
BEGIN
   proced(n, n, n);
   DBMS_OUTPUT.put_line('affichage bloc principal n '|| n);
END;
```

```
affichage procedure para1 111
affichage procedure para1 333
affichage bloc principal n 222
```

Création d'une fonction

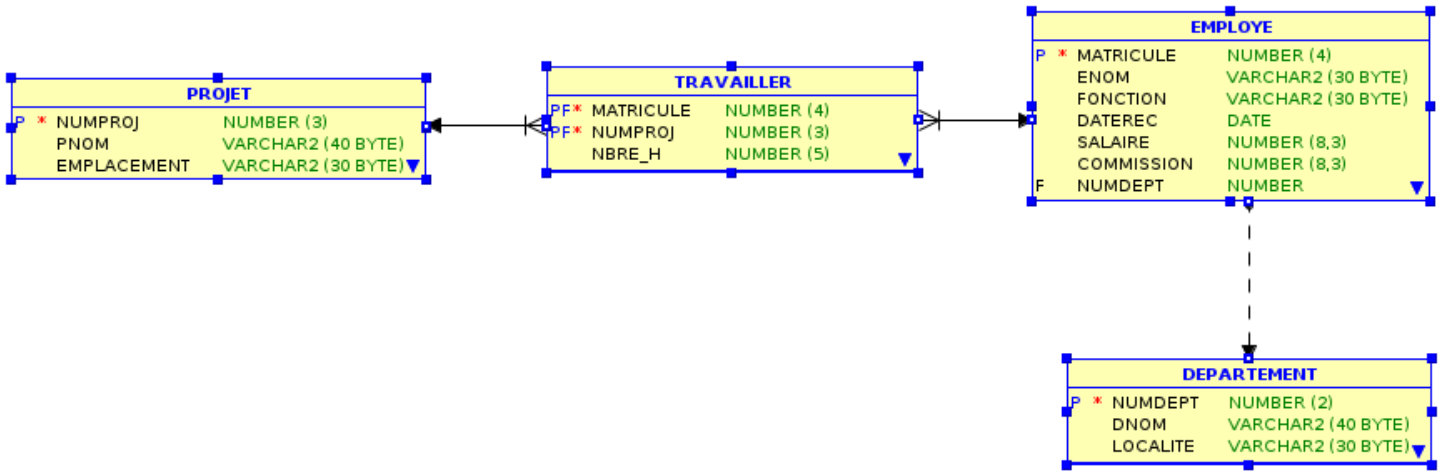
Tout ce qui a été dit sur les **procédures stockées** s'applique aux **fonctions**

Syntaxe

```
CREATE [OR REPLACE] FUNCTION nom_fonction[(paramètre [,paramètre]) ]
   RETURN type_de_données_de_la_valeur_retournée
   IS | AS
   [Section déclaration]
BEGIN
   <CODE PL/SQL>
[EXCEPTION
   Section gestion des exceptions]
END [nom_fonction];
```

Exemple

- Soit le SR suivant :



- Écrire une fonction qui retourne le nombre d'heures travaillées pour un employé particulier

Exemple

Solution

```
CREATE OR REPLACE FUNCTION total_heures (mat IN EMPLOYE.MATRICULE%TYPE)
RETURN NUMBER
IS
  total NUMBER;
BEGIN
   SELECT SUM(NBRE_H) INTO total FROM TRAVAILLER WHERE MATRICULE = mat;
   RETURN (total);
END;
```

Appel des fonctions en PL/SQL I

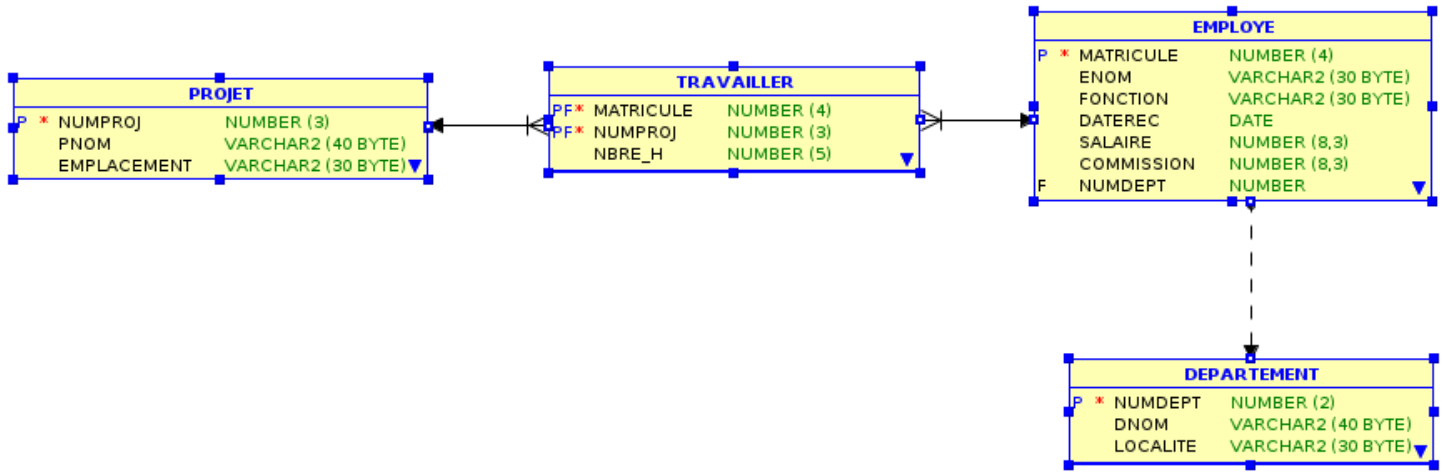
- Comme pour les **procédures stockées**, l'appel des **fonctions** peut se faire depuis un environnement applicatif tel qu'`Oracle Report` et à partir de `PL/SQL`
- Les **fonctions cataloguées** s'utilisent comme les **fonctions prédéfinies d'`Oracle`** (`CONCAT`, `SUBSTR`, `ABS`...). Elles peuvent être placées dans des expressions qui seront injectées dans des requêtes `SQL`
- Une **fonction** peut être placée dans :
  - la liste qui suit un `SELECT`,
  - la condition d'une clause `WHERE` ou `HAVING`,
  - les clauses `GROUP BY`, `ORDER BY`...,
  - la clause `VALUES` d'une commande `INSERT`,
  - la clause `SET` d'une commande `UPDATE`

Appel des fonctions en PL/SQL II

- Par contre, on ne peut pas utiliser les **fonctions** dans la contrainte `CHECK` d'une commande `CREATE` ou `ALTER TABLE`
- Pour être appelée dans une expression `SQL`, une **fonction** utilisateur doit vérifier les contraintes suivantes :
  - La fonction doit être stockée ou cataloguée, en utilisant la commande `CREATE` (Objets du dictionnaire `Oracle`). Elle ne peut pas être une fonction locale, temporaire, déclarée dans un bloc `PL/SQL` ; c'est-à-dire qu'elle n'est plus disponible pour être appelée lorsque l'exécution du bloc se termine
  - Elle ne peut pas être considérée comme une fonction de groupe
  - Tous ses paramètres sont de type `IN`
  - Tous ses paramètres et le retour de la fonction sont de type `NUMBER`, `CHAR`, `DATE` et non pas de type `PL/SQL` comme `RECORD`, `TABLE` ou `BOOLEAN`
  - La fonction ne doit pas utiliser les commandes `INSERT`, `UPDATE` et `DELETE`
  - La fonction ne doit pas appeler un sous-programme (fonction ou procédure) exécutant les commandes `INSERT`, `UPDATE` et `DELETE`

Exemple (I)

- Soit le SR suivant :



- Écrire un bloc anonyme qui permet d'appeler la fonction `total_heures` pour calculer le nombre d'heures total de l'employé de matricule `7800`

Exemple (I)

Solution

```
SET SERVEROUTPUT ON
DECLARE
   mat1 EMPLOYE.MATRICULE%TYPE := 7800;
   tot NUMBER;
BEGIN
   tot := total_heures(mat1);
   DBMS_OUTPUT.PUT_LINE('L'employé de matricule ' || mat1 || ' a travaillé '
      || tot ||' heures');
END;
```

Exemple (I)

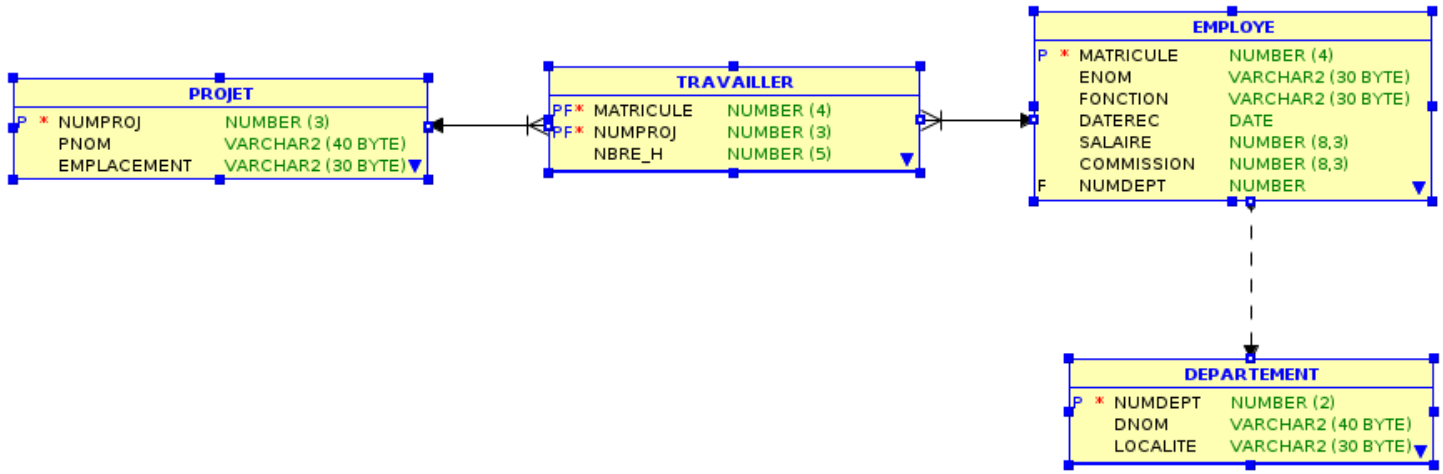
Solution

```
SET SERVEROUTPUT ON
DECLARE
   mat1 EMPLOYE.MATRICULE%TYPE := 7800;
   tot NUMBER;
BEGIN
   tot := total_heures(mat1);
   DBMS_OUTPUT.PUT_LINE('L'employé de matricule ' || mat1 || ' a travaillé '
      || tot ||' heures');
END;
```

```
L'employé de matricule 7800 a travaillé 120 heures
```

Exemple (II)

- Soit le SR suivant :



- Écrire une requête `SQL` qui permet d'afficher pour chaque employé le total des heures travaillées en utilisant la fonction `total_heures`

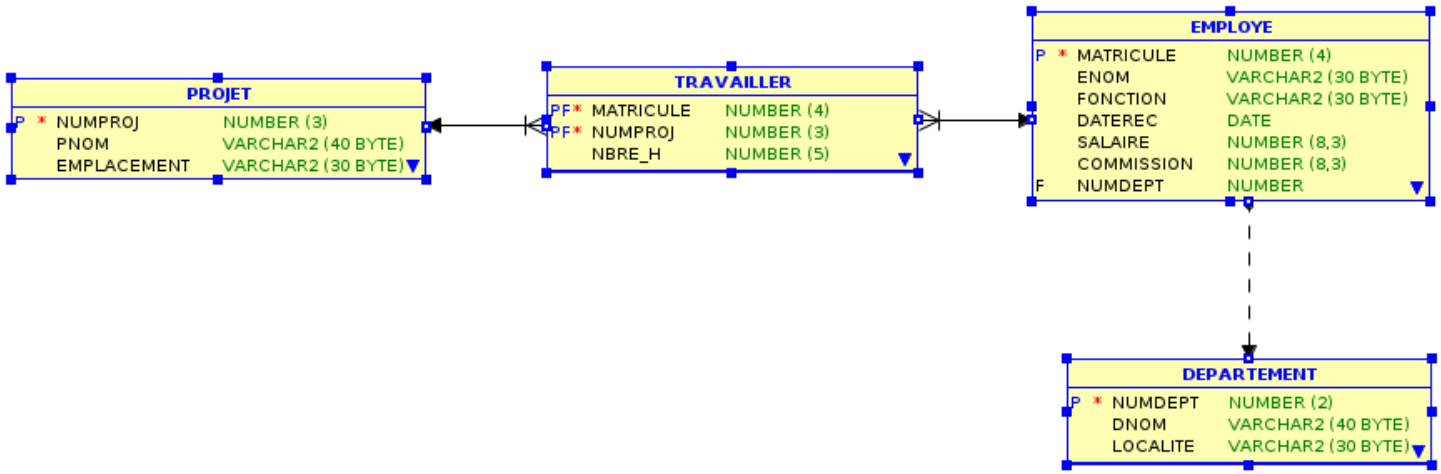
Exemple (II)

Solution

```
SELECT DISTINCT(MATRICULE), total_heures(MATRICULE)
FROM TRAVAILLER;
```

Exemple (III)

- Soit le SR suivant :



- Écrire une fonction `formatage_date` qui prend en paramètre une date et un séparateur, et qui retourne la date dans le format jour mois année, séparé par le séparateur fourni

Exemple (III)

Solution

```
CREATE OR REPLACE FUNCTION formatage_date ( pdate IN DATE, psep  IN CHAR )
      RETURN VARCHAR2
IS
   vdate VARCHAR2(30);
BEGIN
   vdate := TO_CHAR(pdate,'dd') || psep || TO_CHAR(pdate,'mm') || psep ||
     TO_CHAR(pdate,'yy');
RETURN(vdate);
END;
```

Exemple (III)

Execution

```
SELECT formatage_date(SYSDATE,'_')
FROM DUAL;
```

```
 FORMATAGE_DATE(SYSDATE,'_')
-----------------------------
01_06_16
```

Informations sur les procédures et fonctions stockées

Afficher les noms de tous les sous-programmes que vous avez créés

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

Afficher le code source de la fonction `factoriel`

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

Afficher les informations de la procédure `trouver_emp`

```
DESCRIBE TROUVER_EMP;
```

Afficher les erreurs de la fonction `factoriel`

```
SHOW ERRORS FUNCTION factoriel
```

Modification, suppression et droit d'exécution d'un sous-programme

Recompilation d'un sous-programme

```
ALTER {FUNCTION | PROCEDURE} nom_sous_prog COMPILE
```

Suppression d'un sous-programme

```
DROP {FUNCTION | PROCEDURE} nom_sous_prog
```

Droit d'exécution d'un sous-programme

```
GRANT EXECUTE ON nom_sous_prog TO nom_utilisateur;
```


THAT'S ALL
FOLKS !