Plan

- Introduction
- Bloc PL/SQL
- Déclaration des variables
- Structure de contrôle
- Curseurs
- Exceptions
- Transactions
- Fonctions et procédures
- Packages
- Triggers

Exceptions

- Définition
- Exceptions prédéfinies
- Exceptions définies par l'utilisateur

Exceptions

- PL/SQL offre au développeur un mécanisme de gestion des exceptions. Il permet de préciser la logique du traitement des erreurs survenues dans l'un de ses blocs.
- Il s'agit donc d'un point clé dans l'efficacité du langage qui permettra de protéger l'intégrité du système.

➢ C'est un mécanisme permettant de dérouter l'exécution normale du bloc vers la section **Exception**.

➢ Il existe deux types d'exceptions:

- **Exceptions internes:** générées par le moteur du système (division par zéro, connexion non établie, table inexistante, privilèges insuffisants, mémoire saturée, espace disque insuffisant, ...).
- **Exception externes:** générées par le programmeur.

Exceptions

- **Exceptions définies par ORACLE**
  - ✓ Nommées par Oracle
  - ✓ Exemples : **NO_DATA_FOUND, TOO_MANY_ROWS, …**
  - ✓ Se déclenchent automatiquement
  - ✓ Nécessitent de prévoir la prise en compte de l'erreur dans la section **EXCEPTION**
- **Exceptions définies par l'utilisateur :**
  - ✓ Nommés par le programmeur
  - ✓ Arrêt de l'exécution du bloc
  - ✓ Sont déclenchées par une instruction du programme soit automatiquement (**PRAGMA**) ou manuellement par : **Raise**
  - ✓ Nécessitent de prévoir la prise en compte de l'erreur dans la section **EXCEPTION**.
- Oracle fournit les fonctions **SQLCODE** et **SQLERRM**, qui renvoient respectivement le **code** et le **message** d'erreur correspondant à l'exception.

Exceptions

Exceptions prédéfinies

- **ZERO_DIVIDE** : tentative de division par zéro.
- **TOO_MANY_ROWS** : la commande ***SELECT INTO*** retourne plus d'une ligne.
- **NO_DATA_FOUND** : déclenchée si la commande ***SELECT INTO*** ne retourne aucune ligne ou si l'on fait référence à un enregistrement, non initialisé, d'un tableau PL/SQL.
- **LOGIN_DENIED** : connexion à la base est échouée, car le nom d'utilisateur ou le mot de passe est invalide.
- **CURSOR_ALREADY_OPEN** : tentative d'ouverture d'un curseur déjà ouvert.
- **INVALID_NUMBER** : échec de la conversion d'une chaîne de caractères en numérique.
- **INVALID_CURSOR** : opération incorrecte sur un curseur, comme par exemple la fermeture d'un curseur qui n'a pas été ouvert.
- **PROGRAM_ERROR** : problème général dû au PL/SQL.
- **TIMEOUT_ON_RESOURCE** : dépassement du temps dans l'attente de libération des ressources (lié aux paramètres de la base).

Exceptions

Exceptions prédéfinies : Exemple

```
DECLARE
    SAL empl.salaire%type ;
BEGIN
    SELECT salaire into SAL FROM employe WHERE ville ='RABAT';
    ……….
EXCEPTION
    WHEN TOO_MANY_ROWS THEN traitements ;
    WHEN NO_DATA_FOUND THEN traitements ;
    WHEN OTHERS THEN :
            dbms_output.put_line ('Code d'erreur :' || SQLCODE) ;
            dbms_output.put_line ('Message d'erreur : ' || SQLERRM) ;
END ;
```

Exceptions

- **Fonctionnement**: lorsqu'une exception est détectée, il y a:
  - Arrêt de l'exécution du bloc
  - Branchement sur la section exception
  - Parcours des clauses **WHEN** jusqu'au bon choix
  - Exécution des instructions associées
  - Une fois le traitement de l'erreur est terminé, c'est le bloc suivant qui est effectué.

Exceptions

Exceptions prédéfinies (2): Exemples avec le code oracle

- **Acces_INTO_NULL**: on a tenté d'affecter une valeur à un objet non initialisé ***ORA-6530***
- **CASE_NOT_FOUND**: il n'y a pas de choix ***WHEN*** correspondant dans une instruction ***CASE*** et l'option ***ELSE*** n'a pas été définie: ***ORA-6592.***
- **INVALID_NUMBER**: PL/SQL exécute un ordre ***SQL*** qui ne parvient pas à convertir une chaîne de caractère en nombre: ***ORA-01722***.
- **NO_DATA_FOUND**: cette exception est déclenchée en trois cas:
  - Un ordre ***SELECT INTO*** qui ne ramène aucun enregistrement
  - On référence une ligne non initialisé d'une table PL/SQL.
  - On tente de lire après la fin d'un fichier avec le package ***UTL_FILE ORA-01403***
- **TOO_MANY_ROWS**: ***SELECT INTO*** a ramené plus d'une ligne ***ORA-01422***

Exceptions

Exceptions prédéfinies (3) : Exercice

1) Ecrire un bloc PL/SQL qui:

- Lit les deux entiers A et B
- Calcule et affiche la division de A par B

1) Exécuter le bloc pour B=0. que remarque-t-on?

2) Modifier le bloc pour traiter l'exception levée en affichant un message approprié.

Exceptions

Exceptions définies par l'utilisateur

- Déclaration d'une variable de type **EXCEPTION** dans la section **DECLARE**

  Nom_exception **EXCEPTION**

- Définition du traitement associé

  WHEN Nom_exception **THEN** traitement

- Déclenchement
  - Soit associer à cette erreur un code *ORACLE*, elle sera levée automatiquement:
    - **PRAGMA EXCEPTION_INIT** (Nom_exception, Code_erreur)
  - Soit lever manuellement l'exception
    - **RAISE** Nom_exception

Exceptions

Exceptions définies par l'utilisateur (2)

```
DECLARE
    ……..
    Nom_exception EXCEPTION
BEGIN
    ……..--Instructions
If (condition_erreur) THEN RAISE Nom_exception;
    ……
EXCEPTION
    WHEN Nom_exception THEN traitements;
END;
```

Remarques:

- On sort du bloc après l'exécution du traitement d'erreur.
- Les règles de visibilité des exceptions sont les mêmes que celles des variables

Exceptions

Exemple

```
DECLARE
  CURSOR emp_rabat IS SELECT nomemp, sal FROM employe WHERE  ville = 'Rabat' ;
  nom employe.nomemp%TYPE ;
  salaire employe.sal%TYPE ;
  ERR_salaire EXCEPTION
BEGIN
  OPEN emp_rabat ;
    FETCH emp_rabat INTO nom, salaire; -- On récupère le premier enregistrement
    WHILE emp_rabat % found LOOP -- S'il y a un enregistrement récupéré
              If salaire IS NULL THEN
                        RAISE ERR_salaire
             .....                          --Traitement de l'enregistrement récupéré
      FETCH emp_rabat INTO nom, salaire;-- On récupère l'enregistrement suivant
    END  LOOP;
 CLOSE emp_rabat ;
EXCEPTION
    WHEN ERR_salaire THEN INSERT INTO temp (nomempl, 'Salaire non défini');
    WHEN NO_DATA_FOUND THEN INSERT INTO temp (nomempl,'non trouvé');
END;
```

Exceptions

Exception ‘others’

Le module ***EXCEPTION*** permet également de traiter un code erreur qui n’est traité par aucune des exceptions. Le nom générique de cette exception (prédéfinie) est **OTHERS**.

- **Syntaxe :**

```
EXCEPTION
WHEN exception1 THEN
  traitement1;
WHEN exception2 THEN
  traitement2;
WHEN OTHERS THEN
  traitement3;
```

- Deux fonctions permettent de récupérer des informations sur l’erreur Oracle:
- **Sqlcode**: retourne une valeur numérique: numéro de l’erreur
- **Sqlerrm**: renvoie le libellé de l’erreur

Exceptions

Exemple complet

```
DECLARE
   salaire employe.sal%TYPE.
   SAL_nulle EXCEPTION;
   code number;
   message varchar2(50);
BEGIN
   SELECT sal INTO salaire from employe;
    If salaire=0 THEN RAISE SAL_nulle; END IF;
EXCEPTION
   WHEN SAL_nulle THEN
   ….-- générer une erreur de salaire
   WHEN TOO_MANY_ROWS THEN
   …. -- générer trop de lignes
   WHEN NO_DATA_FOUND THEN
   ….-- générer erreur pas de lignes
   WHEN OTHERS THEN
   -- générer toutes les autres erreurs
   code:=sqlcode;
   message:=sqlerrm;
   Dbms_out_put_put_line('erreur'|| code ||message)
END;
```

Plan

- Introduction
- Bloc PL/SQL
- Déclaration des variables
- Structure de contrôle
- Curseurs
- Exceptions
- Transactions
- Fonctions et procédures
- Packages
- Triggers

Transactions

Introduction

- Une transaction est un bloc d'instructions LMD permettant de passer la BD d'un état cohérent à un autre état cohérent.
- Lors d'une transaction, si un problème matériel ou logiciel survient alors aucune des instructions se trouvant dans cette transaction n'est effectuée, quel que soit l'emplacement de la transaction où l'erreur est intervenue.
- Sous Oracle, la majorité des transactions sont programmées en PL/SQL.

Exemple: transfert d'un compte épargne vers un compte courant

Supposons qu'après une panne, le compte épargne a été débité de la somme de 1000 DH sans que le compte courant ait été crédité de ce même montant. Le client dans ce cas ne sera pas content des services de sa banque.

Solution: utilisation du mécanisme transactionnel va empêcher cet épisode fâcheux, en invalidant les opérations effectuées depuis le début de la transaction en cas d'une panne survenue au cours de cette transaction.

Transactions

Caractéristiques

Une transaction assure :

- Cohérence des états: passage d'un état cohérent de la base à un autre état cohérent.
- Atomicité des instructions : sont considérées comme une seule opération, principe du tout ou rien.
- Durabilité des opérations: les mises à jour perdurent même si une panne se produit après la transaction.
- Isolation des transactions entre elles.

Il existe d'autres transactions particulières, qui sont constituées par :

- Ordre SQL du LDD: *CREATE*, *ALTER*, *DROP…*
- Ordre SQL du LCD *: GRANT, REVO*KE.

Transactions

Début et fin d'une transaction

➢Pas d'ordre SQL ou PL/SQL qui désigne le début d'une transaction. Même le **BEGIN** d'un programme PL/SQL ne marque pas forcément son début.

➢En général, une transaction commence à la première commande SQL rencontrée ou à partir de la fin de la transaction précédente.

➢Une transaction se termine explicitement par les instructions SQL **ROLLBACK** ou **COMMIT**.

➢Une transaction se termine implicitement :

- A la première commande SQL du **LCD** ou du **LDD** rencontrée ;
- A la fin normale d'une session utilisateur avec déconnexion ;
- A la fin anormale d'une session utilisateur sans déconnexion.

Transactions

Ordre COMMIT

- Utilisé pour terminer la transaction en cours en appliquant de façon permanente tous les changements.
- Il est recommandé d'utiliser **COMMIT** après chaque ordre d'instruction LMD.

Exemple:

```
SQL> DELETE FROM emp;
11 ligne(s) supprimée(s).
SQL> COMMIT;
```

Transactions

Ordre ROLLBACK

- Annule la transaction et restitue l'état du serveur avant le changement.

Exemple:

```
SQL> DELETE FROM emp;
11 ligne(s) supprimée(s).
SQL> ROLLBACK;
SQL> SELECT COUNT(*) FROM emp;
COUNT(*)
---------
```


ORACLE

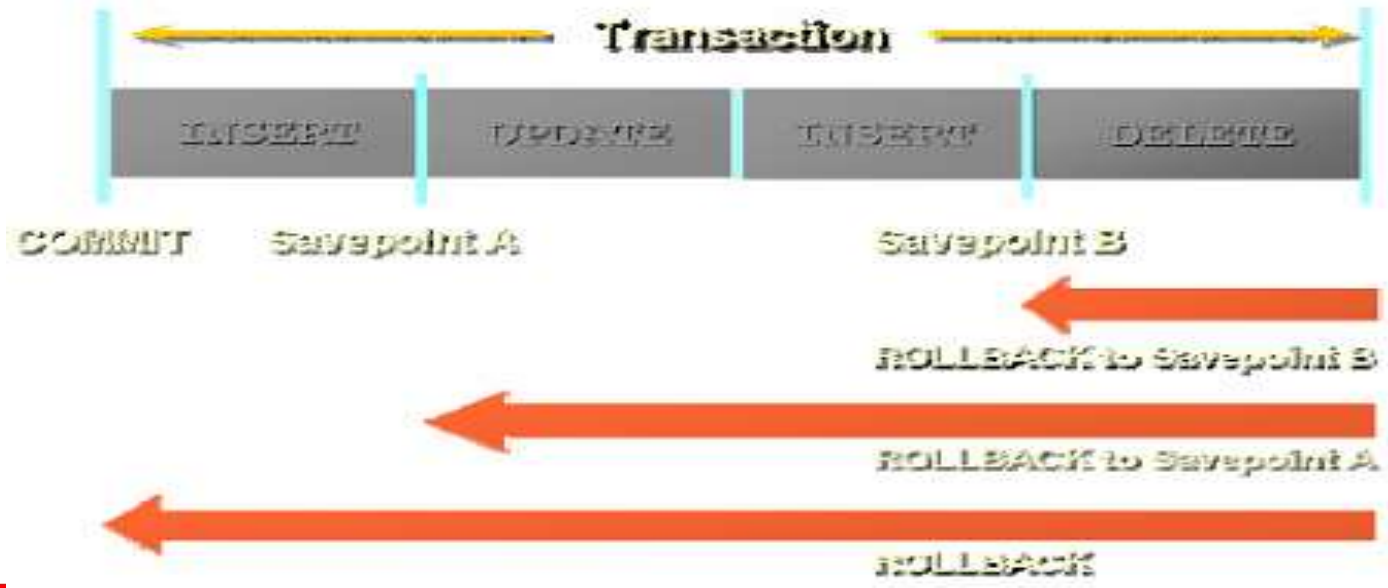
Transactions

Ordre SAVEPOINT est utilisé pour:

- **Annuler une partie de la transaction.**
- **Effectuer des ROLLBACK réduits.**

➔**Ajouter des marqueurs de sauvegarde SAVEPOINT;**

Syntaxe: SAVEPOINT nom;



Transactions

Validité de la transaction en fonction des événements possibles

Événement	Validité
COMMIT Commande SQL (LDD ou LCD) Fin normale d'une session.	Transaction validée
ROLLBACK Fin anormale d'une session.	Transaction non validée

Exemple:

```
commit;
begin
insert into table1 values (…);
end;
/
select * from table1;
```

- Si l'on exécute ce bloc dans SQL*Plus, puis on se déconnecte en cliquant sur "**fermer**" et on se reconnecte, on remarque que le tuple n'est pas stocké dans la table.
- Mais si l'on relance le bloc et on sorte proprement de SQL*Plus avec exit (quit) et on se reconnecte, on voit le tuple est stocké désormais dans la table.

Transactions

Contrôle des transactions

Parfois il est nécessaire de découper une transaction en introduisant des points de validation (***savepoints)*** *qui permettent d'annuler tout ou partie des opérations d'une transaction.*

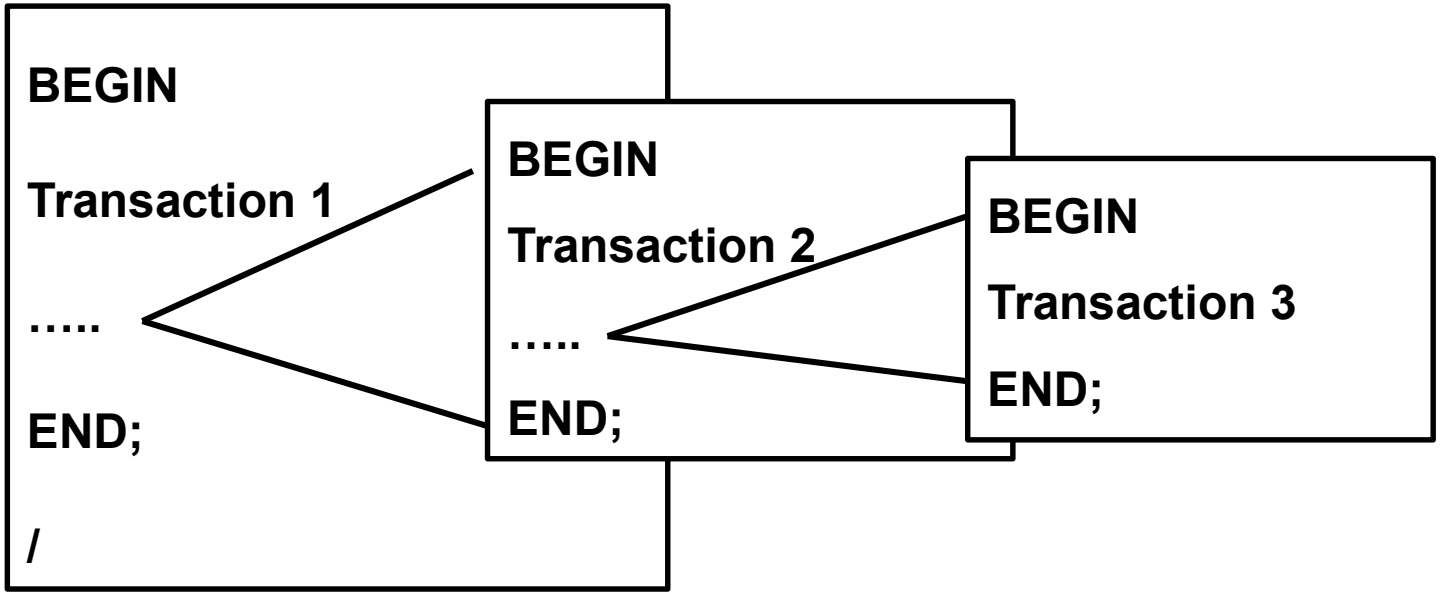
Code PL/SQL	Commentaires
BEGIN INSERT INTO Compagnie VALUES ('C2',2, 'Place Brassens', 'Blagnac', 'Easy Jet');	Première partie de la transaction.
SAVEPOINT P1; UPDATE Compagnie SET nrue = 125 WHERE comp = 'AF'; UPDATE Compagnie SET ville = 'Castanet' WHERE comp = 'C1';	Deuxième partie de la transaction.
SAVEPOINT P2; DELETE FROM Compagnie WHERE comp = 'C1';	Troisième partie de la transaction.
-- ROLLBACK TO P1;	2ère et 3ème partie à invalider.
-- **ROLLBACK TO P2;**	3ème partie à invalider.
ROLLBACK;	Tout à invalider
COMMIT; **END;**	Valide la ou les sous-parties.

Transactions

Transactions imbriquées

➢Il est possible de programmer plusieurs transactions qui se déroulent dans des blocs imbriqués.

➢ Les mécanismes de cohérence, d'atomicité, de durabilité et d'isolation sont aussi respectés.



Procédures et fonctions

- Introduction
- Utilité
- Structure
- Exemples

Procédures et fonctions

Introduction

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

Les procédures (fonctions) permettent de :

- Masquer la complexité du code SQL: simple appel de procédure avec passage de paramètres.
- Sécuriser l'accès aux données : accès seulement à certaines tables à travers les procédures.
- Mieux garantir l'intégrité des données: encapsulation des données par les procédures;
- Optimiser le code : les procédures sont compilées avant l'exécution du programme et elles sont exécutées immédiatement si elles sont dans la SGA (SYSTEM GLOBAL AREA). Une procédure peut être exécutée également par plusieurs utilisateurs.

Procédures

Création d'une procédure

Syntaxe générale :

procedure_general.sql

CREATE OR REPLACE PROCEDURE nom_procedure (liste d'arguments en **INPUT** ou **OUTPUT**) **IS**

déclaration de variables, de constantes, ou de curseurs

BEGIN

...

EXCEPTION

...

END;

/

Procédures

Au niveau de la liste d'arguments :

- Les arguments sont précédés des mots réservés **IN**, **OUT**, ou **IN OUT**,
- Ils sont séparés par des virgules,

IN : paramètre "entrant". il s'agit d'un paramètre dont la valeur est fournie à la procédure stockée. Cette valeur sera utilisée pendant la procédure.

OUT : paramètre "sortant", dont la valeur va être établie au cours de la procédure et qui pourra ensuite être utilisé en dehors de cette procédure.

IN OUT : un tel paramètre sera utilisé pendant la procédure, verra éventuellement sa valeur modifiée par celle-ci, et sera ensuite utilisable en dehors.

Remarques:

- Contrairement au PL/SQL, la zone de déclaration n'a pas besoin d'être précédé du mot réservé **DECLARE**. Elle se trouve entre le **IS** et **le BEGIN.**
- La dernière ligne de chaque procédure doit être composée du seul caractère **/** pour spécifier au moteur le déclenchement de son exécution.

Procédures

Exemple 2: procédure modifiant le salaire d'un employé.

Arguments : Identifiant de l'employée, Taux

modifie_salaire.sql

```
CREATE PROCEDURE modifie_salaire (id in number, taux in number) is
BEGIN
  UPDATE employe SET salaire=salaire*(1+taux) WHERE Id_emp= id;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
   RAISE_APPLICATION_ERROR (-20010,'Employé inconnu :'||to_char(id));
END;
/
```

Procédures

Compilation de la procédure « modifie_salaire »

Il faut compiler le fichier « sql » qui s'appelle ici **modifie_salaire.sql** (le nom du script, dans cet exemple, correspond à celui de la procédure; il faut donc écrire le nom du script sql après la commande **start**).

SQL> **start** modifie_salaire

Procedure created.

Appel de la procédure « modifie_salaire »

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

PL/SQL procedure successfully completed.

Procédures

Exemple 2: procédure modifiant le salaire de chaque employé en fonction de la valeur courante du salaire

```
create or replace procedure salaire is
CURSOR curs1 is select salaire from employe for update;
ancien_salaire number; nouveau_salaire number;
BEGIN
  OPEN curs1;
LOOP
  FETCH curs1 into ancien_salaire;
  EXIT WHEN curs1 % NOTFOUND;
  if ancien_salaire >=0 and ancien_salaire <= 50 then
        nouveau_salaire :=  4*ancien_salaire;
  elsif ancien_salaire >50 and ancien_salaire <= 100 then
     nouveau_salaire := 3*ancien_salaire;
   else nouveau_salaire = :ancien_salaire;
   end if;
update employe set salaire = nouveau_salaire where current of curs1;
 FETCH curs1 into ancien_salaire;
END LOOP;
CLOSE curs1;
END;/
```

Procédures

Modification d'une procédure

➢Si la base de données évolue, il faut recompiler les procédures existantes pour qu'elles tiennent compte de ces modifications.

➢La commande est la suivante:

ALTER PROCEDURE nom_procedure **COMPILE;**

Exemple:

ALTER PROCEDURE modifie_salaire **COMPILE;**

Suppression d'une procédure

DROP PROCEDURE nom_procedure;

Exemple:

DROP PROCEDURE modifie_salaire**;**

Plan

- Introduction
- Bloc PL/SQL
- Déclaration des variables
- Structure de contrôle
- Curseurs
- Exceptions
- Transactions
- Procédures
- Fonctions
- Packages
- Triggers

Fonctions

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

Syntaxe générale : ***Création d'une fonction***

fonction _general.sql

CREATE OR REPLACE FUNCTION nom_function (liste d'arguments en **INPUT** ou **OUTPUT**) **return** type_valeur_retour **IS**

déclaration de variables, de constantes, ou de curseurs

BEGIN

...

return (valeur);

EXCEPTION

...

END;

/

Fonctions

Exemple 1: fonction retournant la moyenne des salaires des employés par bureau (regroupé par responsable du bureau)

moyenne_salaire.sql

```
create or replace function moyenne_salaire (v_id_employe IN NUMBER)
return NUMBER
IS
   valeur NUMBER;
BEGIN
   select avg(salaire) into valeur from employe where empno = v_id_emp
groupe by empno ;
return (valeur);
END;
/
```

Fonctions

Exemple 2: fonction retournant la moyenne des prix avec gestion des exceptions.

```
CREATE OR REPLACE FUNCTION moyennePrix (dd IN DATE ) RETURN
NUMBER IS
  moy Tarif.prix%TYPE := 0; //Tarif est une table
  e EXCEPTION;
BEGIN
   SELECT AVG(prix) INTO moy FROM Tarif WHERE dateDebut = dd;
    IF moy IS NULL THEN RAISE e;
    END IF;
     RETURN moy;
EXCEPTION
     WHEN e THEN RETURN 0;
END;
/
```

Fonctions

Appel à une fonction

Appel de la fonction « moyenne_salaire **»**

```
SQL> Declare
        A emp.empno %type;
        S Number;
BEGIN
        select empno into A from emp;
        S:= moyenne_salaire (A);
dbms_output.put_line(S);
END;
/
```

Fonctions

Modification d'une fonction

➢Si la base de données évolue, il faut recompiler les fonctions existantes pour qu'elles tiennent compte de ces modifications.

➢La commande est la suivante:

ALTER FUNCTION nom_fonction **COMPILE;**

Exemple:

ALTER FUNCTION moyenne_salaire **COMPILE;**

Suppression d'une fonction

DROP FUNCTION nom_fonction;

Exemple:

DROP FUNCTION moyenne_salaire**;**

Packages

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

Packages

❑ *Création des spécifications du package*

CREATE OR REPLACE PACKAGE nom_package **IS**

Spécifications PL/SQL

END nom_package;

Les spécifications PL/SQL contiennent les déclarations des:

- Variables
- Enregistrements
- Curseurs
- Exceptions
- Fonctions
- Procédures
- …

Packages

Exemple

```
CREATE OR REPLACE PACKAGE gest_emp IS
-- Variables publiques
        v_sal EMP.sal%type;
-- Fonctions et procédures publiques
FUNCTION augmentation (numEmp IN EMP.empno%type, pourcent IN
NUMBER) Return NUMBER;
PROCEDURE test_Augmentation (numEMP IN EMP.empno%Type,
pourcent IN NUMBER);
END gest_emp;
```

Packages

❑ *Création du corps du package*

CREATE OR REPLACE PACKAGE BODY nom_package **IS**

Spécifications PL/SQL

END nom_package;

Les spécifications PL/SQL contiennent les déclarations des :

- Variables
- Enregistrements
- Curseurs
- Exceptions
- Fonctions
- Procédures
- …

Packages

Exemple:

```
CREATE OR REPLACE PACKAGE BODY gest_emp IS

E EMP%Rowtype; -- Variable privée

-- Fonction publique
FUNCTION augmentation (numEmp IN EMP.empno%type, pourcent IN
NUMBER) Return NUMBER IS
    salaire EMP.sal%Type;
BEGIN
  select sal into salaire from EMP where empno=numEMP;
  salaire := salaire* (poucent+1);--augmentation virtuelle
  v_sal:=salaire -- affectation de la variable globale publique
  return (salaire); -- retour de la valeur
END augmentation;
```

Packages

Suite de l'exemple

```
--Procédure privée
PROCEDURE affiche_Salaires IS
   CURSOR C_EMP IS select * from EMP;
    E C_EMP%Rowtype;
BEGIN
OPEN C_EMP; FETCH C_EMP INTO E;
While C_EMP % FOUND Loop
   dbms_output.put_Line ('Employé '|| E.ename || '  ' || E.sal);
   FETCH C_EMP INTO E;
End Loop;
Close C_EMP;
END affiche_Salaires;
```

Packages

Suite de l'exemple

```
--Procédure publique
PROCEDURE test_Augmentation (numEMP IN EMP.empno%Type,
pourcent IN NUMBER) IS
   salaire EMP.sal%type;
BEGIN
   select sal into salaire from EMP where empno=numEMP;
   salaire := salaire* (poucent+1);--augmentation virtuelle
   affiche_salaires; -- appel à la procédure privée
END test_Augmentation;
END gest_emp;
/
```

Packages

L'accès à un objet d'un package :

nom_package.nom_objet (liste des paramètres)

Exemple:

```
SQL> Declare
        A emp.sal%type;
Begin
        A:=gest_emp.augmentation (50, 0.1);
…
Dbms_output.put_Line(gest_emp.v_sal);
…
End;
 /
```

Plan

- Introduction
- Bloc PL/SQL
- Déclaration des variables
- Structure de contrôle
- Curseurs
- Exceptions
- Transactions
- Fonctions et procédures
- Packages
- Triggers (déclencheurs)

Triggers

- Définition
- Utilité
- Structure d’un trigger
- Exemples

Triggers

Définition

- Un trigger permet de spécifier les réactions du système d'information lorsque l'on « touche » à ses données. Concrètement il s'agit de définir un traitement (un bloc PL/SQL) à réaliser lorsqu'un événement survient.

➢ Déclencheur :

✓ **Action** ou ensemble d'actions déclenchée(s) **automatiquement** lorsqu'une **condition** se trouve satisfaite après l'apparition d'un **événement**.

➢ Un déclencheur est une règle **ECA**

✓ **Evénement** = mise à jour, suppression, insertion

✓ **Condition** = expression logique vraie ou fausse, optionnelle

✓ **Action** = procédure exécutée lorsque la condition est vérifiée.

Triggers

Remarques:

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

- **BEFORE INSERT**
- **AFTER INSERT**
- **BEFORE UPDATE**
- **AFTER UPDATE**
- **BEFORE DELETE**
- **AFTER DELETE**

- **Utilité des triggers**

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

  ✓ Propagation des MAJ dans une base de données distribuée.

  ✓ Sécurité

  ✓ …

Triggers

Structure d'un trigger

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

Triggers

- **Exemple** : Suppression d'un client

  → On supprime toutes ses commandes

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

Triggers

Structure d'un trigger

La structure de trigger est composée de trois parties :

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

Remarques :

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

ALTER TRIGGER nom_trigger **DISABLE** ;

ALTER TRIGGER nom_trigger **ENABLE** ;

Triggers

Remarque:

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

- ✓ **: old**.attribut : ancienne valeur (**DELETE | UPDATE**)
- ✓ **: new**.attribut : nouvelle valeur (**INSERT | UPDATE**)

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

```
CREATE OR REPLACE TRIGGER   suppression_ligne
BEFORE DELETE ON   produit
FOR EACH ROW
BEGIN
   INSERT INTO Journal VALUES (:old.codprod, :old.libelle, :old.prix, :old.qte);
END ;
```

Triggers

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

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

```
UPDATE PRODUIT
SET Prix = 15.99
WHERE Codprod = 10 ;
```

	Codprod	Prix	....
Ligne avant (OLD)	10	11	

	Codprod	Prix	....
Ligne après (NEW)	10	15.99	

	Codprod	Prix	....
Ligne après (NEW)	10	12.1	

Triggers

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

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

Triggers

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

- **0 < *codcli* < 10000**

```
CREATE OR REPLACE TRIGGER    VERIFIER_NUM_CLIENT
BEFORE  INSERT OR UPDATE OF Codcli ON CLIENT
FOR EACH ROW
WHEN (New.Codcli <= 0) OR (New.Codcli  >= 10000)
BEGIN
    RAISE_APPLICATION_ERROR (-20009, ' Numéro du client incorrect')
END ;
```

Triggers

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

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

Triggers

Exemple complet: mettre à jour automatiquement les voyages (colonne 'qte' de la table «voyage») lorsque l'on modifie (création, modification, suppression) les commandes des clients (table «ligne_com»).

```
CREATE OR REPLACE TRIGGER modif_voyage
AFTER DELETE OR UPDATE OR INSERT ON ligne_com
FOR EACH ROW
DECLARE
BEGIN
IF DELETING THEN
/* On met à jour la nouvelle quantité de voyages annuel par employé */
UPDATE voyage SET
qte = qte + :OLD.qtecmd
WHERE Id_voyage = :OLD.Id_voyage;
END IF;
```

Triggers

Suite de l'exemple

```
IF INSERTING THEN
UPDATE voyage SET
qte = qte - :NEW.qtecmd
WHERE Id_voyage = :NEW.Id_voyage;
END IF;
IF UPDATING THEN
UPDATE voyage SET
qte = qte + (:OLD.qtecmd - :NEW.qtecmd)
WHERE Id_voyage = :NEW.Id_voyage;
END IF;
END;/
```

Triggers

Quelques particularités des TRIGGERS Oracle

- Pas de **SELECT** dans le **WHEN**
- **:NEW, :OLD**
- Corps en **PL/SQL**
- Pas de **COMMIT / ROLLBACK** dans un **TRIGGER**
  - ✓ Procédure PL/SQL **RAISE_APPLICATION_ERROR**
- **IF INSERTING, DELETING, UPDATING.**
- Evénement non standards
  - ✓ **INSTEAD OF**, **STARTUP**, **LOGON**, …