



Ecole Nationale des sciences appliquées Al-Hoceima Université Abdelmalek Essaadi



PL/SQL

S3

2022/2023

PLAN:

- **Introduction**
- **Les variables**
- **Structure de contrôle**
- **Gestion des Curseurs**
- **Gestion des exceptions et les transactions**
- **Les fonctions et procédures**
- **Les packages**
- **Les triggers**
- **Index, index b-tree, index bitmap**

INTRODUCTION

Nécessite de PL/SQL:

- PL/SQL: Procedural Language SQL
- SQL est un langage non procédural
- Parfois les traitements complexes sont difficiles à écrire, si l'on ne peut pas utiliser des variables et les structures de programmation comme les boucles, les conditions, etc.
- Pour cette raison, on doit avoir un langage procédural pour combiner des requêtes SQL avec des variables et des structures de programmation habituelles.

INTRODUCTION

Quelques caractéristique de PL/SQL:

- Extension du langage SQL : des requêtes SQL se combinent avec les structures de contrôles habituelles de la programmation structurée (blocs, boucles, ...)
- Sa syntaxe ressemble a celle du langage Pascal et Ada
- Un programme est constitué de procédures, de fonction, ...
- En générale, l'échange d'information entre les requêtes SQL et le reste du programme est effectué par des variables
- PL/SQL est un langage propriétaire d'Oracle. Il est crée par Oracle et utilisé dans le cadre de bases de données relationnelles.

INTRODUCTION

Quelques caractéristique de PL/SQL (Suite):

- ✓ PL/SQL: langage procédural d'Oracle étend SQL
 - Instruction spécifique a PL/SQL
 - Instructions SQL intégrées dans PL/SQL
 - Instructions du **LRD**: SELECT
 - Instructions de **LMD**: INSERT, UPDATE, DELETE
 - Instructions de **LDD**: CREATE, ALTER, DROP, RENAME, TRUNCATE, ..
 - Instructions du langage de contrôle des transactions (**LCT**): COMMIT, ROLLBACK, SAVEPOINT
 - Fonctions: TO_CHAR, TO_DATE, UPPER, SUBSTR, ...
 -

INTRODUCTION

Instructions spécifiques a PL/SQL

- Définition des variables
- Traitements conditionnels
- Traitement répétitifs (ou boucles)
- Traitement des curseurs
- Traitement des erreurs
- ...

INTRODUCTION

Base et utilisation de PL/SQL:

- PL/SQL : Langage basé sur les paradigmes de programmation procédurale et structuré
- Il est utilisé pour l'écriture des procédures stockées et des déclencheurs (triggers)
- On l'utilise aussi pour écrire des fonctions utilisateurs qui peuvent être exploitées dans les requêtes SQL, ainsi que pour des fonctions prédéfinies
- On l'utilise aussi dans plusieurs outils d'oracle: Forms, Report, ...

Structure d'un programme PL/SQL

Unité de base : blocs

- PL/SQL: n'interprète pas une commande mais un ensemble de commande contenues dans un bloc PL/SQL
- En général, un programme est organisé en blocs d'instructions de types:
 - Procédures anonymes
 - Procédures nommées
 - Fonction nommées
- Un bloc peut contenir plusieurs autres blocs

Structure d'un programme PL/SQL

Structure d'un bloc

```
DECLARE -- Section optionnelle
    -- Définition des variables
BEGIN -- Section obligatoire
    -- Implémentation : Les instructions à exécuter
Exception -- Section optionnelle
    -- Code de gestion des exceptions
END;
/ -- Obligatoire pour que vous puissiez exécuter le script
```

Seuls BEGIN et END
sont obligatoires

Les blocs, comme les
instructions, se terminent
par un « ; »

Structure d'un programme PL/SQL

Remarques :

- Les sections DECLARE et EXCEPTION sont facultatives.
- Chaque instruction se termine par un ;
- Les commentaires : -- sur une ligne ou /* sur plusieurs lignes */

VARIABLES

Variables en PL/SQL

- Identificateurs oracle:
 - Comporte 30 caractères au plus
 - Commence par une lettre
 - Peut contenir des lettres, des chiffres, _, \$, #, etc
- Pas sensible a la casse
- Porté habituelle des langages a blocs
- Doit être déclaré avant d'être utilisé

VARIABLES

Possibilités de placer les commentaires

- - - Commentaire sur une seule ligne
- /* Commentaire sur plusieurs lignes */

Types de variables:

- **Type habituels** qui correspondent aux types SQL ou Oracle : Intégrer, Number, Char, Varchar2, ...
- **Type composites** qui s'adaptent a la récupération des lignes, des colonnes et des tables SQL : %TYPE, %ROWTYPE
- **Type référence** : REF

VARIABLES

Déclaration d'une variable

Syntaxe: nom_variable [CONSTANT] type_donnée [NOT NULL] := [DEFAULT expr];

Exemple:

- Age integer;
- Nom varchar(20);
- dateNaissance date;
- Ok boolean := true;
- ...

NB: Déclarations multiples **interdites**

l, h integer ;

VARIABLES

Déclaration %TYPE

- Possible de déclarer une variable de même type qu'une colonne de table ou une vue
- **Exemple:** `nom emp.name%TYPE;`

Déclaration %ROWTYPE

- Une variable peut contenir toutes les colonnes d'une ligne d'une table
- **Exemple:** `employe emp%ROWTYPE;`

VARIABLES

Exemple

Considérant la table:

EMP(empno, ename, fonction, mgr, dateEmbauche, sal, comm, deptno)

Manipulation des variables composites:

```
employe emp%ROWTYPE;  
nom emp.ename%TYPE;  
  
SELECT INTO employe FROM emp WHERE ename='ABOUFASSIL';  
nom := employe.ename;  
Employe.deptno := 20;  
...  
INSERT INTO emp VALUES employe;
```

VARIABLES

Type RECORD(Enregistrement)

Equivalent a STRUCT du langage C.

Syntaxe:

```
TYPE nomRecord IS RECORD (  
    Champ1 type1,  
    Champ2 type2,  
    Champ3 type3,  
    ...  
);
```


VARIABLES

Exemple (Type RECORD)

```
DECLARE
    TYPE enreg IS RECORD (
        Num emp.empno%TYPE,
        Name emp.ename%TYPE,
        Job emp.job%TYPE
    );
    R_EMP enreg; -- variable record de type enreg

BEGIN
    R_EMP.num := 1;
    R_EMP.nom := 'BOUFASSIL';
    R_EMP.job := 'Ingénieur';
END;
```

VARIABLES

AFFECTATION

PLUSIEURS FAÇONS DE DONNER UNE VALEUR A UNE VARIABLE:

- Opérateur d'affectation (**:=**)
- Directive **INTO** de la requête SELECT.

EXEMPLE

- `Date_embauche := '12/01/2022';`
- `SELECT ename INTO nom FROM emp WHERE empno=37;`

NB:

- SELECT ne renvoie qu'une seule ligne,
- Dans Oracle, il n'est pas possible d'inclure la clause SELECT sans INTO dans une **procédure**
- Pour renvoyer plusieurs lignes, on va utiliser les **curseurs**.

VARIABLES

PROBLÈME DES CONFLITS DE NOMS

Si une variable porte le même nom qu'une colonne d'une table c'est la colonne qui l'emporte

EXEMPLE

```
DECLARE
    ename varchar(30) := 'Asmae';
BEGIN
    -- emp contient un champ ename
    DELETE FROM emp WHERE ename='Ali';
END;
/
```

NB: Pour éviter les conflits de nommages, préfixer les variables PL/SQL par v_.

VARIABLES

AFFICHAGE

- Pour plus de clarté, il est utile d'afficher les valeurs des variables
- Activer le retour écran: **SET SERVEROUTPUT ON;**
- Sortie standard, le paquetage : **DBMS_OUTPUT('...' || ...);**
- Un paquetage est un regroupement de procédures et de fonctions
- Concaténation de chaines : Opérateur ||

STRUCTURE DE CONTRÔLE

- Exécution d'un traitement en fonction d'une condition.

```
IF condition THEN
    instructions1
END IF;
```

```
IF condition THEN
    instructions1
ELSE
    instruction2
END IF;
```

```
IF condition1 THEN
    instruction 1;
ELSIF condition2 THEN
    instruction 2;
ELSIF ...
    ...
ELSE
    instruction n;
END IF;
```

STRUCTURE DE CONTRÔLE

LES TRAITEMENTS (IF ELSE)

- Les opérateurs utilisés dans les conditions sont les mêmes que dans SQL : =, <, ... IS NULL, LIKE, ...
- Dès que l'une des conditions est vraie, le traitement qui suit le THEN est exécuté.
- Si aucune condition n'est vraie, c'est le traitement qui suit le ELSE qui est exécuté

STRUCTURE DE CONTRÔLE

■ LECTURE D'UNE VARIABLE

ACCEPT n number PROMPT 'Veuillez saisir la valeur de n'

■ EXEMPLE:

```
ACCEPT n number PROMPT 'Veuillez saisir la valeur de n'
SET SERVEROUTPUT ON;
DECLARE
    n number;
BEGIN
    n := 3;
    IF (n>0) THEN
        DBMS_OUTPUT.PUT_LINE(' n est strictement positif');
    ELSIF(n=0) THEN
        DBMS_OUTPUT.PUT_LINE(' n est null');
    ELSE
        DBMS_OUTPUT.PUT_LINE(' n est null');
    END IF;
END;
/
```

STRUCTURE DE CONTRÔLE

Les traitements (CASE) Comme l'instruction IF, l'instruction CASE sélectionne une séquence d'instructions à exécuter. Cependant, pour sélectionner la séquence, l'instruction CASE utilise un sélecteur plutôt que plusieurs expressions booléennes.

STRUCTURE DE CONTRÔLE

■ CHOIX

```
CASE expression
    WHEN expr1 THEN instruction1;
    WHEN expr2 THEN instruction2;
    ...
    ELSE instruction;
END CASE;
```

- **NB**: expression peut avoir n'importe quel type simple (ne peut pas par exemple être *record*)

STRUCTURE DE CONTRÔLE

■ EXEMPLE:

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
    n integer;
```

```
BEGIN
```

```
    n := 3;
```

```
CASE n
```

```
    WHEN 1 THEN DBMS_OUTPUT.PUT_LINE('Lundi');
```

```
    WHEN 2 THEN DBMS_OUTPUT.PUT_LINE('Mardi');
```

```
    WHEN 3 THEN DBMS_OUTPUT.PUT_LINE('Mercredi');
```

```
    WHEN 4 THEN DBMS_OUTPUT.PUT_LINE('Jeudi');
```

```
    WHEN 5 THEN DBMS_OUTPUT.PUT_LINE('Vendredi');
```

```
    WHEN 6 THEN DBMS_OUTPUT.PUT_LINE('Samedi');
```

```
    ELSE DBMS_OUTPUT.PUT_LINE('Dimanche');
```

```
END CASE;
```

```
END;
```

```
/
```

STRUCTURE DE CONTRÔLE

- **BOUCLE « TANT QUE »:**

Exécution d'un traitement tant qu'une condition reste vraie.

```
WHILE condition LOOP  
    instructions;  
END LOOP;
```

STRUCTURE DE CONTRÔLE

EXEMPLE (CALCULE DE LA MOYENNE DE 10 ENTIERS):

```
SET SERVEROUTPUT ON
DECLARE
    compteur number(2);
    somme number(2) := 0;
    moyenne number(3, 1);
BEGIN
    compteur := 1;
    WHILE compteur <= 10 LOOP
        somme := somme + compteur;
        compteur := compteur + 1;
    END LOOP;
    moyenne := somme / 10;
    DBMS_OUTPUT.PUT_LINE('La moyenne est : ' || moyenne);
END;
/
```

STRUCTURE DE CONTRÔLE

■ BOUCLE « FAIRE ... TANT QUE »:

Les traitements (LOOP)

- Exécution d'un traitement plusieurs fois, le nombre n'étant pas connu mais dépendant d'une condition.

```
LOOP
    instructions;
EXIT WHEN condition;
    instructions;
END LOOP;
```

STRUCTURE DE CONTRÔLE

■ EXEMPLE (CALCULE DE LA MOYENNE DE 10 ENTIERS):

```
SET SERVEROUTPUT ON
DECLARE
    compteur number(2);
    somme number(2) := 0;
    moyenne number(3, 1);

BEGIN
    compteur := 1;
    LOOP
        somme := somme + compteur;
        compteur := compteur + 1;
    EXIT WHEN compteur > 10;
    END LOOP;
    moyenne := somme / 10;
    DBMS_OUTPUT.PUT_LINE('La moyenne est : ' || moyenne);

END;
/
```

STRUCTURE DE CONTRÔLE

■ BOUCLE « POUR »:

Les traitements (FOR LOOP)

- Exécution d'un traitement un certain nombre de fois. Le nombre étant connu.

```
For compteur IN inf..sup LOOP  
    instructions;  
END LOOP;
```

STRUCTURE DE CONTRÔLE

- **EXEMPLE:**

```
For i IN 1..100 LOOP  
    somme := somme + i;  
END LOOP;  
DBMS_OUTPUT.PUT_LINE('La somme de 1 a 100 est : ' || somme);
```


GESTION DES CURSEURS

■ DÉFINITION

Curseur: C'est une variable spéciale qui pointe sur le résultat d'une requête SQL. La déclaration d'un curseur est liée au texte de la requête.

Curseur: zone de mémoire de taille fixe, utilisée par le noyau d'Oracle pour analyser et interpréter tout ordre SQL.

GESTION DES CURSEURS

- **TYPES DE CURSEUR:**

Il existe deux types de curseurs: **Implicite** & **Explicite**

Implicite : créés et gérés par Oracle à chaque ordre SQL (lorsque la close INTO accompagne le SELECT).

Explicite : créés et gérés par le programmeur afin de pouvoir traiter un SELECT qui retourne plusieurs tuples.

GESTION DES CURSEURS

- **DÉCLARATION DU CURSEUR:**

Association d'un nom de curseur à une requête SELECT;
Se fait dans la section DECLARE d'un bloc PL/SQL;

```
DECLARE  
    CURSOR nom_curseur IS Requête_Select ;
```

- **EXEMPLE:**

```
DECLARE  
    CURSOR checkValidation IS  
    SELECT nom, note FROM etds WHERE note >12;
```

GESTION DES CURSEURS

■ OUVERTURE DU CURSEUR:

- ✓ Alloue un espace mémoire au curseur et positionne les éventuels verrous
- ✓ Après avoir déclaré le curseur, il faut l'ouvrir dans la section exécutable **BEGIN**.
- ✓ L'ouverture du curseur amorce l'analyse de **SELECT** et son exécution.
- ✓ La réponse est calculée et rangée dans un espace temporaire:
OPEN nom_curseur ;

■ EXEMPLE:

```
DECLARE
    CURSOR checkValidation IS
    SELECT nom, note FROM etds WHERE note >12;

BEGIN
    OPEN checkValidation;

END;
/
```

GESTION DES CURSEURS

▪ TRAITEMENT DES LIGNES:

- ✓ L'accès aux données se fait par la clause : **FETCH INTO**
- ✓ **FETCH** nom_curseur **INTO** var1, var2, ...
- ✓ **FETCH** permettant de récupérer une ligne de l'ensemble des lignes associés au curseur et stocker les valeurs dans des variables réceptrices
- ✓ Pour traiter plusieurs tuples, il faut utiliser une boucle.

FERMETURE DU CURSEUR:

- Après le traitement des lignes pour libérer la place mémoire, il faut fermer le curseur: **CLOSE** nom_curseur ;

■ EXAMPLE 1:

```
DECLARE
    CURSOR dept_20 IS
        SELECT ename, sal FROM emp WHERE deptno=20;
        nom emp.ename%TYPE;
        salaire emp.sal%TYPE;
BEGIN
    OPEN dept_20;
    Loop
        FETCH dept_20 INTO nom, salaire ;
        IF salaire>2500
        Then insert into resultat values(nom,salaire);
        End if
        Exit when salaire=5000;

        END LOOP;
    CLOSE dept_20;
END;
/
```

■ EXAMPLE 2:

```
DECLARE
    CURSOR checkValidation IS
        SELECT nom, note FROM etds WHERE note >12;
    nomEtd etds.nom%TYPE;
    noteEtd etds.note%TYPE;

BEGIN
    OPEN checkValidation;
        -- On récupère le premier enregistrement
        FETCH checkValidation INTO nomEtd, noteEtd ;
        -- S'il y a un enregistrement récupéré
        WHILE checkValidation %found LOOP
            -- Traitement de l'enregistrement récupéré
            FETCH checkValidation INTO nomEtd, noteEtd ; -- On récupère l'enregistrement suivant
        END LOOP;
    CLOSE checkValidation;

END;
/
```

- **EXEMPLE 3:** (création et remplissage d'une table 'mention' à partir de la table etudiant)

-- Création de la table

```
CREATE TABLE mention(nom varchar2(10), notegenerale number(7,2));
```

```
DECLARE
```

```
    CURSOR checkValidation IS SELECT nom, note FROM etds;  
    nom etds.nom%TYPE; noteGlob etds.note%TYPE;
```

```
BEGIN
```

```
    OPEN checkValidation;
```

```
        FETCH checkValidation INTO nom, noteGlob;
```

```
        WHILE checkValidation %found LOOP
```

```
            IF noteGlob > 12 THEN
```

```
                INSERT INTO mention VALUES (nom, notegenerale) ;
```

```
            END IF ;
```

```
            FETCH checkValidation INTO nom, noteGlob;
```

```
        END LOOP;
```

```
    CLOSE checkValidation;
```

```
END;
```

```
/
```


■ STATUT D'UN CURSEUR (ATTRIBUT):

- ✓ Les attributs d'un curseur sont des indicateurs sur son état.
- ✓ Quatre attributs permettent d'évaluer l'état du curseur:
 - **%Found** : vrai si le dernier **FETCH** a ramené un tuple.
 - **%NotFound** : vrai si le dernier **FETCH** n'a ramené aucun tuple.
 - **%RowCount** : compte le nombre de **FETCH** exécutés sur un curseur;
 - **%IsOpen** : vrai si le curseur est ouvert;

■ REMARQUE:

Avec un curseur implicite créé par le SGBD Oracle, les mêmes attributs aux colonnes sont disponibles à condition de les préfixer par SQL. Ces attributs réfèrent au dernier curseur implicite utilisé par l'application.

Curseur Implicite	Curseur Explicite
SQL%FOUND	nom-curseur%FOUND
SQL%NOTFOUND	nom-curseur%NOTFOUND
SQL%ISOPEN	nom-curseur%ISOPEN
SQL%ROWCOUNT	nom-curseur%ROWCOUNT

■ L'ATTRIBUT DE CURSEUR %ROWTYPE:

- ✓ L'attribut %ROWTYPE permet de déclarer une variable de même type que l'enregistrement de la table
- ✓ **Syntaxe** : `Nom_de_variable nom_table%ROWTYPE ;`

Exemple 1:

- ✓ **DECLARE** `enrg_emp emp%ROWTYPE;`
- ✓ Avec un curseur :
`CURSOR nom_curseur IS Requete_SELECT ;`
`nom_variable nom_curseur%ROWTYPE ;`
- ✓ Les éléments de la structure (nom_variable) sont identifiés par :
nom_variable.nom_colonne
- ✓ La structure est renseignée par le FETCH :
FETCH `nom_curseur INTO nom_variable`

■ UTILISATION SIMPLIFIÉE DES CURSEURS :

- ✓ L'utilisation FOR LOOP **remplace** OPEN, FETCH et CLOSE.
- ✓ Lorsque le curseur est invoqué, un enregistrement est automatiquement créé avec les mêmes éléments de données que ceux définies dans SELECT.

Exemple : (création et remplissage d'une table 'mention' à partir de la table etudiant)

```
CREATE TABLE mention(nom varchar2(10), notegenerale number(7,2));  
DECLARE  
    CURSOR checkValidation IS SELECT nom, note FROM etds;  
BEGIN  
    FOR enreg_e IN checkValidation LOOP  
        IF enreg_e.noteGlob > 12 THEN  
            INSERT INTO resultat VALUES (enreg_e.nom, enreg_e.notegenerale) ;  
        END IF;  
    END LOOP;  
END;
```

■ EXERCICE :

Ecrire un bloc PL/SQL permettant de:

- Créer la table Emp2 à partir de la table Emp.
- Utiliser un curseur pour sélectionner toutes les colonnes de la table
- Parcourir ce curseur afin d'insérer dans Emp2 les employés gagnant plus que 2000\$.
- Afficher le nombre des employés dans la table Emp2.

```
CREATE TABLE emp2 AS select * FROM emp;
TRUNCATE TABLE emp2;
SET SERVEROUTPUT ON
DECLARE
    CURSOR emp_cur IS SELECT * FROM EMP ;
    ligne emp_cur%rowtype; etudiant_count NUMBER(5) := 0;
BEGIN
    OPEN emp_cur;
    FETCH emp_cur INTO ligne ;
    WHILE emp_cur%FOUND LOOP
        IF ligne.sal>2000 THEN
            etudiant_count := etudiant_count+1;
            INSERT INTO emp2 VALUES
                (ligne.empno,ligne.ename,ligne.job,ligne.mgr,
                 ligne.hiredate,ligne.sal,ligne.comm,ligne.deptno);
        END IF;
        FETCH emp_cur INTO ligne ;
    END LOOP ;
    DBMS_OUTPUT.PUT_LINE ('Le nombre total des employes est ' || emp_cur%ROWCOUNT);
    DBMS_OUTPUT.PUT_LINE ('le nombre des employes ayant le salaire plus que 2000:' || etudiant_count );
    CLOSE emp_cur;
END;/
SELECT ename,sal FROM emp2;
```

■ UTILISATION D'UNE VARIABLE DE TYPE RECORD :

- ✓ Un record permet de définir des types composites.

```
DECLARE
  TYPE enreg IS RECORD (
    v1 TYPE1,
    v2 TYPE2
  );
```

EXEMPLE: Déclaration d'une variable de ce type : nom_variable nom_record

```
DECLARE
  TYPE enrg_emp IS RECORD (
    nom emp.ename %TYPE,
    salaire emp.sal %TYPE
  );
  e enrg_empl
```

■ EXAMPLE:

DECLARE

TYPE *etudiant* **IS RECORD** (

cne NUMBER,

nom VARCHAR2(10),

prenom VARCHAR2(10),

date_naiss DATE,

sexe VARCHAR2(10)

);

et *etudiant*;

BEGIN

et.cne := 144010586;

et.nom:='BOUFASSIL';

et.prenom := 'Asmae';

et.date_naiss := '12/01/1999';

et.sexe := 'FEMININ';

DBMS_OUTPUT.PUT_LINE ('L etudiant de nom:' || et.nom || 'et
prenom:' || et.prenom || 'de CNE:' || et.cne || 'Ne le:' || et.date_naiss || 'de sexe:' || et.sexe);

END;

/

■ MODIFICATION DES DONNÉES:

- ✓ La modification des données se fait habituellement avec **INSERT**, **UPDATE** ou **DELETE**, mais on peut utiliser: **FOR UPDATE** dans la déclaration du curseur.

■ OBJECTIF DU CURSEUR MODIFIABLE:

- ✓ Modification via SQL sur le n-uplet courant
- ✓ Permet d'accéder directement en modification suppression du nuplet récupéré par **FETCH**.

■ REMARQUE:

- ✓ Un curseur qui comprend **plus** d'une table dans sa définition **ne permet pas la modification des tables de BD.**
- ✓ Seuls les curseurs définis sur **une seule table sans fonction d'agrégation et de regroupement** peuvent être utilisés dans une MAJ : delete, update, insert avec le **CURRENT OF CURSOR.**

■ SYNTAXE:

```
CURSOR nomCurseur IS .... FOR UPDATE;  
  - - déclarations des variables et opérations  
WHERE CURRENT OF nomCurseur;
```

■ EXERCICE :

- ✓ Créer un curseur qui permet de supprimer tous les champ NULL de la colonne COMM dans la table EMP, et mettre le résultat dans une table EMP_COMM,

```
CREATE TABLE EMP_COMM AS SELECT ename,sal,comm FROM EMP;  
SET SERVEROUTPUT ON  
DECLARE  
    CURSOR mce IS SELECT ename, sal, comm FROM EMP FOR UPDATE;  
    nom EMP_COMM.ename%TYPE ; salaire EMP_COMM.sal%TYPE ;  
    commission EMP_COMM.comm%TYPE ;  
  
BEGIN  
    OPEN mce;  
    FETCH mce INTO nom, salaire, commision;  
    WHILE mce%FOUND LOOP  
        IF commision IS NULL THEN  
            DELETE FROM EMP_COMM WHERE CURRENT OF mce ;  
        END IF;  
        FETCH mce INTO nom, salaire, commision;  
    END LOOP ;  
    CLOSE emp_cur;  
  
END;  
/
```

GESTION DES EXCEPTIONS

Une exception se produit lorsque le moteur PL/SQL rencontre une instruction qu'il ne peut pas exécuter en raison d'une erreur qui se produit au moment de l'exécution.

GESTION DES EXCEPTIONS

EXEMPLE:

```
DECLARE
BEGIN
    DBMS_OUTPUT.PUT_LINE('Bonjour ');
    DBMS_OUTPUT.PUT_LINE('Le résultat est : '|| 2/0); /*Erreur*/
    DBMS_OUTPUT.PUT_LINE('à tous');
    INSERT INTO emp VALUES (NULL, 'a','b',1,to_DATE(17/11/1981),2,3,10); /*Erreur*/
END;
/
```

Affichage:

Bonjour

Rapport d'erreur -

ORA-01476: le diviseur est égal à zéro

ORA-06512: à ligne 4

01476. 00000 - "divisor is equal to zero"

*Cause:

*Action:

GESTION DES EXCEPTIONS

Si le moteur PL/SQL reçoit une instruction pour diviser un nombre par "0 ", alors le moteur PL/SQL le lancera comme une exception. L'exception n'est levée qu'à l'exécution par le moteur PL/SQL.

GESTION DES EXCEPTIONS

Ces erreurs ne seront pas capturées au moment de la compilation et, par conséquent, elles ne doivent être gérées qu'au moment de l'exécution.

GESTION DES EXCEPTIONS

Les exceptions empêcheront le programme de continuer à s'exécuter, donc pour éviter une telle condition, elles doivent être capturées et gérées séparément. Ce processus est appelé Exception-Handling (Gestion des exceptions), dans lequel le programmeur gère l'exception qui peut se produire au moment de l'exécution.

GESTION DES EXCEPTIONS

Types des exceptions

On a trois sortes d'exceptions:

- 1. Exceptions définies par l'utilisateur
- 1. Exceptions prédéfinies par le système
- 1. Exceptions non-prédéfinies par le système

GESTION DES EXCEPTIONS

Exceptions définies par l'utilisateur

- On doit les déclarer (leur donner un nom) dans la partie DECLARE.
- On doit les déclenchées par RAISE
- On peut les gestionner dans la partie EXCEPTION (aussi par leur nom)

GESTION DES EXCEPTIONS

Exceptions définies par l'utilisateur

- **SYNTAXE:**

```
DECLARE
```

```
    nom_erreur EXCEPTION;           -- On donne un nom à l'erreur
```

```
BEGIN
```

```
    IF ...
```

```
    THEN RAISE nom_erreur;         -- On déclenche l'erreur
```

```
EXCEPTION
```

```
    WHEN nom_erreur                -- Traitement de l'erreur
```

```
    THEN ..
```

```
END;
```

GESTION DES EXCEPTIONS

Exceptions définies par l'utilisateur

EXEMPLE:

```
DECLARE
  a NUMBER;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Saisir un nombre entre 1 et 10');
  a := &nombre ; /* utilisateur a saisi 20*/
  DBMS_OUTPUT.PUT_LINE(a);
END;
/
```

Affichage:

Saisir un nombre entre 1 et 10

20

GESTION DES EXCEPTIONS

Exceptions définies par l'utilisateur

```
DECLARE
  a NUMBER ;
  erreur EXCEPTION;
BEGIN
  a:=&nombre;
  IF a<1 OR a>10 THEN
    RAISE erreur;
  END IF;
EXCEPTION WHEN erreur THEN
  DBMS_OUTPUT.PUT_LINE (' le nombre doit entre 1 et 10 ');
END;
/
```

Affichage:

le nombre doit entre 1 et 10

GESTION DES EXCEPTIONS

Exceptions définies par l'utilisateur

```
DECLARE  
BEGIN  
    UPDATE emp SET ename='AHMED' WHERE empno=100;  
END;  
/
```

Affichage:

Procédure PL/SQL terminée.

GESTION DES EXCEPTIONS

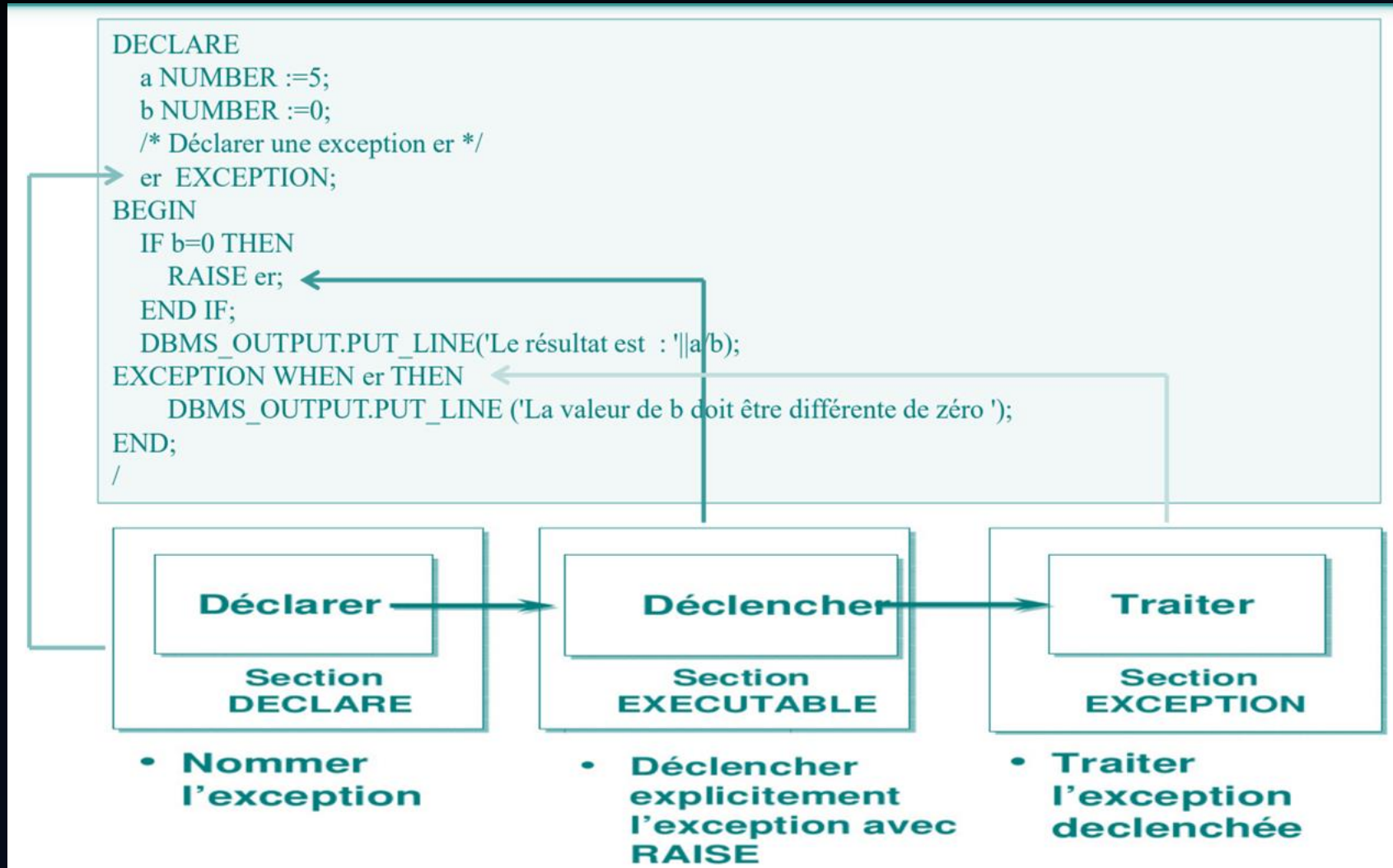
Exceptions définies par l'utilisateur

```
DECLARE
    erreur EXCEPTION;
BEGIN
    UPDATE emp SET ename='AHMED' WHERE empno=100;
    IF SQL%NOTFOUND THEN
        RAISE erreur;
    END IF;
    EXCEPTION
    WHEN erreur THEN
        DBMS_OUTPUT.PUT_LINE ('Erreur, employe n'existe pas !');
END;
/
```

Affichage:

Erreur, employe n'existe pas

GESTION DES EXCEPTIONS



GESTION DES EXCEPTIONS

Exceptions prédéfinies et non-prédéfinies par le système

Exceptions prédéfinies et non- prédéfinies par le système sont des exceptions définies par le système et gérées implicitement par la base de données Oracle. Elles sont définis dans le package Oracle STANDARD.

GESTION DES EXCEPTIONS

Exceptions prédéfinies par le système

- Utiliser le nom standard à l'intérieur du sous- programme de traitement des exceptions.
- NE doivent PAS être déclarées dans la partie DECLARE.
- NE doivent PAS être déclenchées par RAISE
 - elles sont déclenchées par le serveur Oracle
- On peut les gérer dans la partie EXCEPTION (aussi par leur nom)

GESTION DES EXCEPTIONS

Exceptions prédéfinies par le système

NO_DATA_FOUND: Déclenché si la commande SELECT INTO ne retourne aucune ligne

```
DECLARE
  vnom VARCHAR(10);
BEGIN
  SELECT ename INTO vnom FROM emp WHERE empno=1;
END;
/
```

Affichage :

Rapport d'erreur -

ORA-01403: aucune donnée trouvée

ORA-06512: à ligne 4

01403. 00000 - "**no data found**"

*Cause: No data was found from the objects.

*Action: There was no data from the objects which may be due to end of fetch.

GESTION DES EXCEPTIONS

Exceptions prédéfinies par le système

```
DECLARE
  vnom VARCHAR(10);
BEGIN
  SELECT ename INTO vnom FROM emp WHERE empno=1;
  EXCEPTION WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('! il n'y a aucune donnée !' );
END;
/
```

Affichage:


il n'y a aucune donnée

GESTION DES EXCEPTIONS

Exceptions prédéfinies par le système

TOO_MANY_ROWS : Si la commande SELECT INTO retourne plus d'une ligne.

```
DECLARE
  vnom VARCHAR(10);
BEGIN
  SELECT ename INTO vnom FROM emp WHERE deptno=10;
END;
/
```



...	Ename	Deptno
	President						10
	Manager						10
	Clerk						10

GESTION DES EXCEPTIONS

Exceptions prédéfinies par le système

TOO_MANY_ROWS : Si la commande SELECT INTO retourne plus d'une ligne.

```
DECLARE
  vnom VARCHAR(10);
BEGIN
  SELECT ename INTO vnom FROM emp WHERE deptno=10;
END;
/
```

Résultat:

Erreur !

ORA-01422: l'extraction exacte ramène plus que le nombre de
lignes
demandé

GESTION DES EXCEPTIONS

Exceptions prédéfinies par le système

TOO_MANY_ROWS : Si la commande SELECT INTO retourne plus d'une ligne.

```
DECLARE
  vnom VARCHAR(10);
BEGIN
  SELECT ename INTO vnom FROM emp WHERE deptno=10;
  EXCEPTION WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE('! il y a plus d'une ligne ! ');
END;
/
```

Résultat:

il y a plus d'une ligne

GESTION DES EXCEPTIONS

Exceptions prédéfinies par le système

ZERO_DIVIDE : tentative de division par zéro.

```
DECLARE  
  num NUMBER;  
BEGIN  
  num := 12/0;  
END;  
/
```

Résultat:

Erreur !

ORA-01476: le diviseur est égal à zéro

GESTION DES EXCEPTIONS

Exceptions prédéfinies par le système

```
DECLARE
  num NUMBER;
BEGIN
  num := 12/0;
  EXCEPTION WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('il y a une division sur 0' );
END;
/
```

Résultat:

il y a une division sur 0

GESTION DES EXCEPTIONS

Exceptions prédéfinies par le système

OTHERS : Permet de gérer n'importe quelle autre erreur déclenchée par notre programme.

```
DECLARE
BEGIN
  UPDATE emp SET empno = 'a' WHERE empno=7839;
  EXCEPTION WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('! il y a une erreur !' );
END;
/
```

Résultat:

il y a une erreur

GESTION DES EXCEPTIONS

Exceptions prédéfinies par le système

- `CURSOR_ALREADY_OPEN` : votre programme tente d'ouvrir un curseur déjà ouvert. Un curseur doit être fermé avant de pouvoir être rouvert. Un curseur `FOR LOOP` ouvre automatiquement le curseur auquel il se réfère. Ainsi, votre programme ne peut pas ouvrir ce curseur à l'intérieur de la `LOOP`.
- `DUP_VAL_ON_INDEX` : votre programme tente de stocker les valeurs en double dans une colonne de base de données contrainte par un index unique.
- `INVALID_CURSOR` : votre programme tente une opération de curseur illégale telle que la fermeture d'un curseur non ouvert.

GESTION DES EXCEPTIONS

Exceptions prédéfinies par le système

- `ACCESS_INTO_NULL` : Votre programme tente d'attribuer des valeurs aux attributs d'un objet non initialisé .
- `CASE_NOT_FOUND` : aucun des choix dans les clauses `WHEN` d'une instruction `CASE` n'est sélectionné et il n'y a pas de clause `ELSE`.
- `COLLECTION_IS_NULL` : votre programme tente d'appliquer des méthodes de collecte autres que `EXISTS` à une table ou un varray imbriqué non initialisé (atomiquement nul) ou le programme tente d'affecter des valeurs aux éléments d'une table ou d'un varray imbriqué non initialisé.

GESTION DES EXCEPTIONS

Exceptions prédéfinies par le système

- **INVALID_NUMBER** : Dans une instruction SQL, la conversion d'une chaîne de caractères en nombre échoue car la chaîne ne représente pas un nombre valide. (Dans les instructions procédurales, **VALUE_ERROR** est levée.) Cette exception est également levée lorsque l'expression de la clause **LIMIT** dans une instruction **FETCH** en bloc n'est pas évaluée à un nombre positif.
- **LOGIN_DENIED** : votre programme tente de se connecter à Oracle avec un nom d'utilisateur et/ou un mot de passe invalides.

GESTION DES EXCEPTIONS

Exceptions prédéfinies par le système

NO_DATA_FOUND : une instruction **SELECT INTO** ne renvoie aucune ligne, ou votre programme référence un élément supprimé dans une table imbriquée ou un élément non initialisé dans une table indexée par. Les fonctions d'agrégation SQL telles que **AVG** et **SUM** renvoient toujours une valeur ou un null. Ainsi, une instruction **SELECT INTO** qui appelle une fonction d'agrégat ne déclenche jamais **NO_DATA_FOUND**. L'instruction **FETCH** ne devrait finalement renvoyer aucune ligne. Ainsi, lorsque cela se produit, aucune exception n'est déclenchée.

GESTION DES EXCEPTIONS

Exceptions prédéfinies par le système

- NOT_LOGGED_ON : votre programme émet un appel de base de données sans être connecté à Oracle.
- PROGRAM_ERROR : PL/SQL a un problème interne.
- ROWTYPE_MISMATCH : la variable de curseur hôte et la variable de curseur PL/SQL impliquées dans une affectation ont des types de retour incompatibles. Par exemple, lorsqu'une variable de curseur hôte ouverte est transmise à un sous-programme stocké, les types de retour des paramètres réels et formels doivent être compatibles.

GESTION DES EXCEPTIONS

Exceptions prédéfinies par le système

- **SELF_IS_NULL**: votre programme tente d'appeler une méthode MEMBER sur une instance nulle. C'est-à-dire que le paramètre intégré SELF (qui est toujours le premier paramètre passé à une méthode MEMBER) est null.
- **STORAGE_ERROR**: PL/SQL manque de mémoire ou la mémoire a été corrompue.
- **SUBSCRIPT_BEYOND_COUNT**: votre programme référence une table imbriquée ou un élément varray à l'aide d'un numéro d'index supérieur au nombre d'éléments de la collection.

GESTION DES EXCEPTIONS

Exceptions prédéfinies par le système

- **SUBSCRIPT_OUTSIDE_LIMIT** : votre programme référence une table imbriquée ou un élément varray à l'aide d'un numéro d'index (-1 par exemple) qui est en dehors de la plage légale.
- **SYS_INVALID_ROWID**: La conversion d'une chaîne de caractères en un rowid universel échoue car la chaîne de caractères ne représente pas un rowid valide.
- **TIMEOUT_ON_RESOURCE**: un délai d'attente se produit pendant qu'Oracle attend une ressource.

GESTION DES EXCEPTIONS

Exceptions prédéfinies par le système

- `TOO_MANY_ROWS` : une instruction `SELECT INTO` renvoie plusieurs lignes.
- `VALUE_ERROR` : une erreur d'arithmétique, de conversion, de troncature ou de contrainte de taille se produit. Par exemple, lorsque votre programme sélectionne une valeur de colonne dans une variable caractère, si la valeur est plus longue que la longueur déclarée de la variable, PL/SQL abandonne l'affectation et déclenche `VALUE_ERROR`.

GESTION DES EXCEPTIONS

Exceptions prédéfinies par le système

Dans les instructions procédurales, `VALUE_ERROR` est levée si la conversion d'une chaîne de caractères en nombre échoue. (Dans les instructions SQL, `INVALID_NUMBER` est déclenché.)

- `ZERO_DIVIDE` : votre programme tente de diviser un nombre par zéro.

GESTION DES EXCEPTIONS

Exceptions non-prédéfinies par le système

- Chacune de ces erreurs a un numéro d'erreur Oracle standard (ORA-#####) et message d'erreur, mais pas de nom prédéfini
- Ces erreurs peuvent être traitées de deux façons:
 - ✓ En utilisant WHEN OTHERS
 - ✓ En déclarant l'exception (on lui donne un nom - identifiant) et en associant le nom au code d'erreur retourné par Oracle dans la partie DECLARE. Dans ce cas on peut les traiter comme les exceptions prédéfinies.

GESTION DES EXCEPTIONS

Exceptions non-prédéfinies par le système

```
DECLARE  
BEGIN  
    INSERT INTO emp VALUES (NULL, 'a','b',1,to_DATE(17/11/1981),2,3,10);  
END;  
/
```

Affichage:

Erreur !

ORA-01400: impossible d'insérer NULL dans ("SYSTEM"."EMP"."EMPNO")

GESTION DES EXCEPTIONS

Exceptions non-prédéfinies par le système

Traitement 2:

```
DECLARE
    erreur EXCEPTION;
    PRAGMA EXCEPTION_INIT(erreur, -01400);
BEGIN
    INSERT INTO emp VALUES (NULL, 'a','b',1,to_DATE(17/11/1981), 2, 3,10);
    EXCEPTION WHEN erreur THEN
        DBMS_OUTPUT.PUT_LINE('Échec de l insertion');
END;
/
```

Affichage :

Échec de l insertion

GESTION DES EXCEPTIONS

Exceptions non-prédéfinies par le système

- Ces exceptions ont un code et un message associé.
- La façon de gérer ces exceptions est de leur attribuer un nom à l'aide de Pragma EXCEPTION_INIT

- **Syntax:**

```
PRAGMA  
numero_erreur);
```

```
EXCEPTION_INIT(nom_exception,
```


GESTION DES EXCEPTIONS

Exceptions non-prédéfinies par le système

Pragma

- Un **pragma** est une instruction spéciale au compilateur qui est traitée au moment de la compilation au lieu de l'exécution.
- Un pragma appelé **EXCEPTION_INIT** demande au compilateur d'associer ou d'initialiser une exception définie par le programmeur avec un numéro d'erreur Oracle spécifique.

GESTION DES EXCEPTIONS

Exceptions non-prédéfinies par le système

Pragma

- Avec un nom pour cette erreur, vous pouvez alors lever cette exception et écrire un gestionnaire qui interceptera cette erreur.
- Alors que dans la plupart des cas, vous laisserez à Oracle le soin de lever ces exceptions système, vous pouvez également les lever vous-même.

GESTION DES EXCEPTIONS

Exceptions non-prédéfinies par le système

EXCEPTION_INIT

- Lorsque vous utilisez EXCEPTION_INIT, vous devez Fournir un nombre littéral pour le deuxième argument de l'appel de pragma.
- En nommant explicitement cette exception système, le but du gestionnaire d'exceptions est évident.

GESTION DES EXCEPTIONS

Exceptions non-prédéfinies par le système

EXCEPTION_INIT

- Le pragma `EXCEPTION_INIT` améliore la lisibilité de vos programmes en attribuant des noms à des numéros d'erreur autrement obscurs. Vous pouvez utiliser le pragma `EXCEPTION_INIT` plus d'une fois dans votre programme.
- Vous pouvez même affecter plusieurs noms d'exception au même numéro d'erreur.

GESTION DES EXCEPTIONS

SQLCODE et SQLERRM

Lorsqu'une exception se produit, vous pouvez récupérer le code d'erreur ou message d'erreur en utilisant deux fonctions.

- SQLCODE
 - ✓ Renvoie la valeur numérique associé au code de l'erreur.
 - ✓ Vous pouvez l'assigner à une variable de type number.
- SQLERRM
 - ✓ Renvoie le message associé au code de l'erreur.

LES TRANSACTIONS

Transaction de base de données

Une transaction de base de données consiste en une ou plusieurs instructions. Plus précisément, une transaction consiste en l'un des éléments suivants :

- LDD (Langage de définition de données) :
CREATE, ALTER, DROP
- LMD (Langage de manipulation de données) :
SELECT, INSERT, DELETE, UPDATE.
- LCD (Langage de contrôle de données) :
GRANT, REVOKE

LES TRANSACTIONS

Problèmes de cohérence et transaction:

Un SGBD doit pouvoir supporter :

- Plusieurs utilisateurs l'utilisant en parallèle
- Effectuant des opérations d'écriture et de lecture tout en garantissant la cohérence des données

LES TRANSACTIONS

Une transaction est un ensemble d'ordres (SQL) indivisibles, faisant passer la base de données d'un état cohérent à un autre en une seule étape.

LES TRANSACTIONS

Propriétés ACID

- Atomicité: Une transaction réussit SSi toutes ses opérations réussissent.
- Cohérence: Une transaction terminée laisse la base dans un état cohérent où les données sont intègres.
- Isolation: Les transactions doivent être rendues indépendantes les unes des autres.
- Durabilité: Les effets d'une transaction terminée sont persistant.

LES TRANSACTIONS

OPÉRATIONS

ROLLBACK: annule entièrement une transaction : toutes les modifications depuis le début de la transaction sont alors défaites.

COMMIT: valide entièrement une transaction : les modifications deviennent définitives et visibles à tous les utilisateurs.

LES TRANSACTIONS

SAVEPOINT

- Vous pouvez déclarer des marqueurs intermédiaires appelés savepoints dans le contexte d'une transaction.
- Les savepoints divisent une longue transaction en parties plus petites.

LES TRANSACTIONS

SAVEPOINT

- Vous avez ensuite la possibilité d'annuler ultérieurement le travail effectué avant le point actuel de la transaction mais après un savepoint déclaré dans la transaction. Par exemple, vous pouvez utiliser savepoints tout au long d'une longue série complexe de mises à jour, donc si vous faites une erreur, vous n'avez pas besoin de soumettre à nouveau chaque instruction.

LES TRANSACTIONS

Exemple

```
INSERT INTO article VALUES (1,'TV',690,50);  
INSERT INTO article VALUES (2,'Radio',130,10);  
INSERT INTO article VALUES (3,'GSM',350,20);  
INSERT INTO article VALUES (4,'PC',990,39);  
ROLLBACK;  
SELECT * FROM article;
```

LES TRANSACTIONS

Exemple

```
INSERT INTO article VALUES (1,'TV',690,50);  
INSERT INTO article VALUES (2,'Radio',130,10);  
INSERT INTO article VALUES (3,'GSM',350,20);  
INSERT INTO article VALUES (4,'PC',990,39);  
ROLLBACK;  
SELECT * FROM article;
```

Affichage

NUM_ART...	DESIGNA...	PRIX_UNI...	QTE_STOCK
------------	------------	-------------	-----------

LES TRANSACTIONS

Exemple

```
CREATE TABLE article(Num_Article NUMBER PRIMARY KEY,Designation  
VARCHAR(10),Prix_Unitaire NUMBER, Qte_stock NUMBER);  
INSERT INTO article VALUES (1,'TV',690,50);  
INSERT INTO article VALUES (2,'Radio',130,10);  
INSERT INTO article VALUES (3,'GSM',350,20);  
INSERT INTO article VALUES (4,'PC',990,39);  
COMMIT;  
SELECT * FROM article;
```

LES TRANSACTIONS

Exemple

```
CREATE TABLE article(Num_Article NUMBER PRIMARY KEY,Designation
VARCHAR(10),Prix_Unitaire NUMBER, Qte_stock NUMBER);
INSERT INTO article VALUES (1,'TV',690,50);
INSERT INTO article VALUES (2,'Radio',130,10);
INSERT INTO article VALUES (3,'GSM',350,20);
INSERT INTO article VALUES (4,'PC',990,39);
COMMIT;
SELECT * FROM article;
```

Affichage

NUM_ARTICLE	DESIGNATION	PRIX_UNITAIRE	QTE_STOCK
1	TV	690	50
2	Radio	130	10
3	GSM	350	20
4	PC	990	39

LES TRANSACTIONS

SAVEPOINT A;

INSERT INTO article VALUES (3,'GSM',350,20);

INSERT INTO article VALUES (4,'PC',990,39);

SAVEPOINT B;

INSERT INTO article VALUES (1,'TV',690,50);

INSERT INTO article VALUES (2,'Radio',130,10);

ROLLBACK SAVEPOINT B;

SELECT * FROM article;

Affichage

NUM_ARTICLE	DESIGNATION	PRIX_UNITAIRE	QTE_STOCK
3	GSM	350	20
4	PC	990	39

LES TRANSACTIONS

Les transactions : une validation/ annulation automatique

- Une validation automatique (automatic COMMIT) se produit dans les circonstances suivantes:
 - LDD (CREATE, ALTER, DROP).
 - LCD (GRANT, REVOKE).
 - Sortie normale de SQL Developer , sans émettre explicitement d'instructions COMMIT ou ROLLBACK.
- une annulation automatique se produit en cas d'arrêt anormal de SQL Developer



LES FONCTIONS ET PROCÉDURES

- PL/SQL est utilisé aussi pour définir les procédures et les fonctions stockées.
- Une procédure est un bloc PL/SQL nommé qui exécute une ou plusieurs actions
- Les procédures (fonctions) **permettent de** :
 - ✓ Réduire la complexité du code SQL (simple appel de procédure avec passage d'arguments)
 - ✓ Mettre en œuvre une architecture client/serveur de procédures et rendre indépendant le code client de celui des procédures
 - ✓ Mieux garantir l'intégrité des données: encapsulation des données par les procédures
 - ✓ Optimiser le code (les procédures sont compilées avant l'exécution du programme et elles sont exécutées immédiatement si elles se trouvent dans la SGA (zone mémoire gérée par ORACLE). De plus une procédure peut être exécutée par plusieurs utilisateurs (Réutilisation du code).

- Une procédure est une unité de traitement qui contient des commandes SQL relatives au LMD, des variables, des instructions PL/SQL, des constantes, et un gestionnaire d'erreurs.
- Une fonction est une procédure retournant une valeur.

Les procédures (fonctions) permettent de :

- Masquer la complexité du code SQL: simple appel de procédure avec passage de paramètres.
- Sécuriser l'accès aux données : accès seulement à certaines tables à travers les procédures.
- Mieux garantir l'intégrité des données: encapsulation des données par les procédures;

PROCÉDURES

■ Syntaxe

Une procédure est créée avec l'instruction **CREATE OR REPLACE PROCEDURE**.

```
CREATE [OR REPLACE] PROCEDURE procedure_name  
[(parameter_name [IN | OUT | IN OUT] type [, ...])]  
{IS | AS}  
BEGIN  
    < procedure_body >  
END procedure_name;
```

- **Procedure_name** spécifie le nom de la procédure.
- L'option **[OR REPLACE]** modifie une procédure existante.
- La liste de paramètres facultatifs contient **le nom, le mode et les types de paramètres**.
 - ✓ **IN** représente la valeur qui sera transmise de l'extérieur
 - ✓ **OUT** représente le paramètre qui sera utilisé pour renvoyer une valeur en dehors de la procédure.
 - ✓ **IN OUT** représente un argument en entrée sortie. Elle peut être lue et modifiée par la procédure

PROCÉDURES

Argument	Signification
IN	Valeur par défaut. Argument en entrée. Elle ne être modifiée par la procédure.
OUT	Argument en sortie (modifié par la procédure).
IN OUT	Argument en entrée sortie. Elle peut être lue et modifiée par la procédure.
TYPE	Type de l'argument sans spécification de la taille
DEFAULT	Affecte une valeur par défaut à l'argument.
IS	Permet la définition de la procédure.

PROCÉDURES

- **NB:** Contrairement au bloc anonyme de PL/SQL, la zone de déclaration n'a pas besoin d'être précédé du mot réservé DECLARE. Elle se trouve entre le IS et le BEGIN.
- **Syntaxe:**
 - Compilation d'une procédure:
 - @chemin absolu du fichier .sql
 - **start** fichier .sql
 - Exécution de la procédure:
 - **EXECUTE** fichier .sql
- **Remarque:**
 - Pour supprimer une procedure :
 - **DROP PROCEDURE** procedure_name

PROCÉDURES

- Les paramètres réels peuvent être passés de trois manières:
 - ✓ **Notation positionnelle** : le premier paramètre effectif est substitué au premier paramètre formel.
 - ✓ **Notation nommée** : le paramètre actuel est associé au paramètre formel à l'aide du symbole de
 - flèche (\Rightarrow)
 - ✓ **Notation mixte** : On mélange les deux notations en appel de procédure; cependant, la notation de position doit précéder la notation nommée.

PROCÉDURES

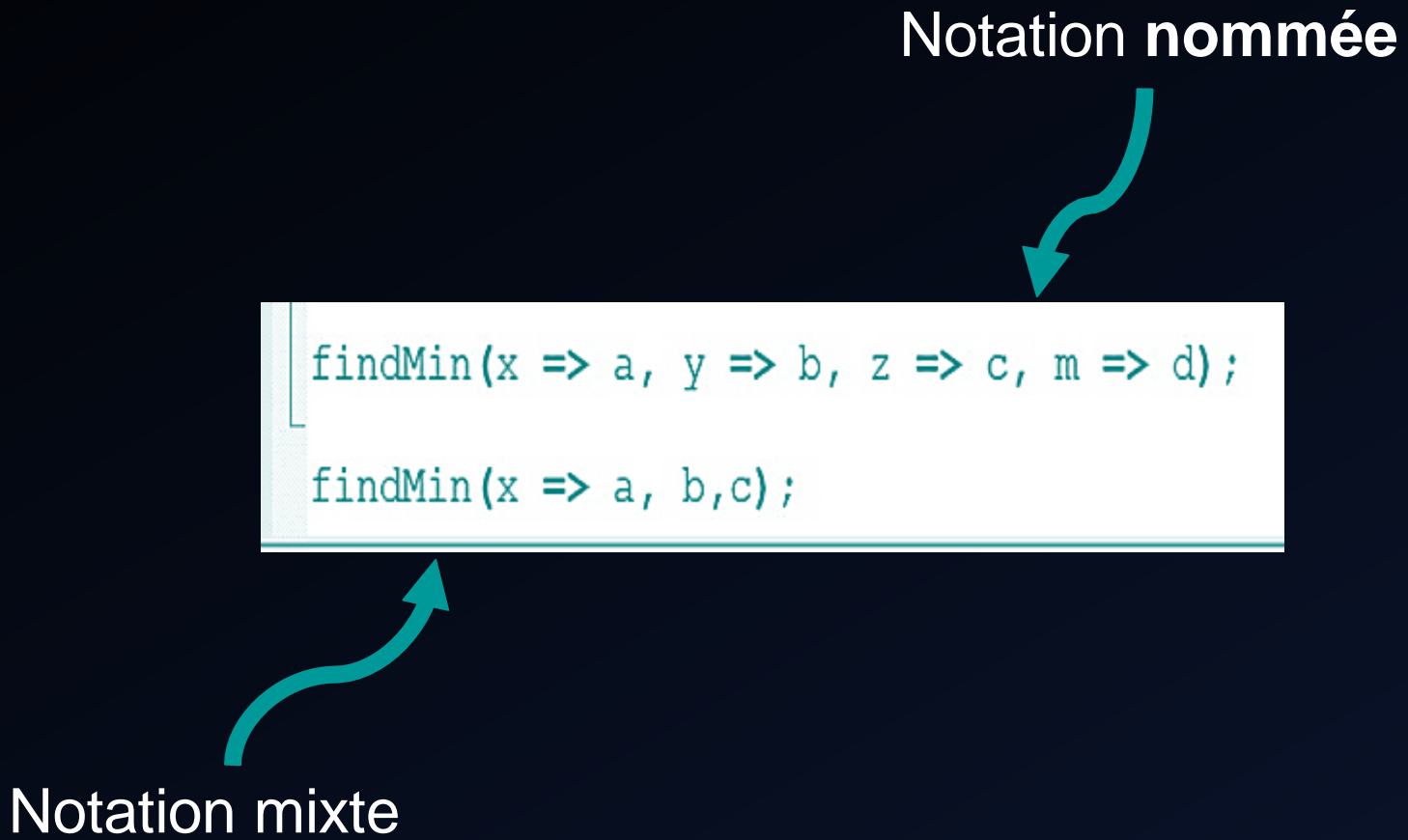
```
DECLARE
    a number;
    b number;
    c number;
    PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
    IF x < y THEN
        z := x;
    ELSE
        z := y;
    END IF;
END;

BEGIN
    a := 23;
    b := 45;
    findMin(a, b, c);
    dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;

/
```

PROCÉDURES

Notation **nommée**



```
findMin(x => a, y => b, z => c, m => d);  
findMin(x => a, b,c);
```

Notation mixte

PROCÉDURES

- Compter le nombre d'employés pour un département donné.

```
SET SERVEROUTPUT ON
```

```
CREATE OR REPLACE PROCEDURE proc_dept (p_no IN dept.deptno%TYPE)
```

```
IS
```

```
v_no NUMBER;
```

```
BEGIN
```

```
    SELECT COUNT(deptno) INTO v_no FROM emp WHERE deptno=p_no;
```

```
    DBMS_OUTPUT.PUT_LINE ('Nombre d'employés : ' || ' ' || v_no);
```

```
END;
```

```
/
```

PROCÉDURES

- Modifie le salaire d'un employé

```
SET SERVEROUTPUT ON
```

```
CREATE OR REPLACE PROCEDURE modifie_salaire ((id in number, taux in number)
```

```
IS
```

```
BEGIN
```

```
    UPDATE employe SET salaire=salaire*(1+taux) WHERE Id_emp= id;
```

```
    EXCEPTION
```

```
        WHEN no_data_found THEN
```

```
            DBMS_OUTPUT.PUT_LINE ('Employé inconnu: ' || to_char(id));
```

```
END;
```

```
/
```

PROCÉDURES

- Compilation de la procédure modifie_salaire:
 - ✓ SQL> **START** modifie_salaire
- Appel de la procédure modifie_salaire :

```
BEGIN  
    modifie_salaire (id => 15,-0.5);  
END;  
/
```

PROCÉDURES

- Si le script contient des erreurs, la commande **show err** permet de visualiser les erreurs.

```
SQL> start modifie_salaire

Warning: Procedure created with compilation errors.

SQL> show err
Errors for PROCEDURE MODIFIE_SALAIRE:

LINE/COL ERROR
-----
4/8      PLS-00103: Encountered the symbol "VOYAGE" when expecting one of the
following:
         := . ( @ % ;
         Resuming parse at line 5, column 21.

SQL> 14

4*      update employe set salaire=salaire*(1+taux)
```

LES FONCTIONS

LES FONCTIONS

- Une fonction est une procédure qui retourne une valeur. La seule différence syntaxique par rapport à une procédure se traduit par la présence du mot clé **RETURN**.
- Une fonction précise le type de donnée qu'elle retourne dans son prototype (signature de la fonction).
- Une fonction est créée avec l'instruction **CREATE OR REPLACE FUNCTION**.

LES FONCTIONS

➤ Syntaxe:

```
CREATE [OR REPLACE] FUNCTION function_name
[(param1 [IN] param_type [, ...])]

RETURN return_datatype IS
    --VAR DECLARATION
BEGIN
    --FUNCTION BODY
    return returned_value -- the same as return_datatype
END
```

- **function_name** spécifie le nom de la fonction.
- L'option **[OR REPLACE]** modifie une fonction existante.
- La liste de paramètres facultatifs contient le nom, le mode et les types de paramètres. **IN** représente la valeur qui sera transmise de l'extérieur. **Pas de parametres en OUT**
- **RETURN** indique le type de donnees que la fonction va retourner

LES FONCTIONS

➤ Exemple: (Trouver le maximum de deux entiers)

```
DECLARE
    a number;
    b number;
    c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
    z number;
BEGIN
    IF x > y THEN
        z := x;
    ELSE
        z := y;
    END IF;
    RETURN z;
END;
BEGIN
    a := 23;
    b := 45;
    c := findMax(a, b);
    dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/
```

LES FONCTIONS

- **Exemple: (La somme de deux entiers)**

```
-- somme.sql
DECLARE
    s number
CREATE or REPLACE FUNCTION somme ( a IN number, b IN number)
return number is
BEGIN
    return a+b;
END;
BEGIN
    select somme(10,20) into s from dual;
    dbms_output.put_line(" La somme est : " || s || " ");
END;
/
```

- **NB:** La table DUAL est une table spéciale d'une seule colonne et d'une seule ligne. Elle est utilisée généralement pour sélectionner les pseudo-colonnes.

LES FONCTIONS

- **Exemple: (fonction retourne le nom d'employé à partir de son numéro)**

```
-- getEmpFromItsNum.sql
```

```
DECLARE
```

```
    s number
```

```
CREATE FUNCTION getEmpName( numEmp Emp.EmpNO%TYPE)
```

```
return Emp.Ename%TYPE Is
```

```
Nom Emp.Ename%Type;
```

```
BEGIN
```

```
    SELECT ENAME INTO nom FROM EMP WHERE EmpNO=numEmp;
```

```
    RETURN nom;
```

```
END;
```

LES FONCTIONS

- **Exemple: (fonction retourne le nom d'employé à partir de son numéro)**

```
-- TEST1_getEmpFromItsNum.sql
```

```
BEGIN
```

```
    dbms_output.put_line("L'employé s'appel: " || getEmpName(7900) );
```

```
END;
```

```
/
```

```
-- TEST2_getEmpFromItsNum.sql
```

```
DECLARE
```

```
empname varchar2(10);
```

```
BEGIN
```

```
    select get_emp_name(7900) into empname from dual;
```

```
    dbms_output.put_line("L'employé s'appel: " || empname );
```

```
END;
```

```
/
```

LES FONCTIONS

Modification d'une fonction:

- Si la base de données évolue, il faut **recompiler** les procédures existantes pour qu'elles tiennent compte de ces modifications. La commande est la suivante: **ALTER FUNCTION nom_fonction COMPILE;**

- **Exemple:**

ALTER FUNCTION moyenne_salaire COMPILE;

- Suppression d'une fonction

Suppression d'une fonction:

DROP FUNCTION nom_fonction;

Exemple: **DROP FUNCTION moyenne_salaire;**

PACKAGES

PACKAGES

- Encapsuler des procédures, des fonctions, des curseurs et des variables comme une unité dans une base de données.
- **Etapas de création:**
 - ***Création des spécifications du package***
 - Spécifier la partie public du package (fonctions, procédures, types, variables, constantes, exceptions et curseurs).
 - ***Création du corps du package***
 - Définir les procédures, les fonctions, les curseurs et les exceptions qui sont déclarées dans les spécifications du package.
 - Définir d'autres objets non déclarés dans les spécifications. Ces objets sont alors privés.

PACKAGES

- *Création des spécifications du package*

CREATE OR REPLACE PACKAGE *nom_package* ***IS***

Spécifications PL/SQL

END *nom_package*;

Les spécifications PL/SQL contiennent les déclarations des :

- Variables
- Enregistrements
- Curseurs
- Exceptions
- Fonctions
- Procédures
- ...

PACKAGES

- Exemple

```
CREATE OR REPLACE PACKAGE gest_emp IS
```

```
-- Variables publiques
```

```
    v_sal EMP.sal%type;
```

```
-- Fonctions et procédures publiques
```

```
FUNCTION augmentation(numEmp IN EMP.empno%type,pourcent IN  
NUMBER) Return NUMBER;
```

```
PROCEDURE    test_Augmentation (numEMP IN EMP.empno%Type,  
pourcent IN NUMBER);
```

```
END gest_emp;
```

PACKAGES

Création du corps du package

-- Définition du corps d'un package (optionnel)

CREATE [OR REPLACE] PACKAGE BODY

nom_package

AS

[-- Définition de types]

[-- Spécification de

curseurs] [-- Déclaration

de variables]

[-- Définition de sous-

programmes] END;

The image features a dark blue background with abstract geometric line art. In the top-left corner, there are several parallel lines forming a stepped, L-shaped pattern. In the bottom-right corner, there are several parallel diagonal lines extending from the bottom edge towards the right edge.

TRIGGERS

TRIGGERS

- Un trigger permet de spécifier les réactions du système d'information lorsque l'on « touche » à ses données. Concrètement il s'agit de définir un traitement (un bloc PL/SQL) à réaliser lorsqu'un événement survient
- Déclencheur :
 - **Action** ou ensemble d'actions déclenchée(s) **automatiquement** lorsqu'une **condition** se trouve satisfaite après l'apparition d'un **événement**.
- Un déclencheur est une règle **ECA**
 - **Événement** = mise à jour, suppression, insertion
 - **Condition** = expression logique vraie ou fausse, optionnelle
 - **Action** = procédure exécutée lorsque la condition est vérifiée.

TRIGGERS

Remarques :

- Avec ORACLE, les **événements** sont prédéfinies et les **conditions** et les **actions** sont spécifiées par le langage procédural PL/SQL.
- Les **événements** sont de six types et ils peuvent porter sur des tables ou des colonnes :
- **BEFORE INSERT**
- **AFTER INSERT**
- **BEFORE UPDATE**
- **AFTER UPDATE**
- **BEFORE DELETE**
- **AFTER DELETE**

Utilité des triggers

Renforcer la cohérence des données d'une façon transparente pour le développeur,

- Propagation des MAJ dans une base de données distribuée.
- Sécurité
- ...

TRIGGERS

Structure d'un trigger

```
CREATE [ OR REPLACE ] TRIGGER      nom_trigger
    { BEFORE | AFTER } // événement
    { INSERT | DELETE | UPDATE [ OF liste de colonnes ] }
    ON table
    [ FOR EACH ROW ]
    [ WHEN ( condition de déclenchement ) ]      // condition
    DECLARE
    ..... BEGIN
    .....      // Actions avec les données
    EXCEPTION
    .....
    END ;
/
```


TRIGGERS

- **Exemple** : Suppression d'un client
 - On supprime toutes ses commandes

```
CREATE TRIGGER suppclint  
BEFORE DELETE ON CLIENT  
FOR EACH ROW  
BEGIN  
    DELETE FROM COMMANDE WHERE Codcli =: OLD.Codcli  
END ;  
/
```

TRIGGERS

Structure d'un trigger

La structure de trigger est composée de trois parties :

- ✓ Un **événement** déclencheur E : action externe sur une table ou sur un tuple qui déclenche le trigger.
- ✓ Une **condition** de déclenchement C : C'est une expression booléenne
- ✓ Une **action** du trigger A : C'est une procédure PL/SQL

Remarques :

1. Lorsqu'un trigger est lancée sur le serveur et qui se termine sans traitement d'exception, l'événement qui l'a déclenché se poursuit correctement.
2. Deux types de triggers peuvent être définis :
 1. Trigger **d'énoncé** : C'est un trigger lancé une seule fois.
 2. Trigger de **tuple** : trigger exécuté autant de fois qu'il y a de tuples à insérer, à modifier ou à supprimer
3. Par défaut un trigger est actif, il peut cependant être désactivé :

ALTER TRIGGER nom_trigger **DISABLE** ;

ALTER TRIGGER nom_trigger **ENABLE** ;

TRIGGERS

Remarque:

Dans les triggers de tuples (**FOR EACH ROW**), on peut manipuler les valeurs traitées, directement **en mémoire** :

✓ : **old.attribut** : ancienne valeur (**DELETE | UPDATE**)

✓ : **new.attribut** : nouvelle valeur (**INSERT | UPDATE**)

Exemple : Trigger pour suppression et sauvegarde dans la table «*Journal*»

```
CREATE OR REPLACE TRIGGER suppression_ligne
```

```
BEFORE DELETE ON      produit
```

```
FOR EACH ROW BEGIN
```

```
    INSERT INTO Journal VALUES (:old.codprod, :old.libelle, :old.prix, :old.qte);
```

```
END ;
```

TRIGGERS

Exemple 1 : lorsqu'une augmentation du prix unitaire (PU) d'un produit est tentée, il faut limiter l'augmentation à 10% du prix en cours.

```
CREATE OR REPLACE TRIGGER BORNER_AUGMENTPU
BEFORE UPDATE OF Prix
ON PRODUIT
FOR EACH ROW
When (New.Prix > Old.Prix * 1.1)
BEGIN
    : New.Prix := :Old.Prix * 1.1 ;
END ;
/
```

UPDATE PRODUIT
SET Prix = 15.99
WHERE Codprod = 10 ;

	Codprod	Prix
Ligne avant (OLD)	10	11	

	Codprod	Prix
Ligne après (NEW)	10	15.99	

	Codprod	Prix
Ligne après (NEW)	10	12.1	

TRIGGERS

Exemple 2 : Utilisation d'un TRIGGER pour le maintien d'une contrainte d'intégrité dynamique. Empêcher une augmentation du PU d'un produit au-delà de 10% du prix en cours.

```
CREATE OR REPLACE TRIGGER BEFORE UPDATE OF Prix  
ON PRODUIT BORNER_AUGMENTPU  
FOR EACH ROW  
When (New.Prix > Old.Prix * 1.1)  
BEGIN  
    RAISE_APPLICATION_ERROR (-20999, ' Violation de la Contrainte')  
END ;  
/
```

TRIGGERS

Exemple 3 : Utilisation d'un TRIGGER pour le maintien d'une contrainte d'intégrité statique.

➤ **$0 < \text{codcli} < 10000$**

CREATE OR REPLACE TRIGGER VERIFIER_NUM_CLIENT

BEFORE INSERT OR UPDATE OF Codcli ON CLIENT

FOR EACH ROW

WHEN (New.Codcli <= 0) OR (New.Codcli >= 10000)

BEGIN

RAISE_APPLICATION_ERROR (-20009, ' Numéro du client incorrect')

END ;

TRIGGERS

Exemple 4 : Lors d'un achat, la quantité à commander d'un produit ne peut pas dépasser la quantité en stock disponible.

```
CREATE OR REPLACE TRIGGER          VERIFIER_STOCK
BEFORE INSERT on Commande        //Table Commande
FOR EACH ROW
DECLARE
S Produit.Qte%type ;
BEGIN
SELECT Qte INTO S FROM Produit WHERE codprod = :New.codprod ;
IF ( :New.Qtecom > S) THEN
RAISE_APPLICATION_ERROR ( -20009, ' Quantité demandée non disponible') ;
END IF ;
END ;
/
```

INDEX

- Stockage des information d'une manières pour facilite la recherche

Syntaxe:

```
CREATE [UNIQUE] INDEX NOM_INDEX ON NOM_TABLE (, [nom_colonne]...) ;
```

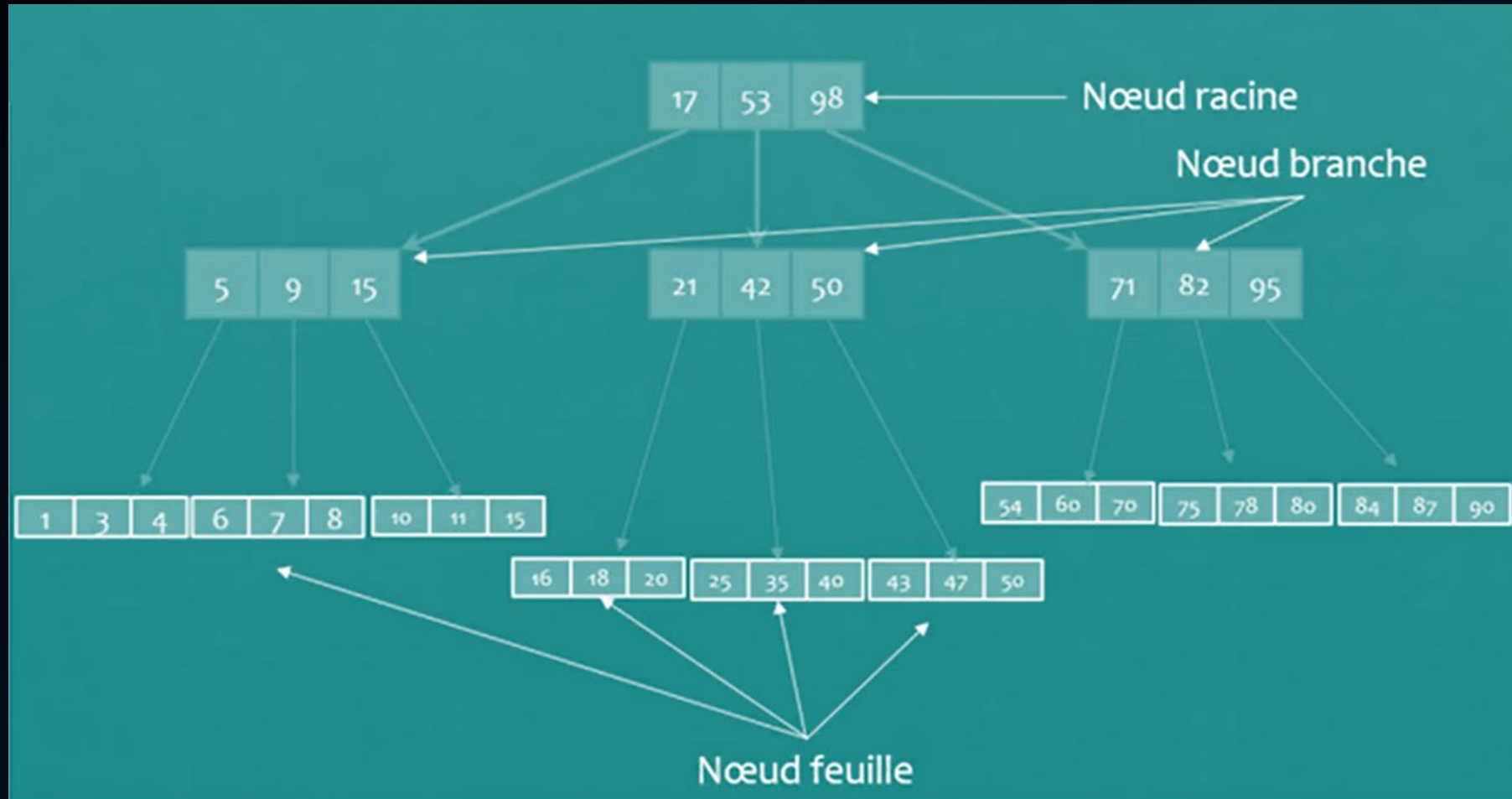
Exemple:

```
CREATE INDEX ACCES_PILOTE ON PILOTE (NOMPIL);
```

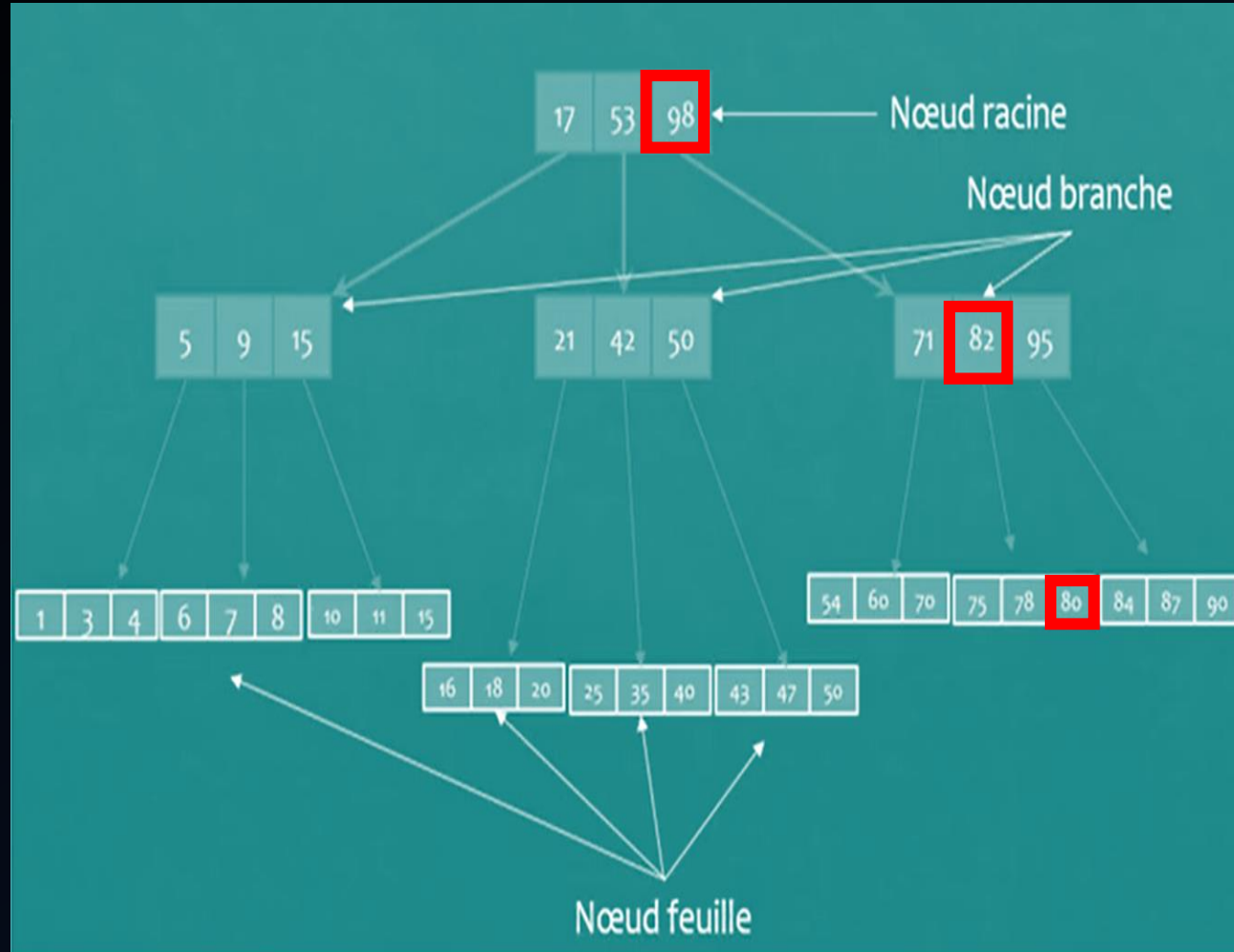

INDEX B-TREE

1. Index B-Tree appelé aussi index Arbre équilibré
2. Il est utilisé pour les requêtes d'égalité: =, <>, >, <,>= et <=
3. Les valeurs des clés de l'index sont enregistrées de façon ordonnée dans une structure arborescente (sous forme d'arbre)Quels sont les types d'index?

INDEX B-TREE



INDEX B-TREE



montre un fragment d'index pour illustrer une recherche sur la clé 80. Le parcours de l'arbre commence au nœud racine du côté gauche. Chaque entrée est traitée dans l'ordre ascendant jusqu'à ce qu'une valeur identique ou plus grande (\geq) que la valeur recherchée (80) soit trouvée. Dans ce graphique, il s'agit de la valeur (98). La base de données suit la référence au nœud branche correspondant et répète la même procédure jusqu'à ce que le parcours atteigne un nœud feuille.

INDEX BITMAP

- Les index bitmap sont traditionnellement considérés comme fonctionnant bien pour les colonnes à faible cardinalité, qui ont un nombre modeste de valeurs distinctes, soit absolues, soit relatives au nombre d'enregistrements contenant les données.
- Le cas extrême de faible cardinalité est celui des données booléennes qui a deux valeurs, Vrai et Faux.

INDEX BITMAP

	Titre	Genre
1	Vertigo	Suspense
2	Brazil	Science-fiction
3	Twin Peaks	Fantastique
4	Underground	Drame
5	Easy Rider	Drame
6	Psychose	Drame
7	Greystoke	Aventures
8	Shining	Fantastique
9	Annie Hall	Comédie
10	Jurassic Park	Science-fiction
11	Metropolis	Science-fiction
12	Manhattan	Comédie
13	Reservoir Dogs	Policier
14	Impitoyable	Western
15	Casablanca	Drame
16	Smoke	Comédie

Un index bitmap considère toutes les valeurs possibles pour un attribut. Pour chaque valeur, on stocke un tableau de bits (dit bitmap) avec autant de bits qu'il y a de lignes dans la table.

Très utile pour les colonnes qui ne possèdent que quelques valeurs distinctes.

INDEX BITMAP

Rechercher les films de type "drame"

1- Sélectionner dans la table les numéros de ligne (RowId) ayant Drame="1"

RowId	Drame	Science-fiction	Comédie
1	0	0	0
2	0	1	0
3	0	0	0
4	1	0	0
5	1	0	0
6	1	0	0
7	0	0	0
8	0	0	0
9	0	0	1
10	0	1	0
11	0	1	0
12	0	0	1
13	0	0	0
14	0	0	0
15	1	0	0
16	0	0	1

	Titre	Genre
1	Vertigo	Suspense
2	Brazil	Science-fiction
3	Twin Peaks	Fantastique
4	Underground	Drame
5	Easy Rider	Drame
6	Psychose	Drame
7	Greystoke	Aventures
8	Shining	Fantastique
9	Annie Hall	Comédie
10	Jurassic Park	Science-fiction
11	Metropolis	Science-fiction
12	Manhattan	Comédie
13	Reservoir Dogs	Policier
14	Impitoyable	Western
15	Casablanca	Drame
16	Smoke	Comédie