LE LANGAGE SQL

1-Définition des acronymes

SQL *:Structured Query Langage*

LDD *(Langage de Définition de Données) :*

création, modification et suppression des définitions des tables

LMD *(Langage de Manipulation de Données) :*

ajout, suppression, modification et interrogation des données

LCD *(Langage de Contrôle de Données) : gestion*

des protections d'accès

- *Fin d'instruction : ;*
- *Commentaires : /* ... */ ou –commentaire*

Base de données exemple

CLIENT (NumCli, Nom, Prénom, DateNaiss, Rue,CP, Ville)

PRODUIT (NumProd, Dési, PrixUni, #NumFour)

FOURNISSEUR (NumFour, RaisonSoc)

COMMANDE (NumCli, #NumProd, Date, Quantité)

Clés primaires

Clés étrangères

2. Définition des données

Définitions des tables

CREATE TABLE Nom_Table (Attribut1 TYPE, Attribut2 TYPE, …,
contrainte_integrité1,
contrainte_integrité2, …);

Type des données :

- NUMBER(n) : Entier à n chiffres
- NUMBER(n, m) : Réel à n chiffres au total (virgule comprise), m après la virgule
- VARCHAR(n) : Chaîne de n caractères (entre ' ')
- DATE : Date au format 'JJ-MM-AAAA'

Définitions des contraintes d'intégrité

• *Clé primaire :*

CONSTRAINT nom_contrainte PRIMARY KEY
(attribut_clé [, attribut_clé2, …])

• *Clé étrangère :*

CONSTRAINT nom_contrainte FOREIGN KEY
(attribut_clé_ét) REFERENCES table(attribut)

• *Contrainte de domaine :*

CONSTRAINT nom_contrainte CHECK (condition)

Définition des données

ex. CREATE TABLE Client (NumCli NUMBER(3),
Nom CHAR(30),
DateNaiss DATE,
Salaire NUMBER(8,2),
NumEmp NUMBER(3),

CONSTRAINT cle_pri PRIMARY KEY (NumCli),
CONSTRAINT cle_etr FOREIGN KEY (NumEmp)
REFERENCES EMPLOYEUR(NumEmp),
CONSTRAINT date_ok CHECK (DateNaiss < SYSDATE));

- **Création d'index** (accélération des accès)

  CREATE [UNIQUE] INDEX nom_index ON
  nom_table (attribut [ASC|DESC], …);

  UNIQUE ® pas de double

  ASC/DESC ® ordre croissant ou décroissant

  ex. CREATE UNIQUE INDEX Icli ON Client (NumCli);

- **Destructions :** DROP TABLE nom_table;
  DROP INDEX nom_index

- **Ajout d'attributs**

  ALTER TABLE nom_table ADD (attribut TYPE, …);

  ex. ALTER TABLE Client ADD (tel NUMBER(8));

- **Modifications d'attributs**

  ALTER TABLE nom_table MODIFY (attribut TYPE, …);

  ex. ALTER TABLE Client MODIFY (tel NUMBER(10));

- **Suppression de contraintes**

  ALTER TABLE nom_table DROP CONSTRAINT
  nom_contrainte;

3. Mise à jour des données

- **Ajout d'un tuple**

  INSERT INTO nom_table
  VALUES (val_att1, val_att2, …);

  ex. INSERT INTO Produit
  VALUES (400, 'Nouveau produit', 78.90);

- **Mise à jour d'un attribut**

  UPDATE nom_table SET attribut=valeur
  [WHERE condition];

  ex. UPDATE Client SET Nom='Dudule'
  WHERE NumCli = 3;

3. Mise à jour des données

- **Suppression de tuples**

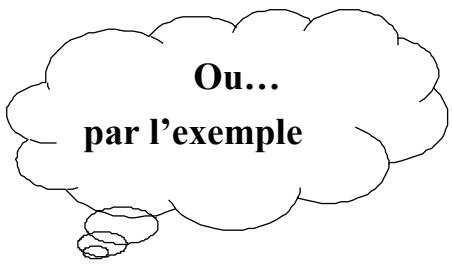
  DELETE FROM nom_table [WHERE condition];

  ex. DELETE FROM Produit;

  ~~—~~ DELETE FROM Client
  WHERE Ville = ‘Lyon’;

4. Interrogation des données

SELECT [ALL|DISTINCT] attribut(s) FROM table(s)
[WHERE condition]
[GROUP BY attribut(s) [HAVING condition]]
[ORDER BY attribut(s) [ASC|DESC]];



- **Tous les tuples d’une table**
  ~~ex.~~ SELECT * FROM Client;

- **Tri du résultat**
  ~~ex.~~ *Par ordre alphabétique inverse de nom*
  SELECT * FROM Client
  ORDER BY Nom DESC;

4. Interrogation des données

- **Calculs** ex. *Calcul de prix TTC*

SELECT PrixUni+PrixUni*0.18 FROM Produit;

- **Projection**

ex. *Noms et Prénoms des clients, uniquement*

SELECT Nom, Prenom FROM Client;

- **Restriction**

ex. *Clients qui habitent à Lyon*

SELECT * FROM Client

WHERE Ville = 'Lyon';

II.4. Interrogation des données

ex. *Commandes en quantité au moins égale à 3*

SELECT * FROM Commande

WHERE Quantite >= 3;

ex. *Produits dont le prix est compris entre 50 et 100 F*

SELECT * FROM Produit

WHERE PrixUni BETWEEN 50 AND 100;

ex. *Commandes en quantité indéterminée*

SELECT * FROM Commande

WHERE Quantite IS NULL;

4. Interrogation des données

ex. *Clients habitant une ville dont le nom se termine par* sur-Saône

```
SELECT * FROM Client
WHERE Ville LIKE '%sur-Saône';
```

'sur-Saône%' ® commence par *sur-Saône*

'%sur%' ® contient le mot *sur*

4. Interrogation des données

ex. *Prénoms des clients dont le nom est Dupont, Durand ou Martin*

```
SELECT Prenom FROM ClientWHERE Nom IN ('Dupont', 'Durand', 'Martin');
```

NB : Possibilité d'utiliser la négation pour tous ces prédicats : **NOT BETWEEN, NOT NULL, NOT LIKE, NOT IN.**

4. Interrogation des données

- **Fonctions d'agrégat**

  Elles opèrent sur un ensemble de valeurs.

  – AVG(), VARIANCE(), STDDEV() : moyenne, variance et écart-type des valeurs

  – SUM() : somme des valeurs

  – MIN(), MAX() : valeur minimum, valeur maximum

  – COUNT() : nombre de valeurs

  ex. *Moyenne des prix des produits*

  SELECT AVG(PrixUni) FROM Produit;

4. Interrogation des données

Opérateur DISTINCT

ex. *Nombre total de commandes*

SELECT COUNT(*) FROM Commande;

SELECT COUNT(NumCli) FROM Commande;

ex. *Nombre de clients ayant passé commande*

SELECT COUNT(DISTINCT NumCli)

FROM Commande;

4. Interrogation des données

Table COMMANDE (simplifiée)

NumCli	*Date*	*Quantite*
1	22/09/99	1
3	22/09/99	5
3	22/09/99	2

COUNT(NumCli) ® Résultat = 3

COUNT(DISTINCT NumCli) ® Résultat = 2

4. Interrogation des données

- **Jointure**

  ex. *Liste des commandes avec le nom des clients*

```
SELECT Nom, Date, Quantite
FROM Client, Commande
WHERE Client.NumCli =Commande.NumCli;
```

4. Interrogation des données

ex. *Idem avec le numéro de client en plus*

```
SELECT C1.NumCli, Nom, Date, Quantite
FROM Client C1, Commande C2
WHERE C1.NumCli = C2.NumCli
ORDER BY Nom;
```

NB : Utilisation d'*alias* (C1 et C2) pour alléger l'écriture + tri par nom.

II.4. Interrogation des données

Jointure exprimée avec le prédicat IN

ex. *Nom des clients qui ont commandé le 23/09*

```
SELECT Nom FROM Client
WHERE NumCli IN (
    SELECT NumCli FROM Commande
    WHERE Date = '23-09-1999'                    );
```

NB : Il est possible d'imbriquer des requêtes.

II.4. Interrogation des données

- **Prédicats EXISTS / NOT EXISTS**

  ex. *Clients qui ont passé au moins une commande*

  [n'ont passé aucune commande]

```
SELECT * FROM Client C1
WHERE [NOT] EXISTS (
        SELECT * FROM Commande C2
        WHERE C1.NumCli = C2.NumCli) ;
```

4. Interrogation des données

- **Prédicats ALL / ANY**

  ex. *Numéros des clients qui ont commandé au moins un produit en quantité supérieure à chacune [à au moins une] des quantités commandées par le client n° 1.*

-

```
SELECT DISTINCT NumCli FROM Commande
WHERE Quantite > ALL [ANY] (
        SELECT Quantite FROM Commande
        WHERE NumCli                = 1);
```

4. Interrogation des données

- **Groupement**

ex. *Quantité totale commandée par chaque client*

```
SELECT NumCli, SUM(Quantite)
FROM Commande
GROUP BY NumCli;
```

ex. *Nombre de produits différents commandés...*

```
SELECT NumCli, COUNT(DISTINCT NumProd)
FROM Commande
GROUP BY NumCli;
```

II.4. Interrogation des données

ex. *Quantité moyenne commandée pour les*

produits faisant l'objet de plus de 3 commandes

```
SELECT NumProd, AVG(Quantite)
FROM Commande
GROUP BY NumProd
HAVING COUNT(*)>3;
```

Attention : La clause HAVING ne s'utilise qu'avec GROUP BY.

II.4. Interrogation des données

- **Opérations ensemblistes**

  INTERSECT, MINUS, UNION

  ex. *Numéro des produits qui soit ont un prix*
  inférieur à 100 F, soit ont été commandés
  par le client n° 2

```
SELECT NumProd FROM Produit WHERE PrixUni<100
UNION
SELECT NumProd FROM Commande WHERE NumCLi=2;
```

4. Interrogation des données

- **Fonctions SQL*Plus**

– ABS(n) : Valeur absolue de n
– CEIL(n) : Plus petit entier ε n
– FLOOR(n) : Plus grand entier δ n
– MOD(m, n) : Reste de m/n
– POWER(m, n) : mn
– SIGN(n) : Signe de n
– SQRT(n) : Racine carrée de n
– ROUND(n, m) : Arrondi à 10-m
– TRUNC(n, m) : Troncature à 10-m
– CHR(n) : Caractère ASCII n° n
– INITCAP(ch) : 1ère lettre en maj.
– LOWER(ch) : c en minuscules
– UPPER(ch) : c en majuscules
– LTRIM(ch, n) : Troncature à gauche
– RTRIM(ch, n) : Troncature à droite
– REPLACE(ch, car) : Remplacement de caractère
– SUBSTR(ch, pos, lg) : Extraction de chaîne
– SOUNDEX(ch) : Cp. Phonétique
– LPAD(ch, lg, car) : Compléter à gauche
– RPAD(ch, lg, car) : Compléter à droite

4. Interrogation des données

- ASCII(ch) : Valeur ASCII de ch
- INSTR(ch, ssch) : Recherche de ssch dans ch
- LENGTH(ch) : Longueur de ch
- ADD_MONTHS(dte, n) : Ajout de n mois à dte
- LAST_DAY(dte) : Dernier jour du mois
- MONTHS_BETWEEN(dt1, dt2) : Nombre de mois entre dt1 et dt2
- NEXT_DAY(dte) : Date du lendemain
- SYSDATE : Date/heure système
- TO_NUMBER(ch) : Conversion de ch en nombre
- TO_CHAR(x) : Conversion de x en chaîne
- TO_DATE(ch) : Conversion de ch en date
- NVL(x, val) : Remplace par val si x a la valeur NULL
- GREATEST(n1, n2…) : + grand
- LEAST (n1, n2…) : + petit
- UID : Identifiant numérique de l'utilisateur
- USER : Nom de l'utilisateur

EXERCICES

Exercice 1

Soit le schéma relationnel de la base de données d'un organisme de voyage.

Station (IdStation,nomStation, capacité, lieu, région, tarif)
Activite (IdAct,nomActivité, prix, #IdStation)
Client (idclient, nom, prénom, ville, solde)
Séjour (idClient, IdStation, datedébut,datefin, nbPlaces)

Exprimer les requêtes suivantes en SQL*Plus.

1. Nom des stations ayant strictement plus de 200 places.
2. Noms des clients dont le nom commence par 'P' ou dont le solde est supérieur à 10 000.
3. Noms et capacité de toutes les stations se trouvant aux Antilles.
4. Nom des stations qui proposent de la plongée.
5. Nom des clients qui habite à Saint Denis.
6. Nom des régions qu'a visité Mr Pascal.
7. Nom et prenoms clients qui ont effectués un sejour dans la station de situé dans la region sud
8. Quelles stations ne proposent pas de la plongée ?
9. Liste des clients (nom, prenom,solde) qui ont pris plus de 2 places en Janvier 2017
10. Le nombre d'activité par station et le nombre de client par ville

Exercice 2

Soit le schéma relationnel de la base FABRICATION.
CLIENT(NOC, NOM, ADRESSE,TELEPHONE)
SERVICE(NOS, INTITULE, LOCALISATION)
PIECE(NOP, DESIGNATION, COULEUR, POIDS)
COMMANDE (NOS, NOC,NOP QUANTITE)

Formuler en SQL*Plus les commandes de création de la structure de cette base, puis exprimer les requêtes suivantes.

1) Donner pour chaque service le poids de la pièce commandée de couleur bleue.
2) Donner le poids moyen des pièces commandées pour chacun des services "Promotion".
3) Donner les pièces de couleur bleue qui sont commandées par le service localisé a Lyon.
4) Donner le maximum des quantités des pièces commandées par les différents services.

Introduction au PL/SQL

PL/SQL (pour PROCEDURAL LANGUAGE/SQL) est un langage procédural d'Oracle corporation étendant SQL. Il permet de combiner les avantages d'un langage de programmation classique avec les possibilités de manipulation de données offertes par SQL.

1 Structure d'un programme PL/SQL

La structure de base d'un programme PL/SQL est celle de bloc (possiblement imbriquée). Il a généralement la forme suivante:

DECLARE

/* section de declaration */

BEGIN

/* corps du bloc de programme

Il s'agit de la seule zone dont la présence est obligatoire */

EXCEPTION

/* gestion des exceptions */

END;

Le corps du programme (entre le BEGIN et le END) contient des instructions PL/SQL (assignements, boucles, appel de procédure) ainsi que des instructions SQL. Il s'agit de la seule partie qui soit obligatoire. Les deux autres zones, dites zone de Déclaration et zone de gestion des exceptions sont facultatives. Les seuls ordres SQL que l'on peut trouver dans un bloc PL/SQL sont: **SELECT, INSERT, UPDATE, DELETE**. Les autres types d'instructions (par exemple CREATE, DROP, ALTER) ne Peuvent se trouver qu''a l'extérieur d'un tel bloc.

Chaque instruction se termine par un "**;**".

Le PL/SQL ne se soucie pas de la casse (majuscule vs. minuscule). On peut inclure des commentaires par **---** (en début de chaque ligne commentée) ou par /*....*/ (pour délimiter des blocs de commentaires).

2 Variables et types

Un nom de variable en PL/SQL comporte au plus 30 caractères. Toutes les variables ont un type On trouve trois sortes de types de variables en PL/SQL. A savoir :

- un des types utilisés en SQL pour les colonnes de tables.
- un type particulier au PL/SQL.
- un type faisant référence `a celui d'une (suite de) colonne(s) d'une table.

Les types autorisés dans PL/SQL sont nombreux. On retiendra principalement:

- Pour les types numériques : **INTEGER, NUMBER** (précision de 38 chiffres par défaut**), NUMBER(x)** (nombres avec x chiffres de précision), **NUMBER(x,y)** (nombres avec x chiffres de précision dont y après la virgule).

- Pour les types alphanumériques : **CHAR(x)** (chaine de caractère de longueur fixe x), **VARCHAR(x)** (chaine de caractère de longueur variable jusqu''a x), **VARCHAR2** (idem que précédent excepte que ce type supporte de plus longues chaines et que l'on est pas obligé de spécifier sa longueur maximale). PL/SQL permet aussi de manipuler des dates (type **DATE**) sous différents formats.

Une déclaration pourrait donc contenir les informations suivantes :

DECLARE

n NUMBER;

mot VARCHAR(20) ;

mot2 VARCHAR2 ;

Une autre specificité du PL/SQL est qu'il permet d'assigner comme type a une Variable celui d'un champ d'une table (par l'operateur %TYPE) ou d'une ligne entière (Operateur %ROWTYPE). Dans la déclaration suivante :

DECLARE

nom emp.name%TYPE;

employe emp%ROWTYPE;

La variable nom est definie comme etant du type de la colonne name de la table emp (qui doit exister au préalable). De même, employé est un vecteur du type d'une ligne de la table emp. A supposer que cette dernière ait trois champs numéro, name, age de type respectifs NUMBER, VARCHAR, INTEGER, la variable employe disposera de trois composantes :employe.numero, employe.name, employe.age, de meme types que ceux de la table.

Soit ce premier programme

```
DECLARE
A NUMBER;
b NUMBER;
BEGIN
a:=a+b;
END;
```

(noter au passage l'instruction d'affectation "a:=a+b")

Un deuxième programme (incluant une requete SQL) :

```
DECLARE
a     NUMBER;
BEGIN
SELECT numero INTO a FROM emp
WHERE noemp=’21’;
END;
```

Ce dernier exemple donne comme valeur `a la variable a le resultat de la requete (qui doit être du meme type). Il est impératif que la requete ne renvoie qu’un et un seul résultat (c’est `a dire qu’il n’y ait qu’un seul employé numero 21). Autrement, une erreur se produit.

3 Operateurs

PL/SQL supporte les operateurs suivants :

- Arithmetique : +, - *, /, ** (exponentielle)
- Concatenation : ||
- Parentheses (controle des priorites entre operations):
    - Affectation: :=
    - Comparaison : =, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN
    - Logique : AND, OR, NOT

EXERCICE 1:

Soit le table: **EMP (EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO)**

Ecrire un programme pl/sql permettant d’insérer un nouvel employé avec comme numéro matricule le plus grand des numero matricule de la table plus une unité et comme salaire la moyenne des salaires du département numéro 30

4 Structures de contrôle

Comme n'importe quel langage procédural, PL/SQL possède un certain nombre de structures de controles evoluees comme les branchements conditionnels et les boucles.

4.1 Les branchements conditionnels

Syntaxe :

IF <condition> THEN

commandes;

[ELSEIF <condition> THEN

commandes;]

[ELSE

commandes;]

END IF;

- Exemple :

IF nomEmploye='TOTO' **THEN**

salaire:=salaire*2;

ELSEIF salaire>10000 THEN

salaire:=salaire/2;

ELSE

salaire:=salaire*3;

END IF;

4.2 Boucles

PL/SQL admet trois sortes de boucles. La première est une boucle potentiellement infinie :

LOOP

commandes;

END LOOP;

Au moins une des instructions du corps de la boucle doit etre une instruction de sortie :

EXIT WHEN <condition>;

Dés que la condition devient vraie (si elle le devient...), on sort de la boucle.

Le deuxieme type de boucle permet de repeter un nombre d´efini de fois un mˆeme traitement :

FOR <compteur> IN [REVERSE] <limite_inf> .. <limite_sup> LOOP

commandes;

END LOOP;

Enfin, le troisieme type de boucle permet la sortie selon une condition predefinie.

WHILE <condition> LOOP

commandes;

END LOOP;

Toutes ces structures de contrôles sont évidemment imbriquables les unes dans les autres. Voici un meme exemple qui traite de trois manieres differentes suivant le type de boucle choisi.

Premier exemple :

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

Deuxieme exemple :

```
DECLARE
x NUMBER(3);
BEGIN
FOR x IN 1..100 LOOP
INSERT INTO employe (noemp, nomemp, job, nodept)
VALUES (x, 'TOTO', 'PROGRAMMEUR', 1);
END LOOP;
END;
```

Troisieme 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;
```

Exercice2: Repeter l'exemple precedent 10 fois en mettant les employes a code paire dans la premiere table et les employés a code impair dans une seconde table

5- Curseurs

Jusqu'a présent, nous avons vu comment récupérer le résultat de requêtes SQL dans des variables PL/SQL lorsque ce résultat ne comporte au plus qu'une seule ligne. Bien évidemment, cette situation est loin de représenter le cas général : les requêtes renvoient très souvent un nombre important et non prévisible de lignes. Dans la plupart des langages procéduraux au dessus de SQL, on introduit une notion de "curseur" pour récupérer (et exploiter) les résultats de requêtes.

Un curseur est une variable dynamique qui prend pour valeur le résultat d'une requête. La méthode pour utiliser un curseur est invariable. En premier lieu, celui-ci doit etre défini (dans la zone de declaration). Exemple :

CURSOR empCur IS SELECT * FROM emp;

Le curseur de nom empCur est chargée dans cet exemple de récupérer le résultat de la requête qui suit. Il peut alors être ouvert lorsqu'on souhaite l'utiliser (dans le corps d'un bloc) :

OPEN empCur;

Pour récupérer les tuples successifs de la requête, on utilise l'instruction :

FETCH empCur INTO employeVar;

La premiere fois, c'est le premier tuple du resultat de la requete qui est affectée comme valeur a la variable employeVar (qui devra etre definie de facon adequate). A chaque execution de l'instruction, un nouveau tuple resultat est charge dans employeVar. Enfin, lorsque le traitement sur le resultat de la requete est terminée, on ferme le curseur.

CLOSE empCur;

L’exemple suivant sélectionne l’ensemble des employés dont le salaire ne dépasse pas 6000 francs et les augmente de 500 francs.

```
DECLARE
CURSOR SalCur IS
SELECT * FROM EMP WHERE SAL<6000.00;
employe     EMP%ROWTYPE;
BEGIN
OPEN SalCur;
LOOP
FETCH SalCur INTO employe;
UPDATE EMP
SET SAL=6500.00          WHERE EMPNO=employe.empno;
EXIT WHEN SalCur%NOTFOUND;
END LOOP;
END;
```

Lorsqu’on souhaite parcourir un curseur dans une boucle pour effectuer un traitement, on peut simplifier l’utilisation de ceux-ci. Le programme suivant, parfaitement l´égal, est équivalent au précédent.

```
DECLARE
CURSOR SalCur IS
SELECT * FROM EMP WHERE SAL<6000.00;
employe     EMP%ROWTYPE;
BEGIN
FOR employe IN SalCur
LOOP
UPDATE EMP
SET SAL=6500.0 WHERE EMPNO=employe.empno;
END LOOP;
END;
```

Dans cette boucle, l'ouverture, la fermeture et l'incrémentation du curseur SalCur sont implicites et n'ont pas besoin d'être specifiquees.

Un certain nombre d'informations sur l'état d'un curseur sont exploitables a l'aide d'attributs predefinis : l'attribut **%FOUND** renvoie vrai si le dernier FETCH a Bien ramene un tuple. L'attribut **%NOTFOUND,** permet de décider si on est arrivé en fin de curseur**. %ROWCOUNT** renvoie le nombre de lignes ramenes du curseur au moment de l'interrogation. Enfin %ISOPEN permet de déterminer si le curseur est ouvert.

EXERCICE 2:

1- Créer une table **AGREGAT**(deptno,deptname, tot_salaire,tot_comm,nbre_employe)

Ecrire un programme pl/sql permettant d'enrichir cette table avec un cursor.

6- La gestion des exceptions

PL/SQL permet de definir dans une zone particuliere (de gestion d'exception), l'attitude que le programme doit avoir lorsque certaines erreurs d´efinies ou pr´ed´efinies se produisent.

Un certain nombre d'exceptions sont predefinies sous Oracle. Citons, pour les plus fréquentes : NO_DATA_FOUND (devient vrai des qu'une requête renvoie un résultat vide), TOO_MANY_ROWS (requête renvoie plus de lignes qu'escompte), CURSOR_ALREADY_OPEN (curseur deja ouvert), INVALID_CURSOR (curseur invalide)...

L'utilisateur peut definir ses propres exceptions. Dans ce cas, il doit definir celles ci dans la zone de declaration. Exemple :

excpt1 EXCEPTION

Puis, cette exception est levée quelque part dans le programme (apres un test non concluant, par exemple), par l'instruction :

RAISE excpt1

Enfin, dans la zone d'exception un traitement est affectée a chaque exception Possible (definie ou predefnie) :

EXCEPTION

WHEN <exception1> [OR <exception2> OR ...] THEN <instructions>

WHEN <exception3> [OR <exception2> OR ...] THEN <instructions>

WHEN OTHERS THEN <instructions>

END;

Evidemment, un seul traitement d'exception peut se produire avant la sortie du bloc.

Autre Exemple : **Calcul du prix TTC des produits et recopie dans la table PRODTTC**

```
DECLARE
nbp NUMBER(3);
aucun_produit EXCEPTION;
CURSOR calcul IS
SELECT numprod, prixuni*1.206 prixttc
FROM produit;
tuple calcul%ROWTYPE;
 BEGIN
-- Comptage des produits
SELECT COUNT(*) INTO nbp FROM produit;
-- Test « il existe des produits » ou pas ?
IF nbp = 0 THEN
RAISE aucun_produit;
END IF;
 -- Recopie des valeurs dans la table prodttc
FOR tuple IN calcul LOOP
INSERT INTO prodttc VALUES
(tuple.numprod, tuple.prixttc);
END LOOP;
-- Validation de la transaction
COMMIT;

EXCEPTION
WHEN aucun_produit THEN
RAISE_APPLICATION_ERROR (-20501, 'Erreur : table client vide');

END;
```

7- Procédures et fonctions

Il est possible de créer des procédures et des fonctions comme dans n'importe quel langage de programmation classique. La syntaxe de création d'une procédure est la Suivante :

CREATE OR REPLACE PROCEDURE <nom>
AS
<Zone de déclaration de variables>
BEGIN
<corps de la procédure>
EXCEPTION
<Traitement des exceptions>
END;

A noter que toute fonction ou procédure créée devient un objet part entière de la base (comme une table ou une vue, par exemple). Elle est souvent appelée "procédure stockée". Elle est donc, entre autres, sensible `a la notion de droit : son céeateur peut décider par exemple d'en permettre l'utilisation `a d'autres utilisateurs. Elle est aussi appelable depuis n'importe quel bloc PL/SQL avec la commande :

EXECUTE [NOM_PROCEDURE] ;

Example

```
CREATE OR REPLACE PROCEDURE Insert_T
AS
x NUMBER ;
  y NUMBER ;
Matricule NUMBER ;
Salaire  NUMBER ;
BEGIN
Select empno into x from scott.emp where ename='JONES' ;
Matricule:=x+1;
Select sal into y from scott.emp where ename='JONES' ;
Salaire := y+ (y*20/100) ;
INSERT INTO T VALUES (matricule, salaire);
Commit ;
END;
```

On peut aussi créer des fonctions. Le principe est le meme, l'en-tête devient :

CREATE OR REPLACE FUNCTION <nom> [(parametres)] RETURN <type du resultat>
IS
BEGIN
<Instructions>;
Return <var>;
END;

Une instruction RETURN <expression> devra se trouver dans le corps pour specifier

Le type du resultat est renvoyé.

Example

```
CREATE OR REPLACE FUNCTION module (a NUMBER, b NUMBER) RETURN NUMBER IS
BEGIN
IF a < b THEN
RETURN a ;
ELSE
RETURN (b-a);
END IF ;
END;
```

Les procédures et fonctions PL/SQL supportent assez bien la surcharge (i.e. co-

existence de procédures de même nom ayant des listes de parametres differentes).

C'est le systeme qui, au moment de l'appel, decidera, en fonction du nombre d'arguments

et de leur types, quelle est la bonne procedure a appeler.

On peut detruire une procedure ou une fonction par:

DROP PROCEDURE <nom de procedure>

EXERCICE

L'entreprise employant les personnes de la table EMP de l'exo n° 1(cours) est délocalisée des États-Unis en France (si, si). Il est donc nécessaire de convertir leur salaire et leur commission en francs (pour simplifier,on admettra 1 USD = 6,5 FF). Tous les employés voient également augmenter leur salaire de 25 % après conversion.

1) Créer une nouvelle table vide EMP_FR de même structure que EMP a partir d'une seule requête.

2) Écrire un programme PL/SQL permettant de recopier tous les tuples de la table EMP dans la table EMP_FR en effectuant au passage les opérations nécessaires sur le salaire et la commission. Traiter le cas où la table EMP est vide comme une exception.

8- Les déclencheurs (“triggers”)

Les déclencheurs ou ”triggers” sont des séquences d’actions définis par le programmeur qui se declenchent, non pas sur un appel, mais directement quand un événement particulier (specifiée lors de la definition du trigger) sur une ou plusieurs tables se Produit.

Un trigger sera un objet stock´e (comme une table ou une procédure) La syntaxe est la suivante:

CREATE [OR REPLACE] TRIGGER <nom>

{BEFORE|AFTER} {INSERT|DELETE|UPDATE} ON <nom de table>

FOR EACH ROW [WHEN (<condition>)]

<corps du trigger>

Un trigger se declenche avant ou apres (BEFORE|AFTER) une insertion, suppresion ou mise a jour (INSERT|DELETE|UPDATE) sur une table (`a noter que l’on peut exprimer des conditions plus complexes avec le OR: INSERT OR DELETE ...).

L’option FOR EACH ROW [WHEN (<condition>)]] fait s’exécuter le trigger `a chaque modification d’une ligne de la table specifiée (on dit que le trigger est de ”niveau ligne”).

Soit l’exemple suivant :

```
CREATE TABLE T1 (num1 INTEGER, num2 INTEGER);
CREATE TABLE T2 (num3 INTEGER, num4 INTEGER);
CREATE TRIGGER inverse
AFTER INSERT ON T1
FOR EACH ROW
BEGIN
IF (:NEW.num1 <=3) THEN
INSERT INTO T2 VALUES(:NEW.num2, :NEW.num1);
END IF;
END inverse;
```

Ce trigger va, en cas d'insertion d'un tuple dans T 1 dont la premiere coordonnee est inferieure `a 3, inserer le tuple inverse dans T 2. Les prefixes **NEW et OLD** (en cas de UPDATE ou de DELETE) vont permettre de faire reference aux valeurs des colonnes apres et avant les modifications dans la table. Ils sont utilisés sous la forme NEW.num dans la condition du trigger et sous la forme : NEW.num dans le corps.

Un trigger peut etre activée ou desactivée par les instructions :

ALTER TRIGGER <nom> {ENABLE|DISABLE};

et détruit par :

DROP TRIGGER <nom>.

9 Quelques remarques

9.1 Affichage

PL/SQL n'est pas un langage avec des fonctionnalités d'entrées sorties évoluées (ce N'est pas son but!). Toutefois, on peut imprimer des messages et des valeurs de variables de plusieurs manières différentes. Le plus pratique est de faire appel `a un package predéfini : DBMS output.

En premier lieu, taper au niveau de la ligne de commande :

SET SERVEROUTPUT ON

Pour afficher, on utilise alors la commande suivante :

DBMS_OUTPUT.PUT_LINE ('Au revoir' || nom || ' a bientôt');

Si on suppose que nom est une variable dont la valeur est "toto", l'instruction Affichera : Au revoir toto a bientôt.

9.2 Debugger

Pour chaque objet créée (procédure, fonction, trigger...), en cas d'erreur de compilation, on peut voir ces erreurs en tapant (si "essai" est un nom de procédure)

SHOW ERRORS PROCEDURE essai;

Exercice 1

Soit le schéma relationnel d'une agence bancaire régionale.

CLIENT (NUMCL, NOM, PRENOM, ADR, #CodeVILLE, SALAIRE, CONJOINT)

Ville (CodeVILLE,nomville,departement)

Écrire un trigger en insertion permettant de contrôler les contraintes suivantes :

− La Ville dans laquelle habite le client doit être Cotonou,calavi,ouidah,ou Porto-Novo;

− Le nom du conjoint doit être le même que celui du client.

Exercice 2

Soit une table quelconque TABL, dont la clé primaire CLENUM est numérique. Définir un trigger en insertion permettant d'implémenter une numérotation automatique de la clé. Le premier numéro doit être 1.

Exercice 3

Soit la table suivante :

METEO (NOM_VILLE, Température, Humidité)

Ecrire un déclencheur qui avant l'insertion d'une nouvelle ville dans la table vérifie :

a. Si la température est la plus grande de toutes les villes, afficher un message d'avertissement.

b. Si la ville existe déjà dans la table, ne pas l'insérer une nouvelle fois mais faire la mis a jour seulement.

***** FIN DU MODULE **********