PL/SQL

INTRODUCTION

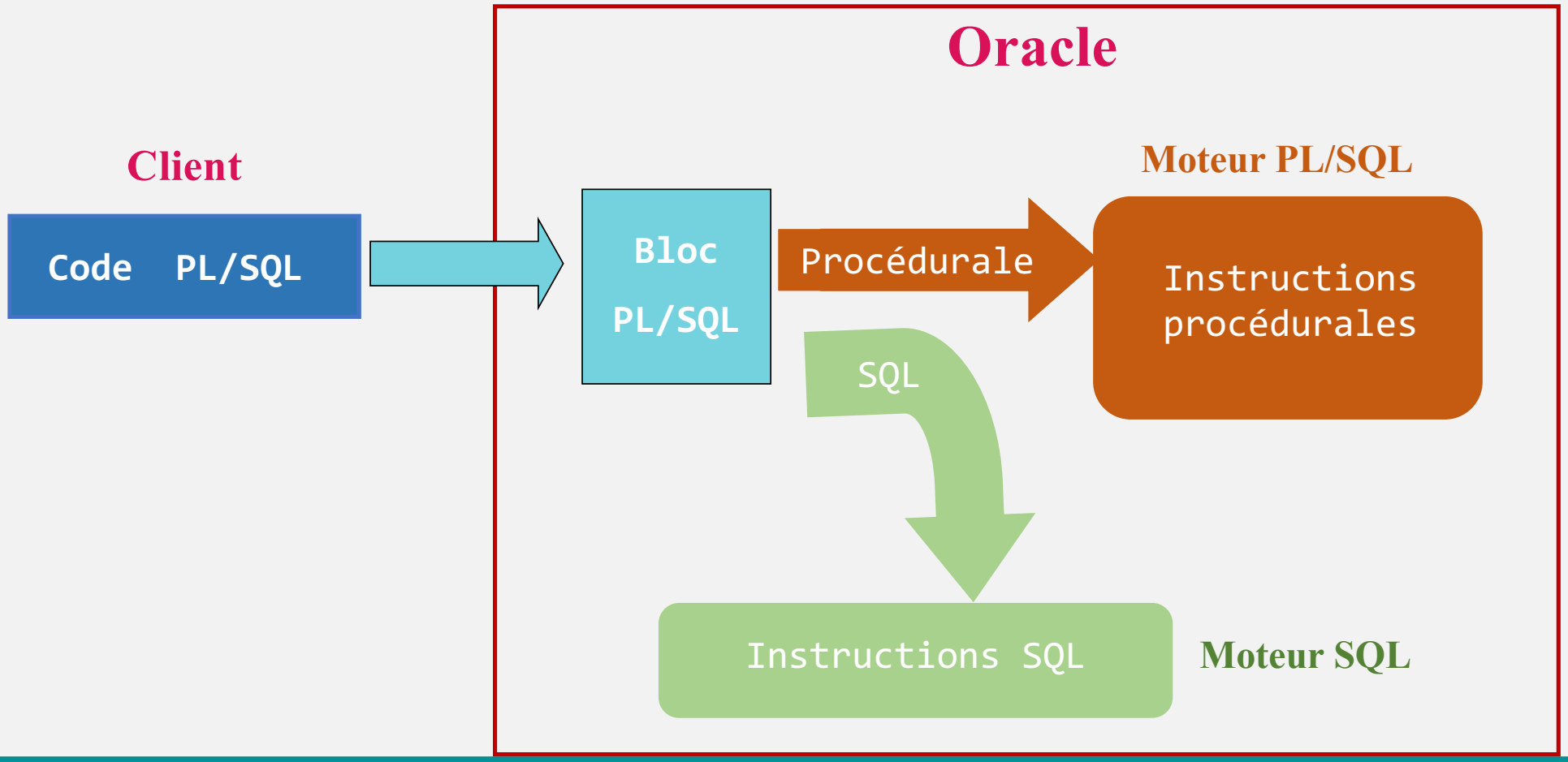
- PL/SQL (Procedural Language / SQL), l'extension procédurale proposée par Oracle pour SQL ,
- Il permet de combiner des requêtes SQL (SELECT, INSERT, UPDATE et DELETE) et des instructions procédurales (boucles, conditions...),

  - Créer des traitements complexes destinés à être stockés sur le serveur de base de données (objets serveur),

  - Comme on le sait, les structures de contrôle habituelles d'un langage (IF, WHILE…) ne font pas partie intégrante de la norme SQL. **Oracle les prend en compte dans PL/SQL**.

INTRODUCTION

➢ ARCHITECTURE



INTRODUCTION

➢ PERFORMANCE DE PL/SQL

- Dans un environnement client/serveur, chaque instruction SQL donne lieu à l'envoi d'un message du client vers le serveur suivi de la réponse du serveur vers le client.

- Un bloc PL/SQL donne lieu à un seul échange sur le réseau entre le client et le serveur. Les résultats intermédiaires sont traités côté serveur et seul le résultat final est retourné au client

INTRODUCTION

➢ Comparaison PL/SQL & SQL

SQL

Langage assertionnel et non procédural.
SELECT, INSERT, UPDATE, DELETE
CREATE, DROP, ALTER

PL/SQL

- Langage procédural, qui intègre des ordres SQL : SELECT, INSERT, UPDATE, DELETE COMMIT, ROLLBACK, SAVEPOINT
- Définition de variables, constantes, expressions, affectations
- Traitements conditionnels, répétitifs
- Traitement de Curseurs
- Traitement des erreurs et d'exceptions
- Etc…

INTRODUCTION

➢ OUTILS DE DÉVELOPPEMENT

Pour développer et exécuter du PL/SQL dans un environnement Oracle, plusieurs outils sont disponibles.

- Installer le SGBDR Oracle Database 11g (ou des versions supérieures 11g) Express Edition
- Installer Oracle SQL Developer

Structure d'un Bloc PL/SQL

- PL/SQL est un langage structuré en blocs, constitués d'un ensemble d'instructions,
- Un bloc PL/SQL peut être "externe", on dit alors qu'il est anonyme, ou alors stocké dans la base de données sous forme de procédure, fonction ou trigger (on lui affecte un nom et peut être réutilisés et appelés plusieurs fois),
- Un bloc PL / SQL contient trois parties :
    1. **Une partie déclarative,**
    2. **Une partie exécutable,**
    3. **Une partie pour la gestion des exceptions.**

Structure d'un Bloc PL/SQL

```
[DECLARE]
..........
 ..........
 BEGIN
..........
..........
[EXCEPTION]
..........
..........

END ;

 /
```

- La zone DECLARE sert à la déclaration des variables, des constantes, ou des curseurs
- La zone BEGIN constitue le corps du programme
- La zone EXCEPTION permet de préciser les actions à entreprendre lorsque des erreurs sont rencontrées.
- Le END répond au BEGIN précédent, il marque la fin du script.
- Le / permet de terminer le bloc PL/SQL

Structure d'un Bloc PL/SQL

- **Section DECLARE : déclaration de**
  - Variables locales simples
  - Variables tableaux
  - Cursors
- **Section BEGIN**
  - Section des ordres exécutables
  - Ordres SQL
  - Ordres PL/SQL
- **Section EXCEPTION**
  - Réception en cas d'erreur
  - Exceptions SQL ou utilisateur

```
[DECLARE]
..........
 ..........
 BEGIN
..........
..........
[EXCEPTION]
..........
..........

END ;

 /
```

Structure d'un Bloc PL/SQL

```
DECLARE --Facultative
   v_salary NUMBER;
   v_name VARCHAR2(100);

BEGIN -- Obligatoire
   SELECT salary INTO v_salary FROM employees WHERE employee_id = 101;
   IF v_salary > 5000 THEN
      DBMS_OUTPUT.PUT_LINE('Salaire élevé.');
   END IF;
EXCEPTION -- Facultative
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Aucun employé trouvé.');

END;
/
```

Structure d'un Bloc PL/SQL

Les commandes **LDD** (*Langage de Définition de Données*) **ne sont pas autorisées directement** dans la section exécutable d'un bloc PL/SQL. En effet, les instructions LDD comme CREATE, ALTER, et DROP ne peuvent pas être exécutées directement dans un bloc PL/SQL, car elles impliquent une modification de la structure de la base de données, ce qui est géré différemment par Oracle.

Pourquoi les LDD sont interdits ?

- Les instructions LDD entraînent un **commit implicite**, ce qui est incompatible avec le traitement transactionnel dans PL/SQL.
- PL/SQL est principalement conçu pour gérer les **données** (via des commandes DML comme SELECT, INSERT, UPDATE, DELETE) et les **logiques procédurales**.

Les Variables en PL/SQL

Les variables en PL/SQL servent à stocker des valeurs temporaires pour exécuter des calculs, manipuler des données, ou transmettre des valeurs entre instructions. Elles doivent être déclarées avant utilisation.

La Déclaration se fait dans la section DECLARE d'un bloc PL/SQL.

Syntaxe :

```
nom_variable TYPE [NOT NULL] [:= valeur_initiale];
```

Les Variables en PL/SQL

➢ Variables simples

- Variables de type SQL

```
nbr      NUMBER(2) ;
nom      VARCHAR(30) ;
minimum CONSTANT INTEGER := 5 ;
salaire NUMBER(8,2) ;
debut    NUMBER NOT NULL ;
```

- Variables de type booléen (TRUE, FALSE, NULL)

```
fin      BOOLEAN ;
reponse BOOLEAN DEFAULT TRUE ;
ok       BOOLEAN := TRUE;
```

Les Variables en PL/SQL

➢ Affichage des variables

- Pour d'afficher une chaine ou le contenu d'une variable

```
SET SERVEROUTPUT ON;
BEGIN
DBMS_OUTPUT.PUT_LINE('Bonjour tout le monde!');
 END ;
/
```

```
Bonjour tout le monde !
```

Les Variables en PL/SQL

```
DECLARE
   nom VARCHAR2(50); -- Déclaration d'une variable texte
   age NUMBER(3);     -- Déclaration d'un entier
BEGIN
   nom := ‘Ahmed'; -- Affectation
   age := 37;
   DBMS_OUTPUT.PUT_LINE('Nom : ' || nom || ', Âge : ' || age);
END;
/
```

Les Variables en PL/SQL

➢ Types PL/SQL spécifiques :

PL/SQL ajoute des types utiles pour le traitement procédural :

- **BINARY_INTEGER** : Nombres entiers (rapide pour les calculs).
- **PLS_INTEGER** : Entier rapide et optimisé pour PL/SQL.
- **%TYPE** : Récupère le type d'une colonne ou variable existante.

```
DECLARE
   nom employees.last_name%TYPE; -- Même type que la colonne last_name
BEGIN
   nom := 'Martin';
END;
```

Les Variables en PL/SQL

➢ Variable Bornée

Permet de définir un nouveau type basé sur un type existant, avec éventuellement des contraintes.

```
SUBTYPE <nom_type> IS TYPE [(constraint)][NOT NULL]
```

```
DECLARE
SUBTYPE type_nombre is number(3,0);-- Sous-type basé sur NUMBER
Commission type_nombre; ---- Variable utilisant ce sous-type
Begin
Commission:=123;
DBMS_OUTPUT.PUT_LINE('Valeur de commission est : '|| Commission);
End; /
```

Les Structures

Une **structure** est un **type composé** en PL/SQL.Elle permet de **stocker plusieurs champs de données** (valeurs de types différents) dans une seule variable. En PL/SQL, ces structures sont définies en utilisant le mot-clé **RECORD**, et on les appelle des **enregistrements**.

- *Syntaxe pour déclarer une structure :*

```
TYPE <nom_structure> IS RECORD (
   <nom_champ1> TYPE [NOT NULL] [:= valeur_par_defaut],
   <nom_champ2> TYPE [NOT NULL] [:= valeur_par_defaut],
   ...
);
```

- *Syntaxe d'utilisation d'un enregitrement*

```
<nom_enregistrement> <nom_structure>
```

Les Variables en PL/SQL

```
DECLARE
   TYPE employe_record IS RECORD (
      nom         VARCHAR2(50),
      salaire     NUMBER(10,2),
      departement VARCHAR2(30)
   );
   emp employe_record; -- Variable de type employe_record
BEGIN
   emp.nom := ‘Ahmed';
   emp.salaire := 5000;
   emp.departement := 'Finance';

   DBMS_OUTPUT.PUT_LINE('Nom : ' || emp.nom || ', Salaire : ' || emp.salaire ||
', Département : ' || emp.departement);
END;
/
```

Nom : Ahmed, Salaire : 5000, Département : Finance

Les Variables en PL/SQL

➢ Variables Basées

Une **variable basée** en PL/SQL est une variable dont le **type** ou la **structure** dépend directement des éléments d'une table ou d'une vue dans la base de données.

Basée sur une colonne (%TYPE) :

- La variable hérite du type de données d'une colonne spécifique.
- **Usage** : Garantit que la variable utilise le même type que la colonne.

```
DECLARE
   vsalaire employe.salaire%TYPE; -- Type basé sur la colonne "salaire" de
la table "employe"
BEGIN
   vsalaire := 5000;
   DBMS_OUTPUT.PUT_LINE('Salaire : ' || vsalaire);
END;
/
```

Les Variables en PL/SQL

➢ Variables Basées

Basée sur une ligne entière (%ROWTYPE) :

- La variable hérite de la structure d'une ligne complète (toutes les colonnes) d'une table ou d'une vue.
- **Usage** : Manipuler facilement toutes les colonnes d'une ligne.

```
DECLARE
   vemploye employe%ROWTYPE; -- Type basé sur une ligne de la table
"employe"
   BEGIN
      vemploye.nom := 'Ahmed';
      vemploye.salaire := 5000;
      DBMS_OUTPUT.PUT_LINE('Nom : ' || vemploye.nom || ', Salaire : ' ||
   vemploye.salaire);
   END;
   /
```

➢ Les Tableaux

les tableaux (appelés **collections**) sont des structures de données permettant de stocker plusieurs valeurs du même type. Ils sont très utiles pour manipuler des ensembles de données en mémoire.

Syntaxe :

```
TYPE <nom_tableau> IS TABLE OF <type_valeur> [NOT NULL]
INDEX BY {PLS_INTEGER | BINARY_INTEGER | VARCHAR2(TAILLE)};
<nom_variable> <nom_tableau>;
```

Les Variables en PL/SQL

- Les Tableaux

```
DECLARE
   TYPE tableau_entiers IS TABLE OF NUMBER
   INDEX BY PLS_INTEGER; -- Déclaration du type de tableau
   nombres tableau_entiers; -- Variable de type tableau
BEGIN
   nombres(1) := 10; -- Affectation d'une valeur
   nombres(2) := 20;
   nombres(3) := 30;

   FOR i IN 1..3 LOOP
      DBMS_OUTPUT.PUT_LINE('Élément ' || i || ' : ' || nombres(i));
   END LOOP;
END;
/
```

Les Variables en PL/SQL

- Tableaux de structures: Un tableau contenant des enregistrements (structures RECORD).

```
DECLARE
   TYPE employe_record IS RECORD (nom  VARCHAR2(50),  salaire NUMBER);
    -- Définition d'un enregistrement
   TYPE tableau_employes IS TABLE OF employe_record INDEX BY BINARY_INTEGER;
            -- Tableau de structures
   employes tableau_employes;
BEGIN
   employes(1).nom := ‘Ahmed';
   employes(1).salaire := 5000;
   employes(2).nom := ‘Malak';
   employes(2).salaire := 6000;

   FOR i IN 1..2 LOOP
      DBMS_OUTPUT.PUT_LINE('Employé ' || i || ' : ' || employes(i).nom || ',
Salaire : ' || employes(i).salaire);
   END LOOP;
END;/
```

Les Variables en PL/SQL

- Tableaux Prédimensionnés : Les tableaux **VARRAY** (prédimensionnés) ont une taille maximale fixée lors de la déclaration.

Syntaxe de déclaration

```
TYPE nom_tableau is varray(TAILLE) of type_valeur not null;
```

Syntaxe d'utilisation

```
<nom_variable> < nom_tableau>
```

```
DECLARE
TYPE tableau is varray(2) of  varchar2(30) not null;
Tab tableau;
Begin
Tab:=tableau('Salmi', 'Nadi');
End;
/
```

Les Variables en PL/SQL

➢ Les Actions Possibles Sur Un Tableau

Action	Description	Syntaxe
Lire une valeur	Obtenir une valeur d'un tableau via un index.	valeur := mon_tableau(index);
Compter les éléments	Obtenir le nombre d'éléments d'un tableau.	mon_tableau.COUNT
Ajouter un élément	Ajouter de nouveaux éléments dans un tableau imbriqué.	mon_tableau.EXTEND(n);
Supprimer un élément	Supprimer un élément spécifique ou tout le tableau.	mon_tableau.DELETE(index); / mo n_tableau.DELETE;
Vérifier l'existence d'un index	Vérifie si un index existe dans le tableau.	mon_tableau.EXISTS(index);
Obtenir le premier/dernier index	Obtenir l'index du premier élément.	mon_tableau.FIRST /LAST
Taille maximale (limite)	Obtenir la limite maximale d'un tableau VARRAY.	mon_tableau.LIMIT
Index d'élèment	Retourne l'index suivant/précèdent dans le tableau.	mon_tableau.NEXT/PRIOR(index)

Ordre SELECT

- Interroger une bases de données Oracle avec le PL/SQL
- Renvoyer un seul enregistrement

```
SELECT expression1 [, ...] INTO variable1 [, ...]
FROM table
WHERE condition;
```

- Cette instruction est utilisée pour récupérer une seule ligne et assigner les colonnes sélectionnées à des variables. Elle lève une erreur si aucune ligne ou plusieurs lignes correspondent à la condition.

ACCÈS À LA BASE DE DONNÉES ET ORDRES SQL

```
DECLARE
   v_nom VARCHAR2(50);
   v_salaire NUMBER;
BEGIN
   SELECT nom, salaire
   INTO v_nom, v_salaire
   FROM employes
   WHERE id_employe = 101;
   DBMS_OUTPUT.PUT_LINE('Nom : ' || v_nom || ', Salaire : ' || v_salaire);
END;
/
```

➢ Ordre SELECT

Si on utilise une **structure (RECORD)**, on peut assigner une ligne complète à cette structure.

```
SELECT * INTO structure
FROM table
WHERE condition;
```

```
DECLARE
   TYPE employe_record IS RECORD ( id_employe NUMBER,  nom VARCHAR2(50),
                                   salaire NUMBER  );

   v_employe employe_record;

BEGIN
   SELECT * INTO v_employe.      FROM employe     WHERE id_employe = 101;

   DBMS_OUTPUT.PUT_LINE('Nom : ' || v_employe.nom || ', Salaire : ' ||
v_employe.salaire);
END;
/
```

➢ Ordre SELECT : plusieurs lignes

BULK COLLECT est utilisé pour récupérer **plusieurs lignes** et les insérer dans un **tableau PL/SQL**.

Cela est plus efficace que de parcourir chaque ligne individuellement.

```
SELECT expression1 [, ...]
BULK COLLECT INTO tableau1 [, ...]
FROM table
WHERE condition;
```

ACCÈS À LA BASE DE DONNÉES ET ORDRES SQL

```
DECLARE
   TYPE tableau_salaire IS TABLE OF NUMBER;
   v_salaire tableau_salaire;
BEGIN
   SELECT salaire
   BULK COLLECT INTO v_salaire
   FROM employes
   WHERE departement = 'Finance';
   FOR i IN 1..v_salaire.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE('Salaire : ' || v_salaire(i));
   END LOOP;
END;/
```

✓ **Récapitulatif ;**

Cas	Syntaxe	Description
Une seule ligne	`SELECT colonne1 [, ...]` `INTO variable1 [, ...]`	Récupère une seule ligne et l'assigne à des variables.
Une structure	`SELECT * INTO structure`	Récupère une ligne entière dans une structure RECORD.

Plusieurs lignes	`SELECT colonne1 [, ...]`
	`BULK COLLECT INTO`
	`tableau1 [, ...]`
		Récupère plusieurs lignes dans un tableau PL/SQL.

Insertion

- **Syntaxe**

```
INSERT INTO <nom_table>  VALUES <variable_enregistrement> ;
```

- **Exemple**

```
BEGIN
   INSERT INTO employes (id_employe, nom, salaire,
departement)
   VALUES (201, ‘Ahmed', 5000, 'IT');
   DBMS_OUTPUT.PUT_LINE('Nouvel employé ajouté.');
END;
/
```

Update

- **Syntaxe**

```
UPDATE <nom_table>  SET <nom_champ>=
<variable_enregistrement>  [ WHERE CONDITION ];
```

- **Exemple**

```
BEGIN
   UPDATE employes
   SET salaire = salaire + 500
   WHERE id_employe = 101;

   DBMS_OUTPUT.PUT_LINE('Salaire mis à jour.');
END;
/
```

Delete

- **Syntaxe**

```
delete <nom_table>  [WHERE CONDITION ];
```

- **Exemple**

```
BEGIN
   DELETE FROM employes
   WHERE id_employe = 201;

   DBMS_OUTPUT.PUT_LINE('Employé supprimé.');
END;
/
```

➢ Les attributs des ordres SQL

Les **attributs SQL%** sont des variables système intégrées dans **PL/SQL**. Ils permettent d'obtenir des **informations sur l'exécution de la dernière instruction SQL** (comme INSERT, UPDATE, DELETE, ou SELECT INTO) dans un bloc PL/SQL.

Ils sont utilisés pour :

1. **Savoir si une commande SQL a modifié des données.**

   Exemples : Combien de lignes ont été mises à jour ? Une suppression a-t-elle été effectuée ?

1. **Gérer les flux logiques en fonction du résultat d'une commande SQL.**

   Exemples : Effectuer des actions alternatives si aucune ligne n'a été affectée.

1. **Déboguer ou vérifier les performances/les erreurs**

   Exemples : Afficher combien de lignes ont été insérées ou mises à jour.

➢ Les attributs des ordres SQL

SQL%FOUND

De Type **Boolean**, elle Retourne TRUE si la dernière commande SQL (comme INSERT, UPDATE ou DELETE) a modifié au moins une ligne.

Usage : Vérifier si une opération a réussi à modifier des données.

```
BEGIN
   UPDATE employes
   SET salaire = salaire + 500   WHERE departement = 'Finance';
   IF SQL%FOUND THEN
      DBMS_OUTPUT.PUT_LINE('L\'opération a modifié des lignes.');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Aucune ligne modifiée.');
   END IF;
END;
/
```

➢ Les attributs des ordres SQL

SQL%NOTFOUND

De Type **Boolean, elle** Retourne TRUE si la dernière commande SQL n'a modifié **aucune ligne**.

Usage : Vérifier si une opération n'a pas affecté de lignes.

```
BEGIN
   DELETE FROM employes
   WHERE id_employe = 999; -- ID inexistant

   IF SQL%NOTFOUND THEN
      DBMS_OUTPUT.PUT_LINE('Aucune ligne supprimée.');
   END IF;
END;
/
```

➢ Les attributs des ordres SQL

SQL%ROWCOUNT

De type **Number,** Retourne le **nombre de lignes** affectées par la dernière commande SQL (INSERT, UPDATE, DELETE).

Usage : Connaître précisément combien de lignes ont été affectées.

```
BEGIN
   UPDATE employes
   SET salaire = salaire * 1.1
   WHERE departement = 'Finance';

   DBMS_OUTPUT.PUT_LINE('Nombre de lignes mises à jour : ' ||
SQL%ROWCOUNT);
END;
/
```

➢ Clause SQL Returning

La clause **RETURNING** en SQL (Oracle) est utilisée dans les commandes **INSERT**, **UPDATE**, ou **DELETE** pour **retourner des données directement** après l'exécution de l'instruction. Cela évite d'exécuter une requête supplémentaire pour récupérer les valeurs modifiées ou supprimées.

```
DECLARE
   v_id NUMBER;
BEGIN
   INSERT INTO employes (nom, salaire, departement)
   VALUES (‘Ahmed', 5000, 'IT')
   RETURNING id_employe INTO v_id;

   DBMS_OUTPUT.PUT_LINE('Nouvel employé ajouté avec ID : ' || v_id);
END;
/
```

➢ Clause SQL Returning

Limites

- Inutilisable avec un ordre INSERT qui insère plusieurs enregistrement à partir d'une sous-requête
- Impossible d'utiliser « * » pour retourner l'ensemble des champs insérées dans l'enregistrement
- **Ne fonctionne qu'avec une seule commande DML** à la fois.

Les **structures de contrôle** en PL/SQL sont des instructions qui permettent de **contrôler le flux d'exécution** d'un programme. Elles déterminent **l'ordre** dans lequel les différentes parties du code sont exécutées, en fonction de conditions ou de répétitions. Elles servent à :

1. **Prendre des décisions dynamiques** : Exécuter des blocs de code différents selon des conditions (par exemple, si un employé dépasse un seuil de salaire, appliquer une règle spécifique).
2. **Gérer des répétitions** : Répéter un ensemble d'instructions pour traiter plusieurs éléments (par exemple, appliquer une opération à chaque employé dans une liste).
3. **Améliorer la lisibilité** : Organiser le code de manière logique et structurée.
4. **Réagir aux événements** : Exécuter des actions spécifiques en cas de situations exceptionnelles ou inattendues.

En résumé, elles sont indispensables pour rendre un programme intelligent, dynamique et efficace.

TRAITEMENTS ITERATIFS if_then_end if

- **Syntaxe**

```
IF condition THEN
      instruction1 ;
      instruction 2 ;
      ……..
      instruction n ;
 END IF;
```

- **Exemple**

```
DECLARE
   A integer := 120;      B  integer := 20;
BEGIN
   IF A>B THEN
   DBMS_OUTPUT.PUT_LINE('A est supérieur à B');
   END IF;
END;
```

➢ TRAITEMENTS ITERATIFS if_then_end if

- **Syntaxe**

```
IF condition1 THEN
   instruction1;
   instruction 2;
ELSE
   instruction3;
END IF;
```

- **Exemple**

```
DECLARE
   A integer := 120;
   B  integer := 20;
BEGIN
   IF A>B THEN
      DBMS_OUTPUT.PUT_LINE('A
      est supérieur à B');
   ELSE
      DBMS_OUTPUT.PUT_LINE('A
      est inférieur à B');
   END IF;
END;
```

➢ TRAITEMENTS ITERATIFS if_then_end if

Permet de tester plusieurs conditions. Exécute le premier bloc où la condition est vraie. Si aucune n'est vraie, exécute le bloc ELSE.

- **Syntaxe**

```
IF condition1 THEN
   -- Instructions si condition1 vraie
ELSIF condition2 THEN
   -- Instructions si condition2 vraie
ELSE
   -- Instructions si aucune condition vraie
END IF;
```

TRAITEMENTS ITERATIFS if_then_end if

- **Exemple**

```
BEGIN
   IF 10 > 20 THEN
      DBMS_OUTPUT.PUT_LINE('10 est supérieur à 20'); -- Non
exécuté
   ELSIF 10 = 10 THEN
      DBMS_OUTPUT.PUT_LINE('10 est égal à 10'); -- Exécuté
   ELSE
      DBMS_OUTPUT.PUT_LINE('Aucune condition n\'est vraie'); -
- Ignoré
   END IF;
END;
/
```

➢ TRAITEMENTS CONDITIONELS AVEC CASE

Alternative plus compacte à IF ... ELSIF. Évalue une série de conditions dans l'ordre. Exécute les instructions associées à la **première condition vraie** ou le bloc ELSE si aucune condition n'est vraie.

- **Syntaxe**

```
CASE
   WHEN condition1 THEN
      -- Instructions si condition1 vraie
   WHEN condition2 THEN
      -- Instructions si condition2 vraie
   ELSE
      -- Instructions par défaut si aucune condition vraie
END CASE;
```

➢ TRAITEMENTS CONDITIONELS AVEC CASE

- **Exemple**

```
DECLARE
   score NUMBER := 85;
BEGIN
   CASE
      WHEN score >= 90 THEN
         DBMS_OUTPUT.PUT_LINE('Excellent'); -- Ignoré
      WHEN score >= 70 THEN
         DBMS_OUTPUT.PUT_LINE('Bien'); -- Exécuté
      ELSE
         DBMS_OUTPUT.PUT_LINE('Insuffisant'); -- Ignoré
   END CASE;
END;
/
```

L'INSTRUCTION LOOP

Répète les instructions **indéfiniment** jusqu'à ce qu'une instruction EXIT mette fin à la boucle.

- **Syntaxe**

```
LOOP
   -- Instructions
   EXIT WHEN condition;
END LOOP;
```

- **Exemple**

```
DECLARE
   compteur NUMBER := 1;
BEGIN
   LOOP
      DBMS_OUTPUT.PUT_LINE('Compteur : ' ||
compteur);
      compteur := compteur + 1;

      EXIT WHEN compteur > 5; -- Sort de la
boucle si la condition est remplie
   END LOOP;
END;
/
```

➢ L'INSTRUCTION WHILE

Répète les instructions **tant qu'une condition est vraie**. Si la condition est fausse dès le départ, la boucle n'est jamais exécutée.

- **Syntaxe**

```
WHILE condition LOOP
    -- Instructions
END LOOP;
```

- **Exemple**

```
DECLARE
   compteur NUMBER := 1;
BEGIN
   WHILE compteur <= 5 LOOP
      DBMS_OUTPUT.PUT_LINE('Compteur : ' ||
compteur);
      compteur := compteur + 1;
   END LOOP;
END;
/
```

➢ L'INSTRUCTION FOR

Répète un ensemble d'instructions pour un **nombre défini d'itérations**. L'index est incrémenté automatiquement.

```
FOR <indice> IN [ REVERSE ]  < valeur_initiale >. . <
valeur_finale> LOOP
      Commande pl/sql;
END LOOP;
```

- **Exemple**

```
BEGIN
   FOR i IN 1..5 LOOP
      DBMS_OUTPUT.PUT_LINE('Valeur de i : ' || i);
   END LOOP;
END; /
```