UNIVERSITE IBN TOFAIL
Faculté des Sciences
Département d'Informatique
Kenitra

Cours : Bases de Données II et Web

SMI S6: Sciences Mathématiques et Informatique

Professeur : Abdelalim SADIQ
Email : sadiq.alim@gmail.com
Web: https://sites.google.com/sadiqalim

Année universitaire 2014-2015

SOMMAIRE

Chapitre 1 : Présentation générale 4

1. Principe de base 4
1.1. Bases de données et SGBD 4
1.2. Objectifs 4
1.3. Modèle de base de données 5
1.4. Niveaux de description des données 6
2. Rappel 6
2.1. Langage SQL 6
2.2. Ob jets manipulés par SQL 7
2.3. Types de données 8
2.4. Langage de définit ion de données 8
2.5. Langage de manipulation de données – Modifier une base 10
2.6. Langage de manipulation de données - Interroger une base 11
3. Les Vues 12
3.1. Définition 12
3.2. Avantage des vues 12
3.3. Création d'une vue 12
3.4. Suppression d'une vue 13
3.5. Modifier une vue 13

Chapitre 2 : PL/SQL 14

1. Structure en blocs d'un programme PL/SQL 14
2. Affichage 15
3. Les types 15
4. Variables 16
5. Types de données composées 16
5.1. Tableaux statiques 17
5.2. Tableau dynamique : 19
5.3. Fonctions et procédures associées à une table : 19
5.4. Structures (tableau des enregistrements) 19
6. Structures de contrôle 20

Traitements conditionnels 20
6.2.Traitements répétitifs 21
7.Procédures et fonctions 22
8. Curseurs 23
8.1.Déclaration de curseurs 23
8.2.Le contrôle d'un curseur 23
8.3.Attributs des curseurs explicites 23
8.4.Paramètre des curseurs 24
9. Gestion des erreurs et exceptions 24
9.1.Exceptions prédéfinies 25
9.2.Définies par l'utilisateur 25
9.3.Utilisation de RAISE_APPLICATION_ERROR 26
10. Procédures stockées 26
10.1.Procédures. 26
10.2.Fonctions 27
10.3.Modification – suppression (d'une procédure ou fonction). 27
11. Package 28
12. Triggers 28

Chapitre 1 : Présentation générale

1. Principe de base

1.1. Bases de données et SGBD

Les bases de données sont actuellement au cœur du système d'information des entreprises. Le stockage de ces données directement dans un ensemble de fichiers soulève de très gros problèmes : Lourdeur d'accès aux données, manque de sécurité, pas de contrôle de concurrence...d'où le recours à un logiciel chargé de gérer les fichiers constituant une base de données, de prendre en charge les fonctionnalités de protection et de sécurité et de fournir les différents types d'interface nécessaires à l'accès aux données.

La première chose à faire est d'établir quelques points de terminologie.

Une **Base de Données** (BD) est un ensemble structuré d'informations mémorisées sur un support permanent et mises à disposition d'un ensemble d'utilisateurs, informaticiens ou non.

Un **Système de Gestion de Base de Données** (SGBD) est un logiciel général qui permet à l'utilisateur de manipuler les données dans des termes abstraits, sans tenir compte de la façon dont l'ordinateur les représente.

1.2. Objectifs

Des objectifs principaux ont été fixés aux SGBD dès l'origine de ceux-ci et ce, afin de résoudre les problèmes causés par le système de fichier classique. Ces objectifs sont les suivants:

Indépendance physique : La façon dont les données sont définies doit être indépendante des structures de stockage utilisées.

Indépendance logique : Un même ensemble de données peut être vu différemment par des utilisateurs différents. Toutes ces visions personnelles des données doivent être intégrées dans une vision globale.

Accès aux données : L'accès aux données se fait par l'intermédiaire d'un Langage de Manipulation de Données (LMD). Il est crucial que ce langage permette d'obtenir des réponses aux requêtes en un temps « raisonnable ». Le LMD doit donc être optimisé, minimiser le nombre d'accès disques, et tout cela de façon totalement transparente pour l'utilisateur.

Administration centralisée des données (intégration) : Toutes les données doivent être centralisées dans un réservoir unique commun à toutes les applications. En effet, des visions différentes des données (entre autres) se résolvent plus facilement si les données sont administrées de façon centralisée.

Non redondance des données : Afin d'éviter les problèmes lors des mises à jour, chaque donnée ne doit être présente qu'une seule fois dans la base.

Cohérence des données : Les données sont soumises à un certain nombre de contraintes d'intégrité qui définissent un état cohérent de la base. Elles doivent pouvoir être exprimées simplement et vérifiées automatiquement à chaque insertion, modification ou suppression des données. Les contraintes d'intégrité sont décrites dans le Langage de Description de Données (LDD).

Partage des données : Il s'agit de permettre à plusieurs utilisateurs d'accéder aux mêmes données au même moment de manière transparente. Si ce problème est simple à résoudre quand il s'agit uniquement d'interrogations, cela ne l'est plus quand il s'agit de modifications dans un contexte multiutilisateurs car il faut : permettre à deux (ou plus) utilisateurs de modifier la même donnée « en même temps » et assurer un résultat d'interrogation cohérent pour un utilisateur consultant une table pendant qu'un autre la modifie.

Sécurité des données : Les données doivent pouvoir être protégées contre les accès non autorisés. Pour cela, il faut pouvoir associer à chaque utilisateur des droits d'accès aux données.

Résistance aux pannes : Que se passe-t-il si une panne survient au milieu d'une modification, si certains fichiers contenant les données deviennent illisibles ? Il faut pouvoir récupérer une base dans un état « sain ». Ainsi, après une panne intervenant au milieu d'une modification deux solutions sont possibles : soit récupérer les données dans l'état dans lequel elles étaient avant la modification, soit terminé l'opération interrompue.

1.3. Modèle de base de données

Modèle hiérarchique : Une base de données hiérarchique est une forme de système de gestion de base de données qui lie des enregistrements dans une structure arborescente de façon à ce que chaque enregistrement n'ait qu'un seul possesseur (par exemple, une paire de chaussures n'appartient qu'à une seule personne). Les structures de données hiérarchiques ont été largement utilisées dans les premiers systèmes de gestion de bases de données. Cependant, à cause de leurs limitations internes, elles ne peuvent pas souvent être utilisées pour décrire des structures existantes dans le monde réel. Les liens hiérarchiques entre les différents types de données peuvent rendre très simple la réponse à certaines questions, mais très difficile la réponse à d'autres formes de questions. Si le principe de relation « 1 vers N» n'est pas respecté (par exemple, un malade peut avoir plusieurs médecins et un médecin a, a priori, plusieurs patients), alors la hiérarchie se transforme en un réseau.

Modèle réseau : Le modèle réseau est en mesure de lever de nombreuses difficultés du modèle hiérarchique grâce à la possibilité d'établir des liaisons de type n-n, les liens entre objets pouvant exister sans restriction. Pour retrouver une donnée dans une telle modélisation, il faut connaître le chemin d'accès (les liens) ce qui rend les programmes dépendants de la structure de données.

Modèle relationnel : Une base de données relationnelle est une base de données structurée suivant les principes de l'algèbre relationnelle. Le père des bases de données relationnelles est Edgar Frank Codd. Chercheur chez IBM à la fin des années 1960, il étudiait alors de nouvelles méthodes pour gérer de grandes quantités de données car les modèles et les logiciels de l'époque ne le satisfaisaient pas. Mathématicien de formation, il était persuadé qu'il pourrait utiliser des branches spécifiques des mathématiques (la théorie des ensembles et la logique des prédicats du premier ordre) pour résoudre des difficultés telles que la redondance des données, l'intégrité des données ou l'indépendance de la structure de la base de données avec sa mise en œuvre physique.

On doutait que les tables puissent être jamais gérées de manière efficace par un ordinateur. Ce scepticisme n'a cependant pas empêché Codd de poursuivre ses recherches. Un premier prototype de Système de gestion de bases de données relationnelles (SGBDR) a été construit dans les laboratoires d'IBM. Depuis les années 80, cette technologie a mûri et a été adoptée par l'industrie. En 1987, le langage SQL, qui étend l'algèbre relationnelle, a été standardisé.

Modèle objet : La notion de bases de données objet ou relationnel-objet est plus récente et encore en phase de recherche et de développement. Elle sera très probablement ajoutée au modèle relationnel.

1.4. Niveaux de description des données

Pour atteindre certains de ces objectifs, trois niveaux de description des données ont été définis par la norme ANSI/SPARC.

Le niveau externe : correspond à la perception de tout ou partie de la base par un groupe donné d'utilisateurs, indépendamment des autres. On appelle cette description le schéma externe ou vue. Il peut exister plusieurs schémas externes représentant différents vues sur la base de données avec des possibilités de recouvrement. Le niveau externe assure l'analyse et l'interprétation des requêtes en primitives de plus bas niveau et se charge également de convertir éventuellement les données brutes, issues de la réponse à la requête, dans un format souhaité par l'utilisateur.

Le niveau conceptuel : décrit la structure de toutes les données de la base, leurs propriétés (i.e. les relations qui existent entre elles : leur sémantique inhérente), sans se soucier de l'implémentation physique ni de la façon dont chaque groupe de travail voudra s'en servir.
Dans le cas des SGBD relationnels, il s'agit d'une vision tabulaire où la sémantique de l'information est exprimée en utilisant les concepts de relation, attributs et de contraintes d'intégrité. On appelle cette description le schéma conceptuel.

Le niveau interne ou physique : s'appuie sur un système de gestion de fichiers pour définir la politique de stockage ainsi que le placement des données. Le niveau physique est donc responsable du choix de l'organisation physique des fichiers ainsi que de l'utilisation de telle ou telle méthode d'accès en fonction de la requête. On appelle cette description le schéma interne.

2. Rappel

2.1. Langage SQL

Le langage SQL (Structured Query Language) peut être considéré comme le langage d'accès normalisé aux bases de données. Il est aujourd'hui supporté par la plupart des produits commerciaux que ce soit par les systèmes de gestion de bases de données micro tel que "Access" ou par les produits plus professionnels tels que "Oracle".

Le succès du langage SQL est dû essentiellement à sa simplicité et au fait qu'il s'appuie sur le schéma conceptuel pour énoncer des requêtes en laissant le SGBD responsable de la stratégie d'exécution. SQL propose un langage de requêtes ensembliste et assertionnel. Néanmoins, le langage SQL ne possède pas la puissance d'un langage de programmation : entrées/sorties, instructions conditionnelles, boucles et affectations. Pour certains traitements il est donc nécessaire de coupler le langage SQL avec un langage de programmation plus complet. De manière synthétique, on peut dire que SQL est un langage relationnel, il manipule donc des tables (i.e. des relations, c'est-à-dire des ensembles) par l'intermédiaire de requêtes qui produisent également des tables.

Les instructions SQL sont regroupées en catégories en fonction de leur utilité et des entités manipulées. Nous pouvons distinguer cinq catégories, qui permettent :

Langage de définition de données (LDD) : est un langage orienté au niveau de la structure de la base de données. Le LDD permet de créer, modifier, supprimer des objets. Il permet également de définir le domaine des données (nombre, chaîne de caractères, date, booléen, . . .) et d'ajouter des contraintes de valeur sur les données. Il permet enfin d'autoriser ou d'interdire l'accès aux données et d'activer ou de désactiver l'audit pour un utilisateur donné.

Les instructions du LDD sont : CREATE, ALTER, DROP, AUDIT, NOAUDIT, ANALYZE, RENAME, TRUNCATE.

Langage de manipulation de données (LMD) : est l'ensemble des commandes concernant la manipulation des données dans une base de données. Le LMD permet l'ajout, la suppression et la modification de lignes, la visualisation du contenu des tables et leur verrouillage.

Les instructions du LMD sont : INSERT, UPDATE, DELETE, SELECT, EXPLAIN, PLAN, LOCK TABLE.

Ces éléments doivent être validés par une transaction pour qu'ils soient pris en compte.

Langage de contrôle de données (DCL) : s'occupe de gérer les droits d'accès aux tables.

Les instructions du DCL sont : GRANT, REVOKE.

Langage de contrôle de transaction (TCL) : gère les modifications faites par le LMD, c'est-à-dire les caractéristiques des transactions et la validation et l'annulation des modifications.

Les instructions du TCL sont : COMMIT, SAVEPOINT, ROLLBACK, SET TRANSACTION

SQL intégré : permet d'utiliser SQL dans un langage de troisième génération (C, Java, PHP, etc.) :

- déclaration d'objets ou d'instructions ;
- exécution d'instructions ;
- gestion des variables et des curseurs ;
- traitement des erreurs.

Les instructions du SQL intégré sont : DECLARE, TYPE, DESCRIBE, VAR, CONNECT, PREPARE, EXECUTE, OPEN, FETCH, CLOSE, WHENEVER.

2.2. Ob jets manipulés par SQL

Identificateurs SQL : utilise des identificateurs pour désigner les objets qu'il manipule : utilisateurs, tables, colonnes, index, fonctions, etc.

Tables : Les relations d'un schéma relationnel sont stockées sous forme de tables composées de lignes et de colonnes**.**

Colonnes : Les données contenues dans une colonne doivent être toutes d'un même type de données. Ce type est indiqué au moment de la création de la table qui contient la colonne. Chaque colonne est repérée par un identificateur unique à l'intérieur de chaque table. Deux colonnes de deux tables différentes peuvent porter le même nom. Il est ainsi fréquent de donner

le même nom à deux colonnes de deux tables différentes lorsqu'elles correspondent à une clé étrangère à la clé primaire référencée.

2.3. Types de données

Les différents types de données spécifiés par SQL et leur disponibilité sur les différents SGBD sont :

a. Types alphanumériques

CHAR : valeurs alpha de longueur fixe
VARCHAR: valeur alpha de longueur maximale fixée

b. Types numériques

NUMERIC : nombre décimal à représentation exacte à échelle et précision facultatives.
INTEGER (ou **INT**): entier long.
SMALLINT : entier court.
FLOAT : réel à virgule flottante dont la représentation est binaire à échelle et précision obligatoire.
REAL : réel à virgule flottante dont la représentation est binaire, de faible précision.
DOUBLE PRECISION : réel à virgule flottante dont la représentation est binaire, de grande précision.
BIT : chaîne de bit de longueur fixe.

c. Types temporels

DATE : date du calendrier grégorien.
TIME : temps sur 24 heures.
TIMESTAMP : combiné date temps.
INTERVAL : intervalle de date / temps.

d. Types " BLOBS "

Longueur maximale prédéterminée, donnée de type binaire, texte long voire formaté, structure interprétable directement par le SGBDR ou indirectement par add-on externes (image, son, vidéo...).

On trouve souvent les éléments suivants :
TEXT : suite longue de caractères de longueur indéterminée.
IMAGE : stockage d'image dans un format déterminé.
OLE : stockage d'objet OLE (Windows).

2.4. Langage de définit ion de données

Le Langage de Définition des Données est la partie de SQL qui permet de créer une base de donnée, décrire les tables et autres objets manipulés par les SGBD. Les Commandes sont : (CREATE) pour créer, (ALTER) pour modifier et (DROP) pour supprimer les éléments du schéma relationnel tel que la base de données, relations, contraintes, …

a. Manipulation de la base de données

CREATE DATABASE

- Syntaxe: **CREATE DATABASE < nom_base_de_données >**
- Effets**:** Crée une base de donnée vide et prête à l'emploi

ALTER DATABASE

- Syntaxe**:** **ALTER DATABASE < nom_base_de_données >**
- Effets: Permet d'effectuer des opérations de maintenance

DROP DATABASE

- Syntaxe : **DROP DATABASE < nom_base_de_données >**
- Effets**:** Efface la base de données entière

b. Manipulation des tables

CREATE TABLE

- Syntaxe:

```
CREATE TABLE nom_table(
    Attribut1   TYPE_Att1,
    Attribut2   TYPE_Att2,
   CONSTRAINT PK_nomTable
                PRIMARY KEY (Attribut1, Attribut2,…)
   CONSTRAINT FK_nomTable_attributi
                FOREIGN KEY (Attributi)
                        REFERENCES nomTablei(Attributi),
   CONSTRAINT CK_nomTable_Attributj
                CHECK (condition_sur_attributj)
) ;
```

- Effets: Création de la table et définition des contraintes :

1. Création de la table **nom_table : CREATE TABLE nom_table**
2. Définir les noms des colonnes de la table et leur type **: Attribut1 TYPE_Att1**
3. Définir les contraintes : **CONSTRAINT** :

- Clé primaire :
  CONSTRAINT PK_nomTable **PRIMARY KEY** (Attribut1, Attribut2,…)
  Avec PK_nomTable **PRIMARY KEY** (Attribut1, Attribut2,…)
- Clé étrangère :
  CONSTRAINT FK_nomTable_attributi **FOREIGN KEY** (Attributi)
  REFERENCES nomTablei(Attributi),
- Contrainte de domaine :
  CONSTRAINT CK_nomTable_Attributj
  CHECK (condition_sur_attributj)

DROP TABLE

- Syntaxe: **DROP TABLE < nom_table >**
- Effets**:** supprimer la table

DESCRIBE (ou DESC)

- Syntaxe: **DESCRIBE (ou DESC) nomTable**
- Effets**:** Afficher la définition d'une table

ALTER /ADD

- Syntaxe: **ALTER TABLE nom_table ADD (attribut1 TYPE, …)**
- Effets**:** Ajout d'attributs dans une table

ALTER / DROP COLUMN

- Syntaxe: **ALTER TABLE nom_table DROP COLUMN nom_attribut;**
- Effets**:** Suppression d'attribut d'une table

ALTER / MODIFY

- Syntaxe: **ALTER TABLE nom_table MODIFY (attribut TYPE);**
- Effets**:** Modification d'attributs

ALTER / DROP CONSTRAINT

- Syntaxe: **ALTER TABLE nom_table DROP CONSTRAINT nom_constraint;**
- Effets**:** Suppression de contraintes

ALTER / ADD CONSTRAINT

- Syntaxe: **ALTER TABLE nom_table ADD CONSTRAINT nom_constraint…;**
- Effets**:** Ajout de contraintes

2.5. Langage de manipulation de données – Modifier une base

a. Insertion de ligne

La commande INSERT permet d'insérer une ligne dans une table en spécifiant les valeurs à insérer. La syntaxe est la suivante :

```
INSERT INTO nom_table(nom_col_1, nom_col_2, …) VALUES (val_1, val_2, …)
```

La liste des noms de colonne est optionnelle. Si elle est omise, la liste des colonnes sera par défaut la liste de l'ensemble des colonnes de la table dans l'ordre de la création de la table. Si une liste de colonnes est spécifiée, les colonnes ne figurant pas dans la liste auront la valeur NULL.

b. Modification de ligne

La commande UPDATE permet de modifier les valeurs d'une ou plusieurs colonnes, dans une ou plusieurs lignes existantes d'une table. La syntaxe est la suivante :

```
UPDATE nom_table SET attribut= Valeur  [WHERE condition];
```

c. Suppression de ligne

La commande DELETE permet de supprimer des lignes d'une table. La syntaxe est la suivante :

```
DELETE FROM nom_table [WHERE condition];
```

2.6. Langage de manipulation de données - Interroger une base

SQL est un langage déclaratif qui permet d'interroger une base de données sans se soucier de la représentation interne (physique) des données, de leur localisation, des chemins d'accès ou des algorithmes nécessaires.

La commande SELECT constitue, à elle seule, le langage permettant d'interroger une base de données. Elle permet de :

- sélectionner certaines colonnes d'une table (projection) ;
- sélectionner certaines lignes d'une table en fonction de leur contenu (sélection) ;
- combiner des informations venant de plusieurs tables (jointure, union, intersection, différence et division) ;
- combiner entre elles ces différentes opérations.

Une requête (i.e. une interrogation) est une combinaison d'opérations portant sur des tables et dont le résultat est lui-même une table dont l'existence est éphémère (le temps de la requête).

La syntaxe d'une requête est la suivante :

```
SELECT [DISTINCT] nom-cols [AS nom-cols]
FROM nom-tables
[ WHERE conditions]
[ GROUP BY nom-cols]
[ HAVING conditions ]
[ ORDER BY nom-cols [ASC|DESC] ]
```

- **SELECT** permet de récupérer l'information contenue dans la base de données qui satisfait les conditions de la requête.
- **DISTINCT** spécifie d'enlever les doublons dans les résultats.
- **nom-cols** spécifie quelles colonnes retenir (projection). Le caractère ***** récupère toutes les colonnes des tables précisées dans la clause **FROM**.
- **AS** permet de donner un nom aux colonnes créées par la requête.
- **WHERE** spécifie les conditions de sélection des tuples (sélection), qui peuvent être n'importe quel prédicat qui doit pouvoir être évalué à vrai ou faux selon les valeurs :

    - Comparaison à une valeur **(<, <=, =, >=, >, <>)**. quantifiés (**ALL, SOME, ANY**)

- Comparaison à une fourchette (**[NOT] BETWEEN**), comparaison partielle (**[NOT] LIKE**)
- Opérateurs logiques: **AND, OR, NOT**
- Conditions d'existence: **nom-col IS [NOT] NULL, [NOT] EXISTS** (sous-requête)
- Conditions d'ensemble: **nom-col [NOT] IN** (liste-de-valeur ou sous-requête)

– **GROUP BY** permet de regrouper les données dans la table selon les colonnes sélectionnées. Peut s'utiliser avec des fonctions d'agrégation: **count, sum, avg, max, min,...** dans la clause **SELECT**
– **HAVING** spécifie les conditions de rétention des regroupements de lignes
– **ORDER BY** permet de trier les données dans la table selon les colonnes sélectionnées en ordre ascendant (**ASC**) ou descendant (**DESC**)

3. Les Vues

3.1. Définition

Une vue est une table contenant des données calculées sur celle d'une autre table, mais n'occupant pas d'espace disque pour les données. Les données d'une vue sont tout le temps à jour. Si vous modifiez les données d'une des tables sur lesquelles est calculée la vue, alors les modifications sont automatiquement répercutées sur la vue.

Une vue peut être considérée comme une requête enregistrée. Ainsi vous pouvez réutiliser une instruction sans avoir à la redéfinir.

3.2. Avantage des vues

Une vue vous permet d'effectuer les tâches suivantes :

- Limiter l'accès d'un utilisateur à certaines lignes d'une table ;
- Limiter l'accès d'un utilisateur à certaines colonnes d'une table ;
- Joindre les colonnes de différentes tables ;
- Une vue peut être utilisée partout où on peut utiliser une table ;

3.3. Création d'une vue

On peut créer une vue en utilisant le langage de définition de données.

La commande **CREATE VIEW** permet de créer une vue en spécifiant le **SELECT** constituant la définition de la vue :

```
CREATE VIEW nom_vue [(nom_col1,...)]
[WITH SCHEMABINDING]
AS
Instruction SELECT [WITH CHECK OPTION];
```

La spécification des noms de colonnes de la vue est facultative. Par défaut, les noms des colonnes de la vue sont les mêmes que les noms des colonnes du **SELECT**.

Si certaines colonnes du SELECT sont des expressions, il faut renommer ces colonnes dans le **SELECT**, ou spécifier les noms de colonne de la vue.

Une fois créée, une vue s'utilise comme une table. Il n'y a pas de duplication des données.

Le **CHECK OPTION** permet de vérifier que la mise à jour ou l'insertion faite à travers la vue ne produisent que des lignes qui font partie de la sélection de la vue.

La clause **WITH SCHEMABINDING** permet d'empêcher la modification ou la suppression des objets référencés par la vue.

3.4. Suppression d'une vue

L'instruction **DROP VIEW** permet de supprimer une vue.

```
DROP VIEW nom_vue;
```

3.5. Modifier une vue

L'instruction **REPLACE VIEW** permet de modifier la définition de la vue.

```
CREATE OR REPLACE VIEW nom_vue;
```

Chapitre 2 : PL/SQL

PL/SQL est une extension du langage SQL propre à Oracle, qui permet de grouper des commandes et de les soumettre au noyau d'Oracle comme un bloc unique de traitement. Cette extension comprend le LMD SQL, des opérateurs, des curseurs et du traitement de transactions, combiné avec des instructions d'un langage de programmation style ADA.

Intérêt

- Langage procédural plus portable.
- Permet d'intégrer du code dans des outils classiques (par exemple, un script SQL*Plus, un programme en C précompilé par Pro*C, peuvent contenir des blocs de sous-programmes en PL/SQL).
- Traitement de transactions.
- Construction de procédures ou fonctions stockées qui améliorent le mode client-serveur par stockage de celles régulièrement utilisées au niveau du serveur.
- Construction de triggers qui renforcent la sécurité (intégrité) des données.
- Améliore la performance : les procédures et fonctions stockées sont compilées et peuvent être communes à plusieurs programmes. Il en découle également un moindre trafic réseau.
- La maintenance et le test des programmes est facilitée (intégration complète dans Oracle Server)

Utilisation

Les morceaux de programme écrits en PL/SQL peuvent être utilisés de deux manières :

- ***Bloc PL/SQL anonyme*** *:* ni nommé, ni enregistré dans la base de données, constitué de code PL/SQL inclus en général là où l'on peut mettre des commandes SQL. Un tel bloc est le plus souvent utilisé de manière interactive (via SQL*Plus, un précompilateur,…) et sert à appeler des procédures (ou fonctions) stockées ou à ouvrir des curseurs variables.
- ***Procédure (ou fonction ou « package »)*** *:* code nommé, enregistré dans le dictionnaire des données, pouvant être appelé et utilisé par plusieurs applications.

Le moteur PL/SQL qui traite les unités de programme écrites en PL/SQL fait partie de plusieurs produits d'Oracle, dont Oracle Server, et les outils Oracle Forms, Oracle Reports, Oracle Graphics de Developper/2000 (famille d'outils pour le développement d'applications clients-serveur).

1. Structure en blocs d'un programme PL/SQL

Tout code écrit dans un langage procédural est formé de blocs. Chaque bloc comprend une section de déclaration de variables, et un ensemble d'instructions dans lequel les variables déclarées sont visibles.

La syntaxe est :

```
DECLARE
        /* declaration des types, variables et constantes  */
        /* declaration des procédures et fonctions  */
        /* declaration des curseurs  */
        /* declaration des exceptions  */
BEGIN
        /* instructions a executer */
EXCEPTION
        /* section de gestion des erreurs */
END
/
```

2. Affichage

Pour afficher le contenu d'une variable, les procédures **DBMS_OUPUT.PUT()** et **DBMS_OUPUT.PUT_LINE()** prennent en argument une valeur à afficher ou une variable dont la valeur est à afficher. Par défaut, les fonctions d'affichage sont désactivées. Il convient, à moins que vous ne vouliez rien voir s'afficher, de les activer avec la commande SQL+ de oracle par : **SET SERVEROUTPUT ON.**

3. Les types

Types classiques

Ce sont les types d'Oracle :

- Numériques : **DECIMAL, DEC, DOUBLE PRECISION, INTEGER, INT, NUMBER, NUMERIC, REAL, SMALLINT.**
- Caractères : **VARCHAR2 (*longueur*), CHAR (*longueur*).**
- Date : **DATE** *codée sur 7 octets*
- **ROW, LONG RAW,**
- **BLOB, CLOB…**
- **ROWID :** l'adresse d'un n-uplet dans la base.

Remarque : une variable non initialisée prend la valeur NULL, ce qui peut conduire à des erreurs. Il vaut donc mieux initialiser les variables dès leur définition.

Exemple :

```
DECLARE     v_cte CONSTANT NUMBER :=10 ;
            v_num NUMBER NOT NULL := 0 ;
```

Types de données supplémentaires

Types simples :

BOOLEAN : prend les valeurs TRUE, FALSE, NULL.
BINARY_INTEGER : entier signé de type binaire à utiliser dans le cas de calcul.
NATURAL : sous-type de BINARY_INTEGER (positif ou nul).
POSITIVE : sous-type de BINARY_INTEGER (positif).

Types complexes : %TYPE et %ROWTYPE

Les variables PL/SQL correspondent le plus souvent à des valeurs stockées dans la base. D'où, les types :

- **%TYPE :** pour déclarer une variable du même type qu'une colonne :

```
nom_variable   nom_table.nom_colonne%TYPE
```

- **%ROWTYPE :** pour déclarer une variable du même type qu'un n-uplet :

```
nom_record  nom_table%ROWTYPE
```

Par exemple, on déclare une variable ***NOM VARCHAR2(20)*** pour recevoir les valeurs de la colonne ***NOM_ETU* de type *VARCHAR2(20)*** de la table ***ETUDIANT*** :

NOM ETUDIANT.NOM_ETU%TYPE

Et pour une variable ***var_etu*** de type **RECORD** (définit dans la section structures) destinée à recevoir les n-uplets de la table **ETUDIANT** :

var_etu ETUDIANT%ROWTYPE

4. Variables

Une variable se déclare de la sorte :

```
nom type [ := initialisation ] ;
```

L'initiation est optionnelle. Nous utiliserons les mêmes types primitifs que dans les tables. Par exemple:

```
DECLARE
      c_var  varchar2 (15) := ' Hello World ! ' ;
BEGIN
      DBMS_OUTPUT.PUT_LINE( c_var ) ;
END;
/
```

Les affectations se font avec la syntaxe **variable := valeur ;**

Affectation d'une valeur à une variable

L'affectation d'une valeur à une variable peut se faire de 3 manières :

- En utilisant l'opérateur **:=**
- Avec l'ordre **FETCH…INTO** *(voir le paragraphe sur les curseurs)*
- Avec l'ordre **SELECT…INTO**

5. Types de données composées

Les deux types de données composites de PL/SQL sont **TABLE** et **RECORD**. Le type de donnée **TABLE** permet à l'utilisateur de définir un tableau PL/SQL. Le type de données **RECORD** permet d'aller au-delà de l'attribut de variable **%ROWTYPE** ; avec ce type, on peut spécifier des champs définis par l'utilisateur et des types de données pour ces champs.

5.1. Tableaux statiques

Création d'un type tableau

Les types tableau doivent être définis explicitement par une déclaration de la forme

```
TYPE  nom_type  IS VARRAY (  taille ) OF  typeElements  ;
```

- **type :** est le nom du type tableau crée par cette instruction
- **taille :** est le nombre maximal d'éléments qu'il est possible de placer dans le tableau.
- **typeElements :** est le type des éléments qui vont être stockés dans le tableau, il peut s'agir de n'importe quel type.

Par exemple, créons un type tableau de nombres indicé de 1 à 10, que nous appellerons **numberTab**

```
TYPE  numberTab  IS  VARRAY (10) OF NUMBER;
```

Déclaration d'un tableau

Dorénavant, le type d'un tableau peut être utilisé au même titre que **NUMBER** ou **VARCHAR2**. Par exemple, déclarons un tableau appelé **tab** de type **numberTab**,

```
DECLARE
        TYPE numberTab IS VARRAY (10) OF NUMBER;
        tab numberTab ;
BEGIN
        /* instructions */
END;
/
```

Allocation d'un tableau

La création d'un type tableau met à disposition un constructeur du même nom que le type créé. Cette fonction réserve de l'espace mémoire pour ce tableau et retourne l'adresse mémoire de la zone réservée, il s'agit d'une sorte de malloc. Si, par exemple, un type tableau **numtab** a été crée, la fonction **numtab()** retourne une tableau vide.

```
DECLARE
          TYPE  numberTab  IS VARRAY (10) OF NUMBER;
          tab  numberTab ;
BEGIN
          tab := numberTab ( ) ;
          /* utilisation du tableau */
END;
/
```

Une fois cette allocation faite, il devient presque possible d'utiliser le tableau...

Dimensionnement d'un tableau

Le tableau retourné par le constructeur est vide. Il convient ensuite de réserver de l'espace pour stocker les éléments qu'il va contenir. On utilise pour cela la méthode **EXTEND(). EXTEND** s'invoque en utilisant la notation pointée. Par exemple,

```
DECLARE
        TYPE numberTab IS VARRAY (10) OF NUMBER;
        tab  numberTab ;
BEGIN
        tab := numberTab ( ) ;
        tab.EXTEND( 4 ) ;
      /* utilisation du tableau */
END;
/
```

Dans cet exemple, **tab.EXTEND(4)** permet par la suite d'utiliser les éléments du tableau t(1), t(2), t(3) et t(4). Il n'est pas possible "d'étendre" un tableau à une taille supérieure à celle spécifie lors de la création du type tableau associé.

Utilisation d'un tableau

On accède, en lecture et en écriture, à l'i-éme élément d'une variable tabulaire nommé T avec l'instruction T (i). Les éléments sont indicés à partir de 1.

Effectuons, par exemple, une permutation circulaire vers la droite des éléments du tableau t.

```
SET SERVEROUTPUT ON
DECLARE
        TYPE numberTab IS VARRAY (10) OF NUMBER;
        tab numberTab ;
        i number ;
        k number ;
BEGIN
        tab := numberTab ( ) ;
        tab .EXTEND( 1 0 ) ;

        FOR i IN 1 . . 1 0 LOOP
            tab ( i ) := i ;
        END LOOP;

                    k := tab ( 1 0 ) ;

          FOR i in REVERSE 2 . . 1 0 LOOP
                    tab ( i ) := tab ( i + 1 ) ;
          END LOOP;

                    tab ( 1 ) := k ;

          FOR i IN 1 . . 1 0 LOOP
                    DBMSOUTPUT.PUT_LINE( tab ( i ) ) ;
          END LOOP;
END;
/
```

5.2. Tableau dynamique :

Tableau de taille dynamique déclaré par

```
TYPE  nom_type  IS  TABLE  OF  type INDEX BY BINARY_INTEGER
```

Exemple : .....

```
TYPE tab IS TABLE OF INTEGER ;
var_tab        tab ;
```

var_tab(1) est un enregistrement de type INTEGER On peut alors écrire :

```
var_tab(1) := 20 ;
```

Remarque *:* L'option **INDEX BY BINARY_INTEGER** est facultative depuis la version 8 de PL/SQL. Si elle est présente, l'indexation ne commence pas nécessairement à 1 et peut être même négative (l'intervalle de valeurs du type BINARY_INTEGER va de – 2 147 483 647 à 2 147 483 647).

5.3. Fonctions et procédures associées à une table :

	TYPE RENVOYÉ	COMMENTAIRES
COUNT	NUMBER	Donne le nombre d'éléments de la table
DELETE	/	Efface les n-uplets dont on donne les indices
EXISTS	BOOLEAN	TRUE si l'indice spécifié appartient à la table
FIRST	BINARY_INTEGER	Indice du premier élément de la table
LAST	BINARY_INTEGER	Indice du dernier élément de la table
NEXT	BINARY_INTEGER	Indice du n-uplet suivant celui dont on donne l'indice
PRIOR	BINARY_INTEGER	Indice du n-uplet précédant celui dont on donne l'indice

5.4. Structures (tableau des enregistrements)

Une structure est un type regroupant plusieurs types. Une variable de type structuré contient plusieurs variables, ces variables s'appellent aussi des champs.

Création d'un type structuré

On définit un type structuré de la sorte :

```
TYPE    nomType    IS  RECORD
(
        /*liste des champs */
) ;
```

nomType est le nom du type structuré construit avec la syntaxe précédente. La liste suit la même syntaxe que la liste des colonnes d'une table dans un CREATE TABLE. Par exemple, construisons le type point (dans IR^2),

```
TYPE point IS RECORD
(
        abscisse NUMBER,
        ordonnee NUMBER
) ;
```

Notez bien que les types servant à définir un type structuré peuvent être quelconques : variables scalaires, tableaux, structures, etc.

Déclaration d'une variable de type structure point est maintenant un type, il devient donc possible de créer des variables de type point, la règle est toujours la même pour déclarer des variables en PL/SQL, par exemple **p point** ; permet de déclarer une variable **p** de type **point**.

Utilisation d'une variable de type structuré

Pour accéder à un champ d'une variable de type structuré, en lecture ou en écriture, on utilise la notation pointée : **v.c** est le champ appelé **c** de la variable structuré appelée **v**. Par exemple,

```
SET SERVEROUTPUT ON
DECLARE
        TYPE point IS RECORD
        (
            abscisse NUMBER,
            ordonnee NUMBER
        ) ;
        p point ;
BEGIN
        p . abscisse  := 1 ;
        p . ordonnee := 3 ;

        DBMSOUTPUT.PUT_LINE( 'p.abscisse = ' ||p.abscisse || '
        and p . ordonnee =' || p . ordonnee ) ;
END;
/
```

Le script ci-dessus crée le type **point**, puis crée une variable **P** de type **point**, et enfin affecte aux champs **abscisse** et **ordonnée** du **point p** les valeurs 1 et 3.

6. Structures de contrôle

6.1. Traitements conditionnels

Le **IF** et le **CASE** fonctionnent de la même façon que dans les autres langages impératifs :

```
IF   /*condition 1 */  THEN
        /*instructions 1 */
ELSE
        /* instructions 2 */
END
```

Voire:

```
IF   /* condition 1 */  THEN
            /* instructions 1 */
ELSIF     /* condition 2 */   THEN
```

```
            /* instructions 2 */
ELSE
            /* instructions 3 */
END IF
```

Les conditions sont les mêmes qu'en SQL. Le **switch** du langage C s'implémente en PL/SQL de la façon suivante :

```
CASE    /* variable */
        WHEN /* valeur 1 */ THEN
                /* instructions 1 */
        WHEN  /* valeur 2 */ THEN
                /* instructions 2 */
        . . .
        WHEN  /* valeur n */ THEN
                /* instructions n */
        ELSE
                /* instructions par défaut */
END CASE
```

6.2. Traitements répétitifs

a. Boucles

L'utilisation de la commande **LOOP** fournit un traitement itératif basé sur des choix logiques. La construction de base des boucles « LOOP » est montrée dans l'exemple suivant :

```
LOOP
        /* instructions  */
END  LOOP
```

Voire

```
<<nom>>
LOOP
            /* instructions  */
END LOOP nom;
```

Pour sortir d'une boucle de ce genre, il faut une commande **EXIT** ou **GOTO** basée sur une condition du traitement. En cas de levée d'exception définie par l'utilisateur, la boucle LOOP s'achève aussi.

```
LOOP
                /* instructions  */
EXIT WHEN   /* condition */
END  LOOP
```

Une boucle peut être nommée comme cela a été montré dans l'exemple en utilisant une étiquette telle que <<nom>> juste avant l'instruction LOOP. Bien que ce ne soit pas obligatoire, l'étiquetage permet de garder une meilleure trace de l'imbrication des boucles.

b. Boucles WHILE

La boucle WHILE vérifie l'état d'une expression PL/SQL qui doit s'évaluer à TRUE, FALSE ou NULL au début de chaque cycle de traitement.

```
WHILE (condition) LOOP
        /* instructions */
END LOOP;
```

Il est possible, en bidouillant d'implémenter la boucle DO ... WHILE...

c. Boucles FOR numériques

Les itérations de boucles peuvent être contrôlées avec des boucles FOR numériques. Ce mécanisme permet au développeur d'établir un intervalle d'entiers pour lesquels la boucle va être itérée.

```
FOR indice IN [REVERSE ]min . . max LOOP
        /* instructions */
END LOOP;
```

Le variable **indice** varie de **min** à **max** avec un pas de 1. Si REVERSE est précisé, indice varie de max à min avec un pas de -1.

7. Procédures et fonctions

Une procédure (non stockée) ne peut s'utiliser dans un bloc anonyme que si elle a été déclarée dans la partie déclaration de ce bloc (ou d'un bloc PL/SQL l'englobant). Sa déclaration peut comporter des arguments dont on déclare le nom, le mode et le type.

```
PROCEDURE nom_procédure [ liste_arguments ] IS
    [ Déclaration_variables_locales ]
BEGIN
        /* instructions */
[Section_exception]
END [nom_procédure] ;
```

argument := nom_argument [IN | IN OUT | OUT] type [{ := | DEFAULT} valeur]

Les modes de transmission des arguments sont :

- **IN :** transmission en lecture seule, mode par défaut.
- **OUT :** le paramètre peut être modifié mais ne peut pas être lu
- **IN OUT :** transmission en lecture – écriture

```
FUNCTION nom_fonction [ liste_arguments ] RETURN type_données IS
        [Déclaration_variables_locales]
BEGIN
        /* instructions */
[Section_exception]
END [nom_fonction] ;
```

8. Curseurs

PL/SQL utilise des curseurs pour tous les accès à des informations de la base de données. Le langage supporte à la fois l'emploi de curseurs implicites et explicites. Les curseurs implicites sont ceux qui sont établis lorsqu'un curseur explicite n'a pas été déclaré. Il faut utiliser des curseurs explicites ou des curseurs de boucles FOR dans toutes les requêtes qui renvoient plusieurs lignes.

Les curseurs implicites sont générés et gérés par le noyau pour chaque ordre SQL. Les curseurs explicites sont créés en programmation PL/SQL et utilisés pas le développeur pour gérer mes requêtes SELECT qui doivent rapporter plusieurs lignes.

8.1. Déclaration de curseurs

Les curseurs sont définis dans la zone des variables de sous-programmes PL/SQL en utilisant l'instruction **CURSOR nome IS**, comme montré dans l'exemple suivant :

```
CURSOR   nom_curseur  IS    /* instruction sql */
```

L'instruction SQL peut être n'importe quelle requête valide. Après l'initialisation d'un curseur, les actions d'un curseur peuvent être contrôlées avec les instructions **OPEN**, **FETCH** et **CLOSE**.

8.2. Le contrôle d'un curseur

Pour utiliser un curseur afin de manipuler des données, il faut utiliser l'instruction **OPEN nom_curseur** pour exécuter la requête et identifier toutes les lignes qui satisfont le critère de sélection. Les extractions ultérieures de lignes sont réalisées avec l'instruction **FETCH.** Lorsque toutes les données sont traitées, l'instruction **CLOSE** clôt toute activité associée avec le curseur ouvert. Ce qui suit est un exemple de contrôle de curseur :

```
OPEN    nom_curseur;
...
FETCH   nom_curseur   INTO  ligne_info;
          ...
          /* traitement de la ligne extraite */
          ...
CLOSE   nom_curseur;
```

Ce code ouvre le curseur **nom_curseur** et traite les lignes extraites. Après l'extraction et le traitement de toute l'information, le curseur est fermé. Le traitement des lignes extraites est typiquement contrôlé par des itérations de boucles comme discuté plus loin.

8.3. Attributs des curseurs explicites

Il y a quatre attributs associés aux curseurs PL/SQL.

- **%NOTFOUND**
- **%FOUND**
- **%ROWCOUNT**
- **%ISOPEN**

Tous les attributs de curseur s'évaluent à **TRUE**, **FALSE** ou **NULL**, en fonction de la situation.

L'attribut **%NOTFOUND** s'évalue à **FALSE** quand une ligne est extraite, **TRUE** si le dernier **FETCH** n'a pas renvoyé une valeur et **NULL** si le curseur **SELECT** n'a pas renvoyé de données.

L'attribut **%FOUND** est l'opposé logique de **%NOTFOUND** par rapport à **TRUE** et **FALSE**, mais s'évalue néanmoins à **NULL** si le curseur ne renvoie pas de données.

%ROWCOUNT peut être utilisé pour déterminer combien de rangées ont été sélectionnées à un moment donné dans le **FETCH**. Cet attribut est incrémenté après la sélection réussie d'une ligne. De plus, **%ROWCOUNT** est à zéro quand le curseur est ouvert pour la première fois.

Le dernier attribut, **%ISOPEN**, est ou bien **TRUE** ou bien **FALSE**, suivant que le curseur associé est ouvert ou non. Avant que le curseur ne soit ouvert et après qu'il soit fermé, **%ISOPEN** vaut **FALSE**. Dans les autres cas, cet attribut s'évalue à **TRUE**.

8.4. Paramètre des curseurs

On peut spécifier des paramètres pour les curseurs de la même manière que pour des sous-programmes.
L'exemple suivant illustre la syntaxe de déclaration de curseurs avec des paramètres :

```
CURSOR nom_curseur (entrée IN NUMBER) IS
SELECT  attrib1, attrib2
FROM table
WHERE  entrée = attrib3;
```

Le mode des paramètres est toujours **IN**, mais les types de données peuvent être n'importe quels types de données valides. Un paramètre de curseur ne peut être référencé que pendant la requête déclarée. La flexibilité au sein des paramètres de curseurs permet au développeur de passer différents nombres de paramètres à un curseur en utilisant le mécanisme des paramètres par défaut. Ceci est illustré dans l'exemple ci-dessous :

```
CURSOR   c_line_item
(order_num INTEGER DEFAULT 100,
line_num INTEGER DEFAULT 1) IS ...
```

En utilisant la déclaration INTEGER DEFAULT, on peut passer tous, un, ou aucun des paramètres de ce curseur en fonction du code appelant.

9. Gestion des erreurs et exceptions

En PL/SQL, un avertissement (warning) ou une condition d'erreur sont appelés **exceptions**. Ces exceptions peuvent être soit définies en interne par Oracle, soit définies par l'utilisateur. Les exceptions courantes ont des noms prédéfinis, par exemple « ZERO_DIVIDE ».

Quand une erreur se produit, une exception est levée : l'exécution normale du programme est arrêtée et le contrôle est transféré au gestionnaire d'exception (partie spécifique du bloc PL/SQL ou du sous-programme concerné). Les exceptions prédéfinies sont levées automatiquement. Les exceptions définies par l'utilisateur doivent être levées explicitement par un ordre **RAISE**.

Après exécution des ordres du gestionnaire d'exception, le contrôle est redonné au bloc qui englobait le sous-programme arrêté lors de la levée de l'exception ou, à défaut, à l'environnement du programme.

Les avantages des exceptions sont multiples :

- Cela évite de répéter plusieurs fois le même test d'erreur au cours d'un programme.
- Cela augmente la fiabilité en évitant les oublis de tests d'erreur.
- Cela améliore la lisibilité du programme.

On distingue trois types d'exceptions en PL/SQL.

9.1. Exceptions prédéfinies

Nom d'exception prédéfinie	Erreur Oracle associée	Code	Valeur SQLCODE
CURSOR_ALREADY_OPEN	Ouverture d'un curseur déjà ouvert	ORA_06511	-6511
DUP_VAL_ON_INDEX	Insertion d'une valeur déjà présente dans une colonne	ORA_00001	-1
INVALID_CURSOR	Opération illégale sur un curseur	ORA_01001	-1001
INVALID_NUMBER	Mauvaise conversion en SQL d'une chaîne de caractères	ORA_01722	-1722
LOGON_DENIED	Mauvais login/password lors de la connexion à Oracle	ORA_01017	-1017
NO_DATA_FOUND	Pas de valeur retournée pour SELECT…INTO ou rèf.	ORA_01403	-1403
NOT_LOGGED_ON	Appel à la base de données avant connexion	ORA_01012	-1012
PROGRAMM_ERROR	Problème interne à PL/SQL	ORA_06501	-6501
STORAGE_ERROR	Mémoire corrompue ou pas assez de mémoire	ORA_06500	-6500
TIMEOUT_ON_RESOURCE	Oracle a attendu trop longtemps une ressource	ORA_00051	-0051
TOO_MANY_ROWS	Plus d'une valeur retournée par SELECT…INTO	ORA_01422	-1422
VALUE_ERROR	Erreur arithmétique, de troncature, de taille, de	ORA_06502	-6502
ZERO_DIVIDE	Division d'un nombre par zéro	ORA_01476	-1476

Pour traiter les erreurs nom nommées, on peut utiliser la clause **WHEN OTHERS THEN**… dans le gestionnaire d'exception avec les fonctions **SQLCODE** et **SQLERRM**. On peut aussi associer une erreur Oracle à un nom d'exception donné par l'utilisateur.

9.2. Définies par l'utilisateur

- Elles sont déclarées dans la partie déclaration sous la forme

```
nom_exception   EXCEPTION ;
```

- Elles sont levées dans la partie programme par

```
RAISE   nom_exception ;
```

- Elles sont définies dans la partie **EXCEPTION** appelée gestionnaire d'exception

```
EXCEPTION
WHEN  nom_exception1  THEN  code_exception1 ;
WHEN  nom_exception2  THEN  code_exception2 ;
```

```
……..
WHEN OTHERS THEN  code_dernière_exception ;
END ;
```

Remarque : WHEN OTHERS (toujours placé à la fin) dans le gestionnaire d'exception permet d'intercepter toute exception. Ici, il permet d'afficher le code et le message d'erreur ORACLE.

9.3. Utilisation de RAISE_APPLICATION_ERROR

Oracle permet de traiter des erreurs définies en code PL/SQL par l'utilisateur : un numéro d'erreur précisé par l'utilisateur et un message sont retournés à l'application. Celle-ci peut traiter l'erreur en fonction de son numéro et du message. C'est la procédure **RAISE_APPLICATION_ERROR** (du package DBMS_STANDARD) qui délivre le message.

La procédure **RAISE_APPLICATION_ERROR** est souvent utilisée par les gestionnaires d'exceptions ou dans la logique du programme. Un programme ne peut appeler cette procédure qu'à l'exécution d'un sous-programme stocké. Cette procédure termine l'exécution de l'action en cours, annule les effets de cette action et retourne un code d'erreur et un message d'erreur.

La syntaxe pour appeler la procédure est :

```
RAISE_APPLICATION_ERROR(numéro_erreur,'texte'[,{TRUE|FALSE}])
```

- **numéro d'erreur** est un nombre entre –20000 et –20999 (valeurs de codes d'erreur Oracle réservées pour l'utilisateur)
- **texte** est un message d'au plus 2 Koctets
- optionnel, **TRUE** indique que l'erreur est placée sur la pile des erreurs précédentes, sinon l'erreur remplace les précédentes (**FALSE** est l'option par défaut).

10. Procédures stockées

Une procédure stockée est composée d'instructions compilées et enregistrées dans la BD. Elle est activée par des événements ou des applications. Elle comporte, outre des ordres SQL, des instructions de langage PL/SQL (branchement conditionnel, instructions de répétition, affectations,…).

L'intérêt d'une procédure stockée est :

- D'alléger les échanges entre client et serveur de BD en stockant au niveau du serveur les procédures régulièrement utilisées.
- D'optimiser les requêtes au moment de la compilation des procédures plutôt qu'à l'exécution.
- De renforcer la sécurité : on peut donner l'autorisation à un utilisateur d'utiliser une procédure stockée sans lui donner les droits directement sur les tables qu'elle utilise.

10.1. Procédures.

On définit une procédure de la sorte

```
CREATE [OR REPLACE] PROCEDURE  nom_procedure  [ liste_argument1 ] {IS|AS}
        [ déclaration_variables_locales ]
BEGIN
```

```
        Instructions PL/SQL
        [Gestions des exceptions]
END [nom_procedure]
```

Syntaxe de liste_argument :

```
nom_argument [ IN|OUT ] type_données [ { := | DEFAULT} valeur ]
```

IN : Paramètre d'entrée
OUT : Paramètre de sortie
IN OUT : Paramètre d'entrée/Sortie

L'exécution de l'ordre CREATE PROCEDURE … déclenche :

- La compilation du code source avec génération de pseudo-code si aucune erreur n'est détectée.
- Le stockage du code source dans la base même si une erreur a été détectée.
- Le stockage du pseudo-code dans la base, ce qui évite la recompilation de la procédure à chaque appel de celle-ci.

10.2. Fonctions

On définit une fonction de la sorte

```
CREATE [OR REPLACE] FUNCTION  nom_fonction [(liste_argument)] {IS|AS} RETURN
type_données {IS|AS}
        [déclaration_variables_locales]
BEGIN
        Instructions PL/SQL
        [Gestions des exceptions]
END [nom_fonction]
```

10.3. Modification – suppression (d'une procédure ou fonction).

On ne peut pas modifier le texte d'une procédure ou fonction, il faut la recréer.

Pour recompiler une procédure/fonction stockée :

```
ALTER {FUNCTION | PROCEDURE}  nom_proc  COMPILE
```

Attention : ceci ne s'applique pas aux procédures/fonctions d'un package.

Pour supprimer :

```
DROP {FUNCTION | PROCEDURE}  nom_proc
```

Invocation

En PL/SQL, une procédure stockée (procédure ou fonction) s'invoque tout simplement avec son nom. Mais on doit utiliser le mot-clé **CALL** ou **EXECUTE.**

```
EXECUTE   nom_proc
```

11. Package

Un package (paquetage) est l'encapsulation d'objets de programmation PL/SQL dans une même unité logique de traitement tels : types, constantes, variables, procédures et fonctions, curseurs, exceptions.

Il comporte:

- Une partie spécification qui déclare types, variables, …, visibles à l'extérieur.
- Un corps qui définit les objets déclarés dans la spécification et ceux non visibles à l'extérieur du package

Il permet une meilleure méthodologie de programmation, permettant de :

- Modifier le corps sans recompiler la spécification, ni changer les programmes qui font appel aux objets du package.
- Définir des variables globales et des curseurs communs à toutes les procédures/fonctions du package.
- Gérer plus efficacement les privilèges à accorder.
- De meilleures performances ; tout le package est chargé en mémoire lors de sa première utilisation.

Syntaxe de création :

```
CREATE [OR REPLACE] PACKAGE nom_package {AS|IS}…
                    Déclarations en PL/SQL
END [nom_package] ;

CREATE  PACKAGE BODY nom_package {AS|IS}…
                    Instructions PL/SQL
END [nom_package];
```

Suppression : DROP PACKAGE nom_package ; (supprime spécifs et corps)

DROP PACKAGE BODY … (supprime juste le corps)

12. Triggers

Un déclencheur (trigger) est un ensemble d'actions déclenchées automatiquement par le SGBD chaque fois qu'un événement défini se produit. Cet ensemble est enregistré dans la base et non dans des programmes d'application.

Un trigger Oracle est une procédure stockée associée à une table et qui est exécutée chaque fois qu'une modification précisée dans le trigger affecte la table.

Type d'actions :

Lors de la création d'un trigger, il convient de préciser quel est le type d'événement qui le déclenche.

Nous réaliserons dans ce cours des triggers pour les événements suivants :

- INSERT
- DELETE
- UPDATE

Moment de l'exécution

On précise aussi si le trigger doit être exécuté avant (BEFORE) ou après (AFTER) l'événement.

Evénements non atomiques

Lors que l'on fait un DELETE ..., il y a une seule instruction, mais plusieurs lignes sont affectées. Le trigger doit-il être exécuté pour chaque ligne affectée (FOR EACH ROW), ou seulement une fois pour toute l'instruction (STATEMENT) ?

- Un FOR EACH ROW TRIGGER est exécuté à chaque fois qu'une ligne est affectée.
- Un STATEMENT TRIGGER est exécuté à chaque fois qu'une instruction est lancée.

Création

```
CREATE OR REPLACE TRIGGER nom_trigger
      [BEFORE | AFTER]   [INSERT | DELETE | UPDATE]  ON nomtable
      [FOR EACH ROW]
     /* declarations */
BEGIN
     /* instructions */
END;
```

Accès aux lignes en cours de modification

Dans les **FOR EACH ROW** triggers, il est possible avant la modification de chaque ligne, de lire l'ancienne ligne et la nouvelle ligne par l'intermédiaire des deux variables structurées old et new.

Par exemple :

- Si nous ajoutons un client dont le nom est toto alors nous récupérons ce nom grâce à la variable **:new.nom**
- Dans le cas de suppression ou modification, les anciennes valeurs sont dans la variable **:old.nom**

Modification

On ne peut modifier la définition du trigger, il faut la remplacer (d'où l'intérêt du CREATE OR REPLACE).

Quand on crée le trigger, il est automatiquement activé. On peut le désactiver, puis le réactiver :

ALTER TRIGGER nom_trigger **{disable|enable|compile}** *pour désactiver/réactiver/recompiler*
ALTER TABLE nom_table DISABLE ALL TRIGGERS *pour désactiver tous les triggers d'une table*
DROP TRIGGER nom_trigger *suppression d'un trigger*

UNIVERSITE IBN TOFAIL
Faculté des sciences
Département d'informatique
Kenitra

Année universitaire : 2014/2015
Filière : SMI
Semestre : 6

Base de données II et Web

Série n°1 : Rappel SQL, Vues

Soit la base de données composée des tables suivantes :

CLIENT (**NumCli**, Nom, Prénom, DateNaiss, Rue, CP, Ville)
PRODUIT (**NumProd**, Desig, PU, #NumFour)
FOURNISSEUR (**NumFour**, RaisonSoc)
COMMANDE **(#NumCli, #NumProd**, DateC, Quantité)

Tableau 1 : Client

NumCli	Nom	Prénom	DateNaiss	Rue	CP	Ville
1	Hilmi	Samir	1984- 02-12	Massira	100023	Rabat
2	Alami	Kamel	1988-12-14		100024	Rabat
3	Saidi	Ali		Nasser	100025	Kenitra

Tableau 2 : Produit

NumProd	Design	PU	NumFour
10	Ordinateur	1100	1
20	Imprimante	2045,25	3

Tableau 3 : Commande

NumCli	NumProd	DateC	Quantite
1	10	2014-02-12	2
3	20	2014-07-16	4
3	10	2014-09-20	6
1	20	2014-11-26	3

Tableau 4 : Fournisseur

NumFour	RaisonSoc
1	SOS Computer
2	Tele-surveillance
3	Toush-Mag

Exercice 1 : Description de la base et des tables

1. Créez la base de données COMMERCE
2. Créez les tables : Client, Fournisseur, Produit et Commande en respectant les conditions suivants :
   - Prix unitaire ne doit pas dépasser 30000DH,
   - La quantité commandée doit être supérieure à zéro.

Exercice 2 : Les vues

1. Créez la **vue CLIENT_RABAT** à partir de la table **CLIENT** ne contenant que les clients habitant à la ville de Rabat Toutes les colonnes sont conservée et portent le même nom que les colonnes de la table. Interroger la vue (DESC et SELECT).
2. Créer la **vue PRODUIT_1 (NP_1, NumFour_1, PrixUnit_1, Desig_1)** à partir de la table **PRODUIT** ne contenant que les produits du fournisseur dont la **raison sociale** est « SOS Computer »

3. Insérer avec INSERT trois nouveaux produits dans la vue **PRODUIT_1** (exemple **Caméra, Scanner et papier**). Tels que **caméra** et **scanner** appartiens au fournisseur dont la raison sociale est « SOS Computer » et **papier** à un autre fournisseur.
    a. Vérifier le contenu de la vue et celui la table PRODUIT. **Conclure ?**
    b. Supprimer ensuite les enregistrements ajoutés dans la table PRODUIT (par l'intermédiaire de la vue **PRODUIT_1**). **Conclure ?**

4. Créer la vue **CLIENT_CMD (NCLI, NOMPRENOM, ADR, REFCMD, MHT, MTVA, MTTC, DATEC)** permettant d'avoir la liste des clients qui ont commandé.
    a. Vérifier le contenu de la vue avec SELECT.
    b. Afficher la liste des clients qui ont commandé au cours du mois de février 2009.
    c. Essayer de mettre à jour la vue **CLIENT_CMD**. **Conclure ?**

5. Créer la vue **CLIENT_NBCMD (NCLI, NOM, PRENOM, ADR, NBRCMD)** permettant d'avoir le nombre de commande pour chaque client.
    a. Vérifier le contenu de la vue avec SELECT.
    b. Afficher la liste des clients qui n'ont pas de commande.

UNIVERSITE IBN TOFAIL
Faculté des sciences
Département d'informatique
Kenitra

Année universitaire : 2014/2015
Filière : SMI
Semestre : 6

Base de données II et Web

Série n°2 : Introduction au PL/SQL, Tableau et Structure, Procédure, Fonction

Exercice 1 :

Ecrivez un programme affectant les valeurs 1 et 2 à deux variables a et b, puis permutant les valeurs de ces deux variables.

Exercice 2 :

Ecrivez un programme plaçant la valeur 10 dans une variable a, puis affichant la factorielle de a.

Exercice 3 :

Ecrivez un programme plaçant les valeurs 48 et 84 dans deux variables a et b puis affichant le pgcd de a et b.

Exercice 4 :

1. Créez un type tableau pouvant contenir jusqu'à 50 entiers.
2. Créez une variable de ce type, faites une allocation dynamique et dimensionnez ce tableau à 20 emplacements.
3. Placez dans ce tableau la liste des 20 premiers carrés parfaits : 1, 4, 9, 16, 25, …
4. Inversez l'ordre des éléments du tableau
5. Affichez le tableau.

Exercice 5 :

1. Créez une structure « RECORD » client reprenant les informations relatives à un client
2. Placez dans ce RECORD les informations liées au client né le « 1988-12-14 »

Exercice 6 :

1. Créez une Fonction **PRIX_TTC** qui calcule le prix TTC des ordinateurs commandé par chaque client.
2. Créez une Procédure **Cli_prod** qui affiche le nom de client suivi de la désignation de produit qu'il a commandé.

UNIVERSITE IBN TOFAIL
Faculté des sciences
Département d'informatique
Kenitra

Année universitaire : 2014/2015
Filière : SMI
Semestre : 6

Base de données II et Web

Série n°3 : Exception, sous-programmes, curseur, Trigger

Soit la table suivante :

PERSONNE (**NumPers**, Nom, Prénom, Code_Postale, #PERE, #MERE)
Père et Mère sont des clés étrangères représentées par des numéros PERSONNES

Exercice 1 :

Ecrire une fonction récursive retournant b^n, avec n entier positif ou nul.

Exercice 2 :

Ecrire une fonction demi-frères prenant deux numéros de personnes en paramètre et retournant vrai si et seulement si ces deux personnes ont un parent en commun.

Exercice 3 :

Ecrire une fonction cousins prenant deux numéros de personnes en paramètre et retournant vrai si et seulement si ces deux individus sont cousins.

Exercice 4 :

Ecrire une procédure récursive affichant le nom de la personne dont le numéro est passe en paramètre et se rappelant récursivement sur le père de cette personne.

Exercice 5 :

En utilisant la table PERSONNE, écrivez une fonction affichant toute la descendance d'une personne. Vous devez utilisez un curseur.

Exercice 6 :

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

1. Le Code Postale dans lequel habite le personne doit être 01, 07, 26, 38, 42, 69, 73, ou 74 ;
2. Le nom du Père doit être le même que celui du personne.