

Bases de données

Cours II – PL/SQL

Guénaël Cabanes

cabanes@lipn.univ-paris13.fr

Partie I

-

Introduction

Qu'appelle-t-on PL/SQL

Procedural Language for SQL

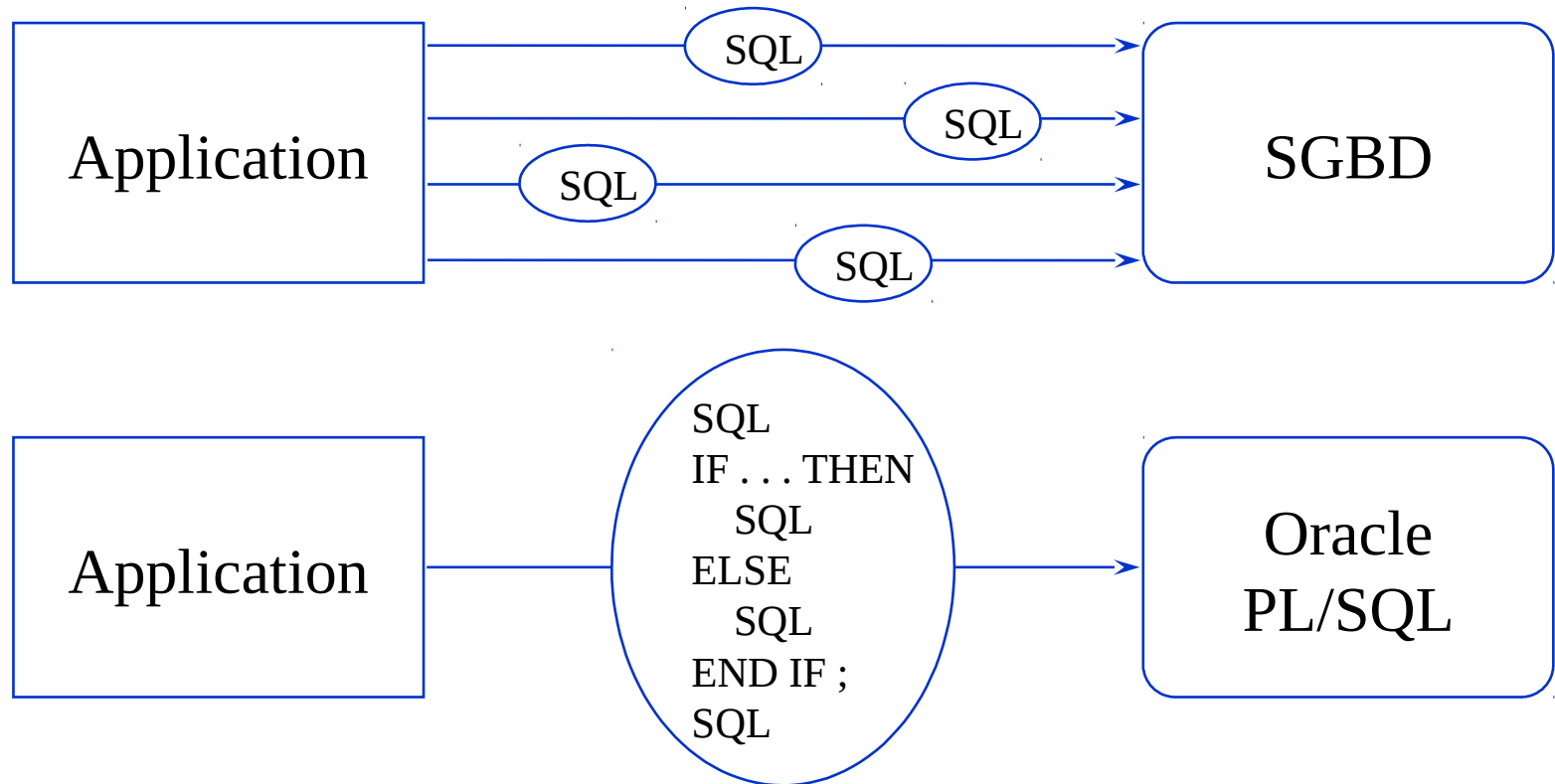
- Le langage PL/SQL est un langage fournissant une interface procédurale au SGBD Oracle.
- Il intègre parfaitement le langage SQL.
- Il permet de réaliser des traitements algorithmiques (ce que ne permet pas SQL).
- Il offre la plupart des mécanismes classiques de programmation (boucles, conditions, variables, ...).

Avantages de PL/SQL

PL/SQL complète SQL qui n'est pas procédural, il offre :

- **Des structures répétitives (WHILE, FOR, ...).**
- **Des structures conditionnelles (IF ELSE THEN, CASE).**
- **La déclaration des curseurs et des tableaux,**
- **La déclaration de variables,**
- **L'affectation de valeurs aux variables,**
- **Les branchements (GOTO, EXIT),**
- **Les exceptions (EXCEPTION).**

Avantages de PL/SQL

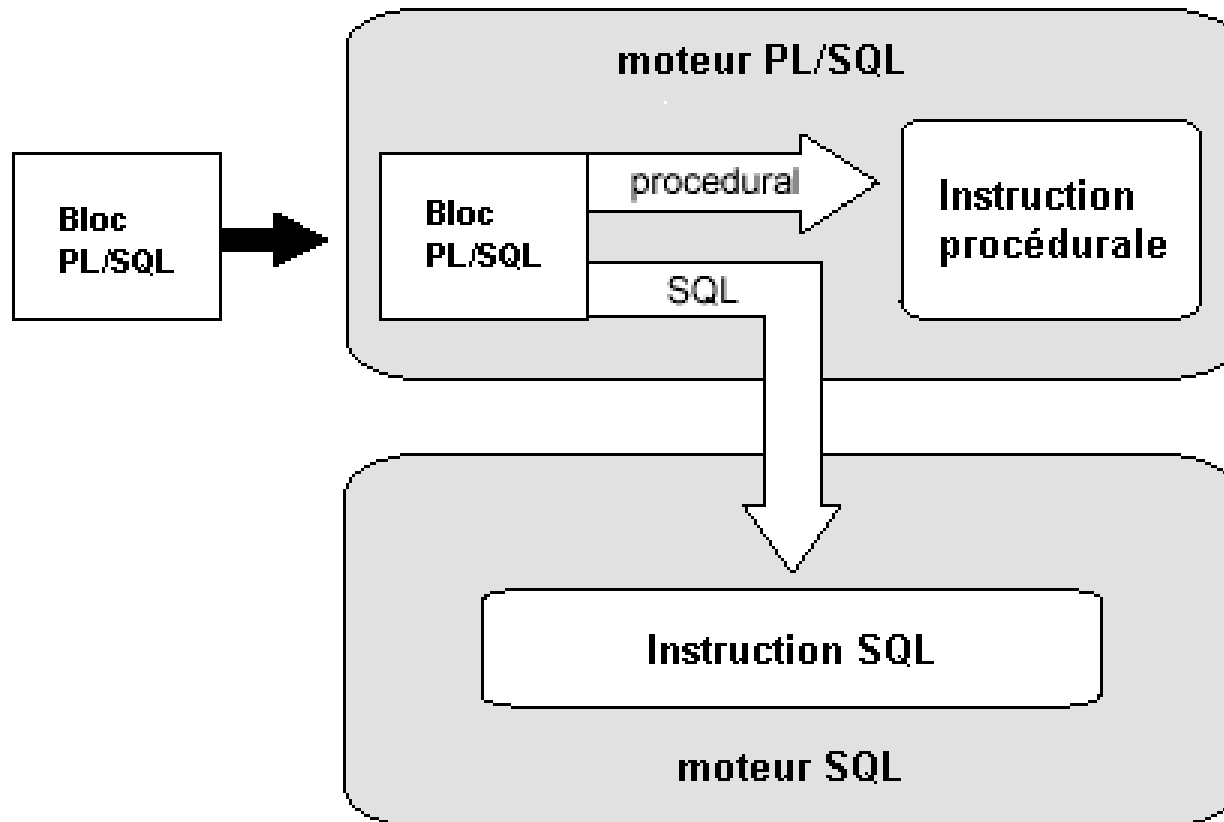


Partie II

-

Le bloc PL/SQL

Environnement PL/SQL



Structure d'un bloc PL/SQL

DECLARE

Variables, curseurs, exceptions, ...

BEGIN

Instructions SQL et PL/SQL

Possibilités de blocs fils (imbrication de blocs)

EXCEPTION

Traitement des exceptions (gestion des erreurs)

END ;

/

Chaque bloc PL/SQL peut être constitué de 3 sections :

- Une section facultative de déclaration et initialisation de types, variables et constantes **DECLARE**.
- Une section obligatoire contenant les instructions d'exécution **BEGIN**.
- Une section facultative de gestion des erreurs **EXCEPTION**.

Structure d'un bloc PL/SQL

Bloc Minimal :

```
BEGIN  
    Null ;  
END ;  
/
```

Le mot clé **BEGIN** détermine le début de la section des instructions exécutables.

Le mot clé **END**; indique la fin de la section des instructions exécutables.

Une seule instruction figure dans ce bloc : **Null**; qui ne génère aucune action. Ce bloc PL/SQL ne fait donc absolument rien !

Exemple d'un bloc PL/SQL

```
DECLARE
  var_x  VARCHAR2(5) ;
BEGIN
  SELECT nom_colonne
  INTO   var_x
  FROM   nom_table ;
EXCEPTION
  WHEN nom_exception THEN
    ..... ;
END ;
/
```

Type de blocs PL/SQL

- Bloc anonyme
- Procédure
- Fonction

Syntaxe d'un bloc PL/SQL

PL/SQL reconnaît les fonctions SQL :

- Les fonctions sur les nombres.
- Les fonctions sur les chaînes de caractères.
- Les fonctions de conversion de type de données.
- Les fonctions de dates.

Syntaxe d'un bloc PL/SQL

Exemples :

- **Recomposer l'adresse d'un employé :**

```
V_AdrComplete := V_Rue || ' ' || V_Ville || ' ' || V_CodePostal ;
```

- **Convertir le nom en majuscule :**

```
V_Nom := UPPER (V_Nom) ;
```

- **Extraction d'une partie de la chaîne :**

```
V_chr := SUBSTR ('PL/SQL',4,3) ;
```

- **Remplacement d'une chaîne par une autre :**

```
V_chr := REPLACE('Serv1/Prod/tb_client','Prod','Valid) ;
```

Syntaxe d'un bloc PL/SQL

Les opérateurs dans PL/SQL :

- Logique
- Arithmétique
- Concaténation
- Parenthèse pour contrôler l'ordre des opérations
- Opérateur d'exponentiation **



Identique à SQL

Syntaxe d'un bloc PL/SQL

Insérer un commentaire :

- Précéder un commentaire écrit sur une seule ligne par ' -- '.
- Placer un commentaire écrit sur plusieurs lignes entre les symboles '/*' et '*/'.

Exemple :

```
DECLARE
```

```
v_sal NUMBER (9,2);
```

```
BEGIN
```

```
    /* Ceci est un commentaire qui peut être écrit sur  
    plusieurs lignes */
```

```
END; -- Ceci est un commentaire sur une seule ligne
```

```
/
```

Section déclaration

- La section déclarative (facultative) d'un bloc débute par le mot clé **DECLARE**.
- Elle contient toutes les déclarations des variables qui seront utilisées localement par la section exécutable, ainsi que leur éventuelle initialisation.
- Cette section ne peut pas contenir d'instructions exécutables. Toutefois, il est possible de définir dans cette section des procédures ou des fonctions contenant une section exécutable.

Toute variable doit avoir été déclarée avant de pouvoir être utilisée dans la section exécutable.

Section exécution

- Délimitée par les mots clé **BEGIN** et **END**; elle contient les instructions d'exécution du bloc PL/SQL, les instructions de contrôle et d'itération, l'appel des procédures et fonctions, l'utilisation des fonctions natives, les ordres SQL, etc.
- Chaque instruction doit être suivi du terminateur d'instruction ';'.

BEGIN

Null ;

END ;

Section de gestion des erreurs

- La section de gestion des erreurs (facultative) débute par le mot clé **EXCEPTION**.
- Elle contient le code exécutable mis en place pour la gestion des erreurs.
- Lorsqu'une erreur intervient dans l'exécution, le programme est stoppé et le code erreur est transmis à cette section.

```
DECLARE
    Chaine  VARCHAR2(15) := 'Hello World' ;
BEGIN
    DBMS_OUTPUT.PUT_LINE( Chaine ) ;
EXCEPTION
    WHEN OTHERS THEN Null ;
END ;
/
```

Section de gestion des erreurs

DECLARE

 Chaine VARCHAR2(15) := 'Hello World' ;

BEGIN

DBMS_OUTPUT.PUT_LINE(Chaine) ;

EXCEPTION

WHEN OTHERS THEN Null ;

END ;

/

- Les erreurs doivent être interceptées avec le mot clé **WHEN** suivi du code erreur ciblé.
- Ici, le code **OTHERS** qui définit toutes les erreurs non interceptées individuellement par les clauses **WHEN** précédentes.
- **DBMS_OUTPUT** permet l’affichage des valeurs d’une variable.

Blocs imbriqués

Les blocs PL/SQL peuvent être imbriqués.

```
DECLARE
...
BEGIN
    DECLARE
    ....
    BEGIN
    .....
        BEGIN
        .....
        END ;
    .....
    END ;
.....
END ;
```

Blocs imbriqués

Blocs imbriqués et porté des variables :

```
...  
x  INTEGER;  
BEGIN  
    .....  
    DECLARE  
    y  NUMBER;  
    BEGIN  
        .....  
    END;  
    .....  
END;
```

Blocs imbriqués

Blocs imbriqués et portée des variables :

```
...  
x  INTEGER;  ————— Portée de x  
BEGIN  
    .....  
    DECLARE  
    y  NUMBER;  ————— Portée de y  
    BEGIN  
        .....  
    END;  
    .....  
END;  —————
```

The diagram illustrates the scope of variables in nested blocks. The outer block is defined by the 'BEGIN' and 'END;' statements. The inner block is defined by its own 'BEGIN' and 'END;' statements. The scope of 'x' is the entire outer block, while the scope of 'y' is only the inner block. The lines for 'x' and 'y' are connected by vertical lines at their respective start and end points, forming a nested structure.

Utilisation dans un script

- Les blocs PL/SQL peuvent être placés dans des fichiers textes avec l'extension .sql qui seront alors chargés dans SQL*Plus.
- Il faut insérer la ligne : **SET serveroutput on** pour pouvoir utiliser la commande **DBMS_OUTPUT.PUT_LINE();**.
- La barre oblique (/) a la fin du bloc PL/SQL force son exécution.

Utilisation dans un script

Exemple de script :

```
SET serveroutput on
```

```
DECLARE
```

```
    Chaine VARCHAR2(15) := 'Hello World' ;
```

```
BEGIN
```

```
    DBMS_OUTPUT.PUT_LINE( Chaine ) ;
```

```
END ;
```

```
/
```


Débogage

SQL ne donne pas de détails si une erreur apparaît lors de l'exécution.

Il faut utiliser l'instruction `SHOW ERRORS` pour obtenir le détail des erreurs rencontrés

Partie III

-

Déclaration et utilisation de variables

Déclaration d'une variable

La section déclarative contient toutes les déclarations des variables qui seront utilisées localement par la section exécutable, ainsi que leur éventuelle initialisation.

Exemple :

```
DECLARE
    Chaine VARCHAR2(15) := 'Salut Monde' ;
BEGIN
    DBMS_OUTPUT.PUT_LINE( Chaine ) ;
END ;
/
```

Déclaration d'une variable

Syntaxe :

```
Nom [CONSTANT] type [NOT NULL] [:= expression];
```

Exemple :

DECLARE

```
v_datenaissance    DATE;  
v_departement      NUMBER(2) NOT NULL := 10;  
v_ville            VARCHAR2(13) := 'Paris';  
c_cumul            CONSTANT NUMBER := 1000;
```

Déclaration d'une variable

Syntaxe :

Nom [**CONSTANT**] type [**NOT NULL**] [:= expression];

- **Nom** : représente le nom de la variable. Le nom de la variable ne peut pas excéder 30 caractères
- **CONSTANT** : indique que la valeur ne pourra pas être modifiée, **expression** doit être indiqué.
- **NOT NULL** : indique que la variable ne peut pas être NULL, **expression** doit être indiqué.
- **type** : représente de type de la variable

Déclaration d'une variable

Les types de base :

- **VARCHAR2 (n)** : Chaîne de caractères de taille max **n**.
- **NUMBER (n , m)** : Nombre de taille **n**, dont **m** décimales.
- **DATE** : Une date.
- **CHAR (n)** : Chaîne de caractères de taille **n**.
- **BOOLEAN** : 0 ou 1.
- **INTEGER** : Nombre entier.

Déclaration d'une variable

L'attribut **%TYPE** donne le type associée à :

- Une colonne d'une table dans la BD.
- Une variable précédemment définie.

Exemples :

```
v_nom      emp.nom%TYPE;
```

```
v_sal_annuel  NUMBER(7,2);
```

```
v_sal_mensuel  v_sal_annuel%TYPE := 2000;
```

Déclaration d'une variable

Pour déclarer une variable globale, il faut faire précéder la référence par un ' : '.

Exemples :

- Stocker le salaire mensuel dans une variable globale SQL*Plus :

`:g_sal_mensuel := v_sal_annuel / 12;`

- Utiliser le salaire mensuel dans un bloc PL/SQL :

`IF :g_sal_mensuel > 1200 THEN`

Assigner une valeur

Avec l'opérateur **:=** dans la section exécution :

Identificateur := expr;

Exemples :

- Affecter une date de naissance :
- Fixer le nom à 'Clément'

Assigner une valeur

Avec l'opérateur **`:=`** dans la section exécution :

Identificateur `:=` expr;

Exemples :

- Affecter une date de naissance :

`v_datenaissance := '23-SEP-2004';`

- Fixer le nom à 'Clément'

`v_nom := 'Clément';`

Assigner une valeur

Avec **SELECT ... INTO** ou **FETCH ... INTO** :

```
DECLARE
```

```
Nom_emp EMP.ENAME%Type ;
```

```
CURSOR C_EMP IS SELECT ename FROM EMP WHERE Empno = 1014;
```

```
BEGIN
```

```
SELECT ename INTO Nom_emp  
FROM EMP WHERE Empno = 1014;
```

```
OPEN C_EMP ;
```

```
FETCH C_EMP INTO Nom_emp ;
```

```
CLOSE C_EMP ;
```

```
END ;
```

Assigner une valeur

Assigner des valeurs avec **SELECT INTO** :

```
SELECT    liste_sélection  
INTO      nom_var [, nom_var, ...]  
FROM      table  
WHERE     condition;
```

Assigner une valeur

Assigner des valeurs avec **SELECT INTO** :

```
SELECT    liste_sélection  
INTO      nom_var [, nom_var, ...]  
FROM      table  
WHERE     condition;
```

Il faut que la requête ne renvoie qu'une seule valeur par colonne !

Assigner une valeur

Exemple 1 :

```
DECLARE
    v_deptno      NUMBER(2);
    v_loc         VARCHAR2(15);
BEGIN
    SELECT      deptno, loc
    INTO        v_deptno, v_loc
    FROM        dept
    WHERE       nom_d = 'INFORMATIQUE';
END;
```

Assigner une valeur

Exemple 2 : Retourne la somme des salaires de tous les employés d'un département donné.

DECLARE

 v_som_sal emp.sal%TYPE;

 v_deptno NUMBER NOT NULL := 10;

BEGIN

END;

Assigner une valeur

Exemple 2 : Retourne la somme des salaires de tous les employés d'un département donné.

```
DECLARE
    v_som_sal    emp.sal%TYPE;
    v_deptno     NUMBER NOT NULL := 10;
BEGIN
    SELECT        SUM(salaire)
    INTO          v_som_sal
    FROM          emp
    WHERE         deptno = v_deptno;
END;
```


Utiliser une variable

Exemple 1 : Ajouter les informations d'un nouvel employé à la table emp.

DECLARE

v_empno NUMBER NOT NULL := 105;

BEGIN

INSERT INTO emp

VALUES (v_empno, 'Clément', 'Directeur', 10);

END;

Utiliser une variable

Exemple 2 : Augmenter le salaire de tous les employés qui ont le poste d'enseignant.

DECLARE

v_augm_sal emp.sal%TYPE := 2000;

BEGIN

END;

Utiliser une variable

Exemple 2 : Augmenter le salaire de tous les employés qui ont le poste d'enseignant.

```
DECLARE
    v_augm_sal  emp.sal%TYPE := 2000;
BEGIN
    UPDATE      emp
    SET         sal := sal + v_augm_sal
    WHERE      job = ' Enseignant ';
END;
```

Utiliser une variable

Exemple 3 : Suppression des lignes appartenant au département 10.

```
DECLARE
    v_deptno    emp.deptno%TYPE := 10;
BEGIN

END;
```

Utiliser une variable

Exemple 3 : Suppression des lignes appartenant au département 10.

```
DECLARE
    v_deptno    emp.deptno%TYPE := 10;
BEGIN
    DELETE FROM emp
    WHERE      deptno = v_deptno;
END;
```

Bind Variables

Les variables attachés (Bind Variables) sont des variables à portée globale définies en dehors d'un bloc PL/SQL.

VARIABLE nom_variable type_variable;

Exemple:

VARIABLE g_sal_mensuel NUMBER(7,2);

Bind Variables

Pour utiliser une Bind Variable dans PL/SQL, il faut faire précéder le nom de la variable par « : »

Exemple:

```
: g_sal_mensuel := v_sal_annuel / 12;
```

Bind Variables et optimisation

Les Bind Variables sont surtout utilisés pour des raisons d'optimisation:

Exemple de requête **SELECT** générées des milliers de fois :

```
SELECT fname, lname, pcode FROM cust WHERE id = 674;  
SELECT fname, lname, pcode FROM cust WHERE id = 234;  
SELECT fname, lname, pcode FROM cust WHERE id = 332;  
...
```

A chaque soumission d'un requête,

- Vérification si la requête a déjà été soumise
- Si oui, récupération du plan d'exécution de la requête, et exécution de la requête

Bind Variables et optimisation

Les Bind Variables sont surtout utilisés pour des raisons d'optimisation:

- Si non,
 - analyse syntaxique de la requête
 - définition des différentes possibilités d'exécution
 - définition du plan d'exécution optimal

→ Processus coûteux en temps CPU, alors que seule la valeur de `id` change !
- Solution : réutiliser le plan d'exécution existant
 - Nécessite d'utiliser des *variables attachées* :
 - Substitution de la valeur par la variable attaché
 - Envoi de la même requête pour toutes les valeurs de `id`
 - Exemple :

```
SELECT fname , lname , pcode FROM cust WHERE id = :cust_no;
```

Partie IV

-

Les Structures de contrôles : conditions et boucles

Structure de contrôle dans PL/SQL

- **IF conditionnel :**
 - IF THEN END IF;
 - IF THEN ELSE END IF;
 - IF THEN ELSIF THEN END IF;
- **Les boucles :**
 - LOOP END LOOP;
 - FOR LOOP END LOOP;
 - WHILE LOOP END LOOP;

Instruction IF

Syntaxe :

```
IF condition THEN  
    énoncés;  
[ELSIF condition THEN  
    énoncés;]  
[ELSE  
    énoncés;]  
END IF;
```

Exemple d'un IF simple :

Mettre le ID de l'employé
'MARK' à 101.

Instruction IF

Syntaxe :

```
IF condition THEN  
    énoncés;  
[ELSIF condition THEN  
    énoncés;]  
[ELSE  
    énoncés;]  
END IF;
```

Exemple 1 :

Mettre le ID de l'employé
'MARK' à 101.

```
IF v_nom = 'MARK' THEN  
    v_ID := 101;  
END IF;
```

Instruction IF

Exemple 2 : Si le nom de l'employé est 'Clément', alors lui attribuer

- **le poste 'Enseignant',**
- **le département n° 102 et**
- **une commission de 25 % sur son salaire actuel.**

Instruction IF

Exemple 2 : Si le nom de l'employé est 'Clément', alors lui attribuer

- le poste 'Enseignant',
- le département n° 102 et
- une commission de 25 % sur son salaire actuel.

```
IF v_nom = 'Clément' THEN  
    v_poste := 'Enseignant';  
    v_deptno := 102;  
    v_nouv_comm := sal * 0.25;  
END IF;
```

Instruction IF

Exemple 3 : Si le nom de l'employé est 'Clément', alors

- **lui attribuer le poste 'Enseignant', le département n° 102 et une commission de 25 % sur son salaire actuel.**
- **sinon afficher le message 'Employé inexistant'.**

Instruction IF

Exemple 3 : Si le nom de l'employé est 'Clément', alors

- lui attribuer le poste 'Enseignant', le département n° 102 et une commission de 25 % sur son salaire actuel.
- sinon afficher le message 'Employé inexistant'.

IF v_nom = 'Clément' **THEN**

 v_poste := 'Enseignant';

 v_deptno := 102;

 v_nouv_comm := sal * 0.25;

ELSE

DBMS_OUTPUT.PUT_LINE('Employé inexistant');

END IF;

Instruction IF

Exemple 4 : si le IF est dans une fonction, on peut utiliser **RETURN** pour renvoyer une valeur.

```
IF v_debut > 100 THEN  
    RETURN ( 2 * v_debut);  
ELSIF v_debut >= 50 THEN  
  
    RETURN ( 5 * v_debut);  
ELSE  
    RETURN (1 * v_debut);  
END IF;
```

Boucle de base

Syntaxe :

```
LOOP                -- délimiteur
  énoncé 1;         -- énoncé
  .....
  EXIT [ WHEN condition]; -- énoncé EXIT
END LOOP;           -- délimiteur
```

Boucle de base

Exemple : Insérer dans la table « article » 10 articles numérotés de 1 a 10 et avec la date d'aujourd'hui.

Boucle de base

Exemple : Insérer dans la table « article » 10 articles numérotés de 1 a 10 et avec la date d'aujourd'hui.

DECLARE

v_Date **DATE;**

v_compteur **NUMBER(2) := 1;**

BEGIN

v_Date := SYSDATE;

LOOP

INSERT INTO article **VALUES** (v_compteur, v_Date);

v_compteur := v_compteur + 1;

EXIT WHEN v_compteur > 10;

END LOOP;

END;

/

Boucle FOR

Syntaxe :

```
FOR indice IN [REVERSE] borne_inf .. Borne_sup LOOP  
    énoncé 1;  
    énoncé 2;  
    .....  
END LOOP;
```

Boucle FOR

Syntaxe :

```
FOR indice IN [REVERSE] borne_inf .. Borne_sup LOOP  
    énoncé 1;  
    énoncé 2;  
    .....  
END LOOP;
```

Remarques :

- Inutile de déclarer l'indice, il est déclaré implicitement.
- L'option **REVERSE** permet de parcourir l'indice à l'envers.

Boucle FOR

Exemple : Insérer Nb articles indexés de 1 à Nb avec la date du système en utilisant la boucle FOR.

Boucle FOR

Exemple : Insérer Nb articles indexés de 1 à Nb avec la date du système en utilisant la boucle FOR.

```
DECLARE
```

```
    v_Date DATE;
```

```
    BEGIN
```

```
    v_Date := SYSDATE;
```

```
    FOR i IN 1 .. &Nb LOOP
```

```
        INSERT INTO article VALUES (i, v_Date);
```

```
    END LOOP;
```

```
    END;
```

```
    /
```

Boucle FOR

Exemple : Insérer Nb articles indexés de 1 à Nb avec la date du système en utilisant la boucle FOR.

DECLARE

v_Date DATE;

BEGIN

v_Date := SYSDATE;

FOR i IN 1 .. &Nb LOOP

INSERT INTO article VALUES (i, v_Date);

END LOOP;

END;

/

Avec &Nb, le système demande une valeur à l'utilisateur au début de la boucle.

Boucle WHILE

Syntaxe :

```
WHILE condition LOOP  
    énoncé 1;  
    énoncé 2;  
    .....  
END LOOP;
```

Remarque :

- La condition est évaluée au début de chaque itération.

Boucle WHILE

Exemple : Insérer dans la table « article » 10 articles numérotés de 1 à 10 et avec la date d'aujourd'hui.

Boucle WHILE

Exemple : Insérer dans la table « article » 10 articles numérotés de 1 à 10 et avec la date d'aujourd'hui.

DECLARE

v_Date DATE;

v_compteur NUMBER(2) := 1;

BEGIN

v_Date := SYSDATE;

While v_compteur < 10 LOOP

INSERT INTO article VALUES (v_compteur, v_Date);

v_compteur := v_compteur + 1;

END LOOP;

END;

/

Boucles imbriquées et Labels

- On peut Imbriquer les boucles à des niveaux multiples.
- Utiliser les labels pour distinguer les blocs et les boucles.
- Quitter la boucle extérieure avec un EXIT référençant le label.
- Le label s'écrit sous la forme << nom_label >>.

Boucles imbriquées et Labels

Exemple :

```
BEGIN
  << bouc_ext>>
  LOOP
    v_compteur := v_compteur + 1;
  EXIT WHEN v_compteur > 10;
  <<bouc_int>>
  LOOP
    .....
    EXIT bouc_ext WHEN total_fait = 1;

    EXIT WHEN int_fait = 1;
    .....
  END LOOP bouc_int;
END LOOP bouc_ext;
END;
/
```

Boucles imbriquées et Labels

Exemple :

```
BEGIN
  << bouc_ext>>
  LOOP
    v_compteur := v_compteur + 1;
  EXIT WHEN v_compteur > 10;
  <<bouc_int>>
  LOOP
    .....
    EXIT bouc_ext WHEN total_fait = 1; -- quitter les deux boucles

    EXIT WHEN int_fait = 1; -- quitter uniquement la boucle interne
    .....
  END LOOP bouc_int;
  END LOOP bouc_ext;
END;
/
```


Partie V



-

Gestion des erreurs

Manipulation des exceptions

- **Le traitement des exceptions est un mécanisme pour manipuler les erreurs rencontrées lors de l'exécution.**
- **Cela permet à l'exécution de continuer si l'erreur n'est pas suffisamment importante pour terminer l'exécution.**
- **Si une erreur est rencontrée et traitée de la sorte, l'exception est traitée, le programme sort du bloc courant puis l'exécution se poursuit.**

Types des exceptions

- **Exceptions Oracle prédéfinies**
 - **Exceptions Oracle non-prédéfinies**
- 
- Déclenchées implicitement**
-
- **Exceptions définies par l'utilisateur**
- 
- Déclenchées explicitement**

Capture des exceptions

Syntaxe :

EXCEPTION

WHEN exception1 [OR exception2 ...] **THEN**

énoncé1;

énoncé2;

.....

[**WHEN** exception2 [OR exception4 ...] **THEN**

énoncé3;

énoncé4;

.....]

[**WHEN OTHERS THEN**

énoncé5;

énoncé6;

.....]

Capture des exceptions prédéfinies

- **Faire référence au nom dans la partie traitement des exceptions.**
- **Quelques exceptions prédéfinies :**
 - **NO_DATA_FOUND**
 - **TOO_MANY_ROWS**
 - **INVALID_CURSOR**
 - **ZERO_DIVIDE**
 - **DUP_VAL_ON_INDEX**

Capture des exceptions prédéfinies

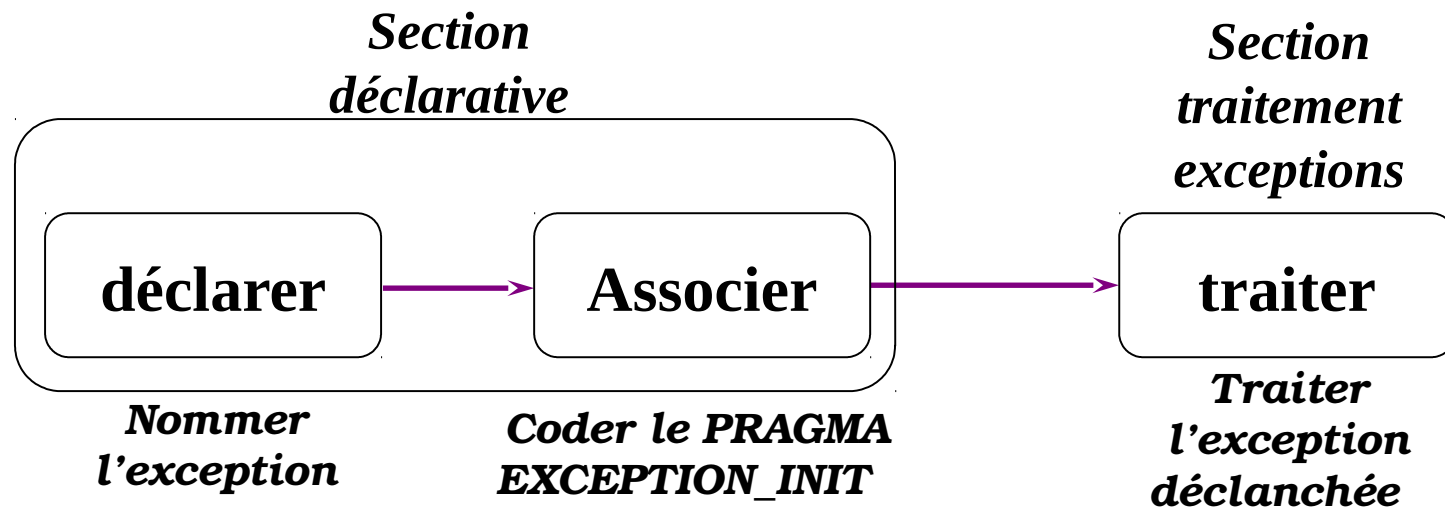
Exemple :

```
BEGIN
    .....
    COMMIT;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE (TO_CHAR (etudno) || 'Non valide');

    WHEN TOO_MANY_ROWS THEN
        énoncé1;
        DBMS_OUTPUT.PUT_LINE (' Données invalides');

    WHEN OTHERS THEN
        énoncé2;
        DBMS_OUTPUT.PUT_LINE (' Autres erreurs ');
    ROLLBACK;
END ;
```

Exceptions non-prédéfinies

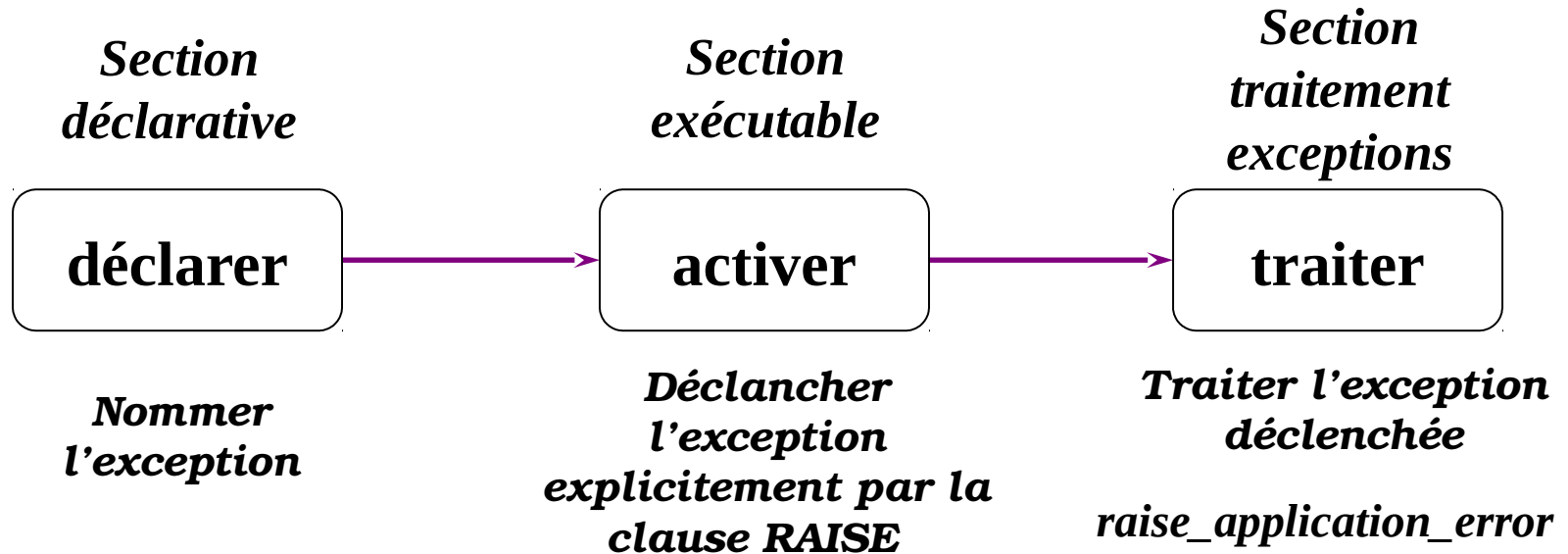


Exceptions non-prédéfinies

Exemple : Capture de l'erreur n° 2291 (violation de la contrainte intégrité).

```
DECLARE
    cont_integrit_viol    EXCEPTION;
    PRAGMA EXCEPTION_INIT(cont_integrit_viol, -2291);
BEGIN
    .....
EXCEPTION
    WHEN cont_integrit_viol    THEN
        DBMS_OUTPUT.PUT_LINE ('violation de contrainte
d'intégrité');
END;
/
```


Exceptions définies par l'utilisateur



Exceptions définies par l'utilisateur

La commande **RAISE_APPLICATION_ERROR** permet d'afficher un message et un code d'erreur pour une exception définie par l'utilisateur.

Syntaxe :

RAISE_APPLICATION_ERROR(code_erreur, message);

- Le code erreur doit être entre -20000 and -20999.
- Le message d'erreur sera affiché comme pour une erreur classique.
- Le code PL/SQL s'arrête immédiatement et affiche l'erreur.

Exceptions définies par l'utilisateur

Exemple :

```
DECLARE
  x NUMBER;
  x_trop_petit EXCEPTION;
BEGIN
  .....
  IF x < 5 THEN RAISE x_trop_petit;
  END IF;
  .....
EXCEPTION
  WHEN x_trop_petit THEN
    raise_application_error(-20002, ' la valeur de x est trop petite !! ');
  .....
END;
/
```

Les fonctions de capture

- **SQLCODE**
 - Retourne la valeur numérique du code de l'erreur.
- **SQLERRM**
 - Retourne le message associé au numéro de l'erreur.

Les fonctions de capture

Exemple :

```
DECLARE
    v_code_erreur    NUMBER;
    v_message_erreur VARCHAR2(255);
BEGIN
    .....
EXCEPTION
    .....
WHEN OTHERS THEN
    v_code_erreur := SQLCODE;
    v_message_erreur := SQLERRM;
    INSERT INTO erreurs
    VALUES (v_code_erreur, v_message_erreur);
END;
/
```

Partie VI

-

Les curseurs

Les curseurs

Un curseur est un pointeur vers une zone mémoire SQL privée allouée pour le traitement d'une instruction SQL. Le curseur permet de traiter un à un les enregistrements (lignes de tables) ramenés par l'instruction SQL en question.

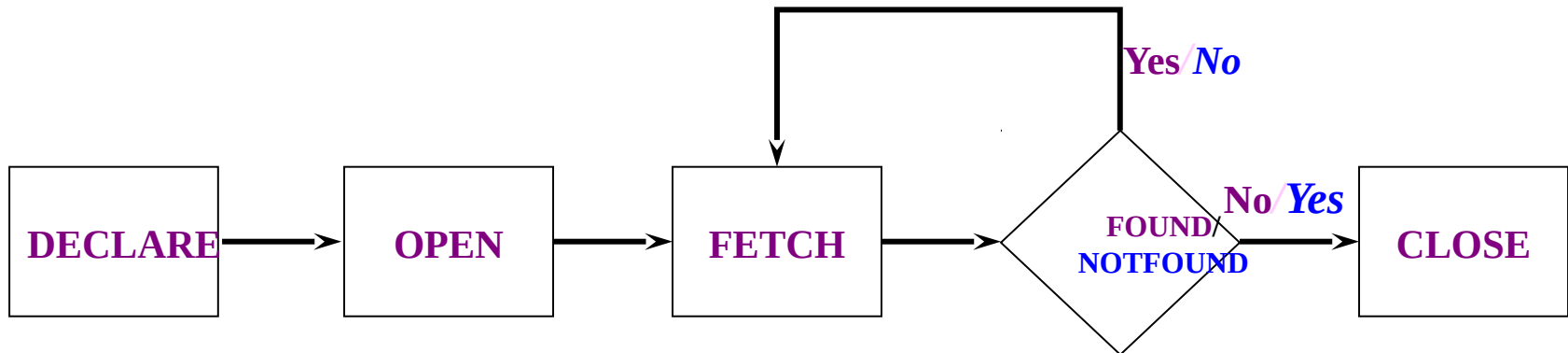
Les curseurs

Un curseur est un pointeur vers une zone mémoire SQL privée allouée pour le traitement d'une instruction SQL. Le curseur permet de traiter un à un les enregistrements (lignes de tables) ramenés par l'instruction SQL en question.

Deux types de curseurs peuvent être distingués :

- **Curseur implicite** : lorsqu'un utilisateur lance une commande SQL, Oracle génère un curseur pour le traitement de cette commande. Ce curseur créé et géré par Oracle est dit curseur implicite.
- **Curseur explicite** : si vous souhaitez gérer une commande SQL au sein de votre code PL/SQL, vous pouvez créer explicitement un curseur pour traiter une à une les lignes ramenées par la commande.

Contrôle des curseurs explicites



- *Crée une zone SQL*

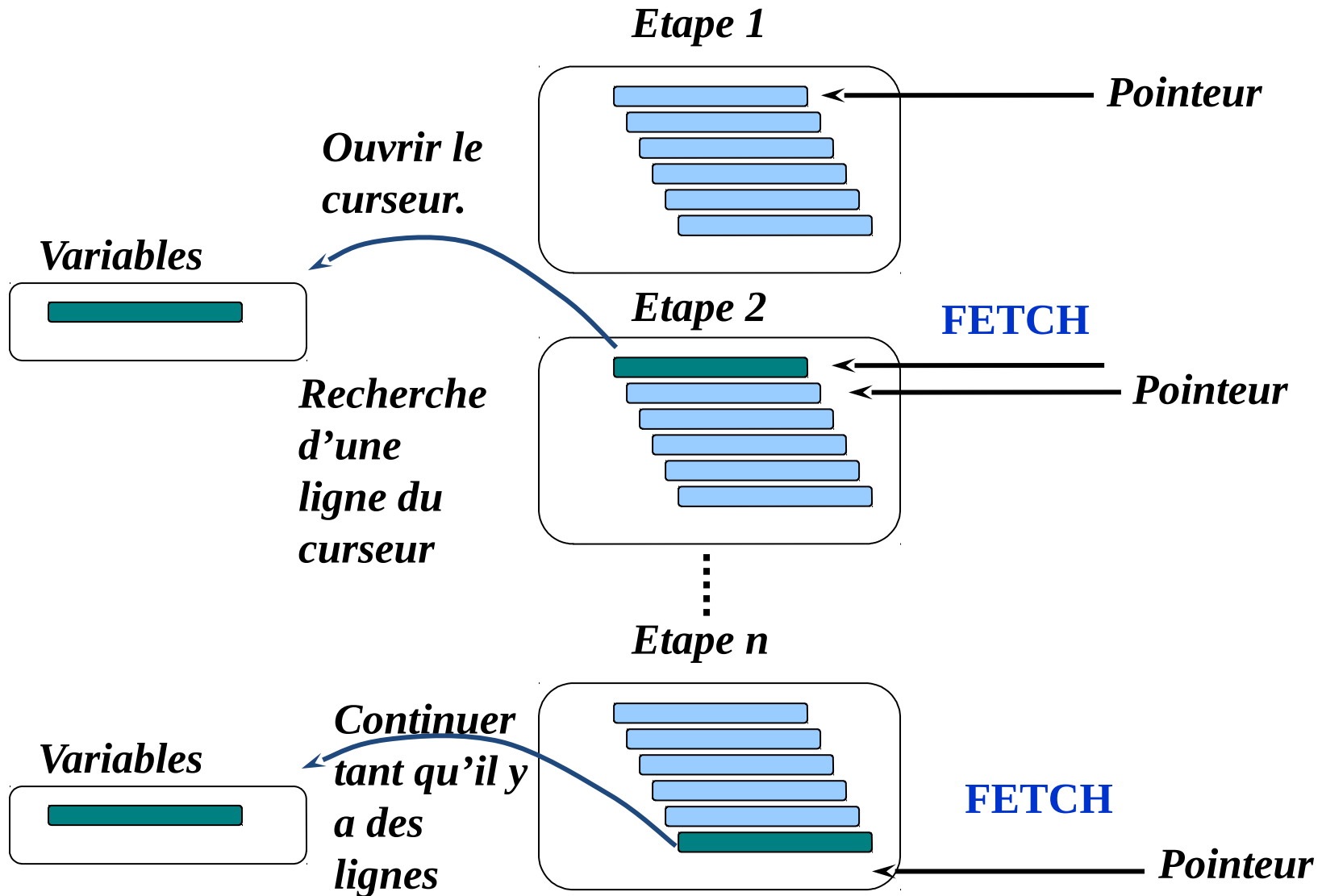
- *Pointer sur la première ligne du curseur*

- *Charger la ligne en cours dans des variables*

- *Test d'existence des lignes*
- *Retour à FETCH si ligne trouvée*

- *Libérer le curseur.*

Contrôle des curseurs explicites



Déclaration des curseurs

Syntaxe :

```
CURSOR nom_du_curseur IS un  
énoncé SELECT;
```

Déclaration des curseurs

Syntaxe :

```
CURSOR nom_du_curseur IS un  
énoncé SELECT;
```

- Ne pas inclure la clause **INTO** dans la déclaration du curseur.
- Si le traitement des lignes doit être fait dans un ordre spécifique, on utilise la clause **ORDER BY** dans la requête.

Déclaration des curseurs

Exemple :

```
DECLARE  
    CURSOR C1 IS  
    SELECT RefArt, NomArt, QteArt  
    FROM Article  
    WHERE QteArt < 500;
```

Ouverture du curseur

Syntaxe :

```
OPEN nom_du_curseur;
```

Ouverture du curseur

Syntaxe :

```
OPEN nom_du_curseur;
```

- **Ouvrir le curseur pour exécuter la requête et identifier l'ensemble actif.**
- **Utiliser les attributs des curseurs pour tester le résultat du **FETCH**.**

Recherche des données

Syntaxe :

```
FETCH nom_du_curseur  
INTO variable1, [variable2, ...];
```


Recherche des données

Syntaxe :

```
FETCH nom_du_curseur  
INTO variable1, [variable2, ...];
```

- Rechercher les informations de la ligne **en cours** et les mettre dans des variables.
- Les lignes sont traitées dans l'ordre de la table, à chaque **FETCH** la ligne suivante est traitée.

Recherche des données

Exemple 1 :

```
FETCH c1 INTO v_RefArt, v_NomArt, v_QteArt;
```

Recherche des données

Exemple 2 :

.....

LOOP

OPEN Cur_Etud;

FETCH Cur_Etud **INTO** Rec_Etud;

{traitements des données recherchées}

.....

END LOOP;

.....

Fermeture du curseur

Syntaxe :

```
CLOSE  
nom_du_curseur;
```

Fermeture du curseur

Syntaxe :

```
CLOSE  
nom_du_curseur;
```

- **Fermer le curseur après la fin du traitement des lignes.**
- **Rouvrir le curseur si nécessaire.**
- **On ne peut pas rechercher des informations dans un curseur si ce dernier est fermé.**

Les attributs du curseurs explicite

Obtenir les informations d'état du curseur :

Attribut	Type	Description
%ISOPEN	BOOLEAN	Prend la valeur TRUE si le curseur est ouvert.
%NOTFOUND	BOOLEAN	Prend la valeur TRUE si le FETCH le plus récent ne retourne aucune ligne.
%FOUND	BOOLEAN	Prend la valeur TRUE si le FETCH le plus récent retourne une ligne.
%ROWCOUNT	NUMBER	Retourne le nombre de lignes traitées jusqu'ici.

L'attribut %ISOPEN

- La recherche des lignes n'est possible que si le curseur est ouvert.
- Utiliser l'attribut **%ISOPEN** avant un **FETCH** pour tester si le curseur est ouvert ou non.

Exemple :

```
IF NOT C1%ISOPEN THEN  
    OPEN C1  
END IF;  
LOOP  
    FETCH C1 .....
```

Les attributs %FOUND, %NOTFOUND et %ROWCOUNT

- Utiliser l'attribut **%ROWCOUNT** pour fournir le nombre exact des lignes traitées.
- Utiliser les attributs **%FOUND** et **%NOTFOUND** pour formuler le test d'arrêt de la boucle.

Les attributs %FOUND, %NOTFOUND et %ROWCOUNT

Exemple :

```
LOOP
    FETCH curs1 INTO v_etudid, v_nom;
    IF curs1%ROWCOUNT > 20 THEN
        .....
    END IF;
    EXIT WHEN curs1%NOTFOUND;
END LOOP;
```

Les records

Un record est une variable qui contient une ligne entière d'une table.

Les records

Un record est une variable qui contient une ligne entière d'une table.

Exemple :

```
DECLARE
  CURSOR Etud_Curs IS
  SELECT etudno, nom, age, ard
  FROM   etud WHERE age < 26;
  Etud_Record Etud_Curs%ROWTYPE;           -- définition du record
BEGIN
  OPEN Etud_Curs;
  .....
  FETCH Etud_Curs INTO Etud_Record;       -- remplissage du record
  IF Etud_Record.age < 18 THEN             -- utilisation du record
  .....
END;
```

Les boucles FOR des curseurs

Syntaxe :

```
FOR nom_record IN nom_curseur  
  LOOP  
    -- traitement des informations  
  END LOOP;
```

- Un raccourci pour le traitement des curseurs explicites.
- OPEN, FETCH et CLOSE se font de façon implicite.
- Ne pas déclarer le record, il est déclaré implicitement.

Les boucles FOR des curseurs

Exemple :

```
DECLARE
  CURSOR Cur_Etud IS
    SELECT *
    FROM Etud ;
BEGIN
  FOR Rec_Etud IN Cur_Etud LOOP
    DBMS_OUTPUT.PUT_LINE(Rec_Etud.nom || ' ' || Rec_Etud.adr);
  END LOOP;
END;
/
```

Les attributs des curseurs implicites

Ils permettent de tester les résultats des énoncés SQL :

SQL%ROWCOUNT	Nombre de lignes affecté par l'énoncé SQL le plus récent (renvoie un entier).
SQL%FOUND	Attribut booléen qui prend la valeur TRUE si l'énoncé SQL le plus récent affecte une ou plusieurs lignes.
SQL%NOTFOUND	Attribut booléen qui prend la valeur TRUE si l'énoncé SQL le plus récent n'affecte aucune ligne.
SQL%ISOPEN	Prend toujours la valeur FALSE parce que PL/SQL ferment les curseurs implicites immédiatement après leur exécution.

Les attributs des curseurs SQL

Exemple : Supprimer de la table ITEM des lignes correspondants au lot 605.
Afficher le nombre de lignes supprimées.

```
DECLARE
    v_lot    NUMBER := 605;
BEGIN
    DELETE FROM item WHERE lot= v_lot;
    DBMS_OUTPUT.PUT_LINE (SQL%ROWCOUNT || 'Lignes supprimées');
END;
/
```

Partie VII

-

Déclencheurs (Triggers)

Les triggers (Déclencheurs)

- **Un trigger est un programme PL/SQL qui s'exécute automatiquement avant ou après une opération LMD (**Insert**, **Update**, **Delete**).**
- **Contrairement aux procédures, un trigger est déclenché automatiquement suite à un ordre LMD.**

Événement-Condition-Action

Un trigger est activé par un événement :

- Insertion, suppression ou modification sur une table

Si le trigger est activé, une condition est évaluée :

- Prédicat qui doit retourner vrai

Si la condition est vraie, l'action est exécutée :

- Insertion, suppression ou modification de la base de données

Composants du trigger

À quel moment se déclenche le trigger ?

- **BEFORE** : le code dans le corps du triggers s'exécute avant les évènements de déclenchement LMD.
- **AFTER** : le code dans le corps du triggers s'exécute après les évènements de déclenchement LMD.

Les composants du trigger

Les évènements du déclenchement :

Quelles sont les opérations LMD qui causent l'exécution du trigger?

- **INSERT**
- **UPDATE**
- **DELETE**
- **La combinaison des ces opérations**

Les composants du trigger

Le corps du trigger est défini par un bloc PL/SQL :

```
DECLARE  
BEGIN  
EXEPTION  
END;
```

Les composants du trigger

Syntaxe :

```
CREATE [OR REPLACE] TRIGGER <Nom_Trigger>
[BEFORE | AFTER] [INSERT [OR] DELETE [OR] UPDATE]
ON <Nom_Table>
[FOR EACH ROW][WHEN <Condition>]
DECLARE
BEGIN
EXCEPTION
END ;
/
```

Les composants du trigger

Exemple :

```
CREATE OR REPLACE TRIGGER StartFacture
  AFTER INSERT ON Facture
  FOR EACH ROW
  DECLARE
    VNbInsert Number ;
  BEGIN
    SELECT Nb_Insert INTO VNbInsert
      FROM Statistique
      WHERE Nom_Table='Facture' ;
    UPDATE Statistique
      SET Nb_Insert = VNbInsert+1
      WHERE Nom_Table='Facture' ;
  EXCEPTION
    WHEN No_Data_Found THEN
      INSERT INTO Statistique VALUES (1,'Facture') ;
  END ;
/
```

Manipulation des triggers

Activer ou désactiver un Trigger :

ALTER TRIGGER <Nom_Trigger> [**ENABLE** | **DISABLE**] ;

Supprimer un Trigger :

DROP TRIGGER <Nom_Trigger> ;

Déterminer les triggers de votre BD:

SELECT Trigger_Name FROM User_Triggers ;

Les attributs :Old et :New

Ces deux attributs permettent de gérer l'ancienne et la nouvelle valeurs manipulées.

Insert(.....) ... \models :New.nom_att

Delete

Where(.....) ... \models :Old.nom_att

Update ...

Set (.....) ... \models :New.nom_att

Where(.....) ... \models :Old.nom_att

Attention au « : » avant Old et New dans la section d'exécution !

Les attributs :Old et :New

Exemple 1 : Donner un trigger qui met à jour la table classe suite à une insertion d'un nouvel étudiant.

Etudiant (Id_Etu, Nom,..., Id_Classe)

Classe (Id_Classe, Nbr_Etu)

Les attributs :Old et :New

Exemple 1 : Donner un trigger qui met à jour la table classe suite à une insertion d'un nouvel étudiant.

Etudiant (Id_Etu, Nom,..., Id_Classe)

Classe (Id_Classe, Nbr_Etu)

```
CREATE OR REPLACE TRIGGER MajNbEtud
  AFTER INSERT ON Etudiant
  FOR EACH ROW
BEGIN
  UPDATE Classe
    SET Nbr_Etud = Nbr_Etud+1
    WHERE Id_Classe = :New.Id_Classe;
END ;
/
```

Les attributs :Old et :New

Exemple 2 : Donner un trigger qui met à jour la table classe suite à une insertion d'un nouvel étudiant si il a plus de 20 ans.

Etudiant (Id_Etu, Nom, Age, ..., Id_Classe)

Classe (Id_Classe, Nbr_Etu)

Les attributs :Old et :New

Exemple 2 : Donner un trigger qui met à jour la table classe suite à une insertion d'un nouvel étudiant si il a plus de 20 ans.

Etudiant (Id_Etu, Nom, Age, ..., Id_Classe)

Classe (Id_Classe, Nbr_Etu)

```
CREATE OR REPLACE TRIGGER MajNbEtud
  AFTER INSERT ON Etudiant
  FOR EACH ROW
  WHEN New.Age > 20
BEGIN
  UPDATE Classe
    SET Nbr_Etud = Nbr_Etud+1
    WHERE Id_Classe = :New.Id_Classe;
END ;
/
```

Les attributs :Old et :New

Exemple 2 : Donner un trigger qui met à jour la table classe suite à une insertion d'un nouvel étudiant si il a plus de 20 ans.

Etudiant (Id_Etu, Nom, Age, ..., Id_Classe)

Classe (Id_Classe, Nbr_Etu)

```
CREATE OR REPLACE TRIGGER MajNbEtud
  AFTER INSERT ON Etudiant
  FOR EACH ROW
  WHEN New.Age > 20                                -- Dans le "WHEN", pas de ":"
BEGIN
  UPDATE Classe
    SET Nbr_Etud = Nbr_Etud+1
    WHERE Id_Classe = :New.Id_Classe;               -- Dans BEGIN, ":" obligatoire
END ;
/
```

Les prédicats

inserting, updating et deleting

- **Inserting :**
True : Le trigger est déclenché suite à une insertion
False : Sinon
- **Updating :**
True : le trigger est déclenché suite à une mise à jour
False : sinon
- **Deleting :**
True : le trigger est déclenché après une suppression
False : sinon

Les prédicats

inserting, updating et deleting

Exemple :

```
CREATE OR REPLACE TRIGGER MajNbEtud
  AFTER INSERT OR DELETE ON Etudiant
  FOR EACH ROW
BEGIN
  IF Inserting THEN
    UPDATE Classe SET Nbr_Etud = Nbr_Etud+1
    WHERE Id_Cla = :New.Id_Cla ;
  END IF ;
  IF Deleting THEN
    UPDATE Classe SET Nbr_Etud = Nbr_Etud-1
    WHERE Id_Cla = :Old.Id_Cla ;
  END IF ;
END ;
/
```


Partie VIII

-

Fonctions et procédures

Les sous-Programmes

- Un sous programme est une séquence d'instruction PL/SQL qui possède un nom.
- On distingue deux types de sous programmes:
 - *Les procédures*
 - *Les fonctions*

Les sous-Programmes

- Une **procédure** est un sous programme qui exécute un code PL/SQL et ne retourne pas de résultat.
- Une **fonction** est un sous programme qui exécute un code PL/SQL et qui retourne des résultats. Une fonction ne peut pas faire de transactions.

Les procédures

Syntaxe :

```
DECLARE
    ...
    PROCEDURE <Nom_Proc>[(P1,...,Pn)] IS
        [Déclarations locales]
    BEGIN
        ...
    EXCEPTION
        ...
    END ;
BEGIN
    /* Appel a la procédure
    ...
EXCEPTION
    ...
END ;
/
```

Les procédures

Syntaxe :

P1,...,Pn suivent la syntaxe :

<Nom_Arg> [IN | OUT | IN OUT] <Type> Où :

IN : Paramètre d'entrée

OUT : Paramètre de sortie

IN OUT : Paramètre d'entrée/Sortie

Par défaut le paramètre est **IN**

Les procédures

Exemple :

DECLARE

```
PROCEDURE NouvSal (PNum IN Emp.Emp_Id %Type, PAug NUMBER ) IS
    VSa NUMBER (7,2) ;
BEGIN
    SELECT Sal INTO VSa FROM Emp WHERE emp_Id=PNum ;
    UPDATE Emp SET Sal = VSa+PAug WHERE Emp_Id=PNum ;
    COMMIT ;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employé inexistant') ;
END ;
```

```
BEGIN
    NouvSal(7550,500) ;
EXCEPTION
    WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE('Erreur');
END ;
/
```

Les procédures

Exemple :

DECLARE

VErr NUMBER ;

PROCEDURE NouvSal(PNum Emp.Emp_Id %TYPE, PAug NUMBER , PErr **OUT** NUMBER) **IS**

VSaI NUMBER (7,2) ;

BEGIN

SELECT Sal **INTO** VSaI **FROM** Emp **WHERE** emp_Id=PNum ;

UPDATE Emp **SET** Sal = VSaI+PAug **WHERE** Emp_Id=PNum ;

COMMIT ; PErr :=0

EXCEPTION

WHEN NO_DATA_FOUND THEN PErr :=1 ;

END ;

BEGIN

NouvSal(7550,500,VErr) ;

IF VErr=0 **THEN** DBMS_OUTPUT.PUT_LINE('Opération Effectuée') ;

ELSE DBMS_OUTPUT.PUT_LINE('Employé inexistant') ;

END IF ;

EXCEPTION

WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE('Erreur');

END ;

/

Les fonctions

Syntaxe

```
DECLARE
  [Déclarations globales]
  FUNCTION <Nom_fonc>[(P1,...,Pn)] RETURN Type IS
    [Déclarations locales]
  BEGIN
    ...
    RETURN valeur;
  EXCEPTION
    ...
  END ;
BEGIN
  /* Appel a la fonction
  ....
  EXCEPTION
    ....
  END ;
/
```


Les fonctions

Exemple :

```
DECLARE
VNomComplet VARCHAR2(40) ;
VErr NUMBER ;
FUNCTION NomComplet(PNum Emp. Emp_Id % TYPE, PErr OUT NUMBER )
RETURN VARCHAR2 IS
VLastName Emp.Last_Name %Type ; VFirstName Emp.First_Name %Type ;
BEGIN
    SELECT Last_Name, First_Name INTO VLastName, VfirstName
    FROM Emp WHERE Emp_Id=PNum ;
    PErr :=0 ;
    RETURN VLastName || ' ' || VFirstName ;
EXCEPTION
    WHEN NO_DATA_FOUND THEN PErr :=1 ; RETURN Null ;
END ;
BEGIN
VNomComplet := NomComplet(&Num, VErr) ;
IF VErr = 0
    THEN DBMS_OUTPUT.PUT_LINE('Nom Complet est : ' || VNomComplet) ;
    ELSE DBMS_OUTPUT.PUT_LINE('Employé inexistant') ;
END IF ;
END ;
/
```

Les procédures et les fonctions stockées

- **Sont des blocs PL/SQL qui possèdent des noms.**
- **Consistent à ranger le bloc PL/SQL compilé dans la base de données (CREATE).**
- **Peuvent être réutilisées sans être recompilées (EXECUTE).**
- **Peuvent être appelées de n'importe quel bloc PL/SQL.**
- **Peuvent être regroupées dans un package.**

Les procédures stockées

Syntaxe :

```
CREATE [ OR REPLACE ] PROCEDURE <Nom_Proc>[(P1,...,Pn)] IS  
  [Déclarations des variables locales]  
  BEGIN  
    ...  
  EXCEPTION  
    ...  
  END ;  
/
```

⊢ Procedure Created ⊢ *La procédure est correcte*

Ou

⊢ Procedure Created with compilation errors ⊢ *Corriger les erreurs* ⊢
SHOW ERRORS;

Les procédures stockées

Exemple :

```
CREATE [ OR REPLACE ] PROCEDURE
AjoutProd (PrefPro Prod.RefPro%TYPE,..., PPriUni Prod.PriUni%TYPE,
PErr OUT Number) IS

BEGIN
  INSERT INTO Prod VALUES(PrefPro,...,PPriUni) ;
  COMMIT ;
  PErr :=0 ;

EXCEPTION
  WHEN OTHER THEN
    PErr:=1;

END ;
/
```

Appel des procédures stockées

Syntaxe :

La procédure stockée est appelée par les applications soit:

- En utilisant son nom dans un bloc PL/SQL (autre procédure).
- Par **EXECUTE** dans SQL*Plus.

- Dans un bloc PL/SQL :

```
BEGIN  
    <Nom_Procedure>[<P1>, ..., <Pn>];  
END ;
```

- Sous SQL*PLUS :

```
EXECUTE <Nom_Procedure>[<P1>, ..., <Pn>];
```

Appel des procédures stockées

Exemple :

```
ACCEPT VRefPro          -- demande une valeur à l'utilisateur
ACCEPT VPriUni
.....
DECLARE
  VErr NUMBER;
BEGIN
  AjoutProd(&VRefPro,...,&VPriUni, VErr) ;
  IF VErr=0 THEN
    DBMS_OUTPUT.PUT_LINE('Opération Effectuée');
  ELSE DBMS_OUTPUT.PUT_LINE('Erreur');
  END IF ;
END ;
/
```

Les fonctions stockées

Syntaxe :

```
CREATE [ OR REPLACE ] FUNCTION <Nom_Fonc>[(P1,...,Pn)]  
    RETURN Type IS  
    [Déclarations des variables locales]  
    BEGIN  
        Instructions SQL et PL/Sql  
        RETURN (Valeur)  
    EXCEPTION  
        Traitement des exceptions  
    END ;  
/
```

⊢ **function Created** ⊢ *La fonction est correcte*

Ou

⊢ **function Created with compilation errors** ⊢ *Corriger les erreurs* ⊢

SHOW ERRORS;

Les fonctions stockées

Exemple :

```
CREATE [ OR REPLACE ]
FUNCTION NbEmp (PNumDep Emp.Dept_Id%Type, PErr OUT Number)
RETURN Number IS
    VNb Number(4) ;

BEGIN
    SELECT Count(*) INTO VNb FROM Emp WHERE Dept_Id=PNumDep;
    PErr :=0
    RETURN VNb;

EXCEPTION
    WHEN OTHER THEN
        PErr :=1 ;
        RETURN Null ;

END ;
/
```


Appel des fonctions stockées

Syntaxe :

La fonction stockée est appelée par les applications soit :

- dans une expression dans un bloc PL/SQL .
- par la commande **EXECUTE** (dans SQL*PLUS).

- Dans un bloc PL/SQL :

BEGIN

<var> := <Nom_fonction>[<P₁>, ..., <P_n>]

END ;

- Sous SQL*PLUS :

EXECUTE :<var> := <Nom_fonction> [<P₁>, ..., <P_n>]

Appel des fonctions stockées

Exemple :

ACCEPT VDep

DECLARE

VErr Number;

VNb Number(4) ;

BEGIN

VNb := NbEmp(&VDep, VErr) ;

IF VErr=0 THEN

DBMS_Output.Put_Line('Le nombre d'employées est : ' || VNb);

ELSE

DBMS_Output.Put_Line('Erreur');

END IF ;

END ;

/

Appel des fonctions stockées

Exemple :

```
SQL> VARIABLE VNb NUMBER(4)
```

```
SQL> EXECUTE :VNb :=NbEmp(&VDep, VErr) ;
```

Procédure PL/SQL terminée avec succès.

```
SQL> PRINT VNb
```

```
      VNB  
-----  
      300
```

Appel des fonctions stockées

Exemple :

```
SQL> VARIABLE VNb NUMBER(4)
```

```
SQL> EXECUTE :VNb :=NbEmp(&VDep, VErr) ;
```

Procédure PL/SQL terminée avec succès.

```
SQL> PRINT VNb
```

```
      VNB  
-----  
      300
```

Les commandes **VARIABLE** et **PRINT** permettent de déclarer des variables (Bind Variables) et d'afficher leurs valeurs sous SQL*Plus.

Suppression des procédures et des fonctions stockées

Syntaxe :

```
DROP PROCEDURE nomprocedure;  
DROP FUNCTION nomfonction;
```

Procédures et fonctions stockées

Quelques commandes utiles :

- `SELECT` object_name, object_type `FROM` user_objects;
- `DESC` nomprocedure
- `DESC` nomfonction

Packages

- Un objet PL/SQL qui stocke d'autres types d'objet : procédures, fonctions, curseurs, variables, ...
- Consiste en deux parties :
 - Spécification (déclaration)
 - Corps (implémentation)
- Ne peut pas être appelé, ni paramétré ni imbriqué
- Permet a Oracle de lire plusieurs objets a la fois en memoire

Créer un Package: spécification

Contient la déclaration des curseurs, variables, types, procédures, fonctions et exceptions

```
CREATE [OR REPLACE] PACKAGE <Nom_Package>  
    IS [Déclaration des variables et Types]  
        [Déclaration des curseurs]  
        [Déclaration des procédures et fonctions]  
        [Déclaration des exceptions]  
END[<Nom_Package>];  
/
```


Exemple de spécification

```
Create Or Replace Package PackProd
  Is Cursor CProd Is Select RefPro , DesPro
    From Produit ;
  Procedure AjoutProd( PrefPro Prod.RefPro%Type ,
    ..., PErr Out Number );
  Procedure ModifProd( PrefPro Prod.RefPro%Type ,
    ..., PErr Out Number );
  Procedure SuppProd( PrefPro Prod.RefPro%Type ,
    ..., PErr Out Number );
  Procedure AffProd ;
EndPackProd ;
/
```

Créer un Package: corps

On implémente les procédures et fonctions déclarées dans la spécification

```
Create [Or Replace] Package Body <Nom_Package>    Is  
    [Implémentation procédures | fonctions]  
End [<Nom_Package>];  
/
```

Exemple de corps

```
Create Or Replace Package Body PackProd
Is
  Procedure AjoutProd (PrefPro Prod.RefPro%Type,
                      ..., PErr Out Number)
  Is
  Begin
      Insert Into Prod Values(PrefPro ,..., PPriUni);
      Commit;
      PErr:=0;
  Exception
      When Dup_Val_On_Index Then    PErr:=1;
      When Others Then    PErr:= 1;
  End;

  Procedure ModifProd (PrefPro Prod.RefPro%Type,
                      ..., PErr Out Number)
      Is B Boolean;
  Begin
      ...
  EndPackProd;
/
```

Utilisation du package

Les procédures et les fonctions définies dans un package sont appelées de la façon suivante :

`<NomPackage>.<NomProcédure>[(Paramètres)];`

`Var:= <NomPackage>.<NomFonction>[(Paramètres)];`

Utilisation du package

Les procédures et les fonctions définies dans un package sont appelées de la façon suivante :

`<NomPackage>.<NomProcedure>[(Paramètres)];`

`Var:= <NomPackage>.<NomFonction>[(Paramètres)];`

Utilisation du package

Exemple :

```
Accept VRef Prompt ' ..... ' ;
Accept VPri Prompt ' ..... ' ;
Declare
    VErr Number ;
Begin
    PackProd.ModifProd(&VRef, ..., &VPri, VErr );
    If VErr= 0 Then
        DBMS_Output.Put_Line( ' Traitement effectué ' );
    Else
        DBMS_Output.Put_Line( ' Erreur ' );
    End If ;
End ;
/
```

Partie IX

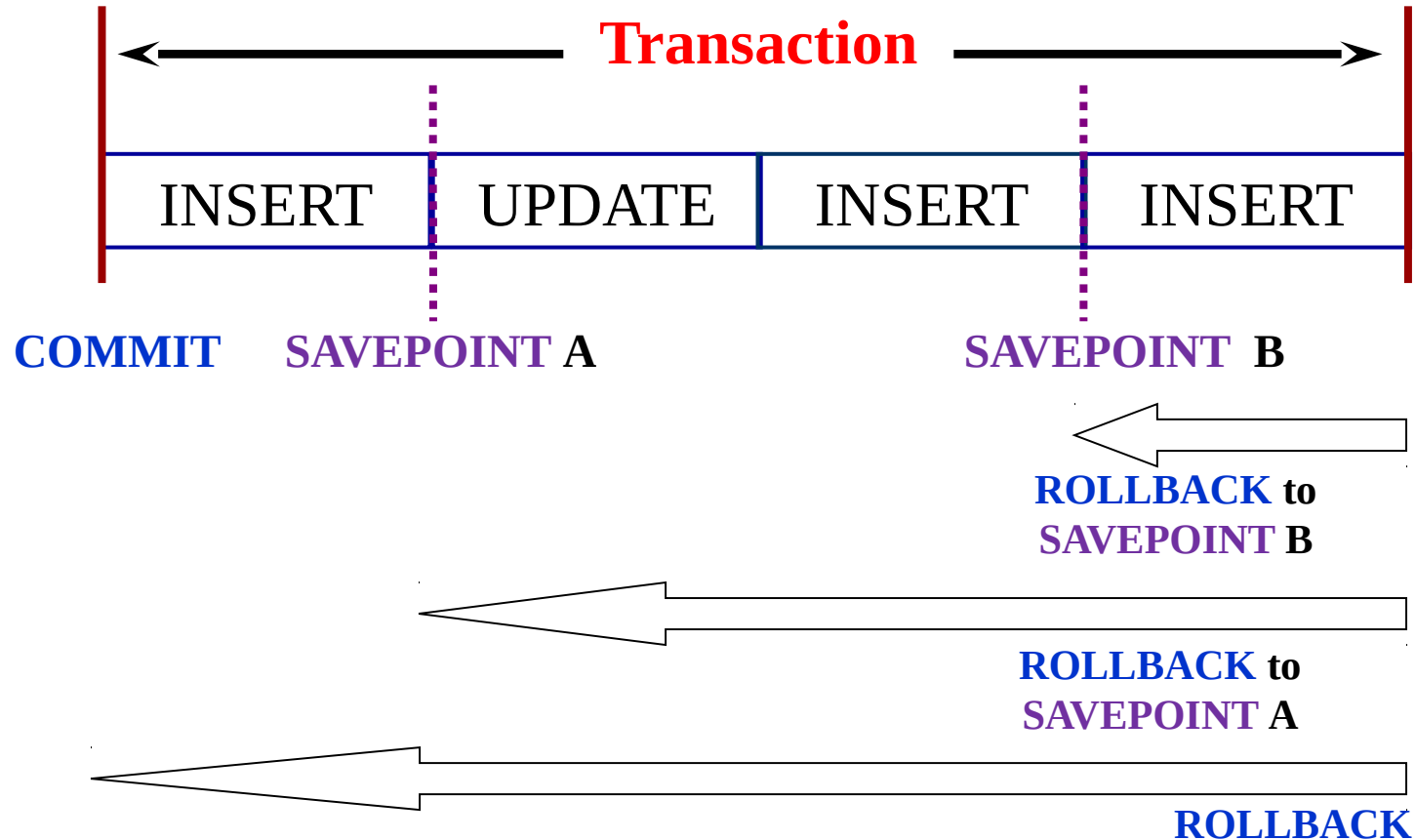
-

Contrôle de transactions

COMMIT et ROLLBACK

- Une transaction commence avec la première commande SQL après un **COMMIT** ou un **ROLLBACK**.
- Utiliser le **COMMIT** ou le **ROLLBACK** de SQL pour terminer une transaction.
- **COMMIT** applique tout ce qui a été fait dans la transaction, **ROLLBACK** annule toutes les opérations.

La commande ROLLBACK



Contrôle de transactions

Déterminer le traitement des transactions pour le bloc PL/SQL suivant :

```
BEGIN
  INSERT INTO temp VALUES (1, 1, 'ROW 1');
  SAVEPOINT a;
  INSERT INTO temp VALUES (2, 2, 'ROW 2');
  SAVEPOINT b;
  INSERT INTO temp VALUES (3, 3, 'ROW 3');
  SAVEPOINT c;
  ROLLBACK TO SAVEPOINT b;
  COMMIT;
END;
/
```