PL/SQL

Université Sultan Moulay Slimane
Faculté Poly disciplinaire
Khouribga
Année universitaire 2021/2022

Pourquoi PL/SQL ?

- SQL est un langage non procédural
- Les traitements complexes sont parfois difficiles à écrire si on ne peut utiliser des variables et les structures de programmation comme les boucles et les alternatives
- On ressent vite le besoin d'un langage procédural pour lier plusieurs requêtes SQL avec des variables et dans les structures de programmation habituelles

Principales caractéristiques de PL/SQL

- Extension de SQL : des requêtes SQL cohabitent avec les structures de contrôle habituelles de la programmation structurée (blocs, alternatives, boucles)
- La syntaxe ressemble au langage Ada
- Un programme est constitué de procédures et de fonctions
- Des variables permettent l'échange d'information entre les requêtes SQL et le reste du programme

Utilisation de PL/SQL

- PL/SQL peut être utilisé pour l'écriture des triggers (Oracle accepte aussi le langage Java), procédures stockées, …
- Il convient aussi pour écrire des fonctions utilisateurs qui peuvent être utilisées dans les requêtes SQL (en plus des fonctions prédéfinies)
- Il est aussi utilisé dans des outils Oracle, Forms et Report en particulier

Normalisation du langage

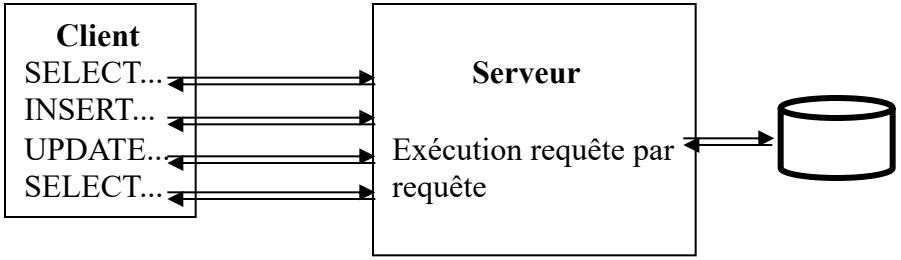
- PL/SQL est un langage propriétaire de Oracle
- PostgreSQL utilise un langage très proche pl/pgsql
- Ressemble au langage normalisé PSM (*Persistant Stored Modules)*
- *Pl/sql est un langage 4g*
- Tous les langages L4G des différents SGBDs se ressemblent

Définition d'un bloc

- Un programme est structuré en blocs d'instructions de 3 types :
  1- procédures anonymes
  2- procédures nommées
  3- fonctions nommées
- Un bloc peut contenir d'autres blocs

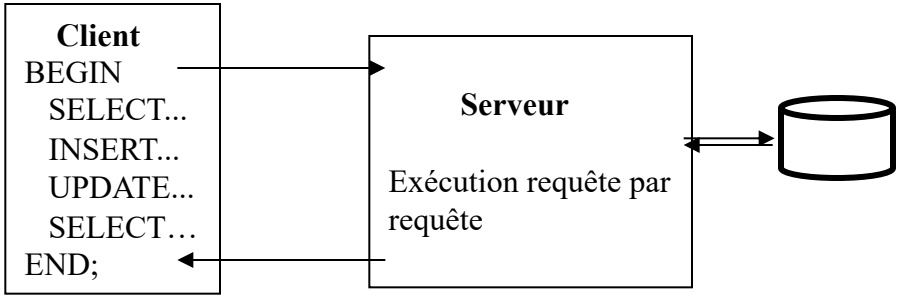
❑ Mode SQL

- Langage de manipulation de donnée(LDD, LMD, LCD)



❑ Mode PL/SQL

- Extension de SQL
- Langage L4G
  - Modulaire (paquetages, sous-programmes)
  - Gestion des erreurs (exceptions)
  - TAD (type abstrait de données)



Editeurs de code

- **Oracle SQL Developer**
  - Oracle SQL Developer is a free, integrated development environment that simplifies the development and management of Oracle Database in both traditional and Cloud deployments. SQL Developer offers complete end-to-end development of your PL/SQL applications, a worksheet for running queries and scripts, a DBA console for managing the database, a reports interface, a complete data modeling solution, and a migration platform for moving your 3rd party databases to Oracle.
- **The basic command line SQL*PLUS**
  - Write the script and save it in a sql extention (ex. myScript.sql)
  - Run SQL*PLUS command line and log in
  - Run the following command : start path to myScript +myScript.sql

  OR
  - Run : @path/myScript.sql

Création d'un utilisateur

CREATE USER utilisateur IDENTIFIED {BY mot de passe | EXTERNALLY }
[DEFAULT TABLESPACE tablespace]
[TEMPORARY TABLESPACE tablespace]
[QUOTA { entier [K,M] | UNLIMITED } ON tablespace…]
[PROFILE profile][PASSWORD EXPIRE]

Tablespace par défaut : SYSTEM

Quota en Ko ou en Mo dans ce tablespace pour cet utilisateur

Sans profil, l'utilisateur reçoit le profil par défaut, qui est, sauf chgt, sans limitation de ressource

PASSWORD EXPIRE oblige l'utilisateur à changer son mot de passe avant sa prochaine connexion

Exemple: create user

```
CREATE USER user_irisi
        IDENTIFIED BY azerty12
        DEFAULT TABLESPACE data
        TEMPORARY TABLESPACE temp
        QUOTA UNLIMITED ON data
        QUOTA UNLIMITED ON indx
        PASSWORD EXPIRE;
```

Créer un tablespace

```
create  TABLESPACE "tablespace2"
DATAFILE 'C:\app\admin\oradata\commande\tablespace2.DBF' SIZE 2G
AUTOEXTEND ON NEXT 100M MAXSIZE 5000M
LOGGING ONLINE
PERMANENT BLOCKSIZE 8192
EXTENT MANAGEMENT LOCAL UNIFORM SIZE 10M
SEGMENT SPACE MANAGEMENT AUTO;
```

Modifier un utilisateur

```
ALTER USER utilisateur
        IDENTIFIED {BY mot de passe | EXTERNALLY }
        [DEFAULT TABLESPACE tablespace]
        [TEMPORARY TABLESPACE tablespace]
        [QUOTA { entier [K,M] | UNLIMITED } ON tablespace…]
        [PROFILE profile]
        [DEFAULT ROLE { role,…| ALL [EXCEPT role,…]| NONE}]
        [PASSWORD EXPIRE];
```

Supprimer un utilisateur

DROP USER utilisateur [CASCADE];

CASCADE supprime les objets créés par l'utilisateur

Utilisateur prédéfinis

Il y a deux super-utilisateurs (administrateurs) prédéfinis : sys et system

Le rôle prédéfini sysdba (ou dba dans d'anciennes versions) peut être octroyé par les utilisateurs sys et system

Vues du dictionnaire

Table DBA_USERS qui comporte notamment les colonnes :USERNAME, PROFILE, LOCK_DATE, EXPIRY_DATE, DEFAULT_TABLESPACE, TEMPORARY_TABLESPACE

Table DBA_TS_QUOTAS contient les quotas alloués aux utilisateurs

Profil Utilisateur

Un profil est un ensemble nommé de limitations de ressources permet de contrôler les ressources utilisés par un utilisateur

CREATE PROFILE profile LIMIT

SESSIONS_PER_USER {entier | DEFAULT | UNLIMITED}

connexions en parallèle par utilisateur

CPU_PER_SESSION {entier | DEFAULT | UNLIMITED} *en centièmes de secondes*

CONNECT_TIME {entier | DEFAULT | UNLIMITED} *en secondes*

IDLE_TIME {entier | DEFAULT | UNLIMITED}

temps max d'inactivité en continu pour une session en secondes

LOGICAL_READS_PER_SESSION {entier | DEFAULT | UNLIMITED}

nb max de blocs de données lus par session

LOGICAL_READS_PER_CALL {entier | DEFAULT | UNLIMITED}

nb max de blocs de données lus par appel au noyau

COMPOSIT_LIMIT {entier | DEFAULT | UNLIMITED}

PRIVATE_SGA {entier [K,M] | DEFAULT | UNLIMITED};

taille allouée à l'espace privé d'une session dans la SGA

Gestion des comptes utilisateur

A partir de la version 8 d'Oracle

Limitation des paramètres d'utilisation des comptes

Options visibles dans la table DBA_PROFILE

CREATE PROFILE profile LIMIT

FAILED_LOGIN_ATTEMPTS {entier | DEFAULT | UNLIMITED}

PASSWORD_LOCK_TIME {entier | DEFAULT | UNLIMITED}

nb de jours de blocage d'un compte après un nb d'échecs consécutifs

PASSWORD_GRACE_TIME {entier | DEFAULT | UNLIMITED}

nb de jours donnés pour changer un mot de passe (après avertissement)

PASSWORD_LIFE_TIME {entier | DEFAULT | UNLIMITED}

PASSWORD_REUSE_TIME {entier | DEFAULT | UNLIMITED};

nb de jours avant que le mot de passe courant puisse être réutilisé

Gestion des privilèges

Privilèges système : concernent la connexion à la base Oracle et permettent à un utilisateur un certain nombre d'actions sur la définition d'objets de la base

Privilèges objets : permettent des accès aux objets désignés, accordés par le propriétaire de ces objets

Privilèges système

GRANT {privilège système | rôle }
[{privilège système | rôle },…]
TO
{utilisateur | rôle | public} [,{utilisateur | rôle} ,…]
[WITH ADMIN OPTION];

Avec WITH ADMIN OPTION, les privilèges reçus peuvent être transmis à d'autres utilisateurs ou rôles

REVOKE privilège FROM {utilisateur|rôle|public};

Privilèges système

Privilèges système	CREATE	ALTER	DROP	Divers
CLUSTER	Oui			
ANY CLUSTER	oui	oui	Oui	
DATABASE	Oui			
DATABASE LINK	Oui			
PUBLIC DATABASE LINK	oui		Oui	
ANY INDEX	oui	oui	Oui	
LIBRARY	oui		Oui	
ANY LIBRARY	Oui		oui	
PROCEDURE	oui			Execute
ANY PROCEDURE	oui	oui	oui	Execute

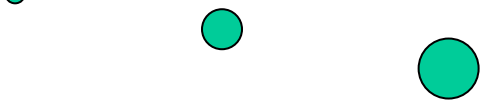
Privilèges système

Privilèges système	CREATE	ALTER	DROP	Divers
PROFILE	oui	oui	Oui	
RESOURCE COST		Oui		
ROLE	Oui			
ANY ROLE		oui	oui	Grant
ROLLBACK SEGMENT	Oui	oui	oui	
SESSION	oui	Oui		
SEQUENCE	Oui			
ANY SEQUENCE	oui	oui	Oui	select
SNAPSHOT				
ANY SNAPSHOT				
SYNONYM				
ANY SYNONYM				

Structure d'un bloc PL/SQL

- **Syntaxe d'un bloc anonyme**

```
[ DECLARE
        -- Déclaration types, constantes, variables ]
  BEGIN
        -- Instructions PL/SQL (et ou SQL-LMD)
[ EXCEPTION
        -- Traitement ou récupération des erreurs ]
  END;
  /
```



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

NB: seuls BEGIN et END sont Obligatoires.

Structure des blocs (3)

Les blocs PL/SQL peuvent être imbriqués:

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

Structure des blocs (3)

- **Les blocs comme les instructions se terminent par un ‘;’.**
- **Seuls ‘Begin’ et ‘End’ sont obligatoire.**

Premier programme PL/Sql

```
SQL>    SET SERVEROUTPUT ON
SQL>    BEGIN
2          dbms_output.putline('Bonjour');
3          END;
4          /
Bonjour
```

- Un programme contient au moins BEGIN <une instruction> END;
- dbms_output.putline permet d'écrire sur la console SQLPlus.
- La sortie écran ne marche que si le serveur d'impression est ouvert. Faire SET SERVEROUTPUT ON sous SQLPlus pour basculer sur le mode sortie console.
- La ligne 4 commence par un /. C'est la demande d'exécution du programme tapée (donc la fin de saisie)

Structure d'un bloc PL/SQL

❑ Exemple

ETUDIANTS

numetudiant	nometudiant	nbformation
1	Jojo	1
2	Mimi	2
3	Polo	0

FORMATIONS

numformation	nomformation
DW	Data Warehouse
OR	BD Objet-Relationnelles

INSCRIPTIONS

numformation	numetudiant	dateinscription
DW	2	22/09/2003
DW	1	19/09/2003
OR	2	22/09/2003

❑ « Insertion de Polo »

```
ACCEPT ne PROMPT 'Numero de l''etudiant a inscrire:'
ACCEPT nf PROMPT 'Numero de l''enseignement:'

BEGIN
  INSERT INTO INSCRIPTIONS
  VALUES (&nf, '&ne', SYSDATE);
END;
/
```

```
...
Numero de l'etudiant a inscrire: 3
Numero de l'enseignement: DW
...
```

Types de données

- **Types simples habituels correspondants aux types SQL2 :**
  - CHAR,CHAR(*n*),STRING,VARCHAR2(*n*),…
  - INTEGER,NATURAL,NUMBER,NUMBER(*n*),FLOAT,REAL,NUMBER(*n*, *m*),…
  - BOOLEAN
  - DATE
  - …
- **Types composites adaptés à la récupération des colonnes et lignes des tables SQL :**
  - RECORD
  - TABLE
  - VARRAY
- **Types références**
  - REF <type objet>
- **Types LOB (« Large Objects »)**
  - BFILE
  - BLOB, CLOB, NCLOB

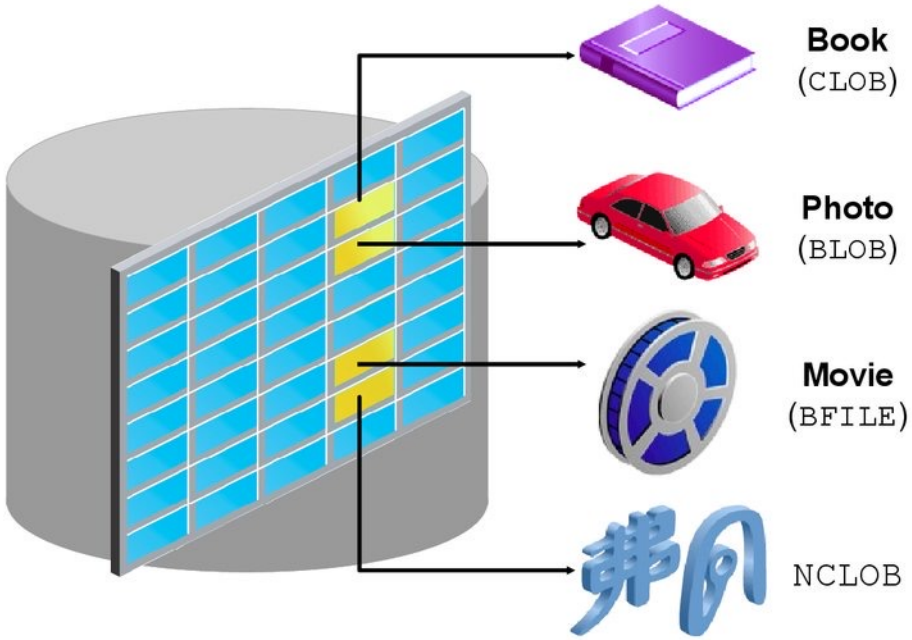
Data Type	Description
CHAR [(*maximum_length*)]	Base type for fixed-length character data up to 32,767 bytes. If you do not specify a maximum length, the default length is set to 1.
VARCHAR2 (*maximum_length*)	Base type for variable-length character data up to 32,767 bytes. There is no default size for VARCHAR2 variables and constants.
NUMBER [(*precision*, *scale*)]	Number having precision *p* and scale *s*. The precision *p* can range from 1 through 38. The scale *s* can range from –84 through 127.
BINARY_INTEGER	Base type for integers between –2,147,483,647 and 2,147,483,647

Data Type	Description
PLS_INTEGER	Base type for signed integers between –2,147,483,647 and 2,147,483,647. PLS_INTEGER values require less storage and are faster than NUMBER values. In Oracle Database 10*g*, the PLS_INTEGER and BINARY_INTEGER data types are identical. The arithmetic operations on PLS_INTEGER and BINARY_INTEGER values are faster than on NUMBER values.
BOOLEAN	Base type that stores one of the three possible values used for logical calculations: TRUE, FALSE, and NULL
BINARY_FLOAT	Represents floating-point number in IEEE 754 format. It requires 5 bytes to store the value.
BINARY_DOUBLE	Represents floating-point number in IEEE 754 format. It requires 9 bytes to store the value.

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE
- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND

LOB data types variables

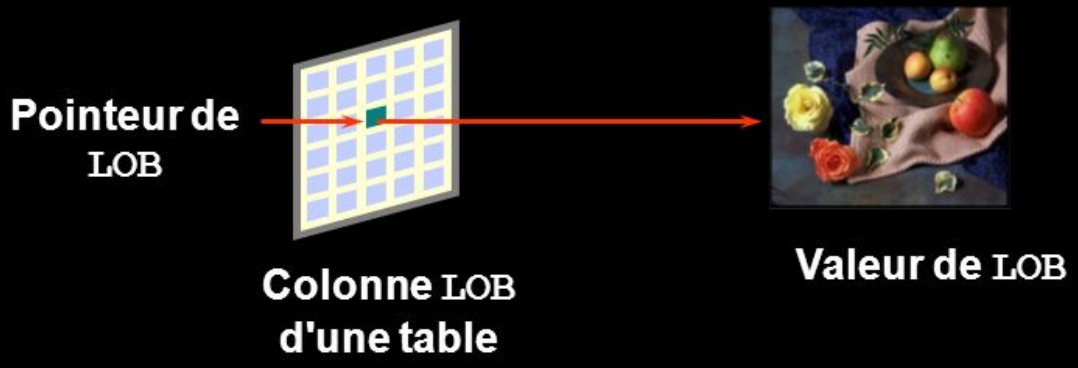
LOB Data Type Variables



ORACLE

Composition d'un type LOB

La colonne LOB conserve un pointeur vers la valeur de LOB



ORACLE

Les variables

- **Identificateurs Oracle :**
  - 30 caractères au plus
  - commence par une lettre
  - peut contenir lettres, chiffres, _, $ et #
- **Pas sensible à la casse.**
- **Portée habituelle des langages à blocs.**
- **Doivent être déclarées avant d'être utilisées.**

Les variables (2)

- **Déclaration :**

Nom_variable type_variable;

- **Initialisation:**

Nom_variable := valeur;

- **Déclaration et initialisation :**

Nom_variable type_variable := valeur;

Les variables

- **var [CONSTANT] datatype [NOT NULL] [:= | DEFAULT expr];**

- **Adopter des conventions pour nommer des objets.**
- **Initialiser les constantes et les variables déclarées NOT NULL.**
- **Initialiser les identifiants en utilisant l'opérateur d'affectation (:=) ou le mot réservé DEFAULT.**
- **Déclarer au plus un identifiant par ligne.**
- **Le type peut être primitif ou objet.**

Commentaires

- **-- Pour une fin de ligne**
- **/* Pour plusieurs lignes */**

Programming Guidelines

Make code maintenance easier by:

- Documenting code with comments
- Developing a case convention for the code
- Developing naming conventions for identifiers and other objects
- Enhancing readability by indenting

Category	Case Convention	Examples
SQL statements	Uppercase	`SELECT, INSERT`
PL/SQL keywords	Uppercase	`DECLARE, BEGIN, IF`
Data types	Uppercase	`VARCHAR2, BOOLEAN`
Identifiers and parameters	Lowercase	`v_sal, emp_cursor, g_sal, p_empno`
Database tables and columns	Lowercase	`employees, employee_id, department_id`

Les types de variables

- **VARCHAR2**
  - Longueur maximale : 32767 octets
  - Syntaxe:

    Nom_variable VARCHAR2(30);

Exemple:
```
name            VARCHAR2(30);
name            VARCHAR2(30) := 'toto';
```

- **NUMBER**

  Nom_variable NUMBER(long,dec);

  avec Long : longueur maximale

  Dec : longueur de la partie décimale

Exemple:
```
num_tel         number(10);
toto            number(5,2)=142.12;
```

Les types de données scalaires

- **Hold a singe value**
- **Have no internal components**

- **We can arrange it in 4 categories:**
- **Character**
- **Data**
- **Number**
- **boolean**

Les types de variables (2)

- **DATE**

  Nom_variable DATE;

  - Par défaut DD-MON-YY (18-DEC-02)
  - Fonction TO_DATE

  Exemple :

```
start_date := to_date('29-SEP-2003','DD-MON-YYYY');
start_date := to_date('29-SEP-2003:13:01','DD-MON-
                      YYYY:HH24:MI');
```

- **BOOLEAN**
  - TRUE, FALSE ou NULL

Les types de variables (3)

- **identificateur [CONSTANT] type [:= valeur];**

- **Exemples :**
  - **age integer;**
  - **nom varchar(30);**
  - **dateNaissance date;**
  - **ok boolean := true;**

- **Déclarations multiples interdites :**
  - **i, j integer;**

Exemples

```
c           CHAR( 1 );
name        VARCHAR2(10) := 'Scott';
cpt         BINARY_INTEGER := 0;
total       NUMBER( 9, 2 ) := 0;
order       DATE := SYSDATE + 7; -- + 7 jours
Ship        DATE;
pi          CONSTANT NUMBER ( 3, 2 ) := 3.14;
done        BOOLEAN NOT NULL := TRUE;
ID          NUMBER(3) NOT NULL := 201;
PRODUIT     NUMBER(4) := 2*100;
V_date      DATE := TO_DATE('17-OCT-01','DD-MON-YY');
V1          NUMBER := 10;
V2          NUMBER := 20;
V3          BOOLEAN := (v1>v2);
Ok          BOOLEAN := (z IS NOT NULL);
```

Les types Prédéfinis

Les types **BINARY_INTEGER** et **PLS_INTEGER** conviennent aux entiers signés (domaine de valeurs de -2^{31} à 2^{31}, soit -2.147.483.647 à +2.147.483.647). Ces types requièrent moins d'espace de stockage que le type **NUMBER**.

Les types **PLS_INTEGER** et **BINARY_INTEGER** ne se comportent pas de la même manière lors d'erreurs de dépassement (*overflow).* **PLS_INTEGER** *déclenchera l'exception ORA*-01426 : dépassement numérique. **BINARY_INTEGER** ne provoque aucune exception si le résultat est affecté à une variable NUMBER.

PLS_INTEGER est plus performant au niveau des opérations arithmétiques que les types NUMBER et BINARY_INTEGER qui utilisent des librairies mathématiques.

Sous-types

Chaque type de données PL/SQL prédéfini a ses caractéristiques (domaine de valeurs, fonctions applicables…). Les sous-types de données permettent de restreindre certaines de ces caractéristiques à des données. Un sous-type n'introduit pas un nouveau type mais en restreint un existant. Les sous-types servent principalement à rendre compatibles des applications à la norme SQL ANSI/ISO ou plus pertinentes certaines déclarations de variables.

PL/SQL propose plusieurs sous-types prédéfinis et il est possible de définir des sous-types personnalisés.

Sous-type	Type restreint	Caractéristiques
CHARACTER	CHAR	Mêmes caractéristiques.
INTEGER	NUMBER(38,0)	Entiers sans décimales.
PLS_INTEGER	NUMBER	Déjà étudié.
BINARY_INTEGER	NUMBER	Déjà étudié.
NATURAL, POSITIVE	BINARY_INTEGER	Non négatif.
NATURALN, POSITIVEN		Non négatif et non nul.
SIGNTYPE		Domaine de valeurs {–1, 0, 1}.
DEC, DECIMAL, NUMERIC	NUMBER	Décimaux, précision de 38 chiffres.
DOUBLE PRECISION, FLOAT, REAL		Flottants.
INTEGER, INT, SMALLINT		Entiers sur 38 chiffres.

Caractères

Les types CHAR et NCHAR permettent de stocker des chaînes de caractères de taille fixe.

Les types VARCHAR2 et NVARCHAR2 permettent de stocker des chaînes de caractères de taille variable (VARCHAR est maintenant remplacé par VARCHAR2).

Les types NCHAR et NVARCHAR2 permettent de stocker des chaînes de caractères Unicode (*multibyte), méthode de codage universelle qui fournit une valeur de code unique pour chaque* caractère quels que soient la plate-forme, le programme ou la langue. Unicode est utilisé par XML, Java, JavaScript, LDAP, et WML. Ces types Oracle sont proposés dans le cadre NLS (*National Language Support).*

Les types CLOB et NCLOB permettent de stocker des flots de caractères (exemple : du texte).

sysdate

- The Oracle SYSDATE function returns the current date and time of the Operating System (OS) where the Oracle Database installed.

- The SYSDATE function returns the current date and time value whose type is DATE.
- The format of the returned date time value depends on the value of the NLS_DATE_FORMAT parameter.
- Examples
  - ✓ The following example returns the current date and time of the OS where the Oracle Database resides:
  - ✓ SELECT TO_CHAR(SYSDATE, 'MM-DD-YYYY HH24:MI:SS') FROM dual;
  - ✓ In this example, we used the TO_CHAR() function to format the current system date and time value returned by the SYSDATE function.

- The following table illustrates the arithmetic of the SYSDATE function:

SYSDATE Math	Description
WHERE (date) > SYSDATE - 8/24;	Past 8 hours
WHERE (date) > SYSDATE - 30;	Past 30 days
WHERE (date) > SYSDATE - 30/1440;	Past 30 minutes
8/24	8 hours
15/24/60/60	15 seconds
1/24/60	One minute
1/24	One hour
TRUNC(SYSDATE+1/24,'HH')	1 hour starting with the next hour

Remarks: The Oracle SYSDATE function cannot be used in the condition of a CHECK constraint. In this tutorial, you have learned how to use the Oracle SYSDATE function to get the current system date and time.

Quelques conventions en PL/SQL

- **Deux variables peuvent partager le même nom si elles sont dans des portées distinctes.**
- **Les noms des variables doivent être différents des noms des colonnes des tables utilisées dans un bloc:**
  - **v_empno (variab**le)
  - g_dept**no (glo**b**a**le)
  - c_emp (CURSOR)
- **L'identifiant est limité à 30 caractères, le premier caractère devant être une lettre.**

Utilisation des variables

- **On utilise des variables pour :**
  - Le stockage temporaire de données
  - La manipulation de valeur stockées
  - La possibilité de les réutiliser
  - Simplifier la maintenance du code
  - Variable typée dynamiquement au moyen d'attributs spéciaux
    - **`%ROWTYPE`** ou **`%TYPE`**
- **Les variables Abstract Data Type et les collections seront abordées ultérieurement**
  - Oracle9i est également un SGBD orienté objet

Bloc PL/SQL Anonyme accédant à la base

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
          v_name VARCHAR2(10);
      BEGIN
          SELECT ename INTO v_name
          FROM emp; -- WHERE empno=7839;
          DBMS_OUTPUT.PUT_LINE(v_name);
      EXCEPTION
          WHEN OTHERS
              THEN NULL;
      END;
/

PL/SQL Procedure successfully completed.
```

Imprimer ‘

Begin

Dbms_output:put_line(‘Father’’s day’);

End;

--when your string contains an apostrophe it is recommanded to use the q’ notation

Select ‘today is the father’’s day from dual;

Select q‘(today is the father’s day)’ from dual;

Select q‘[today is the father’s day]’ from dual;

Select q‘x today is the father’s day x’ from dual;

Cas à plusieurs variables hôtes

```
SET SERVEROUTPUT ON
SQL> DECLARE
            v_ename         VARCHAR2(12);
            v_sal  NUMBER(7,2);
      BEGIN
            SELECT    ename,sal   INTO
                v_ename,v_sal
              FROM emp WHERE ROWNUM = 1;

            DBMS_OUTPUT.PUT_LINE(v_ename);
            DBMS_OUTPUT.PUT_LINE(v_sal);
      EXCEPTION
            WHEN OTHERS
                  THEN NULL;
      END;
/
```

Optionnel à ce stade

Variables et Constantes

- **Déclaration dans la partie déclarative d'un bloc PL/SQL**

- **Variables**

```
DECLARE
   date_naissance DATE;
   compteur INTEGER:=0;          -- initialisation du compteur à zéro
   compteur INTEGER DEFAULT 0; -- initialisation du compteur à zéro
   ok BOOLEAN DEFAULT TRUE;
   id CHAR(5) NOT NULL :='ali';
```

- **Constantes**

```
DECLARE
   pi CONSTANT FLOAT:=3.14;
```

- Déclarations multiples interdites : ~~i, j integer;~~
- null est valeur par défaut, si pas d'initialisation (sauf si NOT NULL spécifié)
- Mais les variables les plus intéressantes en PLSQL sont celles qui vont contenir des données de la base. Variable de même type qu'un attribut (colonne) ou de même type qu'un schéma (ligne).

Les types de variables (4)

- **Types composites adaptés à la récupération des colonnes et lignes des tables SQL :**

  %TYPE, %ROWTYPE.

Typage dynamique `%TYPE`

- **Déclarer une variable à partir :**
  - D'une autre variable déjà déclarée
  - D'une définition d'un attribut de la base de données
- **Préfixer `%TYPE` avec :**
  - La table et la colonne de la base de données
  - Le nom de la variable déclarée précédemment
- **PL/SQL évalue le type de donnée et la taille de la variable.**
- **Inspiré du langage ADA**

Déclaration %TYPE

- **On peut déclarer qu'une variable est du même type qu'une colonne d'une table ou d'une vue (ou qu'une autre variable) :**

```
nom emp.name%TYPE;
```

L'Attribut %TYPE - Exemple

```
DECLARE
      ename              scott.emp.ename%TYPE;
      job                emp.job%TYPE;
      balance            NUMBER( 7, 2 );
      mini_balance       balance%TYPE := 10;
      rec                emp%ROWTYPE
```

- Le type de données de la colonne peut être inconnu.
- Le type de données de la colonne peut changer en exécution.
- Facilite la maintenance.

Variable typée dynamiquement

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
            v_ename                emp.ename%TYPE;
      BEGIN
            SELECT ename
            INTO v_ename
              FROM emp
           WHERE ROWNUM = 1;
            DBMS_OUTPUT.PUT_LINE(v_ename);
      EXCEPTION
            WHEN OTHERS
                  THEN NULL;
      END;
/
```

Déclaration %ROWTYPE

- **Une variable peut contenir toutes les colonnes d'une ligne d'une table:**

```
employe emp%ROWTYPE;
```

- **déclare que la variable employe contiendra une ligne de la table emp.**

Exemple d'utilisation

```
employe emp%ROWTYPE;
nom emp.name%TYPE;
select * INTO employe
from emp
where matr = 900;
nom := employe.name;
employe.dept := 20;
---------
insert into emp
values employe;
```

Attribut %TYPE & %ROWTYPE

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
      rec     emp%ROWTYPE;
      address VARCHAR2(64);
      income  emp.sal%TYPE; -- rec.sal%TYPE;
     BEGIN
      SELECT *
        INTO rec FROM emp
       WHERE ROWNUM = 1;
              income := rec.sal*12;
             address := rec.ename || CHR(10) ||
                           income  || CHR(10) ||
                TO_CHAR(rec.hiredate,'DD/MM/YYYY');
      DBMS_OUTPUT.PUT_LINE(address);
   END;
/
SMITH
9600
17/12/1980
```

Manipulation dans une variable PL

Bind Variables

- **Bind variables are:**
  - ✓ Created in the environment
  - ✓ Also called host variables
  - ✓ Created with the VARIABLES keyword
  - ✓ Used in SQL statements and PL/SQL blocks
  - ✓ Accessed even after the PL/SQL blocks is executed
  - ✓ Referenced with a preceding colon
- **Variable b_result NUMBER**
- **BEGIN**
  - ✓ SELECT (SALARY*12) + NVL(COMMISSION_PCT,0) INTO :b_result FROM employees WHERE employees_id =144;
  - ✓ END;
  - ✓ /PRINT b_result // donee le résultat du block précédent

Affectation

- **Plusieurs façons de donner une valeur à une variable:**
  - :=
  - par la directive INTO de la requête SELECT
- **Exemples :**
  - **dateNaissance := ’10/10/2004’;**
  - **select name INTO nom from emp where matr = 509;**

Les opérateurs

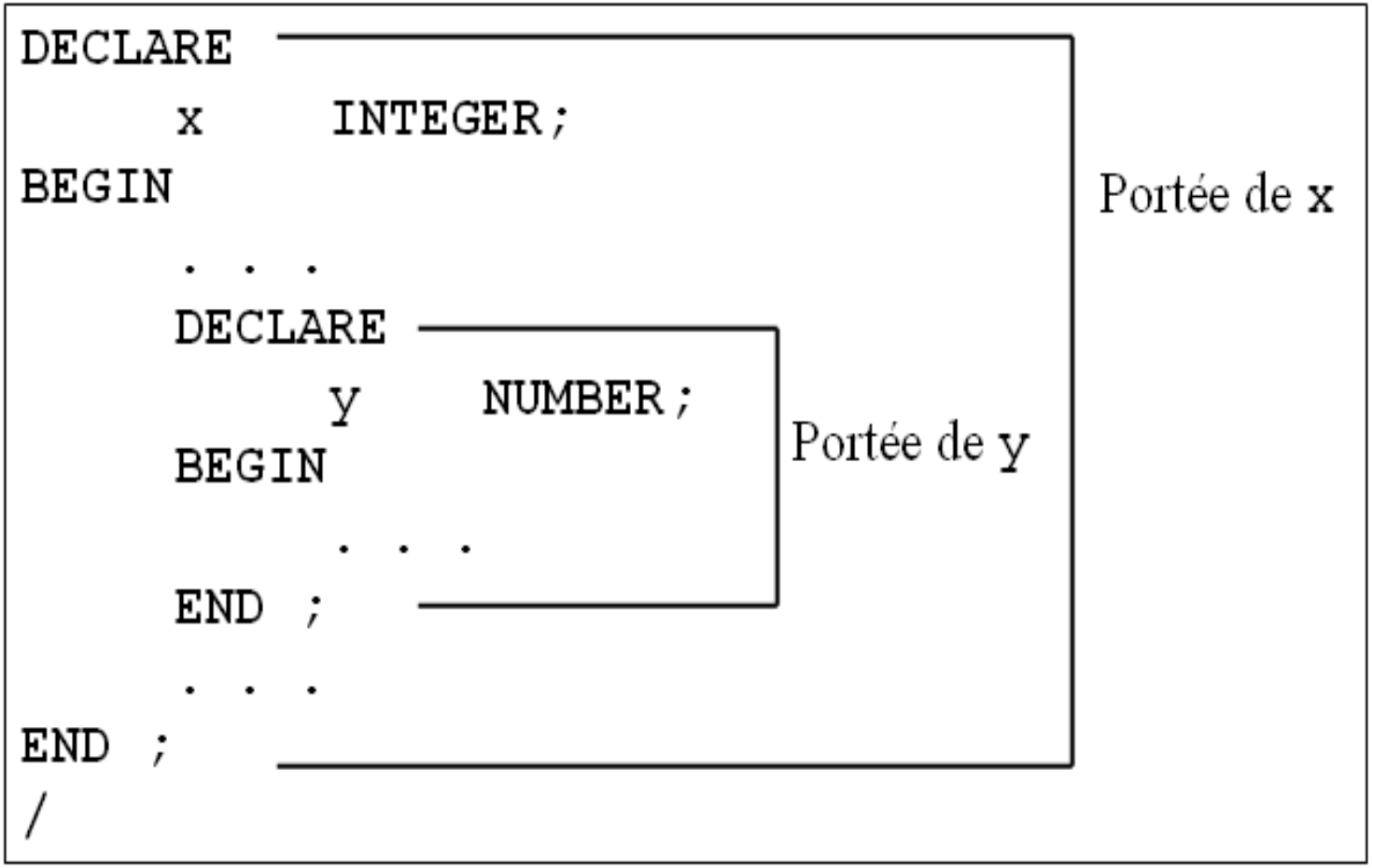
PL/SQL supporte les opérateurs suivants :

- **Arithmétique : +, - *, /,**
- **MOD(m,n) modulo**
- **Concaténation : ||**
- **Parenthèses (contrôle des priorités entre opérations): ()**
- **Comparaison : =, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN**
- **Logique : AND, OR, NOT**
- **Affectation: :=**

Portée

- **Les instructions peuvent être imbriquées là où les instructions exécutables sont autorisées.**
- **La section `EXCEPTION` peut contenir des blocs imbriqués.**
- **Les boucles possède chacune une portée**
  - ✓ les incréments y sont définis
- **Un identifiant est visible dans les régions où on peut référencer cet identifiant :**
  - ✓ Un bloc voit les objets du bloc de niveau supérieur.
  - ✓ Un bloc ne voit pas les objets des blocs de niveau inférieur.

Portée


DECLARE
x INTEGER;
BEGIN
. . .
DECLARE
y NUMBER;
BEGIN
. . .
END ;
. . .
END ;
/
Portée de x
Portée de y

Règles générales

- **Commenter le code**
- **Adopter une convention de casse**
- **Développer une convention d'appel pour les identifiants et autres objets**
- **Indenter le code**

Instruction SQL en majuscules

Mots-clés en majuscules

Type de données en majuscules

Identifiant et paramètres en minuscules

Tables & colonnes en minuscules

Conventions d'appel possible

Identifiant	Nom	Exemple
Variable	v_name	v_sal
Constante	c_name	c_pi
Cursor	name_cursor	Emp_cursor
Exception	e_name	E_too_many
Table type	name_table_type	sum_table_type
Table	name_table	emp_tot_table
Record Type	name_record_type	emp_record_type
Record	name_record	emp_record
Substitution	p_name	p_sal
Globale	g_name	g_deptno

REcap

❑ Référence au dictionnaire des données : %TYPE et %ROWTYPE

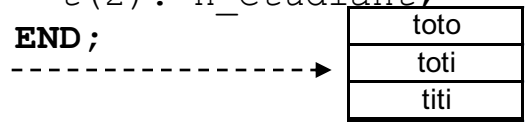
```
DECLARE
   date_ins INSCRIPTIONS.dateinscription%TYPE;/*type de la colonne dateinscriptio
                                                dans la table INSCRIPTIONS*/
   ligne INSCRIPTIONS%ROWTYPE; /* type composite d 'un enregistrement
                                  de la table INSCRIPTIONS */
```

❑ Tableau

```
DECLARE
   -- déclaration du type tableau
   TYPE type_vecteur
      IS TABLE OF VARCHAR2(10)
      INDEX BY BINARY_INTEGER;

   -- déclaration d'une variable tableau
   t type_vecteur:=('toto', 'titi', 'toti');
```


```
 n_etudiant VARCHAR2(10);
BEGIN
  n_etudiant:=t(1);
  t(1):=t(2);
  t(2):=n_etudiant;
END;
```



❑ Type structuré

```
DECLARE
   -- déclaration du type
   TYPE type_structure IS RECORD (numero INTEGER NOT NULL,
                                  nom ETUDIANTS.nometudiant%TYPE);
   -- déclaration d'une variable
   etudiant type_structure;
```

Exemple d'utilisation

```
employe     emp%ROWTYPE;
nom         emp.nom%TYPE;
Select  *   INTO   employe
                        from       emp
                                   where      matr = 900;
 nom    :=   employe.nom;
employe.dept  :=  20;
…
insert  into   emp   values   employe;
```

Le paquetage DBMS_OUTPUT

Les procédures de ce paquetage vous permettent d'écrire des lignes dans un tampon depuis un bloc PL/SQL anonyme, une procédure ou un déclencheur.

Le contenu de ce tampon est affiché à l'écran lorsque le sous-programme ou le déclencheur est terminé.

L'utilité principale est d'afficher à l'écran des informations de trace ou de débogage

La taille maximum du tampon est de 1 million de caractères

La taille maximum d'une ligne est de 255 caractères

La capacité maximum d'une ligne étant de 255 caractères, la procédure DEBUG permet de s'affranchir de cette limitation

Fonctions et procédures du paquetage

DBMS_OUTPUT.ENABLE (taille_tampon IN INTEGER DEFAULT 20000)

Cette procédure permet d'initialiser le tampon d'écriture et d'accepter les commandes de lecture et d'écriture dans ce tampon.

taille_tampon représente la taille maximum en octets allouée au tampon.

Les valeurs doivent être indiquées dans une plage de valeur allant de 2000 (minimum) et 1 million (maximum). Sa valeur par défaut est 20000.

Exceptions générées

ORA-20000: Buffer overflow, limit of (buffer_limit) bytes.

ORU-10027:

DBMS_OUTPUT.DISABLE: Cette procédure désactive les appels de lecture et écriture dans le tampon et purge ce dernier.

Procédures d'écriture dans le tampon

DBMS_OUTPUT.PUT (item IN NUMBER)

DBMS_OUTPUT.PUT (item IN VARCHAR2)

DBMS_OUTPUT.PUT (item IN DATE)

DBMS_OUTPUT.PUT_LINE (item IN NUMBER)

DBMS_OUTPUT.PUT_LINE (item IN VARCHAR2)

DBMS_OUTPUT.PUT_LINE (item IN DATE)

DBMS_OUTPUT.NEW_LINE

Composite data types

- **Can hold multiple values (unlike scalar types)**
- **Are of two types:**
  - ✓ PL/SQL records
  - ✓ PL/SQL collections
    - INDEX BY tables or associative arrays
    - Nested tables
    - VARRAY

Records

- **A PL/SQL record is a composite data structure that is a group of related data stored in fields.**
- **Each field in the PL/SQL record has its own name and data type.**
- **Declaring a PL/SQL record**
  - ✓ Programmer-defined records
  - ✓ Table-based record. %Rowtype
  - ✓ Cursor-based record.

Type RECORD

- Equivalent à struct en langage C
  - Déclaration

TYPE *nomRecord IS RECORD (champ1 type1, champ2 type2, …*);

- Utilisation du type RECORD

```
TYPE t_emp2 IS RECORD (num_matr integer, nom varchar(30));
v_employe t_emp2;
v_employe.num_matr := 500;
```

Exemple de record

```
DECLARE
TYPE t_EMP IS RECORD
(       v_emp_id employees.emplee_id%type,
        v_first_name employees.first_name%type,
        v_last_name employees.last_name%type
);

V_emp t_EMP;
BEGIN
```

Conflits de noms

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

```
DECLARE
    prenom varchar(30) := 'ali';
    BEGIN
    delete from emp where prenom = prenom;
```

- Pour éviter ce genre de contradiction, tout simplement ne pas donner de nom de colonne à une variable !

Visibilité des Variables

- une variable est visible dans le bloc où elle a été déclarée et dans les blocs imbriqués si elle n'a pas été redéfinie.

```
DECLARE
        var1 NUMBER(10);
        var2 CHAR(10);
BEGIN
        DECLARE
                var1 CHAR(10);
                var3 DATE;
         BEGIN
                … var1 CHAR(10)
                … var2
                … var3
        END;
        … var1 NUMBER(3)
        … var2 CHAR(10)
       … var3  CHAR(10) – génère une erreur
 END;
```

Variables de session

- **Outre les variables locales, un bloc Pl/sql peut utiliser d'autres variables: les variables de sessions(globales). La directive sql*plus à utiliser en début de bloc est *VARIABLE*. Dans le code PL/SQL il faut faire préfixer le nom de la variable de session par « : ».**
- **L'affichage de cette variable sous SQL*Plus est réalisé par la directive PRINT.**

```
SET SERVEROUTPUT ON;
VARIABLE nombre Number;
DECLARE
     Num number(3) :=0;
 BEGIN
       :nombre := Num+1;
       DBMS_OUTPUT.PUT_LINE(:nombre);
 END;
/
PRINT :nombre;
```

Fonction SQL en PL/SQL

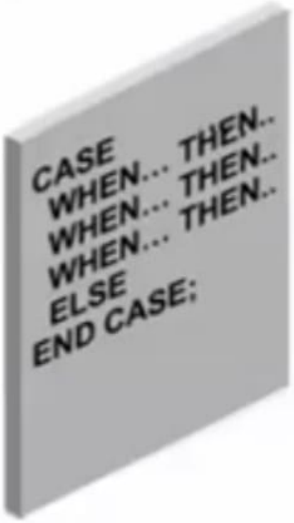
- **Available in procedural statements:**
  - ✓ Single-row functions
- **Exemples: v_ename:=substr(ename,1,5);**
  v_lname:=lenh(first_name);
  v_comm:=nvl(comm,0);
  v_date:=add_months(hiredate,3);
- **Not available in procesural statements:**
  - DECODE
  - Group functions
  - But we can use it in SQL statement insie PL/SQL not in a variables

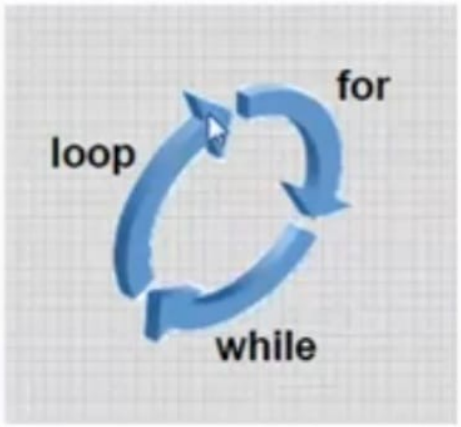
Types LOB

- **Avec les type Large Object on peut stocker des blocs de données non-structurées (textes, images, clips vidéos, sons) d'une taille pouvant aller jusqu'à 4 Go.**

CLOB	Stocke de gros blocs de données de type caractère 8 bits dans la base de données
BLOBE	Stocke de gros objets binaires : photos
BFILE	Stocke de gros objets binaires dans des fichiers du système opérateur à l'extérieur de la base de données

Controlling Flow of Execution





Structures de contrôle

- **Instructions conditionnelle**

```
IF condition1 THEN
    liste d'instructions1;
[ ELSIF condition2 THEN
    liste d'instructions2; ]
…
[ ELSE
    liste d'instructionsn; ]
END IF;
```

```
IF condition THEN
    instructions1;
[ ELSE
    instructionsn; ]
END IF;
```

```
IF condition THEN
    instructions
END IF;
```

- **Contrôle séquentiel**

```
BEGIN
    …
    GOTO label;
    …
    <<label>>
    instructions;
    …
END;
```

Structures de contrôle

- **Nested IF Statements**

```
DECLARE
sales NUMBER(8,2) := 12100; quota NUMBER(8,2) := 10000; bonus NUMBER(6,2);
    emp_id NUMBER(6) := 120;
BEGIN
IF sales > (quota + 200) THEN bonus := (sales - quota)/4;
ELSE IF sales > quota THEN bonus := 50; ELSE bonus := 0;
END IF; END IF;
UPDATE employees SET salary = salary + bonus WHERE employee_id = emp_id;
END; /
```

Using the IF-THEN-ELSIF Statement

```
DECLARE sales NUMBER(8,2) := 20000; bonus NUMBER(6,2); emp_id NUMBER(6)
    := 120;
BEGIN IF sales > 50000 THEN bonus := 1500; ELSIF sales > 35000 THEN bonus :=
    500; ELSE bonus := 100; END IF;
UPDATE employees SET salary = salary + bonus WHERE employee_id = emp_id;
END; /
```

Structures de contrôle

- **Extended IF-THEN Statement**

```
DECLARE
grade CHAR(1);
BEGIN grade := 'B';
IF grade = 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
ELSIF grade = 'B' THEN DBMS_OUTPUT.PUT_LINE('Very
    Good');
ELSIF grade = 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
ELSIF grade = 'D' THEN DBMS_OUTPUT. PUT_LINE('Fair');
 ELSIF grade = 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
END IF;
END; /
```

Structures de contrôle

Exemple

```
SET SERVEROUTPUT ON;
ACCEPT Chaine PROMPT 'Choisir votre sexe :F ou M: ';
BEGIN
   IF Upper('&Chaine') = 'F' THEN
        Goto Message1;
END IF;
IF Upper('&Chaine') = 'M' THEN
        Goto Message2;
END IF;
« Message1 »
DBMS_OUTPUT.PUT_LINE('Femme');
« Message2 »
DBMS_OUTPUT.PUT_LINE('Homme');
END;
/
```

Choix

```
CASE expression
WHEN expr1 THEN instructions1;
WHEN expr2 THEN instructions2;
…
ELSE instructionsN;
END CASE;
```

- **expression peut avoir n'importe quel type simple**

Les itérations

- **PL/SQL offre la possibilité d'écrire des boucles à l'aide d'instructions LOOP, WHILE ou FOR.**

Les itérations (2)

1ere forme:

```
LOOP
        instructions
EXIT WHEN expression
END LOOP ;
```

2eme forme:

La boucle FOR permet des boucles sur un ensemble fini comme un ensemble d'entiers ou un ensemble d'enregistrements issus d'une requête.

```
FOR i IN 1..10 LOOP
        DBMS_OUTPUT . PUT_LINE ( 'iteration ' ||i ) ;
END LOOP;
```

Les itérations (3)

3eme forme:

Enfin, le troisième type de boucle permet la sortie selon une condition prédéfinie.

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

Les itérations (4)

- **Exemple:**

```
DECLARE
        x NUMBER(3):=1;
BEGIN
   WHILE x<=100 LOOP
        INSERT INTO employe(noemp, nomemp, job, nodept)
        VALUES (x, 'TOTO', 'PROGRAMMEUR', 1);
        x:=x+1;
   END LOOP;
END;
```

Affichage

- **Activer le retour écran**
  - set serveroutput on size 10000
- **Affichage**
  - dbms_output.put_line(chaîne);
  - Utilise || pour faire une concaténation

Affichage (2)

```
DECLARE
compteur number(3);
i number(3);
BEGIN
select count(*) into compteur from clients;
FOR i IN 1..compteur LOOP
        dbms_output.put_line('Nombre : ' || i );
END LOOP;
END;
```

Affichage (3)

```
DECLARE
        i number(2);
BEGIN
        FOR i IN 1..5 LOOP
            dbms_output.put_line('Nombre : ' || i );
        END LOOP;
END;
```

La fonction prédéfini LENGTH

```
SET SERVEROUTPUT ON
DECLARE
        x VARCHAR (5);
        y VARCHAR2(5);
BEGIN
        x := 'toto'; y := 'toto';
        IF ( x = y) THEN
                DBMS_OUTPUT.PUT_LINE('equals');
        ELSE
                DBMS_OUTPUT.PUT_LINE('not equals');
        END IF;
        DBMS_OUTPUT.PUT_LINE(LENGTH(x));
        DBMS_OUTPUT.PUT_LINE(LENGTH(y));
END;
/
```

Autre exemple : **CONCAT**

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
          nom          VARCHAR2(10);
          salaire      NUMBER(7,2);
     BEGIN
          SELECT ename,sal INTO nom,salaire
          FROM emp WHERE ROWNUM = 1;
          DBMS_OUTPUT.PUT_LINE(nom || salaire);
          DBMS_OUTPUT.PUT_LINE(
                    CONCAT (nom,salaire));
     END;
/
```

Une autre façon de concaténer des chaînes de caractères

Un bloc peut contenir plusieurs requêtes

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
                nom             VARCHAR2(10);
                salaire         NUMBER(7,2);
        BEGIN
                SELECT ename INTO nom
                FROM emp WHERE ROWNUM = 1;
                DBMS_OUTPUT.PUT_LINE(nom);

                SELECT sal INTO salaire
                FROM emp WHERE ROWNUM = 1;
                DBMS_OUTPUT.PUT_LINE(salaire);
        EXCEPTION
                WHEN OTHERS
                        THEN NULL;
        END;
/
```

Structures de contrôle: exemple

```
DECLARE
 a number(2) := 10;
 BEGIN
   <<loopstart>>
dbms_output.put_line ('starting');
-- while loop execution
WHILE a < 30 LOOP
dbms_output.put_line ('value of a: ' || a);
a := a + 1; IF a = 15 THEN a := a + 1; END IF;
if  a=18 then
GOTO loopstart;
END IF;
END LOOP;
END;
/
```

Structures de contrôle: exemple

Reprenons le même exemple mais en changant l'emplacement de l'étiquete

```
DECLARE
 a number(2) := 10;
 BEGIN
-- while loop execution
WHILE a < 30 LOOP
dbms_output.put_line ('value of a: ' || a);
a := a + 1; IF a = 15 THEN a := a + 1; END IF;
IF  a=18 THEN
GOTO loopstart;
END IF;
END LOOP;
<<loopstart>>
dbms_output.put_line ('value of a: ' || a);
END;
/
```

Structures de contrôle: exemple

```
SET SERVEROUTPUT ON;
declare
        n integer;
begin
        select count(*) into n from clients where noclient = 5;
        if n > 0 then
              goto job1;
        else
              goto job2;
       end if;
       <<job1>>
       dbms_output.put_line('job1 – Dept has employees');
       return;
       <<job2>>
       dbms_output.put_line('job2 – Dept has no employees');
       return;
end;
/
```

```
declare
    n_numb number := 4;
  begin
    if n_numb < 5 then goto small_number;
    else goto large_number;
    end if;
    n_numb := 25; -- goto jumps this line.
   <<small_number>>
   dbms_output.put_line ('Small Number.');
   goto end_message;
   <<large_number>>
   dbms_output.put_line ('Large Number.');
   goto end_message;
   n_numb := 0;  -- goto jumps this line.
   <<end_message>>
   dbms_output.put_line ('The End.');
 end;
 /
```

Restrictions with GOTO Statement

GOTO Statement in PL/SQL imposes the following restrictions:

- **A GOTO statement cannot branch into an IF statement, CASE statement, LOOP statement or sub-block.**
- **A GOTO statement cannot branch from one IF statement clause to another or from one CASE statement WHEN clause to another.**
- **A GOTO statement cannot branch from an outer block into a sub-block (that is, an inner BEGIN-END block).**
- **A GOTO statement cannot branch out of a subprogram. To end a subprogram early, either use the RETURN statement or have GOTO branch to a place right before the end of the subprogram.**
- **A GOTO statement cannot branch from an exception handler back into the current BEGIN-END block. However, a GOTO statement can branch from an exception handler into an enclosing block.**

Instructions

❑ Les itérations

```
LOOP
   liste d'instructions;
   …
        [ IF condition THEN
            …
             EXIT;
           END IF; ]
   …
  [ EXIT WHEN condition; ]
END LOOP;
```

❑ Itération avec label

```
<<sortie>>
LOOP
   …
   LOOP
      …
      EXIT sortie WHEN …;    -- sortie de toutes les boucles
   END LOOP;
   …
END LOOP sortie;
```

Instructions

Exemples:

```
DECLARE
Compteur NUMBER(4) := 0;
BEGIN
LOOP

    DBMS_OUTPUT.PUT_LI
    NE(Compteur);
Compteur := Compteur + 1;
EXIT WHEN Compteur > 5;
END LOOP;
END;
/
```

Exemples avec étiquete:

```
DECLARE
Compteur NUMBER(4) := 0;
BEGIN
« exemple »
LOOP
 DBMS_OUTPUT.PUT_LINE(Compteur);
Compteur := Compteur + 1;
EXIT  exempleWHEN Compteur > 5;
END LOOP exemple;
END;
/
```

Instructions

```
WHILE condition LOOP
   liste d'instructions;
END LOOP;
```

- **Itération « Tant que »**

The WHILE-LOOP statement executes the statements in the loop body as long as a condition is true:

WHILE condition LOOP
sequence_of_statements
END LOOP;

Before each iteration of the loop, the condition is evaluated. If it is TRUE, the sequence of statements is executed, then control resumes at the top of the loop. If it is FALSE or NULL, the loop is skipped and control passes to the next statement.

- **Itération « Pour que »**

```
FOR compteur INTO [REVERSE] borne_inf..borne_sup LOOP
   liste d'instructions;
END LOOP;
```

Instructions: exemple

Exemple FOR:

BEGIN FOR i IN REVERSE 1..3 LOOP -- assign the values 1,2,3 to i DBMS_OUTPUT.PUT_LINE (TO_CHAR(i)); END LOOP; END; /

- **Inside a FOR loop, the counter can be read but cannot be changed.**

  BEGIN FOR i IN 1..3 LOOP -- assign the values 1,2,3 to i IF i < 3 THEN DBMS_OUTPUT.PUT_LINE (TO_CHAR(i)); ELSE i := 2; -- not allowed, raises an error END IF; END LOOP; END; /

Instructions

- **Using WHILE-LOOP for Control**

```
CREATE TABLE temp (tempid NUMBER(6), tempsal NUMBER(8,2), tempname VARCHAR2(25));
DECLARE sal employees.salary%TYPE := 0;
 mgr_id employees.manager_id%TYPE;
lname employees.last_name%TYPE;
 starting_empid employees.employee_id%TYPE := 120;
 BEGIN
SELECT manager_id INTO mgr_id FROM employees WHERE employee_id = starting_empid;
WHILE sal <= 15000 LOOP -- loop until sal > 15000
SELECT salary, manager_id, last_name INTO sal, mgr_id, lname FROM employees WHERE employee_id = mgr_id;
END LOOP;
INSERT INTO temp VALUES (NULL, sal, lname); -- insert NULL for tempid
COMMIT;
EXCEPTION WHEN NO_DATA_FOUND THEN INSERT INTO temp VALUES (NULL, NULL, 'Not found'); -- insert NULLs
COMMIT;
END;
/
```

Instructions

- **Choix**

```
CASE expression
    WHEN expr1 THEN instructions1;
    WHEN expr2 THEN instructions2;
     …
    ELSE instructionsN;
    END CASE;
```

expression peut avoir n'importe quel type simple (ne peut pas par exemple être un RECORD)

Instructions: exemple avec case

- **Using the CASE-WHEN Statement**

```
DECLARE
   grade CHAR(1);
BEGIN
   grade := 'B';
   CASE grade
     WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
     WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
     WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
     WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
     WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
      ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
      END CASE;
   END;
/
```

Instructions: exemple avec case

- **Using the Searched CASE Statement**

```
    DECLARE grade CHAR(1);
BEGIN
    grade := 'B';
    CASE WHEN grade = 'A' THEN
                DBMS_OUTPUT.PUT_LINE('Excellent');
            WHEN grade = 'B' THEN
                DBMS_OUTPUT.PUT_LINE('Very Good');
            WHEN grade = 'C' THEN
                DBMS_OUTPUT.PUT_LINE('Good');
            WHEN grade = 'D' THEN
                DBMS_OUTPUT.PUT_LINE('Fair');
            WHEN grade = 'F' THEN
                DBMS_OUTPUT.PUT_LINE('Poor');
            ELSE
               DBMS_OUTPUT.PUT_LINE('No such grade');
    END CASE;
END;
 -- rather than using the ELSE, in the CASE, could use the following -- EXCEPTION -- WHEN
      CASE_NOT_FOUND THEN -- DBMS_OUTPUT.PUT_LINE('No such grade'); /
```

Instructions: exemple exit

- **Using an EXIT Statement**

```
DECLARE
   credit_rating NUMBER := 0;
BEGIN
  LOOP credit_rating := credit_rating + 1;
    IF credit_rating > 3 THEN
      EXIT; -- exit loop immediately
    END IF;
 END LOOP; -- control resumes here
DBMS_OUTPUT.PUT_LINE ('Credit rating: ' ||  TO_CHAR(credit_rating));
IF credit_rating > 3 THEN
    RETURN; -- use RETURN not EXIT when outside a LOOP
END IF;
DBMS_OUTPUT.PUT_LINE ('Credit rating: ' || TO_CHAR(credit_rating));
END;
```

/

Using the EXIT-WHEN Statement

- The EXIT-WHEN statement lets a loop complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the loop completes and control passes to the next statement after the loop.
- Until the condition is true, the loop cannot complete. A statement inside the loop must change the value of the condition. In the previous example, if the FETCH statement returns a row, the condition is false. When the FETCH statement fails to return a row, the condition is true, the loop completes, and control passes to the CLOSE statement.
- The EXIT-WHEN statement replaces a simple IF statement. For example, compare the following statements:
  IF count > 100 THEN EXIT; ENDIF;

  EXIT WHEN count > 100;

These statements are logically equivalent, but the EXIT-WHEN statement is easier to read and understand.

Using the EXIT-WHEN Statement

- **Using EXIT With Labeled Loops**

```
DECLARE
  s PLS_INTEGER := 0;
  i PLS_INTEGER := 0;
  j PLS_INTEGER;
BEGIN
 <<outer_loop>>
  LOOP i := i + 1;
          j := 0;
         <<inner_loop>>
         LOOP
               j := j + 1; s := s + i * j; -- sum a bunch of products
              EXIT inner_loop WHEN (j > 5);
              EXIT outer_loop WHEN ((i * j) > 15);
         END LOOP inner_loop;
   END LOOP outer_loop;
DBMS_OUTPUT.PUT_LINE('The sum of products equals: ' || TO_CHAR(s)); END; /
```

SQL: Sequence(générateur de nombre)

- Permet d'obtenir des valeurs incrémentales
- Vérrouillage automatique en cas de concurrence d'accès
- Valeur suivante: nomSequene.NEXTVAL
- Valeur courente: nomSequene.CURRTVAL
- Création:

```
CREATE SEQUENCE nomSequence
                START WITH valeur_départ
                INCREMENT BY increment;
```

- Insertion:

```
INSERT INTO table VALUES(nomSequence.NEXTVAL,….);
```

- Suppression:

```
DROP SEQUENCE nomSequence;
```

SQL: Objet sequence

- **Exemple:**

```
SQL> CREATE TABLE test(numero number, nom varchar2(20));
Table crée
SQL> CREATE SEQUENCE seqTest START WITH 6 INCREMENT BY 3;
Séquence crée.
SQL> INSERT INTO test VALUES(seqTest.NEXTVAL, 'jack');
SQL> select * from test;
   numero    nom
---------   ----------------
      6    FADILI
      9    jack
SQL> select seqTest.CURRVAL from dual;
   CURRVAL
-----------
9
```

Pseudo-table DUAL

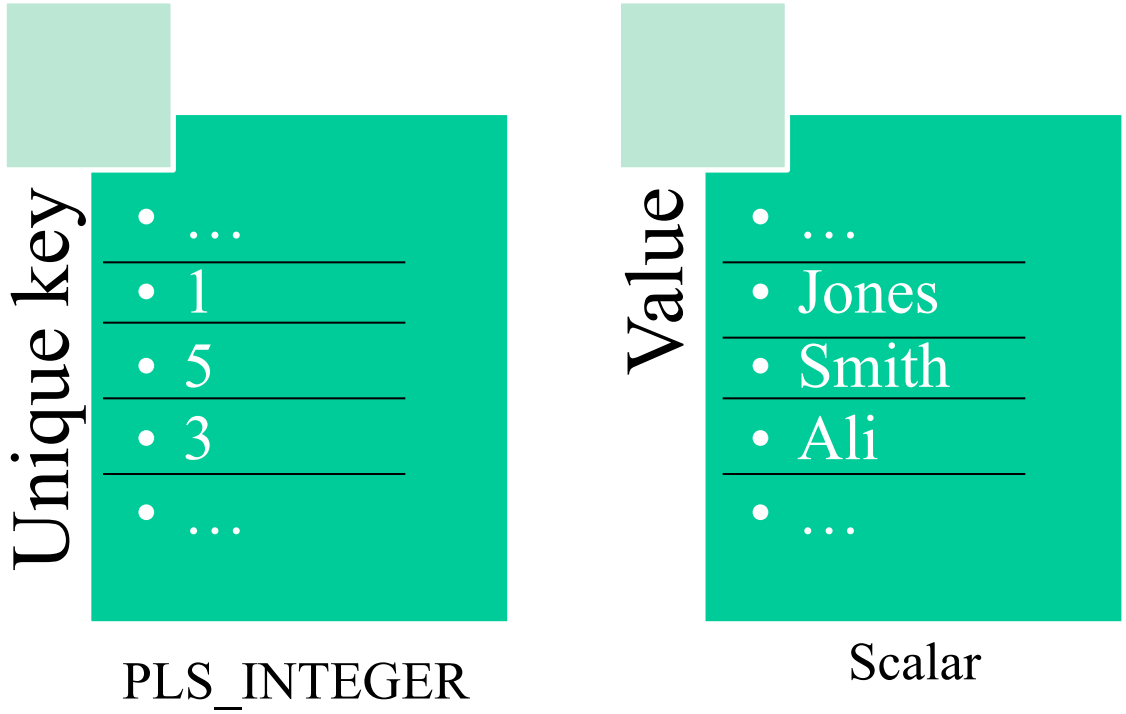
- La table DUAL est une table utilisable par tous (en lecture seulement) et qui appartient à l'utilisateur SYS. Le paradoxe de DUAL réside dans le fait qu'elle est couramment sollicitée, mais les interrogations ne portent jamais sur sa seule colonne (DUMMY définie en VARCHAR2 et contenant un seul enregistrement avec la valeur « X »). En conséquence, DUAL est qualifiée de pseudo-table (c'est la seule qui soit ainsi composée).
- L'interrogation de DUAL est utile pour évaluer une *expression de la manière suivante : « SELECT* expression FROM DUAL » (seule l'instruction SELECT est permise sur DUAL). Comme DUAL n'a qu'un seul enregistrement, les résultats fournis seront uniques (si aucune jointure ou opérateur ensembliste ne sont utilisés dans l'interrogation).

Les collections

- Ces types de données n'existent qu'en PL/SQL et n'ont pas d'équivalent dans la base Oracle. Il n'est pas possible de stocker un enregistrement directement dans la base
- Une collection est un ensemble ordonné d'éléments de même type. Elle est indexée par une valeur de type numérique ou alphanumérique. Elle ne peut avoir qu'une seule dimension (mais en créant des collections de collections on peut obtenir des tableaux à plusieurs dimensions)
- On peut distinguer trois types différents de collections :
  - ✓ Les tables (***INDEX-BY*** TABLES) qui peuvent être indicées par des variables numériques ou alpha-numériques
  - ✓ Les tables imbriquées(***NESTED TABLES***) qui sont indicées par des variables numériques et peuvent être lues et écrites directement depuis les colonnes d'une table
  - ✓ Les tableaux de type ***VARRAY***, indicés par des variables numériques, dont le nombre d'éléments maximum est fixé dès leur déclaration et peuvent être lus et écrits directement depuis les colonnes d'une table

INDEX BY Tables or Associative Arrays

- **Are PL/SQL structures with two columns:**
  - ✓ Primary key of integer or string data type
  - ✓ Column of scalar or record data type
- **Are unconstrained size. However, the size depends on the values that the key data type can hold**



INDEX BY Tables examples

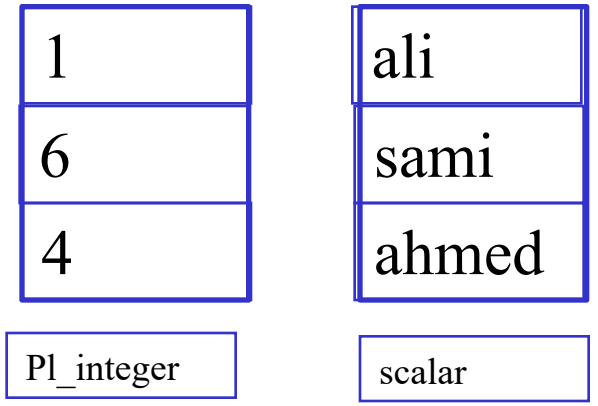
```
Declare
type tab_no is table of varachr2(100)
Index by pls_integer;

V_tab_no tab_no;

Begin

V_tab_no(1):='ali';
V_tab_no(6):='sami';
V_tab_no(4):='ahmed';

Dbms:output.put_line(v_tab_no(1));
Dbms:output.put_line(v_tab_no(6));
Dbms:output.put_line(v_tab_no(4));
```



INDEX BY Tables examples

```
Declare
type tab_no is table of pls_integer
Index by varchar2(100);

V_tab_no tab_no;

Begin
V_tab_no('ali'):=1;
V_tab_no('sami'):=6;
V_tab_no('hmed'):=4;

Dbms_output.put_line(v_tab_no('ali'));
Dbms_output.put_line(v_tab_no('sami'));
Dbms_output.put_line(v_tab_no('hmed'));
end;
```

Inverse de l'exemple précèdent

Using INDEX BY Table Methods

- **The following methods make INDEX BY tables easier to use:**
  - ✓ EXISTS PRIOR
  - ✓ COUNT NEXT
  - ✓ FIRST DELETE LAST
  - ✓ Syntax: table_name.method_name[(parameter)]

METHOD	Description
EXISTS(n)	Returns TRUE if the element in a PL/SQL table exists
COUNT	Returns the number of elements that a PL/SQL table currently contains
FIRST	- Returns the first (smallest) index number in a PL/SQL table - Returns NULL if the PL/SQL table is empty
LAST	- Returns the last (largest) index number in a PL/SQL table - Returns NULL if the PL/SQL table is empty
PRIOR(n)	Return the index number that precedes index n in PL/SQL table
NEXT(n)	Return the index number that succeeds index n in PL/SQL table
DELETE	- DELETE removes all elements from a PL/SQL table - DELETE(n) removes the nth element from a PL/SQL table - DELETE(m,n) removes all elements in the range m…n from a PL/SQL table

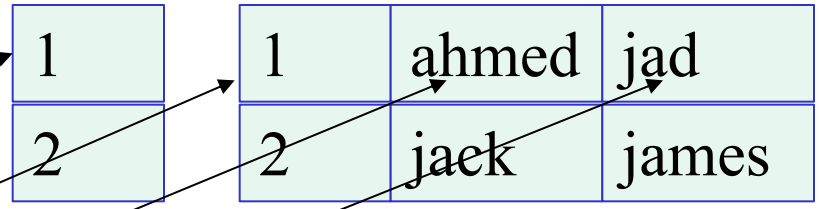
INDEX BY Table of Records(associative tables)

```
Declare
Type tab_no is table of employees%rowtype
Index by pls_integer;

v_tab_no tab_no;
v_total number;

BEGIN
v_tab_no(1).employee_id:=1;
v_tab_no(1).first_name:='ahmed;
v_tab_no(1).last_name:='jad';

v_tab_no(2).employee_id:=2;
v_tab_no(2).first_name:='jack';
V_tab_no(2).last_name:='james';
```



Dbms_output.put_line(v_tab_no(1).employee_id||v_tab_no(1).first_name||v_tab_no(1).last_name);
Dbms_output.put_line(v_tab_no(2).employee_id||v_tab_no(2).first_name||v_tab_no(2).last_name);

Nested Tables

- **No index in nested table (unlike index by table)**
- **It is valid data type in SQL (unlike index by table, only used in PL/SQL)**
- **Initialization required**
- **Extend required**
- **Can be stored in DB**
- **Syntax**

✓ TYPE type_name IS TABLE OF
{column_type | variable%type
| table.column%type} [NOT NULL]
| table%ROWTYPE

Nested Tables

- **Example1:**

```
Declare
Type t_locations IS table of varchar2(100);
Loc t_locations;

Begin
loc:=t_locations('jordan','uae','morocco');

Dbms_output.put_line(loc(1));
Dbms_output.put_line(loc(2));
Dbms_output.put_line(loc(3));

End;
```

Nested Tables

- **Example2:**

```
Declare
Type t_locations IS table of varchar2(100);
Loc t_locations;

Begin
loc:=t_locations('jordan','uae','morocco');
Loc.extend; -- essayer sans cette ligne ➔ ERREUR
loc(4):='egypt';
Dbms_output.put_line(loc(1));
Dbms_output.put_line(loc(2));
Dbms_output.put_line(loc(3));
Dbms_output.put_line(loc(4));
End;
```

VARRAY

- **Very close to the nested tables**
- **But we use varray when i know in advance its lenght**
- **We cannot use delete to a varray type**
- **We can use just a trim to it**

```
Declare
Type t_locations IS varray(size) of varchar2(100);
Loc t_locations;

Begin
loc:=t_locations('jordan','uae','morocco');

Dbms_output.put_line(loc(1));
Dbms_output.put_line(loc(2));
Dbms_output.put_line(loc(3));
End;
```

VARRAY

- **Example**

```
Declare
Type t_locations IS varray(3) of varchar2(100);
Loc t_locations;

Begin
loc:=t_locations('jordan','uae','morocco');
Loc.trim(1); --this delete one element from the last
Dbms_output.put_line(loc(1));
Dbms_output.put_line(loc(2));
--Dbms_output.put_line(loc(3));
End;
```

Characteristics for each type of collections

Collection type	Number of elements	Subscript type	Dense or sparse	Where created	Can be object type attribute
Associative array(index by table)	Unbounded	String or integer	Either	Only in PL/SQL block	No
Nested table	Unbounded	Integer	Starts dense, can become sparse	Either in PL/SQL block or at schema level	Yes
Variable-size array(varray)	Bounded	Integer	Always dense	Either in PL/SQL block or at schema level	Yes

Déclarations et initialisation

- **Les collections de type NESTED TABLE et INDEX-BY TABLES:**

Elles sont de taille dynamique et il n'existe pas forcément de valeur pour toutes les positions.

Déclaration d'une collection de type nested table:

TYPE nom type IS TABLE OF type élément [NOT NULL] ;

Déclaration d'une collection de type index by:

TYPE nom type IS TABLE OF type élément [NOT NULL] INDEX BY index_by_type ;

***index_by_type* représente l'un des types suivants :**

BINARY_INTEGER

PLS_INTEGER(9i)

VARCHAR2(taille)

LONG

Exemple

```
declare
    -- collection de type nested table
    TYPE TYP_NES_TAB is table of varchar2(100) ;
     -- collection de type index by
     TYPE TYP_IND_TAB is table of number index by binary_integer ;
     tab1 TYP_NES_TAB ;
      tab2 TYP_IND_TAB ;
 Begin
       tab1 := TYP_NES_TAB('Lundi','Mardi','Mercredi','Jeudi' ) ;
       for i in 1..10 loop
            tab2(i):= i ;
       end loop ;
 End;
 /
```

Les collections de type VARRAY

Ce type de collection possède une dimension maximale qui doit être précisée lors de sa déclaration. Elle possède une longueur fixe et donc la suppression d'éléments ne permet pas de gagner de place en mémoire. Ses éléments sont numérotés à partir de la valeur 1

Déclaration d'une collection de type VARRAY:

TYPE nom type IS VARRAY (taille maximum) OF type élément [NOT NULL] ;

```
declare
   -- collection de type VARRAY
   TYPE TYP_VAR_TAB is VARRAY(30) of varchar2(100) ;
   tab1 TYP_VAR_TAB := TYP_VAR_TAB('','','','','','','','','','');
   Begin
       for i in 1..10 loop     tab1(i):= to_char(i) ;    end loop ;
End;
/
```

Les enregistrements

TYPE nom type IS RECORD (nom_champ type_élément [[NOT NULL] := expression] [, ….]) ;

Nom_variable nom_type ;

Comme pour la déclaration des variables, il est possible d'initialiser les champs lors de leur déclaration

```
declare
    -- Record –
    TYPE T_REC_EMP IS RECORD (Num emp.empno%TYPE, Nom
    emp.ename%TYPE,  Job emp.job%TYPE );
     R_EMP T_REC_EMP ; -- variable enregistrement de type T_REC_EMP
Begin
    R_EMP.Num := 1 ;
    R_EMP.Nom := 'Scott' ; R_EMP.job := 'GASMAN' ;
 End;
/
```

Exemple

```
declare
   -- Record --
   TYPE T_REC_EMP IS RECORD (Num emp.empno%TYPE, Nom
    emp.ename%TYPE, Job emp.job%TYPE );
   -- Table de records –
   TYPE TAB_T_REC_EMP IS TABLE OF T_REC_EMP index by
    binary_integer ;
   t_rec TAB_T_REC_EMP ; -- variable tableau d'enregistrements
Begin
   t_rec(1).Num := 1 ;
   t_rec(1).Nom := 'Scott' ;
   t_rec(1).job := 'GASMAN' ;
   t_rec(2).Num := 2 ;    t_rec(2).Nom := 'Smith' ;     t_rec(2).job := 'CLERK' ;
End;
/
```

Exemple

```
Declare
    TYPE Temps IS RECORD ( heures SMALLINT, minutes SMALLINT,
    secondes SMALLINT );
    TYPE Vol IS RECORD
    (
          numvol PLS_INTEGER,
          Numavion VARCHAR2(15),
         Commandant Employe, -- type objet
         Passagers ListClients, -- type nested table
         depart Temps, -- type record
         arrivee Temps -- type record
     );
Begin
    ...
End ;
```

Initialisation des collections

Les collections de type NESTED TABLE et VARRAY doivent être initialisées avant toute utilisation (*à l'exception des collections de type INDEX-BY TABLE*).

Pour initialiser une collection, il faut se référer à son constructeur. Celui-ci, créé automatiquement par Oracle et porte le même nom que la collection.

```
Declare
   -- Déclaration d'un type tableau VARRAY de 30 éléments de type
   --Varchar2(100)
   TYPE TYP_VAR_TAB is VARRAY(30) of varchar2(100) ; -- Déclaration et initialisation d'une variable de type TYP_VAR_TAB t
    ab1 TYP_VAR_TAB := TYP_VAR_TAB('','','','','','','','','','');
```

Accès aux éléments d'une collection

La syntaxe d'accès à un élément d'une collection est la suivante :

Nom_collection(*indice*)

Indice peut être un *littéral*, une *variable* ou une *expression*

```
Declare
   Type TYPE_TAB_EMP IS TABLE OF Varchar2(60) INDEX BY
    BINARY_INTEGER ;
   emp_tab TYPE_TAB_EMP ;
   i pls_integer ;
Begin
  For i in 0..10 Loop
  emp_tab( i+1 ) := 'Emp ' || ltrim( to_char( i ) ) ;
  End loop ;
End ;
/
```

LTRIM (str): Efface tous les espaces blancs situés en début de chaîne.

RTRIM (str): Efface tous les espaces blancs situés en fin de chaîne.

Accès aux éléments d'une collection

```
Declare
    Type TYPE_TAB_JOURS IS TABLE OF PLS_INTEGER INDEX BY
    VARCHAR2(20) ;
    jour_tab TYPE_TAB_JOURS ;
Begin
    jour_tab( 'LUNDI' ) := 10 ;
    jour_tab( 'MARDI' ) := 20 ;
    jour_tab( 'MERCREDI' ) := 30 ;
End ;
/
```

Accès aux éléments d'une collection

```
Declare
    Type TYPE_TAB_EMP IS TABLE OF EMP%ROWTYPE INDEX BY
    BINARY_INTEGER ;
    Type TYPE_TAB_EMP2 IS TABLE OF EMP%ROWTYPE INDEX BY
    BINARY_INTEGER ;
    tab1 TYPE_TAB_EMP := TYPE_TAB_EMP( ... );
    tab2 TYPE_TAB_EMP := TYPE_TAB_EMP( ... );
    tab3 TYPE_TAB_EMP2 := TYPE_TAB_EMP2( ... );
Begin
   tab2 := tab1 ; -- OK
   tab3 := tab1 ; -- Illégal : types différents
   ...
End ;
```

Accès aux éléments d'une collection

- **Les collections ne peuvent pas être comparées entre elles.**
- **Exemple:**

```
Declare
     Type TYPE_TAB_STRING IS TABLE OF Varchar2(10) ;
      tab1 TYPE_TAB_STRING := TYPE_TAB_STRING( '1','2','3' );
      tab2 tab1%TYPE := TYPE_TAB_STRING( '1','2','3' );
 Begin
     If tab1 = tab2 Then null ;
     End if ;
 End ;
/
```

Les collections TABLE

Une table PL/SQL

- ***c'est une collection ordonnée d'élément du même type***
- **accessible uniquement en PL/SQL**
- **stockés en mémoire, ils peuvent grandir dynamiquement**
- ***ils utilisent des index non consécutif***
- **ils ont le même format que des champs d'une table**
- on n'utilise pas sql pour s'en servir
- Définition :
  TYPE **nom_tableau_pl** IS TABLE **OF** datatype [NOT NULL]
  INDEX BY BINARY_INTEGER; // **nom_tableau_pl** nom du tableau
  datatype type (curseur, record, variable...)
  Declaration :
  mon_tableau **nom_tableau_pl** ;

Les collections TABLE: exemple

```
-- table de multiplication par 8 et par 9...
      declare
   type tablemul is record ( par8 number, par9 number);
   type tabledentiers is table of tablemul index by binary_integer;
   ti tabledentiers;
   i number;
     begin
         for i in 1..10 loop
          ti(i).par9 := i*9 ;
          ti(i).par8:= i*8;
          dbms_output.put_line (i||'*8='||ti(i).par8||' '||i||'*9='||ti(i).par9 );
        end loop;
     end;
   /
```

Multidimentional arrays

- **Ecrire le code en commentaire dans un éditeur de code puis analyses le.**

Fonctions pour les tableaux

- PL/SQL propose un ensemble de fonctions qui permettent de manipuler des tableaux (également disponibles pour les *nested tables et varrays). Ces fonctions sont les suivantes*:
- Fonction Description

  EXISTS(*x*) *Retourne TRUE si le xeme élément du tableau existe.*

  COUNT Retourne le nombre d'éléments du tableau.

  FIRST / LAST Retourne le premier/dernier indice du tableau (NULL si tableau vide).

  PRIOR(x) / NEXT(*x*) *Retourne l'élément avant/après le xeme élément du tableau.*

  DELETE

  DELETE(*x*)

  DELETE(*x,y*) Supprime un ou plusieurs éléments au tableau.
- Il n'est pas possible actuellement d'appeler une de ces fonctions dans une instruction SQL (SELECT, INSERT, UPDATE ou DELETE).

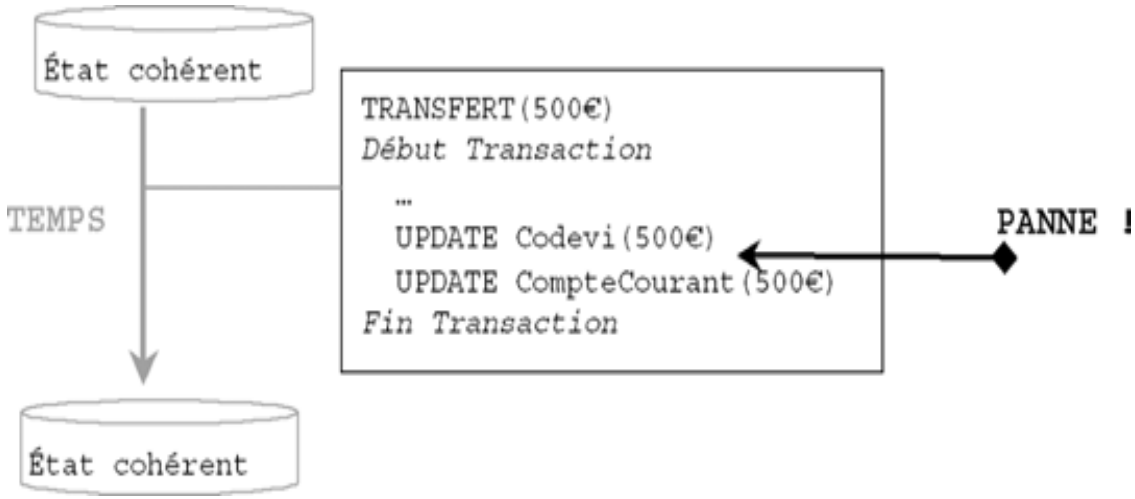
Méthodes pour les tableaux et varray

nom_table_sql.COUNT	retourne le nombre d'élements de la table
nom_table_sql.DELETE	supprime toute la table
nom_table_sql.DELETE (n)	supprime l'élément n
nom_table_sql.DELETE (n,m)	supprime les éléments de n à m
nom_table_sql.EXISTS (n)	retourne TRUE si l'élément n existe
nom_table_sql.EXTEND	
nom_table_sql.EXTEND	
nom_table_sql.EXTEND	
nom_table_sql.FIRST	retourne le premier index
nom_table_sql.LAST	retourne la valeur du dernier index
nom_table_sql.TRIM	élément suivant
nom_table_sql.TRIM(n)	
nom_table_sql.NEXT(n)	
nom_table_sql.PRIOR(n)	élément précédent

Transactions

- Une transaction est un bloc d'instructions LMD faisant passer la base de données d'un état cohérent à un autre état cohérent. Si un problème logiciel ou matériel survient au cours d'une transaction, aucune des instructions contenues dans la transaction n'est effectuée, quel que soit l'endroit de la transaction où est intervenue l'erreur.
- Le modèle le plus simple et le plus frappant d'une transaction est celui du transfert d'un compte épargne vers un compte courant. Imaginez qu'après une panne votre compte épargne a été débité de la somme de 500 € sans que votre compte courant ait été crédité de ce même montant ! Vous ne seriez pas très content, sans doute, des services de votre banque. Le mécanisme transactionnel empêche cet épisode fâcheux en invalidant toutes les opérations faites depuis le début de la transaction si une panne survient au cours de cette même transaction.

Transactions



Caractéristiques

Une transaction assure :

- l'atomicité des instructions qui sont considérées comme une seule opération (principe du tout ou rien) ;
- la cohérence (passage d'un état cohérent de la base à un autre état cohérent) ;
- l'isolation des transactions entre elles (lecture consistante, mécanisme décrit plus loin) ;
- la durabilité des opérations (les mises à jour perdurent même si une panne se produit après la transaction).

Début et fin d'une transaction

- Il n'existe pas d'ordre PL/SQL ou SQL qui marque le début d'une transaction. Ainsi le BEGIN d'un programme PL/SQL n'est pas forcément synonyme de son commencement.
- Une transaction débute à la première commande SQL rencontrée ou dès la fin de la transaction précédente.
- Une transaction se termine explicitement par les instructions SQL COMMIT ou ROLLBACK. Elle se termine implicitement :
  - à la première commande SQL du LDD ou du LCD rencontrée ;
  - à la fin normale d'une session utilisateur avec déconnexion ;
  - à la fin anormale d'une session utilisateur (sans déconnexion).

Exemple:

COMMIT:

Commande SQL (LDD ou LCD)

Fin normale d'une session.

ROLLBACK:

Fin anormale d'une session.

Transaction: Point de validation

- Vous pouvez tester rapidement une partie de ces caractéristiques en écrivant le bloc suivant qui insère une ligne dans une de vos tables :

COMMIT;

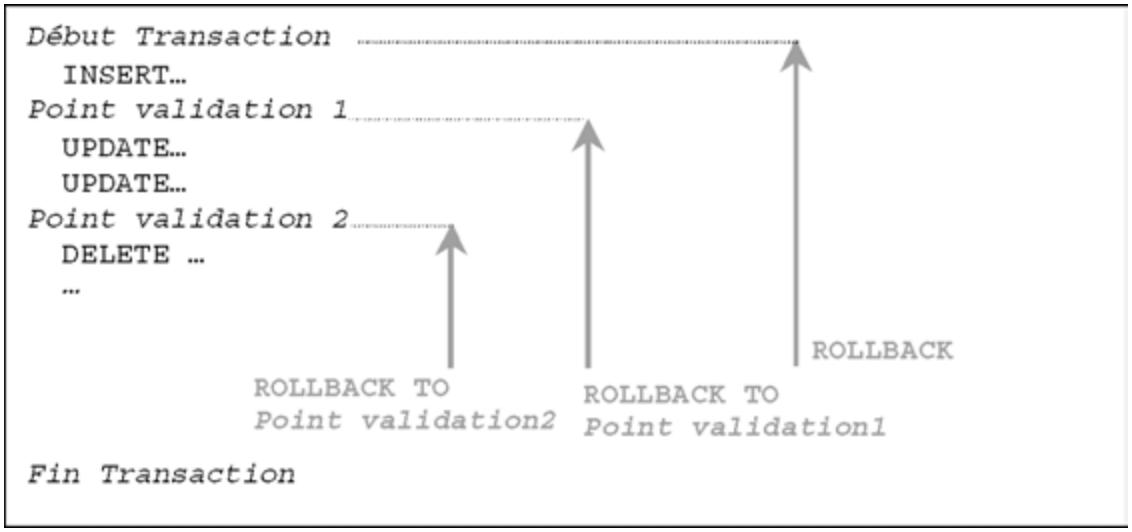
BEGIN

INSERT INTO *TableàVous VALUES (…);*

END;

/

SELECT * FROM *TableàVous;*



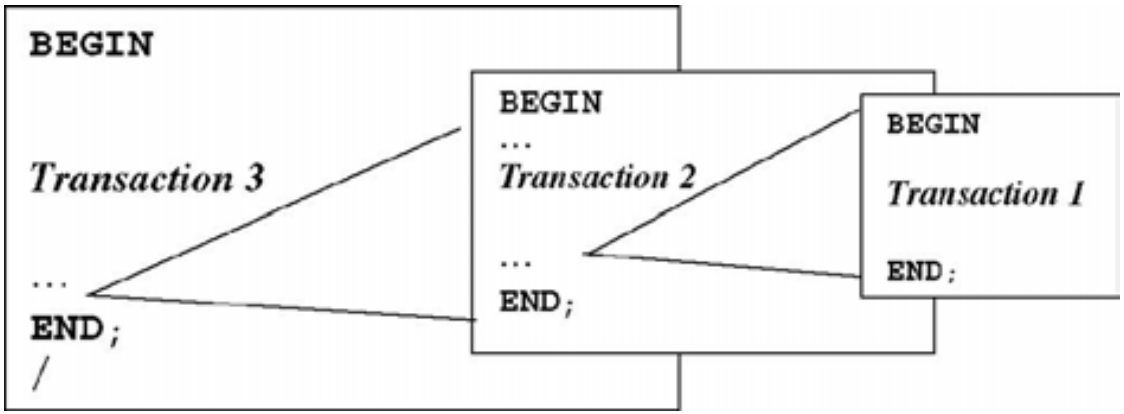
Transaction: Exemple

Code PL/SQL	Commentaires
BEGIN INSERT INTO Compagnie VALUES ('C2',2, 'Place Brassens', 'Blagnac', 'Easy Jet');	Première partie de la transaction.
SAVEPOINT P1; UPDATE Compagnie SET nrue = 125 WHERE comp = 'AF'; UPDATE Compagnie SET ville = 'Castanet' WHERE comp = 'C1';	Deuxième partie de la transaction.
SAVEPOINT P2; DELETE FROM Compagnie WHERE comp = 'C1';	Troisième partie de la transaction.
-- ROLLBACK TO P1;	Première partie à valider.
-- ROLLBACK TO P2;	Deuxième partie à valider.
-- ROLLBACK TO P3	Troisième partie à valider.
ROLLBACK;	Tout à invalider.
COMMIT; END;	Valide la ou les sous-parties.

Exemple: savepoint

```
Begin
    insert into EMP(empno, ename, job) values( 9991, 'Dupontont', 'CLERK' );
    insert into EMP(empno, ename, job ) values( 9992, 'Duboudin', 'CLERK' );
   SAVEPOINT mise_a_jour ;
   Update EMP Set sal = 2500 Where empno > 9990 ;
   ROLLBACK TO SAVEPOINT mise_a_jour ;
   Commit ;
 End ;
 / Procédure PL/SQL terminée avec succès.
SQL> Select * From EMP Where empno > 9990 ;
 EMPNO ENAME JOB   MGR HIREDATE SAL COMM DEPTNO
 --------- --------- --------   --------- ------- --------- --------- ---------
9991      Dupontont CLERK
9992      Duboudin  CLERK
```

Transactions imbriquées



BASES DE DONNÉES AVANCÉES: LES CURSEURS

Introduction

- **Manipulation de tuples:**
  - Les directives INSERT INTO, UPDATE et DELETE FROM peuvent être utilisées sans restriction avec des variables PL/SQL (scalaires, %ROWTYPE)
- **Lecture de tuples Chargement d'une variable à partir de la lecture d'un unique enregistrement dans la base (exception si 0 ou plusieurs enregistrements en réponse)**

```
DECLARE
heure_depart Vols.depart%TYPE;
BEGIN
SELECT Vols.depart INTO heure_depart
FROM Vols WHERE Vols.id = 'AF3517' ;
END;
```

Curseurs

- **Oracle crée des zones de travail pour exécuter les ordres SQL, stocker leurs résultats et les utiliser: Curseur**
- **Toutes les requêtes SQL sont associées à un curseur:**
  - ✓ Un curseur est un pointeur vers un résultat d'une requête
  - ✓ Le curseur peut être implicite (pas déclaré par l'utilisateur) ou explicite.
  - ✓ Les curseurs explicites permettent de manipuler l'ensemble des résultats d'une requête.

- **Where does the oracle server Process SQL statements?**
  - ✓ The oracle server allocates a private memory area called *context area* for processing SQL statements. The SQL statement is parsed and processed in this area. Information required for processing and information retrieved after processing are all stored in this area. You have no control over this area because it is internally managed by the oracle server.
  - ✓ A cursor is a pointer to the context area. However, this cursor is an implicit cursor and is automatically managed by the Oracle server. When the executable block issues a SQL statement, PL/SQL creates an implicit cursor.

Attributs des curseurs implicites

- **Tous les curseurs ont des attributs que l'utilisateur peut utiliser:**
  - ✓ SQL%ROWCOUNT : nombre de lignes traitées par la requête
  - ✓ SQL%FOUND : vrai si au moins une ligne a été traitée par la requête SQL%NOTFOUND : vrai si aucune ligne n'a été traitée par la requête

Pourquoi utiliser les curseur

- **Les instructions de type SELECT ... INTO ... manquent de souplesse, elles ne fonctionnent que sur des requêtes retournant une et une seule valeur. Ne serait-il pas intéressant de pouvoir placer dans des variables le résultat d'une requête retournant plusieurs lignes ?**

Fonctionnalités

- **Toutes les requêtes SQL sont associées à un curseur.**
- **Ce curseur représente la zone mémoire utilisée pour *parser et exécuter la requête.***
- **Le curseur peut être implicite (pas déclaré par l'utilisateur) ou explicite.**
- **Les curseurs explicites servent à retourner plusieurs lignes avec un select.**

Attributs des curseurs

- **Tous les curseurs ont des attributs que l'utilisateur peut utiliser:**
  - ✓ %ROWCOUNT : nombre de lignes traitées par le curseur
  - ✓ %FOUND : vrai si au moins une ligne a été traitée par la requête ou le dernier fetch
  - ✓ %NOTFOUND : vrai si aucune ligne n'a été traitée par la requête ou le dernier fetch
  - ✓ %ISOPEN : vrai si le curseur est ouvert (utile seulement pour les curseurs explicites)

Types de curseurs

- Curseurs implicites:
  - Créer automatiquement par Oracle lorsque la clause **INTO** accompagne le **select**
  - Pour toute requête SQL du LMD et pour les interrogations qui retournent un seul enregistrement
- Curseurs explicite
  - Déclaré et manipulés par le programmeur pour pouvoir traiter le résultat de requêtes retournant plus d'un tuple
  - L'utilisation d'un curseur explicite nécessite les étapes suivantes:
    1. Déclaration du curseur: section DECLARE
    2. Ouverture du curseur: section BEGIN
    3. Traitement des lignes: section BEGIN
    4. Fermeture du curseur: section BEGIN ou EXCEPTION

Curseurs

- **Définitions**
  - les échanges entre l'application et la base de données sont réalésés grâce à des curseurs: se sont des zones de travail capables de stocker plusieurs enregistrements et de gérer l'accès à des enregistrements
  - Stocke le résultat d'une requête retournant plusieurs enregistrements
  - Permet un traitement séquentiel des enregistrements retournés
- **Déclaration**
- **Utilisation**

```
CURSOR nomcurseur IS requête;

OPEN nomcurseur;
LOOP
    FETCH nomcurseur INTO variable;
    EXIT WHEN condition;
    instructions;
    …
END LOOP;
CLOSE nomcurseur;
```

Curseur implicite: exemple

```
BEGIN
        FOR x IN (SELECT * FROM emp WHERE sal < 100) LOOP
            dbms_Output.Put_Line(x.eName ||' '||x.sal||'... should REALLY be
            raised :D');
        END LOOP;
END; /
```

Curseurs

- **Attributs associés aux curseurs**

Tous les curseurs ont des attributs que l'utilisateur peut utiliser

%FOUND est égal à vrai (« TRUE ») si la commande FETCH retourne un enregistrement

%NOTFOUND est égal à faux (« FALSE ») si la commande FETCH retourne un enregistrement

%ROWCOUNT retourne le nombre d'enregistrements constituant le curseur

%ISOPEN indique si le curseur est dans un état ouvert (utilisable)

- **Gestion automatique d'un curseur**

```
OPEN …
LOOP
    FETCH …
    EXIT WHEN …
    …
END LOOP;
CLOSE …
```

```
FOR variable INTO nomcurseur LOOP
    …
    EXIT WHEN …
    …
END LOOP;
```

Curseurs implicites

- Nom: **SQL**; actualisé après chaque requête LMD et chaque select non associé à un curseur explicite
- A travers ses attributs, permet au programme PL/SQL d'obtenir des infos sur l'exécution des requêtes:

- SQL%FOUND: TRUE si la dernière requête LMD à affecté au moins 1 enregistrement
- SQL%NOTFOUND: TRUE si la dernière requête LMD n'a affecté aucun enregistrement
- SQL%ROWCOUNT: nombre de lignes affectées par la requête LMD

```
DELETE FROM pilote WHERE age >= 55;
DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT ||'pilotes partes à la retraite');
```

Curseurs explicites

❑ Les étapes de la vie d'un curseur

Les étapes d'utilisation d'un curseur explicite pour traiter un ordre select suivantes:

✓ Déclaration du curseur: CURSOR … IS …

✓ Utilisation du curseur:

- ouverture du curseur: OPEN
- Traitement des lignes: FETCH ou FOR
- Fermeture du curseur: CLOSE

Curseur explicite: Déclaration

- Tout curseur explicite utilisé dans un bloc PL/SQL doit être déclaré dans la section DECLARE du bloc, en précisant son nom et l'ordre sql associé.
- La syntaxe de déclaration d'un curseur explicite est la suivante:

CURSOR NOM_DU_CURSEUR IS REQUETE;

La requête peut contenir touts les ordres SQL d'intérrogation de données, y compris les opérateurs ensembliste UNION, INTERSECT ou MINUS.

DECLARE

CURSOR Liste_des_Pilotes IS Select * from pilote;

NB: Les expressions, calcul ou fonction SQL, dans la clause select de la requête du curseur, doivent comporter un alias pour pouvoir être référencées.

Curseur explicite exemple

- **La table utilisée est: employe (..., nomemp, sal, ville, ...)**

```
DECLARE
CURSOR emp_rabat IS select nomemp, sal FROM employe WHERE ville='rabat';
nom emplye.nomemp%type; salaire emplye.sal%type;
BEGIN
        OPEN emp_rabat;
        FETCH emp_rabat INTO nom, salaire;
        WHILE emp_rabat%found LOOP
        ….

        END LOOP;

END;
/
```

- La table utilisée est: employe (..., nomemp, sal, ville, ...)

```
DECLARE
CURSOR emp_rabat IS select nomemp, sal FROM employe WHERE ville='rabat';
nom emplye.nomemp%type; salaire emplye.sal%type;
BEGIN
        OPEN emp_rabat;
        FETCH emp_rabat INTO nom, salaire;
        WHILE emp_rabat%found LOOP
                IF salaire>3000 THEN
                        INSERT INTO resultat VALUES (nom,salaire);
                END IF;
                FETCH emp_rabat INTO nom, salaire;
        END LOOP;
END;
/
```

```
drop table employe;
create table employe(nomemp varchar2(20), sal number, ville varchar2(20));
insert into employe values('a',1,'rabat');
insert into employe values('b',2,'paris');
drop table resultat
create table resultat(nom varchar2(20),salaire number);


DECLARE
CURSOR emp_rabat IS select nomemp, sal FROM employe WHERE ville='rabat';
nom employe.nomemp%type; salaire employe.sal%type;
BEGIN
        OPEN emp_rabat;
        FETCH emp_rabat INTO nom, salaire;
        WHILE emp_rabat%found LOOP
                IF salaire>1 THEN
```

Curseur explicite: utilisation

OPEN Nom_curseur; -- ouverture du curseur

LOOP

    FETCH nom_curseur INTO variable; -- Traitement des lignes d'un curseur

    EXIT WHEN conditions;

Instructions;

…

END LOOP;

CLOSE nom_curseur; -- fermeture du curseur

Remarque:

L'utilisation de **FOR LOOP** remplace **OPEN,FETCH and CLOSE.**

Lorsque le curseur est invoqué, un enregistrement est automatiquement crée avec les mêmes éléments de données que ceux définies dans l'instruction SELECT

Curseur explicite: Attributs

- Pour chaque exécution d'un ordre de manipulation du curseur, le noyau renvoie une information appelée statut, qui indique si l'ordre a été exécuté avec succès ou non Cette information est disponible dans le programme par l'intermédiaire de 4 attributs rattachés à chaque curseur.

%FOUND	Est égal à vrai(« true ») si la commande FETCH retourne un enregistrement
%NOTFOUND	Est egal à faux(« FALSE ») si la commande FETCH retourne un enregistrement
%ISOPEN	Indique si le curseur est en état ouvert(utilisable)
%ROWCOUNT	Retourne le nombre d'enregistrements constituant le curseur

La syntaxe de consultation d'un attribut est : NOM_CURSOR%Attribut;

Curseur explicite: FETCH

- La commande FETCH ne retourne qu'un enregistrement. Pour récupérer l'ensemble des enregistrements de l'ordre SQL, il faut prévoir une boucle.

FETCH NOM_CURSOR INTO {NOM_ENREGISTREMENT| NOM_VARIABLE[,…]};

```
DECLARE
  Nom Pilote.plnom%TYPE;
  CURSOR Liste_pilote IS Select plnom From Pilote order by plnom;
  -- Enreg Liste_pilote%Rowtype;
BEGIN
  OPEN liste_pilote;
  LOOP
  FETCH liste_pilote into nom;   -- fetch liste_pilote into enreg;
  Exit when liste_pilote%notfound;
  dbms_output.put_line(nom);  -- traitement ou afficahge
  --dbms_output.put_line(Enreg.plnom);
   End loop;
   End liste_pilote;   End; /
```

Hints when oening a cursor

```
DECLARE
        CURSOR c_emp_cursor IS
        SELECT employee_id, last_name FROM employees
        WHERE departement_id =30;
…
BEGIN
  OPEN c_emp_cursor;
```

The OPEN statement executes the query associated with the cursor. Identifies the active set, and positions the cursor pointer at the first row. The OPEN statement is included in the executable section of the PL/SQL block

OPEN is an executable statement that performs the following operations:

1. Dynamically allocates memory for a context area
2. Parses the SELECT statement
3. Binds the input variables(sets the values for the input variables by obtaining their memory addresses)
4. Identifies the active set(the set of rows that satisfy the search criteria). Rows in the active set are not retrieved into variables when the open statement is executed. Rather, the FETCH statement retrieves the rows from the cursor to the variables.
5. positions the pointer to the first row in the active set.

Hints when fetching data from a cursor

Fetching Data from the cursor:

- the FETCH statement retrieves the rows from the cursor one at a time. After each fetch, the cursor advances to the next row in the active set. You can use the %NOTFOUND attribute to determine wheter the entire active set has been retrieved

- the FETCH statement performs the following operations:
  1. Reads the data for the current row into the output PL/SQL variables
  2. Advances the pointer to the next row in the active set

Hints when closing the cursor

The CLOSE statement disables the cursor, releases the context area, and ‘undefines’ the active set.

Close the cursor after completing the processing of the FETCH statement. You can reopen the cursor if required. A cursor can be reopened only if it is closed. If you attempt to fetch data from a cursor after it has been closed, then an INVALID_CURSOR exception will be raised.

Note: although it is possible to terminate the PL/SQL block whithout closing cursors. You should make it a habit to close any cursor that you declare explicitly to free up resources.

There is a maximum limit on the number of open cursors per session, which is determinated by the OPEN_CURSORS parameter in the database parameter file. (OPEN_CURSORS = 50 by default)

Example

Curseur implicite: FOR

- **La boucle FOR déclare implicitement la variable de parcours, ouvre le curseur, réalise les FETCH successifs et ferme le curseur.**

```
FOR nom_enregistrement in NOM_CURSEUR LOOP
    INSTRUCTIONS;
END LOOP;
```

```
SET SERVEROUTPUT ON;
DECLARE
  CURSOR liste_pilote IS Select plnom from pilote order by plnom;
BEGIN
  FOR v_pilote IN liste_pilote LOOP
  DBNS_OUTPUT.PUT_LINE(v_pilote.plnom); -- traitment ou affichage
  END LOOP;
END
/
```

Les curseurs paramétrés

- **Un curseur paramétré peut servir plusieurs fois avec des valeurs des paramètres différentes**
- **On doit fermer le curseur entre chaque utilisation de paramètres différents(sauf si on utilise « FOR » qui ferme automatiquement le curseur)**

CURSOR nomCurseur (param1, param2,…) IS …;

```
SET SERVEROUTPUT ON;
DECLARE
CURSOR c(Numpilote Integer) IS SELECT plnom from pilote WHERE npilote = Numpilote;
BEGIN
    FOR v_pilote in c(1) LOOP
       DBMS_OUTPUT.PUT_LINE(v_pilote.plnom);
    END LOOP;
    FOR v_pilote in c(2) LOOP
       DBMS_OUTPUT.PUT_LINE(v_pilote.plnom);
    END LOOP;
END;
/
```

Curseurs et Verrouillage

- Objectifs: Lorsqu'un curseur est ouvert: verrouiller l'accès aux colonnes référencées des lignes retournées par la requête afin de pouvoir les modifier.

CURSOR CursorName[(parametres]) IS SELECT listeColonnes1 FROM tableName WHERE condition

FOR UPDATE [OF listeColonnes2] [NOWAIT|WAIT NB_SECONDES]

- Absence de OF: toutes les colonnes sont verrouillées
- NOWAIT demande à Oracle de verrouiller les enregistrements correspondants immédiatement si les enregistrements sont déjà verrouillés; alors l'ouverture du curseur provoque une erreur.
- WAIT demande à Oracle de verrouiller les enregistrements correspondants si les enregistrements sont déjà verrouillés; alors le programme attend NB_SCONDES pour le déverrouillage, si nom l'ouverture du curseur provoque une erreur

Modification des lignes verrouillées

- **Restrictions: DISTINCT, GROUP BY, opérateurs ensemblistes et fonctions de groupe ne sont pas utilisables dans les curseurs FOR UPDATE**
- **La clause CURRENT OF permet d'accéder directement en modification ou en suppression à la ligne qui vient de ramener l'ordre FETCH ou FOR**

```
DECLARE
        CURSOR C1 IS SELECT plnom, sal FROM pilote
        FOR UPDATE OF sal WAIT 100;
         prime pilote.sal%type := 1000;
 BEGIN
        FOR salaireActuel in C1 LOOP
                update pilote  SET sal = sal + prime
                WHERE CURRENT OF C1;
                END LOOP;
END;
/
```

Les curseurs dynamiques

- **Un curseur dynamique n'est pas lié à une requête comme un curseur statique.**
- **Une variable de type curseur permet au curseur d'évoluer au cours du programme en lui associant diverses clauses SQL.**

Déclaration

```
Declare
        TYPE nomTypeCurDyn IS REF CURSOR;
         nomCurseur nomTypeCurDyn;
BEGIN
        … OPEN nomCurseur FOR 'requête dynamique;
```

Utilisation

```
            LOOP
             FETCH nomCurseur  INTO …
             EXIT WHEN …
               …
             END LOOP;
             CLOSE nomCurseur;
END;
 /
```

Curseurs RECAP

❑ Parcours complet

```
DECLARE
   CURSOR liste_etudiants IS                          -- déclaration du curseur
      SELECT numetu, nometu FROM ETUDIANTS;
BEGIN
   FOR enreg INTO liste_etudiants LOOP                -- parcours du curseur
      DBMS_OUTPUT.PUT_LINE(enreg.numetu || ' - '      -- traitement (affichage)
                           || enreg.nometu);
   END LOOP;
END;
```

❑ Parcours personnalisé

```
DECLARE
   CURSOR liste_etudiants IS                          -- déclaration du curseur
      SELECT numetu, nometu FROM ETUDIANTS;
   enreg liste_etudiants%ROWTYPE;
BEGIN
   OPEN liste_etudiants;                              -- ouverture du curseur
   LOOP
      FETCH liste_etudiants INTO enreg;               -- consommation d'une ligne
      EXIT WHEN liste_etudiants%NOTFOUND;             -- test de sortie de boucle
      DBMS_OUTPUT.PUT_LINE(enreg.numetu || ' - '      -- traitement (affichage)
                           || enreg.nometu);
   END LOOP;
   CLOSE liste_etudiants;                             -- fermeture du curseur
END;
```

CURSOR RECAP

```
DECLARE
   TYPE curseurDyn IS REF CURSOR;  -- type of cursor is dynamique
    ListePilote curseurDyn;  -- declaration of cursor
    Num pilote.npilote%TYPE;
    Nom pilote.plnom%TYPE;
BEGIN
    OPEN liste_pilote FOR SELECT npilote, plnom FROM pilote;
     LOOP
     FETCH liste_pilote INTO num, nom;
     EXIT WHEN liste_pilote%NOTFOUND;
     DBMS_OUTPUT.PUT_LINE(num||'-'||nom);
     END LOOP;
     CLOSE liste_pilote;
END;
/
```

Curseurs Récapitulatif

Exemple de curseur dynamique

```
DECLARE
   TYPE curseur_dynamique IS REF CURSOR; -- type curseur
dynamique
   liste_etudiants curseur_dynamique;    -- déclaration du
curseur
   enreg liste_etudiants%ROWTYPE;
BEGIN
   OPEN liste_etudiants FOR              -- ouverture du
curseur
      SELECT numetu, nometu FROM ETUDIANTS;
   LOOP
      FETCH liste_etudiants INTO enreg;  -- consommation
d'une ligne
      EXIT WHEN liste_etudiants%NOTFOUND; -- test de sortie
de boucle
      DBMS_OUTPUT.PUT_LINE(enreg.numetu || ' - '  --
traitement (affichage)
                           || enreg.nometu);
   END LOOP;
   CLOSE liste_etudiants; -- fermeture du curseur
END;
```

Summary: Using FOR UPDATE in Cursor

- FOR UPDATE Clause is always last statement in select
- Syntax:

SELECT …

FROM

FOR UPDATE [OF column_reference] [NOWAIT | WAIT n];

- Use explicit locking to deny access to other sessions for the duration of a transactiom.
- Lock the rows before the update or delete.

Summary: Using FOR UPDATE in Cursor

```
--This is a normal select
Select employee_id from employees
Where employee_id in (100,200)
Order by 1;
/* when you put for update, then no one can do any DML(I,U,D) for theses records, untill you finish your transaction(commit, rollback)
After you execute the next statement try to open another session and do like
update employees
set salary = salary+10;
*/
```

Summary: Using FOR UPDATE in Cursor

Select * from employees where employee_id in (100,200)

Order by 1

***For update*; -- this will give no error and no result but makes it searching if someone catching these records**

Commit;-- unlock

Select * from employees where employee_id in (100,200)

Order by 1

***For update nowait*; -- this will return for you error if someone catching these records**

Commit;-- unlock

Select * from employees where employee_id in (100,200)

Order by 1

***For update wait 30*; -- this will wait 30 seconds to give access to someone's catching these records**

Commit;-- unlock

Summary: For Update Cursors

```
---for upadate +current of : always togeter
DECLARE
        CURSOR c_emp_dept30 is
        SELECT employee_id, first_name, salary FROM employees
        Where department_id=30
        For update;
BEGIN
For i in c_emp_dept30
Loop
        update employees
        set salary=salary+1
        where current of c_emp_dept30; --either this instruction or the following one
        where employee_id=i.employee_id
End loop;
Commit;
END;
```

Gestion des exceptions

- **Si un programme affiche un flux inhabituel et inattendu pendant l'exécution, ce qui peut entraîner l'arrêt anormal du programme, la situation est considérée comme une exception. Ces erreurs doivent être interceptées et traitées dans la section EXCEPTION du bloc PL/SQL. Les gestionnaires d'exceptions peuvent supprimer l'arrêt anormal avec une action alternative et sécurisée.**
- **La gestion des exceptions est l'une des étapes importantes de la programmation d'une base de données. Les exceptions non gérées peuvent entraîner des pannes d'applications imprévues, avoir un impact sur la continuité des activités et frustrer les utilisateurs finaux.**

- **Il existe deux types d'exceptions: *définies par le système* et *définies par l'utilisateur*. Alors que la base de données Oracle déclenche implicitement une exception définie par le système, une exception définie par l'utilisateur est explicitement déclarée et déclenchée dans l'unité de programme.**
- **En outre, Oracle fournit deux fonctions utilitaires, SQLCODE et SQLERRM, pour récupérer le code d'erreur et le message de l'exception la plus récente.**

EXCEPTIONS

- **Structure:**

```
EXCEPTION
   WHEN nomexception1 THEN instructions1;
   WHEN nomexception2 THEN instructions2;
   …
 [ WHEN OTHERS THEN instructionsn; ]
```

Exemple: programme complet

```
SET SERVEROUTPUT ON;

ACCEPT nv PROMPT 'Numéro du vol'
BEGIN
 insert into vol values ( '&nv',to_date('01-05-2009 15:35:00','DD-MM-YYYY
HH24:MI:SS'),to_date('01-05-2009 19:00:00','DD-MM-YYYY HH24:MI:SS'),
'TOKYO','BEJING' );
DBMS_OUTPUT.PUT_LINE('Données Enregistrées');
 EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
    DBMS_OUTPUT.PUT_LINE('Doublon interdit dans une clé primaire');
    WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Autres types d'erreurs détectées');
END;
 /
```

Exceptions définies par le système

- **Comme son nom l'indique, les exceptions définies par le système sont définies et gérées implicitement par la base de données Oracle. Elles sont définis dans le package Oracle STANDARD. Chaque fois qu'une exception se produise dans un programme, la base de données récupère l'exception appropriée dans la liste disponible. Toutes les exceptions définies par le système sont associées à un code d'erreur négatif (sauf 1 à 100) et à un nom abrégé, qui est utilisé lors de la spécification des gestionnaires d'exceptions. Par exemple, le programme PL/SQL suivant inclut une instruction SELECT pour sélectionner les détails de l'employé 8376. Il déclenche une exception NO_DATA_FOUND car l'ID d'employé 8376 n'existe pas.**

```
DECLARE
   Nom VARCHAR2 (100);
   Salaire NUMBER;
   EMPID NUMBER := 8376;
BEGIN
   SELECT ENAME, SAL INTO Nom, Salaire
FROM EMP WHERE EMPNO = EMPID;
END;
/
```

```
DECLARE
 *
 ERROR at line 1:
 ORA-01403: no data found
 ORA-06512: at line 8
```

- **Réécrivons le bloc PL/SQL précédent pour inclure une section EXCEPTION et gérer l'exception NO_DATA_FOUND:**

```
DECLARE
   Nom VARCHAR2 (100);
   Salaire NUMBER;
   EMPID NUMBER := 8376;
BEGIN
   SELECT ENAME, SAL INTO Nom, Salaire FROM EMP WHERE EMPNO = EMPID;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('Aucun employé n existe avec l identifiant '||EMPID);
END;
/
```

Exception Name	Reason	Error Number
CURSOR_ALREADY_OPEN	When you open a cursor that is already open.	ORA-06511
INVALID_CURSOR	When you perform an invalid operation on a cursor like closing a cursor, fetch data from a cursor that is not opened.	ORA-01001
NO_DATA_FOUND	When a SELECT...INTO clause does not return any row from a table.	ORA-01403
TOO_MANY_ROWS	When you SELECT or fetch more than one row into a record or variable.	ORA-01422
ZERO_DIVIDE	When you attempt to divide a number by zero.	ORA-01476

Exception

- **Quelques exceptions prédéfinies**

	Nom	Code d'erreur	Code SQL
	CURSOR_ALREADY_OPEN	ORA-06511	-6511
(*)	DUP_VAL_ON_INDEX	ORA-00001	-1
	INVALID_CURSOR	ORA-01001	-1001
	INVALID_NUMBER	ORA-01722	-1722
	LOGIN_DENIED	ORA-01017	-1017
(*)	NO_DATA_FOUND	ORA-01403	-1403
	NOT_LOGGED_ON	ORA-01012	-1012
	PROGRAM_ERROR	ORA-06501	-6501
	STORAGE_ERROR	ORA-06500	-6500
	TIMEOUT_ON_RESOURCE	ORA-00051	-51
(*)	TOO_MANY_ROWS	ORA-01422	-1422
	VALUE_ERROR	ORA-06502	-6502
	ZERO_DIVIDE	ORA-01476	-1476

Exception

❑ Exemple

ETUDIANTS		
numetu	**nometu**	**nbf**
1	Jojo	1
2	Mimi	2
3	Polo	0

FORMATIONS	
numf	**nomf**
OR11	Oracle 11g
BDR	BD Objet-Relationnelles

INSCRIPTIONS		
numf	**numetu**	**datei**
OR11	2	22/09/2003
BDR	1	19/09/2003
OR11	2	22/09/2003

```
SET SERVEROUTPUT ON; -- Affichages visibles sur le poste client

ACCEPT ne PROMPT 'Numero de l''etudiant a ajouter:'
ACCEPT nm PROMPT 'Nom de l''etudiant:'

BEGIN
  INSERT INTO etudiant VALUES (&nf, '&nm', SYSDATE);
EXCEPTION
   WHEN DUP_VAL_ON_INDEX THEN
      DBMS_OUTPUT.PUT_LINE('Violation clé primaire');
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Erreur détectée');
END;
/
```

Exception définies par l'utilisateur

1. ***- Déclaration (Segment de déclaration, nommer l'exception)***

```
DECLARE
    nomexception EXCEPTION;
```

2. ***- Déclenchement de l'exception(produire: segment exécutable)***

```
…
IF condition THEN
    RAISE nomexception;
END IF;
…
```

3. ***- Traitement de l'exception(segment traitement des exceptions): traiter l'exception produite***

```
EXCEPTION
    WHEN nomexception THEN instructions;
```

Exception: Exemple

ETUDIANTS		
numetu	**nometu**	**nbf**
1	Jojo	1
2	Mimi	2
3	Polo	0

```
SET SERVEROUT ON;

ACCEPT ne  PROMPT 'Numero de l''etudiant a enregistrer:'
ACCEPT nme PROMPT 'Numero de l''etudiant a enregistrer:'

DECLARE
  n INTEGER;
  except_cle EXCEPTION;
BEGIN
  SELECT COUNT(*) INTO n
  FROM ETUDIANTS WHERE numetu = &ne;
  IF (n>0) THEN
     RAISE except_cle;
  ELSE
     INSERT INTO ETUDIANTS VALUES (&ne, '&nme', 0);
  END IF;
EXCEPTION
   WHEN except_cle THEN
      DBMS_OUTPUT.PUT_LINE('Ce numero d''etudiant existe déjà!');
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Autres Erreurs détectees');
END;
/
```

EXCEPTION: Exemple

```
DECLARE
        CURSOR liste_pilote IS select  nom FROM pilote;
 BEGIN
        OPEN liste_pilote;
        FOR v_pilote  IN liste_pilote  LOOP
           DBMS_OUTPUT.PUT_LINE(v_pilote.plnom);
        END LOOP;
EXCEPTION
        WHEN CURSOR_ALREADY_OPEN THEN
            DBMS_OUTPUT.PUT_LINE('le curseur est déjà ouvert');
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('autres erreurs c'est produite');
END;
/
```

Les Exceptions et les blocs imbriqué

Il n'est pas possible de déclarer la même exception deux fois dans le même bloc. Toutefois, dans le cas de blocs imbriqués, vous pouvez déclarer la même exception dans la section EXCEPTION de chaque bloc

```
DECLARE
        EXC1 Exception ;
BEGIN
        DECLARE
                EXC1 Exception ;
        BEGIN
                ... EXCEPTION
                WHEN EXC1 Then ...
        END ;
        EXCEPTION
            WHEN EXC1 Then ...
END ;
```

EXCEPTION

WHEN NO_DATA_FOUND Then ...

WHEN OTHERS THEN

If SQLCODE = …

Then …

Elsif SQLCODE = ...

Then ... ...

End if ;

END;

Fonctions SQLCODE et SQLERRM

Le code erreur numérique Oracle ayant généré la plus récente erreur est récupérable en interrogeant la fonction SQLCODE.

Le libellé erreur associé est récupérable en interrogeant la fonction SQLERRM

```
Declare
   LC$Chaine varchar2(10) ;
Begin
  LC$Chaine := rpad('a', 30, 'b') ;
Exception
    when others then
          dbms_output.put_line( 'Code erreur : ' || to_char( SQLCODE )) ;
          dbms_output.put_line( 'libellé erreur : ' || to_char( SQLERRM )) ;
End ;
 /
Code  erreur : -6502
libellé erreur : ORA-06502:
PL/SQL: numeric or value error: character string buffer too small Procédure PL/SQL terminée avec succès.
```

RPAD and LPAD

LPAD(char1,n,[char2]): Fait précéder la chaîne de caractères char1 de la chaîne de caractères char2 (espace si char2 non spécifié) jusqu'à la longueur n. Si n est inférieur à la longueur de char1, tronque char1 à la longueur n.

LPAD('Juin', 9,'=')= '=====Juin‘

LPAD('Juin',2,'=')='Ju'

```
begin
    for i in 1 .. 15
    loop
      dbms_output.put_line( lpad('string', i) || '<' );
   end loop;
end;
/
```

Associer un code ERR à une var Exception

On peut associer un code erreur Oracle à nos propres variables exception à l'aide du mot clé PRAGMA EXCEPTION_INIT, dans le cadre de la section déclarative de la façon suivante :

Nom_exception EXCEPTION ;
PRAGMA EXCEPTION_INIT(nom_exception, -code_error_oracle);

Exemple :
Lorsque l'on tente d'insérer plus de caractères dans une variable que sa déclaration ne le permet, Oracle déclenche une erreur -6502. Nous allons "nommer" cette erreur en LE$trop_long et l'intercepter dans la section exception

Associer un code ERR à une var Exception

```
 Declare
      LC$Chaine varchar2(10) ;
      LE$trop_long exception ;
      pragma exception_init( LE$trop_long, -6502 ) ;
 Begin
     LC$Chaine := rpad( ' ', 30) ;
 Exception
    when LE$trop_long then
    dbms_output.put_line( 'Chaîne de caractères trop longue') ;
 End ;
/
Chaîne de caractères trop longue
Procédure PL/SQL terminée avec succès.
SQL>
```

Poursuite de l'exécution après l'interception d'une exception

Une fois dans la section EXCEPTION, il n'est pas possible de retourner dans la section exécution juste après l'instruction qui a généré l'erreur.

Par contre il est tout à fait possible d'encadrer chaque groupe d'instructions voire même chaque instruction avec les mots clé

BEGIN … EXCEPTION … END;

Cela permet de traiter l'erreur et de continuer l'exécution

Poursuite de l'exécution après l'interception d'une exception

```
Declare
   LC$Ch1 varchar2(20) := 'Phrase longue';
   LC$Chaine varchar2(10) ;
   LE$trop_long exception ;
   pragma exception_init( LE$trop_long, -6502 ) ;
 Begin
   Begin
      LC$Chaine := LC$Ch1;
      Exception
     when LE$trop_long then
     LC$Chaine := Substr( LC$Ch1, 1, 10 ) ;
   End ;   -- poursuite du traitement –
  dbms_output.put_line(LC$Chaine ) ;
End ;
 / Phrase lon
```

Procédure PL/SQL terminée avec succès.

Utiliser votre propre message d'erreur

Vous pouvez également définir vos propres messages d'erreur avec la procédure RAISE_APPLICATION_ERROR

DBMS_STANDARD.raise_application_error(numero_erreur, message[, {TRUE | FALSE}])

numero_erreur représente un entier négatif compris entre -20000 et -20999

message représente le texte du message d'une longueur maximum de 2048 octets

TRUE indique que l'erreur est ajoutée à la pile des erreurs précedentes

FALSE indique que l'erreur remplace toutes les erreurs précédentes

Sous-programmes: Procédures cataloguées

Principe

- Bloc PL/SQL nommé stocké dans la BD
- Utilisation par plusieurs utilisateurs
- Exécution à partir des applications ou d'autres procédures cataloguées

Structure

```
CREATE [ OR REPLACE ] PROCEDURE nomprocedure
    [(parametre1,…, parametren)]
 [AUTHID { CURRENT_USER | DEFINER }]
 [IS]
    [ variables_internes ]
 BEGIN
    instructions
 [ EXCEPTION …]
 END [ nomprocedure ];
 /

nomvariable [IN|OUT|IN OUT] type [{:=|DEFAULT} valeur]
```

Procédures

- CREATE indique que l'on veut créer une procédure stockée dans la base
  La clause facultative OR REPLACE permet d'écraser une procédure existante portant le même nom
  nom procédure est le nom donné par l'utilisateur à la procédure
  AUTHID indique sur quel schéma la procédure s'applique : CURRENT_USER
  Indique que la procédure utilise les objets du schéma de l'utilisateur qui appelle la procédure

  DEFINER(défaut): Indique que la procédure utilise les objets du schéma de création de la procédure

Procédures

IN: indique que le paramètre n'est pas modifiable par la procédure

OUT: indique que le paramètre est modifiable par la procédure

IN OUT: indique que le paramètre est transmis par le programme appelant et renseigné par la procédure

Par défaut, les paramètres sont transmis par copie, c'est à dire qu'un espace mémoire est créé pour recevoir une copie de la valeur

Sous-Programmes

❑ Exemple de procédure

```
CREATE OR REPLACE PROCEDURE insere_etudiant
          ( ne  NUMBER,                    ( ne ETUDIANTS.numetu%TYPE,
            nme ETUDIANTS.nometu%TYPE) IS    nme ETUDIANTS.nometu%TYPE )
   n INTEGER;
   except_cle EXCEPTION;
BEGIN
   SELECT COUNT(*) INTO n
   FROM ETUDIANTS WHERE numetu = ne;
   IF (n>0) THEN
      RAISE except_cle;
   ELSE
      INSERT INTO ETUDIANTS VALUES (ne, nme, 0);
   END IF;
EXCEPTION
   WHEN except_cle THEN
      DBMS_OUTPUT.PUT_LINE('Numero d''etudiant existant!');
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Erreur detectee');
END insere_etudiant;
/
```

Procédures cataloguées: Exemple

```
CREATE OR REPLACE PROCEDURE insert_pilote
   (num in Number) IS NumVol vol.npilote%TYPE;
   Excep_key Exception;
BEGIN
   Select count(*) INTO numVol FROM vol WHERE nvol = num;
   IF numVol >0 THEN   RAISE Except_key;
    ELSE
        Insert INTO vol values(num,1,2,'rak','paris','0816','1021',SYSDATE);
     DBMS_OUTPUT.PUT_LINE('Données Enregistrées');
    END IF;
EXCEPTION
    WHEN Excep_key THEN
            DBMS_OUTPUT.PUT_LINE('Num de vol existant');
    WHEN OTHERS THEN
           DBMS_OUTPUT.PUT_LINE('ERREUR détéctée');
 END insertPilote;
 /
```

Compilation

Pour compiler ces sous-programmes à partir de l'interface SQL*Plus, il faut rajouter le symbole / en première colonne après chaque dernier END. Si le message suivant apparaît: Avertissement : *Fonction/Procédure créée avec erreurs de compilation,* deux techniques peuvent être utilisées pour visualiser les erreurs de compilation :

- faire SHOW ERRORS sous SQL*Plus ;
- interroger la vue USER_ERRORS (SELECT LINE,POSITION,TEXT FROM USER_ERRORS WHERE NAME='*nomFonction/nomProcédure';).*

Une fois que le message *Fonction/Procédure créée. apparaît, le sous-programme est* correctement compilé et stocké en base.

Sous-programmes

- **Appels des sous-programmes**

  - A partir de SQL*Plus

```
EXECUTE nomprocedure [(parametre1,…, parametren)];
```

  - A partir de PL/SQL

```
DECLARE res typeresultat;
BEGIN
   …
   nomprocedure [(parametre1,…, parametren)];
   res := nomfonction[(parametre1,…, parametren)];
   …
END;
/
```

- **Suppression de sous-programmes**

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

Appel d'unr procedure à partir de SQL*Plus

```
/* create a procedure that take emp_id as parameter and return the first_name and salary.
We use bind variable to print the first_name and salary*/
Create or replace procedure query_emp
(p_emp_id employees.employee_id%type,
p_f_name out employees.first_name%type,
p_sal out employees.salary%type)
Is
Begin
        select first_name,salary into p_f_name, p_sal from employees
        where employee_id=p_emp_id;
Exception
When others then
Dbms_output.put_line(sqlcode);
Dbms_output.put_line(sqlerrm);
End;
--we should declare 2 bind variables
Variable b_first_name varchar2(100)
Variable b_sal numer

Execute query_emp(105,:b_first_name,:b_sal)
Print b_first_name b_sal;
```

Appel d'unr procedure à partir d'un bloc pl/sql

```
/* c'est la méthode la plus pratique*/
Declare
V_first_name employees.first_name%type;
V_sal employees.salary%type;
Begin
Query_emp(105,v_first_name,v_sal);
Dbms_output.put_line(v_first_name);
Dbms_output.put_line(v_sal);
End;
```

In and Out in procedure

```
Create or replace procesure formate_tel
(p_tel in out varchar2)
Is
Begin
P_tel:=substr(p_tel,1,3)|| '('||substr(p_tel,4,2)||')'|| substr(p_tel,1,7);
end;
---------- utilisation d'une variable de bind
Variable b_tel varchar2(20);
Execute :b_tel:='12345677890';
Execute format_tel(:b_tel);
Print b_tel;
---------- deuxième méthode (bloc pl/sql)
Declare
V_tel varchar2(100) := '232434345';
Begin
Formate_tel(v_tel)
dbms:_output.put_line(v_tel);
End;
```

Comparing the parameter modes

IN	OUT	IN OUT
Default mode	Must be specified	Must be specified
Value is passed into subprogram	Returned to calling environment	Passed into subprogram; returned to calling environment
Formal parameter acts as a constant	Unitialized variable	Intialized variable
Actual parameter can be a literal, expression, constant, or initialized variable	Must be a variable	Must be a variable
Can be assigned a default value	Cannot be assigned a default value	Cannot be assigned a default value

Stored functions

- **A function :**
  - ✓ Is a named PL/SQL block that returns a value (compute a value)
  - ✓ Can be stord in the database as a schema object for repeated execution
  - ✓ Is called as part of an expression or is used to provide a paramater value
- **Examples:**
  - ✓ We can create function to return the salary for an employee
  - ✓ Function to retrieve the fall name for the employee
  - ✓ Function to calculate the average of salary
  - ✓ Function to cimpute the tax for a salary

Sous-programmes: Fonctions cataloguées

- Principe
  - Programme PL/SQL stocké dans la BD retournant un résultat
  - Même fonctionnement que les procédures
- Structure

```
CREATE [ OR REPLACE ] FUNCTION nomfonctiom
    [(parametre1,…, parametren)]
RETURN typeresultat
 IS|AS
    [ variables_internes ]
 BEGIN
    instructions
 [ EXCEPTION …]
  RETURN expression;
 END [ nomfonction];
 /
```

PL/SQL Block

- Ghost variable not allowed, also substitute variables &
- It should be at least one return expression in executable section
- **Return datatype should be whithout size**
- Out / IN OUT can be used, but this not good practice

```
nomvariable [IN|OUT|IN OUT] type [{:=|DEFAULT} valeur]
```

The difference between Procedures and Functions

Procedures	Functions
Execute as a PL/SQL statement	Invoke as part of an expression
Do not contain RETURN clause in the header	Must contain a RETURN clause in the header
Can pass values (if any) using output parameters	Must return a single value
Can contain a RETURN statement whithout a value	Must contain at least one RETURN statement

To perform an action	To return a value
Can not be used in select	Can be used in select But it should not include OUT/IN OUT parameters

Aprocedure that have one parameter(OUT) would be better rewritten as a function

Exemples

```
-- create a function to return the salary for an employee
-- we need one parameter (in) number(emploee_id)
-- the return value should be also number it is the salary
Create or replace function get_sal(p_emp_id number)  return number
Is v_sal number;
Begin
        select salary into v_sal from employees where employee_id=emp_id;
Return v_sal; end;
-- now we have many methods to invoke the function
--1 as part of expression
Declare
v_sal number;
begin
v__sal:=get_sal(100);
Dbms_output.put_line(v_sal);
END;
```

Continuation of the example

```
--2 as parameter value
begin
Dbms_output.put_line(get_sal(100));
END;
..also we can do this
begin
Execute dbms_output.put_line(get_sal(100));
END;
--3 using host variable
Variable b_salary:=get_sal(100);
Execute :b_salary:=get_sal(100)
Print b_salary
```

Continuation of the example

```
--4 as part of select (not a good practice)
Select get_sal(100) from dual;
Select employee_id, first_name, get_sal(employee_id)
from employees
Where department_id=20;
-------------------------
Select * from user_objects where object_name='GET_SAL';

Select line, text from user_source where name='GET_SAL';
--now let try this
Begin
Dbms_output.put_line(get_sal(9999));
End; // exception no data found
Select get_sal(9999) from dual; // no data found not raised in select query
because the function doesnt handle exception
```

Create the function whith exception block

```
Create or replace function get_sal(p_emp_id number)  return number
Is v_sal number;
Begin
    select salary into v_sal from employees where employee_id=emp_id;
    Return v_sal;
    Exception
    When no_data_found then
    Return -1;
end;
```

Sous-programmes: fonctions

❑ Exemples de fonction

```
CREATE OR REPLACE FUNCTION nb_etudiants() RETURN INTEGER IS
   n INTEGER;
BEGIN
   SELECT COUNT(*) INTO n
   FROM ETUDIANTS;
   RETURN n;
END nb_etudiants;
/
```

❑ Récursivité

```
CREATE OR REPLACE FUNCTION factorielle ( n INTEGER ) RETURN INTEGER IS
BEGIN
   IF n = 1 THEN
      RETURN 1;
   ELSE
      RETURN n * factorielle(n-1);
   END IF;
END factorielle;
/
```

Sous-programmes: Exemple 2

```
CREATE OR REPLACE FUNCTION moy_salaire
        RETURN FLOAT
 IS
        v_moy FLOAT;
 BEGIN
        SELECT AVG(sal) INTO v_moy FROM pilote;
        RETURN v_moy;
 END moy_salaire:
 /
```

- Utilisation en SQL*PLUS:

  select plnom from pilote where (sal>moy_salaire());

- utilisation en PL/SQL:

  select moy_salaire from dual;

Fonction: exemple

```
CREATE OR REPLACE PROCEDURE PlusExpérimenté (pcomp IN OUT VARCHAR2, pnomPil OUT VARCHAR2,
      pheuresVol OUT NUMBER)
 IS p1 NUMBER;
BEGIN
    IF (pcomp IS NULL) THEN
    SELECT COUNT(*) INTO p1 FROM Pilote WHERE nbHVol = (SELECT MAX(nbHVol) FROM Pilote);
   ELSE   SELECT COUNT(*) INTO p1 FROM Pilote WHERE nbHVol = (SELECT MAX(nbHVol) FROM Pilote
      WHERE comp = pcomp) AND comp = pcomp;
  END IF;
  IF p1 = 0 THEN
   DBMS_OUTPUT.PUT_LINE('Aucun pilote n''est le plus expérimenté');
  ELSIF p1 > 1 THEN
    DBMS_OUTPUT.PUT_LINE('Plusieurs pilotes sont les plus expérimentés');
   ELSE
   IF (pcomp IS NULL) THEN
     SELECT nom, nbHVol, comp INTO pnomPil, pheuresVol, pcomp FROM Pilote WHERE nbHVol = (SELECT
      MAX(nbHVol) FROM Pilote);
   ELSE  SELECT nom, nbHVol INTO pnomPil, pheuresVol FROM Pilote  WHERE nbHVol = (SELECT
      MAX(nbHVol) FROM Pilote WHERE comp = pcomp) AND comp = pcomp;
   END IF;
   END IF;
End PlusExpérimenté;
```

Avantages of User-defined Functions in SQL statements

- **Can extend SQL where activities are too complex, too awkward, or unavailable with SQL**
- **Can increase efficiency when used in the WHERE clause to filter data, as opposed to filtering the data in the application**
- **Can manipulate data values**
- **Calling user-defined finctions in sql statements:**
  - ✓ The SELECT list or clause of a query
  - ✓ Conditional expressions of the WHERE and Having clauses
  - ✓ The CONNECT BY, START WHEN, ORDER BY, and GROUP BY clauses of a query
  - ✓ The VALUES clause of the INSERT statement
  - ✓ The SET clause in the UPDATE statement

Restrictions when calling Functions fron SQL expressions

- **User-efined functions that are callable from SQL expressions must :**
  - ✓ Be stored in the DB
  - ✓ Accept only IN parameters with valid SQL data types, not PL/SQL-specific types (RECORD,TABLE,Boolean)
  - ✓ Return valid SQL data types, not PLSQL-specific types
- **When calling functions in SQL statements:**
  - ✓ Parameters must be specified with positional notation (before 11g but after we can use naming)
  - ✓ You must own the function or have the EXECUTE privilege
- **Can not be used in chack constraint (create table/alter table)**
- **Can not be used as default value for a column**

Controlling side effects when calling functions ***(not procedure)*** from SQL expressions

- **Functions called from:**
  - ✓ A SELECT statement cannot contain DML statements
  - ✓ An UPDATE or DELETE statement on a table T cannot query or contain DML on the same table T
  - ✓ SQL statements cannot end transactions (that is, cannot execute COMMIT or ROLLBACK operations)
- **Note : calls to subprograms that break rhese restrictions are also not allowed in the function**

Paquetages(packages)

- **Principe**
  - Regroupement d'un ensemble applicatifs
  - Contient : fonctions, procédures, variables, constantes, curseurs, types, exceptions
- **Structure**

Paquetage

SPECIFICATIONS

-Interface accessible au programme appelant
- Ne contient que les déclarations des procédures ou fonctions publiques
- variables globales
-Curseurs globaux

CORPS

- Corps des procédures ou des fonctions citées dans la partie spécification
- Nouvelles procédures ou fonctions privées accessibles uniquement par des procédures ou fonctions du package

- **Intérêts**
  - ✓ Encapsulation données/programmes, accès uniquement aux composants publics
  - ✓ Persistance des variables pour la session
  - ✓ Modularité : modifications du corps sans toucher aux spécifications
  - ✓ Performant : un appel charge le paquetage en mémoire et exécution dans le noyau serveur

What are PL/SQL Packages?

- **A package is a schema object that groups logically related PL/SQL types, variables and subprograms.**
- **Packages usually have two parts**
  - ✓ A specification
  - ✓ A body
- **The specification is the interface to the package. It declares the types, variables, constants, exceptions, cursors and subprograms that can be referenced from outside the package**
- **The body defines the queries for the cursors and the coe for the subprograms**
- **Enable the oracle server to read multiple objects into memory at once.**

Paquetages

- **Spécification**

```
CREATE PACKAGE nompaquetage [IS]
   -- types, variables, constantes, curseurs publics
   -- spécifications de sous-programmes publics
END [ nompaquetage ];
/
```

- **Corps**

```
CREATE PACKAGE BODY nompaquetage [IS]
   -- types, variables, constantes, curseurs privés
   -- corps de sous-programmes publics & privés
END [ nompaquetage ];
/
```

- **Suppressions**

```
DROP PACKAGE BODY nompaquetage;
```
-----→ corps

```
DROP PACKAGE nompaquetage;
```
-----→ corps + spécification

Paquetages: Exemple1

✓ Spécification

```
CREATE PACKAGE gestion_etudiants IS

   TYPE t_inscriptions IS RECORD (numetu ETUDIANTS.numetu%TYPE,
                                  numf   FORMATIONS.numf%TYPE,
                                  datei  DATE);

   CURSOR C_inscrits RETURN t_inscriptions;

   PROCEDURE enregistrement(ne ETUDIANTS.numetu%TYPE,
                            nm ETUDIANTS.nometu%TYPE);

   PROCEDURE exclusion(ne ETUDIANTS.numetu%TYPE);

   PROCEDURE inscription(ne ETUDIANTS.numetu%TYPE,
                         nf FORMATIONS.numf%TYPE);
END gestion_etudiants;
/
```

Paquetages: Exemple1

- **Exemple**
  - ✓ Corps

```
CREATE PACKAGE BODY gestion_etudiants IS

   CURSOR C_inscrits RETURN t_inscriptions IS
      SELECT * FROM INSCRIPTIONS ORDER BY datei;

   PROCEDURE enregistrement(ne ETUDIANTS.numetu%TYPE,
                            nm ETUDIANTS.nometu%TYPE) IS
      BEGIN
      INSERT INTO ETUDIANTS VALUES (ne, nm, 0);
      END enregistrement;

   PROCEDURE exclusion(ne ETUDIANTS.numetu%TYPE) IS
      BEGIN
      DELETE FROM INSCRIPTIONS WHERE numetu = ne;
      DELETE FROM ETUDIANTS WHERE numetu = ne;
      END exclusion;

   PROCEDURE inscription(ne ETUDIANTS.numetu%TYPE,
                         nf FORMATIONS.numf%TYPE) IS
      BEGIN
      INSERT INTO INSCRIPTIONS VALUES (nf, ne, SYSDATE);
      END inscription;

END gestion_etudiants;
/
```

Paquetages: Exemple2

```
CREATE OR REPLACE PACKAGE gestionPilote AS
                        -- Objets
    nbreVol integer;
    CURSOR accesPilotes(salPilote  pilote.sal%TYPE)
    RETURN pilote%ROWTYPE;
                        -- FONCTION
    FUNCTION nbreVolAvion(TypeAvion IN Avion.typeAv%TYPE)
    RETURN integer;
                        -- Procédure
    RPOCEDURE AccorderPrime(prime float);
  END gestionPilote;
/
```

Paquetages: Exemple2

```
CREATE OR REPLACE PACKAGE BODY gestionPilote AS
        -- curseur
    CURSOR accesPilote(salPilote pilote.sal%TYPE)
    RETURN pilote%ROWTYPE
    IS SELECT * FROM pilote WHERE sal = salPilote;
        -- fonction
     FUNCTION nbreVolAvion(TypeAvion IN Avion.typeAv%TYPE)
      RETURN integer IS
                Begin
                    select count(*) into nbreVol from avion a, vol v
                    where a.navion = v.navion and typeav = typeAvion;
                    return nbreVol;
                 End nbreVolAvion;
        -- procédure
      PROCEDURE accorderPrime(prime float) IS
            Begin
            UPDATE pilote SET sal=sal+prime  WHERE npilote in select npilote from vol group by npilote having
   count(*) >=4);
End Accorder_prime;
END gestionPilote;
/
```

Référence au contenu d'un package

- seuls les objets et sous programmes publics peuvent être référencés depuis l'extérieur
- syntaxe:

```
nomPackage.nomObjet
nomPackage.nomSousProgramme(…)
```

```
set serveroutput on;
Accept v_sal prompt'Entrer un salaire'
 BEGIN
        for enreg in gestionPilote.accesPilotes(&v_sal) LOOP
          DBMS-OUTPUT.PUT_LINE(enreg.plnom);
          end loop;
 End;
 /
```

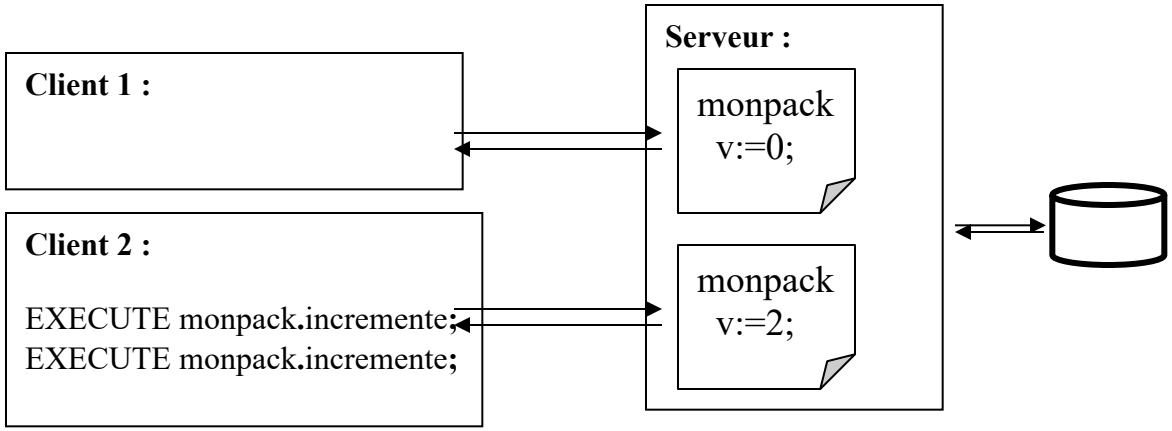
```
Select gestionPilotes.nbreVolAvion('A300') from dual;
Execute gestionPilotes.accorder_Prime(200);
```

Paquetages

❑ Persistance des variables

```
CREATE OR REPLACE PACKAGE monpack IS
   v INTEGER:=0;
   PROCEDURE incremente;
END monpack ;
/

CREATE OR REPLACE PACKAGE BODY monpack IS
   PROCEDURE incremente IS
      BEGIN
      v:=v+1;
      END incremente;
END monpack ;
/
```



Les Déclencheurs

- Un déclencheur est un bloc PL/SQL associé à une vue ou une table, qui s'exécutera lorsqu'une instruction du langage de manipulation de données (DML) sera exécutée
- Un déclencheur définit une action qui doit s'éxécuter quand survient un événement dans la base de données.

  Il peut servir a :

  - ✓ ajouter des contraintes sur les valeurs des attributs d'une table,
  - ✓ enregistrer des changements (suivi)
  - ✓ ajouter des règles de gestion,...
- Les déclencheurs (*triggers) existent depuis la version 6 d'Oracle. Ils sont compilables depuis* la version 7.3 (auparavant, ils étaient évalués lors de l'exécution).
- À la différence des sous-programmes, l'exécution d'un déclencheur n'est pas explicitement opérée par une commande ou dans un programme, c'est l'événement de mise à jour de la table (ou de la vue) qui exécute automatiquement le code programmé dans le déclencheur. On dit que le déclencheur « se déclenche » (l'anglais le traduit mieux : *fired trigger).*

Déclencheurs : définition

- Quand et comment les utiliser

Un déclencheur peut être déclenché :

- a la création, suppression ou modification d'un objet (CREATE, DROP, ALTER),
- a la connexion, déconnexion d'un utilisateur,
- démarrage ou arrêt d'une instance...

- Durée d'activité

L'action associée à un déclencheur est un bloc PL/SQL enregistré dans la base.

Un déclencheur est opérationnel jusqu'a la suppression de la table a laquelle il est lié.

- Le traitement mentionné dans un trigger peut s'effectuer :
  - pour chaque ligne concernée par l'événement,(trigger de niveau ligne)
  - une seule fois pour l'ensemble des lignes concernées par l'événement(trigger de niveau table).
- Ligne

FOR EACH ROW : trigger de niveau ligne, sinon de niveau table.

WHEN condition : pour chaque ligne, trigger déclenché si vraie.

À quoi sert un déclencheur ?

❑ Un déclencheur permet de :

- Programmer toutes les règles de gestion qui n'ont pas pu être mises en place par des contraintes au niveau des tables. Par exemple, la condition : *une compagnie ne fait voler un pilote que s'il a totalisé plus de 60 heures de vol dans les 2 derniers mois sur le type d'appareil du vol en question, ne pourra pas être programmée par une contrainte et nécessitera* l'utilisation d'un déclencheur.
- Déporter des contraintes au niveau du serveur et alléger ainsi la programmation client.
- Renforcer des aspects de sécurité et d'audit.
- Programmer l'intégrité référentielle et la réplication dans des architectures distribuées avec l'utilisation de liens de bases de données (*database links).*

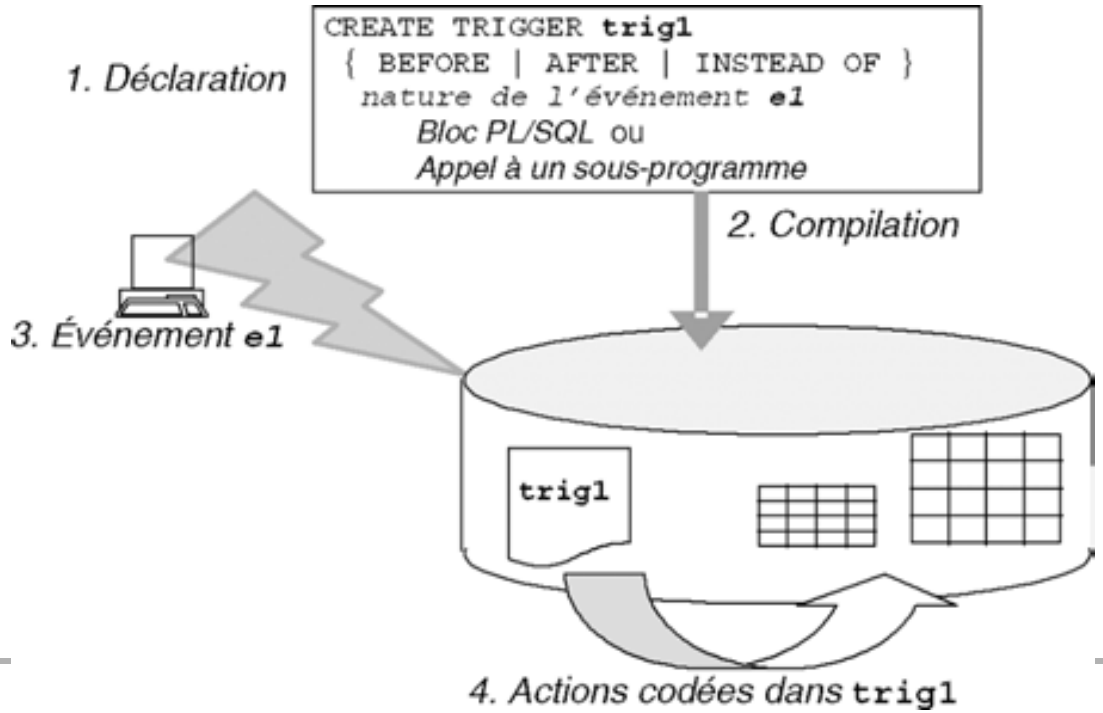
Déclencheurs: Généralités

- Les événements déclencheurs peuvent être :
  - ✓ une instruction INSERT, UPDATE, ou DELETE sur une table (ou une vue). On parle de déclencheurs LMD ;
  - ✓ une instruction CREATE, ALTER, ou DROP sur un objet (table, index, séquence, etc.). On parle de déclencheurs LDD ;
  - ✓ le démarrage ou l'arrêt de la base (*startup ou shutdown), une erreur spécifique (NO_DATA*_FOUND, DUP_VAL_ON_INDEX, etc.), une connexion ou une déconnexion d'un utilisateur.

  On parle de déclencheurs d'instances.

Mécanisme général

- La figure suivante illustre les étapes à suivre pour mettre en oeuvre un déclencheur. Il faut d'abord le coder (comme un sous-programme), puis le compiler (il sera stocké ainsi en base).
- Par la suite, au cours du temps, et si le déclencheur est actif (nous verrons qu'il est possible de désactiver un déclencheur même s'il est compilé), chaque événement (qui caractérise le déclencheur) aura pour conséquence son exécution.



Syntaxe

- Pour pouvoir créer un déclencheur dans votre schéma, vous devez disposer du privilège CREATE TRIGGER
- Un déclencheur est composé de trois parties : la description de l'événement traqué, une éventuelle restriction (condition) et la description de l'action à réaliser lorsque l'événement se produit. La syntaxe de création d'un déclencheur est la suivante :

Syntaxe

CREATE [OR REPLACE] TRIGGER [*schéma.] nomDéclencheur*
{ BEFORE | AFTER | INSTEAD OF }
{ { DELETE | INSERT | UPDATE [OF *col1 [,col2]…] }*
[OR { DELETE | INSERT | UPDATE [OF *col1 [,col2]…] }]…*
ON { [*schéma.] nomTable* | *nomVue }*
[REFERENCING
{ OLD [AS] *nomVieux* | *NEW [AS] nomNew* | *PARENT [AS] nomParent }*
[OLD [AS] *nomVieux* | *NEW [AS] nomNew* | *PARENT [AS] nomParent]…]*
[FOR EACH ROW] }
|
{ événementBase [OR événementBase]… |
actionStructureBase [OR actionStructureBase]… }
ON { [*schéma.] SCHEMA* | *DATABASE } }*
[WHEN (*condition)]*
{ Bloc PL/SQL (variables BEGIN instructions END ;)
| CALL *nomSousProgramme(paramètres) }*

Syntaxe

Les options de cette commande sont les suivantes :

- BEFORE | AFTER | INSTEAD OF précise la chronologie entre l'action à réaliser par le déclencheur LMD et la réalisation de l'événement (exemple BEFORE INSERT programmera l'exécution du déclencheur avant de réaliser l'insertion).
- DELETE | INSERT | UPDATE précise la nature de l'événement pour les déclencheurs LMD.
- ON {[schéma.] nomTable | nomVue} spécifie la table, ou la vue, associée au déclencheur LMD.
- REFERENCING permet de renommer des variables.
- FOR EACH ROW différencie les déclencheurs LMD au niveau ligne ou au niveau état.
- *événementBase identifie la nature d'un déclencheur d'instance (STARTUP ou SHUTDOWN* pour exécuter le déclencheur au démarrage ou à l'arrêt de la base), d'un déclencheur d'erreurs (SERVERERROR ou SUSPEND pour exécuter le déclencheur dans le cas d'une erreur particulière ou quand une transaction est suspendue) ou d'un déclencheur de connexion (LOGON ou LOGOFF pour exécuter le déclencheur lors de la connexion ou de la déconnexion à la base).
- *actionStructureBase spécifie la nature d'un déclencheur LDD (CREATE, ALTER,* DROP, etc. pour exécuter par exemple le déclencheur lors de la création, la modification ou la suppression d'un objet de la base).
- ON {[schéma.]SCHEMA | DATABASE}} précise le champ d'application du déclencheur (de type LDD, erreur ou connexion). Utilisez DATABASE pour les déclencheurs qui s'exécutent pour quiconque commence l'événement, ou SCHEMA pour les déclencheurs qui ne doivent s'exécuter que dans le schéma courant.
- WHEN conditionne l'exécution du déclencheur.

Déclencheurs LMD

Nature de l'événement	État (*statement trigger*) sans FOR EACH ROW	Ligne (*row trigger*) avec FOR EACH ROW
BEFORE	Exécution une fois avant la mise à jour	Exécution avant chaque ligne mise à jour
AFTER	Exécution une fois après la mise à jour	Exécution après chaque ligne mise à jour

Déclencheurs de lignes (row triggers)

Un déclencheur de lignes est déclaré avec la directive FOR EACH ROW. Ce n'est que dans ce type de déclencheur qu'on a accès aux anciennes valeurs et aux nouvelles valeurs des colonnes de la ligne affectée par la mise à jour prévue par l'événement.

Quand et quoi...

- Quand

- BEFORE et AFTER

- pour trigger de niveau table(statement) : déclenché avant ou après l'événement

- pour trigger de niveau ligne : exécute avant ou après la modification de CHAQUE ligne concernée

- INSTEAD OF : spécifique aux vues.

- Quoi

corpsTrigger : bloc PL/SQL effectue quand le trigger est déclenché En plus :

IF INSERTING THEN ... END IF;

IF DELETING THEN ... END IF;

IF UPDATING THEN ... END IF;

Valeurs des attributs

- Quelles valeurs sont testées?

Dans la clause WHERE ou dans le corps, on peut se référer a la valeur d'un attribut avant ou après que soit effectuée la modification déclenchant le trigger :

:OLD.nomAttribut : la valeur avant la transaction UPDATE ou DELETE

:NEW.nomAttribut : la valeur après la transaction UPDATE ou INSERT

Tables exemple

- Une table contenant des etudiants (numero, nom, prenom...) avec leur moyenne.
- Des tables reportEtudiant (pour le suivi), Alerte.

Table etudiant			
NumEtudiant	Nom	Prenom	Moyenne
123	DUPONT	JULES	4
234	DUPOND	ALFRED	5
567	DURAND	JULIE	14
598	DURANT	ALFREDINE	16

Exemple de déclencheur de niveau table

```
CREATE TRIGGER triggerEtudiant
    BEFORE INSERT or UPDATE ON etudiant
BEGIN
    DBMS OUTPUT.ENABLE(10000);
    IF INSERTING THEN
        IF USER ! = 'SCOTT'
            THEN RAISE_APPLICATION_ERROR
            (-20001,'Utilisateur non autorise.');
        END IF;
    END IF;
    IF UPDATING THEN
        DBMS OUTPUT.PUT LINE('Mise a jour de l "étudiant.');
    END IF;
END;
```

Exemple de déclencheur de niveau ligne

```
CREATE TRIGGER auditEtudiant
    AFTER UPDATE OR DELETE ON etudiant
    FOR EACH ROW
BEGIN
    INSERT INTO reportEtudiant
            VALUES (SYSDATE, :OLD.NumEtud, :OLD.nom,
                    :OLD.prenom, :OLD:moyenneEtud);
END;
```

Exemple de déclencheur de niveau ligne avec clause WHERE

```
CREATE TRIGGER moyenneMax
    AFTER UPDATE ON etudiant
    FOR EACH ROW
    WHEN NEW.moyenneEtud > 2*OLD.moyenneEtud
BEGIN
    INSERT INTO alerte(datealerte, numEtudiant, message)
          VALUES ( SYSDATE, :OLD.nom,
                  'MOYENNE a plus que double');
END;
```

Exemple: for each row

```
Create or replace trigger print_salary_changes
    Before update on Emp_tab
    For each row
    When (new.empno > 0)
Declare
    sal_diff number;
Begin
    sal_diff := :new.sal - :old.sal;
    Dbms_output.put('ancienne:'||:old.sal || 'nouvelle' || :new.sal|| 'difference'||sal_diff);
End;
/
```

Ce trigger est lorsque la table emp_tab est mise à jour. Pour chaque modification(lignes mises à jour), le trigger calcule puis affiche respectivement l'ancien salaire, le nouveau salaire est la différence entre les deux salaires la condition précisée (when new.empno>0) restreint le déclenchement du trigger(exécuté uniquement si empno > 0)

Table mutante

- Il faut savoir qu'il n'est pas permis de consulter ou modifier une table mutante. Cette restriction préservera le trigger ligne de lire des données inconsistantes. Donc le trigger ne devrais jamais accéder à la table sur la quelle il porte, autrement que par OLD ou NEW.
- Exemple:

```
Create or replace trigger emp_count
    After delete on Emp_tab
    For each row
    When (new.empno > 0)
Declare
    n integer;
Begin
    select count(*) into n from Emp_tab;
    Dbms_output.put('il y'a ' ||n|' employés ');
End;
/
```

Dans cet exemple la table mutante (celle qui est modifiée par DELETE) est Emp_tab. Or une clause SELECT porte justement sur cette table. Le risque est d'obtenir un résultat complétement incohérent. Cette consultation est donc interdite

:OLD & :NEW

- Dans trigger ligne il est possible de modifier les éléments NEW.
- OLD & NEW sont disponible dans BEFORE & AFTER ROW TRIGGERS
- La valeur de NEW peut être assigné dans un TRIGGER ROW BEFOR mais pas dans un trigger AFTER
- Si vous spécifié un TRIGGER ROW que sont déclenchement dépend d'une colonne précise dans une table, il est tout à fait logique que les éléments OLD et NEW ne porte que sur cette colonne
- Parmi les limites des triggers

- DDL statement are not allowed in the body of a trigger.
- No transaction control statement are allowed in the trigger: ROLLBACK, COMMIT, SAVEPOINT cannot be used

Ordre

Ordre d'activation :

Si plusieurs triggers sur une même table, l'ordre d'activation est :

1 BEFORE niveau table

2 BEFORE niveau ligne, aussi souvent que de lignes concernées

3 AFTER niveau ligne, aussi souvent que de lignes concernées

4 AFTER niveau table

Code PL/SQL	Commentaires
CREATE TRIGGER TrigInsQualif **BEFORE INSERT ON Qualifications** FOR EACH ROW	Déclaration de l'événement déclencheur.
DECLARE v_compteur Pilote.nbHVol%TYPE; v_nom Pilote.nom%TYPE;	Déclaration des variables locales.

BEGIN
SELECT nbQualif, nom INTO v_compteur, v_nom FROM Pilote WHERE brevet =:**NEW.brevet** ;
IF v_compteur < 3 THEN
UPDATE Pilote SET nbQualif = nbQualif + 1 WHERE brevet =:**NEW.brevet** ;
ELSE RAISE_APPLICATION_ERROR (-20100, 'Le pilote ' || v_nom || ' a déjà 3 qualifications!');
END IF;
END;
/
	Corps du déclencheur.
	Extraction et mise à jour du pilote concerné par la qualification.
	
	Renvoi d'une erreur utilisateur.

Code PL/SQL	Commentaires
CREATE TRIGGER TrigInsQualif BEFORE INSERT ON Qualifications FOR EACH ROW	Déclaration de l'événement Déclencheur
DECLARE v_compteur Pilote.nbHVol%TYPE; v_nom Pilote.nom%TYPE;	Déclaration des variables locales.

BEGIN
SELECT nbQualif, nom INTO v_compteur, v_nom
FROM Pilote WHERE brevet = :NEW.brevet;
IF v_compteur < 3 THEN
UPDATE Pilote SET nbQualif = nbQualif + 1
WHERE brevet = :NEW.brevet;
ELSE
RAISE_APPLICATION_ERROR(-20100, 'Le pilote ' ‖
:NEW.brevet ‖ ' a déjà 3 qualifications!');
END IF;
EXCEPTION
WHEN NO_DATA_FOUND THEN
RAISE_APPLICATION_ERROR(-20101, 'Pas de pilote de code brevet ' ‖ :NEW.brevet); WHEN OTHERS THEN
RAISE;
END;
/
	Corps du déclencheur.
	Extraction et mise à jour
	du pilote concerné par la qualification.
	
	Renvoi d'une erreur utilisateur.
	
	Si erreur au SELECT.
	
	Retour de l'erreur courante.

Quand utiliser la directive :OLD ?

- Chaque enregistrement qui tente d'être supprimé d'une table qui inclut un déclencheur de type DELETE FOR EACH ROW, est désigné par :OLD au niveau du code du déclencheur. L'accès aux colonnes de ce pseudo-enregistrement dans le corps du déclencheur se fait par la notation pointée.

SQL dynamique

- **Principe**
  - ✓ Construction de requêtes SQL dynamique dans un bloc PL/SQL
  - ✓ Exemple : « mise à jour d'une table dont le nom est passé en paramètre »
  - ✓ Permet d 'utiliser les ordres du LDD (CREATE, DROP, ALTER) qui altèrent le schéma de la base
- **Structure**

```
EXECUTE IMMEDIATE requête [ INTO var1,…,varn ]
   [ USING [IN|OUT|IN OUT] parametre1
        [, [IN|OUT|IN OUT] parametre2
        …
        [, [IN|OUT|IN OUT] parametren ]…]]
```

SQL dynamique

❑ Exemple d'une requête paramétrée

```
DECLARE
  req VARCHAR2(50);
BEGIN
  req:= 'INSERT INTO ETUDIANTS VALUES (:p1,:p2,0)';
  EXECUTE IMMEDIATE req USING 3, 'Polo';
END;
/
```

❑ Exemple de stockage du résultat

```
DECLARE
  req VARCHAR2(50);
  etud ETUDIANTS%ROWTYPE;
BEGIN
  req:= 'SELECT * FROM ETUDIANTS WHERE numetu = :pid';
  EXECUTE IMMEDIATE req INTO etud USING 3;
END;
/
```

SQL Dynamique exemples

```
CREATE or REPLACE FUNCTION sql_dyn(v_table IN VARCHAR2,
    clauseWhere IN Varchar2) RETURN integer IS V_requete
```

SQL dynamique

- **Exemple d'une requête sans variable**

```
BEGIN
EXECUTE IMMEDIATE 'DELETE FROM ETUDIANTS WHERE numetu = :pid' USING 3;
END;
/
```

- **Exemple d 'une requête du LDD**

```
BEGIN
EXECUTE IMMEDIATE 'DROP TABLE ETUDIANTS';
END;
/
```